

# Incremental Process Discovery

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der  
RWTH Aachen University zur Erlangung des akademischen Grades  
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

**Daniel Schuster, M. Sc. M. Sc.**

aus Kassel

Berichter: Univ.-Prof. Prof. h. c. Dr. h. c. Dr. ir. Wil M. P. van der Aalst  
Prof. Dr. ir. Boudewijn F. van Dongen

Tag der mündlichen Prüfung: 19. Juni 2024

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.



---

# Abstract

---

Many organizational processes rely on information systems to support operational functions such as administration, finance, production, and logistics. These systems track process executions in great detail, generating event data that contain valuable information about process executions. Process mining analyzes these event data and yields crucial insights into the processes, such as process models, conformance diagnostics, and performance metrics. Process analysts and owners can use the derived insights to understand how processes are executed in practice and ultimately optimize them, for example, by reducing cycle times, improving resource allocation, and enhancing conformity. Overall, process mining aims to improve processes through data-driven approaches.

Process discovery is concerned with learning process models from event data and is a fundamental task within process mining. However, most existing process discovery algorithms are fully automated, i.e., they operate as black boxes from the users' perspective, discover process models in a one-shot fashion, devoid of user interaction, and often discover subpar models, particularly when applied to real-world data. Moreover, these process discovery algorithms fail to exploit domain knowledge beyond event data.

This thesis presents a framework for incremental process discovery that allows users to learn and refine process models from event data iteratively. Thereby, users can observe intermediate process models learned so far. Further, users can manually edit intermediate process models before they are fed back into the incremental process discovery framework for further learning. Moreover, users can selectively incorporate process behaviors from event data. In short, we propose an incremental process discovery framework that allows users to interact and steer the discovery phase of a process model. We further extend the incremental process discovery framework as follows. First, we allow the gradual addition of process execution fragments alongside complete process executions. Most automated process discovery algorithms assume complete process executions that span the process from start to end. In contrast, process execution fragments describe a small part of an entire process execution. The second extension allows for the freezing of model components, which allows users to constrain the incremental discovery approach by preventing it from altering frozen model parts during incremental process discovery.

Given users' pivotal role in gradually selecting process behaviors for inclusion in the process model, we introduce novel visualizations for process execution variants. Central to process mining, these variants group individual process executions that have identical arrangements of the activities executed. Considering that activities within a process can run concurrently and overlap, yielding partially ordered event data, we propose visualizations to illustrate such activity relationships. Additionally, this thesis contributes to the field of process querying. We propose a query language for process execution variants that allow the specification of complex control flow patterns among activities. When executing a query, process execution variants satisfying the specified constraints are returned. In short, the proposed query language supports the handling of large event data volumes, enhances the filtering and selection of process execution variants, and, thus,

facilitates users during incremental process discovery.

Next to process discovery and event data handling, this thesis contributes to conformance checking, a further fundamental process mining task. Conformance checking techniques are used to compare observed with modeled process behavior and are crucial to incremental process discovery, providing information and diagnostics on how well the so-far learned process model aligns with the provided event data. We extend the concept of alignments, i.e., a state-of-the-art conformance checking technique, to accommodate process execution fragments. We define infix and postfix alignments and show their computation. Infix and postfix alignments are critical as they enable incremental process discovery with trace fragments.

Moreover, we present Cortado, an open-source process mining software tool that implements the algorithms and techniques proposed in this thesis in an integrated and comprehensive fashion. Through Cortado, we showcase how the methods and algorithms presented in this thesis serve the overall goal of incremental process discovery. Finally, this thesis presents a case study applying Cortado and, therefore, the various contributions of this thesis in a real-life scenario.



---

# Kurzfassung

---

Ein Großteil der Prozesse und betrieblichen Abläufe in Organisationen, beispielsweise Verwaltungs-, Finanz-, Produktions- und Logistikprozesse, wird durch Informationssysteme unterstützt. Diese Systeme zeichnen die Ausführung betrieblicher Prozesse detailliert auf und erzeugen Ereignisdaten, die wertvolle Informationen über die Prozessausführung enthalten. Process Mining analysiert diese Ereignisdaten, um Erkenntnisse in den Prozess zu gewinnen, beispielsweise Prozessmodelle, Konformitätsstatistiken und zeitliche Performancestatistiken. Prozessanalytiker und -verantwortliche nutzen die gewonnenen Erkenntnisse wiederum, um die Ausführung von Prozessen zu verstehen und diese letztlich zu optimieren, zum Beispiel durch Verringerung der Zykluszeiten, Verbesserung der Ressourcenzuweisung und Erhöhung der Konformität. Allgemein zielt Process Mining darauf ab, Prozesse auf datengetriebene Weise zu verbessern.

Process Discovery befasst sich mit dem Lernen von Prozessmodellen aus Ereignisdaten und ist eine grundlegende Aufgabe innerhalb von Process Mining. Die meisten bestehenden Process Discovery Algorithmen sind jedoch vollständig automatisiert, d. h. sie arbeiten aus der Sicht der Benutzer als Blackboxen, ermitteln Prozessmodelle in einer einstufigen Weise ohne Benutzerinteraktion und ermitteln häufig unzureichende Modelle, insbesondere bei Anwendung auf reale Daten. Darüber hinaus nutzen diese Prozesserkennungsalgorithmen kein über die Ereignisdaten hinausgehendes Domänenwissen.

In dieser Dissertation wird ein Framework für inkrementelles Process Discovery vorgestellt, welches es Anwendern ermöglicht, schrittweise ein Prozessmodell aus Ereignisdaten zu lernen. Dabei können die Benutzer die bisher gelernten intermediären Prozessmodelle einsehen. Darüber hinaus können die Nutzer die intermediären Modelle bei Bedarf manuell bearbeiten, bevor diese erneut in das inkrementelle Process Discovery Framework zum weiteren Lernen eingespeist werden. Außerdem können Anwender schrittweise das Prozessverhalten, d. h. die aufgezeichneten Prozessausführungen aus den Ereignisdaten auswählen, welches dem Prozessmodell hinzugefügt wird. Kurz gesagt, das vorgeschlagene inkrementelle Process Discovery Framework befähigt Anwender zur Interaktion und Steuerung der Discovery-Phase eines Prozessmodells aus Ereignisdaten. Darüber hinaus schlagen wir zwei Erweiterungen des inkrementellen Process Discovery Frameworks vor. Erstens erlauben wir das schrittweise Hinzufügen von Prozessausführungsfragmenten zusätzlich zu vollständigen Prozessausführungen. Die meisten automatisierten Algorithmen im Process Discovery gehen von vollständigen Prozessausführungen aus, die den Prozess von Anfang bis Ende umfassen. Im Gegensatz dazu beschreiben Prozessausführungsfragmente einen kleinen Teil einer gesamten Prozessausführung. Als zweite Erweiterung führen wir die Möglichkeit des Einfrierens von Modellteilen ein, die es Anwendern ermöglicht, den inkrementellen Process Discovery Ansatz einzuschränken, indem dieser daran gehindert wird, eingefrorene Modellteile während der inkrementellen Prozessentdeckung weiter zu verändern.

Da die schrittweise Auswahl des Prozessverhaltens durch Anwender, welches in das Prozessmodell aufgenommen werden soll, für die inkrementelle Prozessentdeckung von

zentraler Bedeutung ist, schlagen wir ferner neue Visualisierungen für Prozessausführungsvarianten vor. Varianten sind ein zentrales Konzept im Process Mining, die einzelne Prozessausführungen mit identischer Anordnung der ausgeführten Aktivitäten bündeln. Vor dem Hintergrund, dass Aktivitäten innerhalb eines Prozesses zeitlich parallel laufen können, auch als partiell geordnete Ereignisdaten bekannt, schlagen wir Visualisierungen zur Darstellung solcher Aktivitätsbeziehungen vor. Darüber hinaus leistet diese Arbeit einen Beitrag zu dem Forschungsgebiet Process Querying. Wir präsentieren eine Abfragesprache für Prozessausführungsvarianten, die die Spezifikation von komplexen Kontrollflussmustern über Aktivitäten ermöglicht. Bei der Ausführung einer Abfrage werden Prozessausführungsvarianten zurückgegeben, die den spezifizierten Bedingungen entsprechen. Die vorgeschlagene Abfragesprache unterstützt den Umgang mit großen Mengen von Ereignisdaten, erleichtert die Filterung und Auswahl von Prozessausführungsvarianten und trägt somit zur Unterstützung von Nutzern bei der Anwendung von inkrementellem Process Discovery bei.

Neben der Prozessentdeckung und der Verarbeitung von Ereignisdaten leistet diese Dissertation einen Beitrag zum Conformance Checking, einer weiteren grundlegenden Aufgabe innerhalb des Process Mining neben Process Discovery. Conformance Checking Techniken erlauben den Abgleich von aufgezeichneten Prozessverhalten mit modelliertem Prozessverhalten und sind daher entscheidend für inkrementelles Process Discovery, da diese Techniken Informationen darüber liefern, inwiefern das bisher gelernte Prozessmodell die bereitgestellten Ereignisdaten abdeckt. Wir erweitern das Konzept der Alignments, welche eine State of the Art Conformance Checking Methode sind, um Prozessausführungsfragmente. Wir definieren Infix- und Postfix-Alignments und zeigen, wie diese berechnet werden können. Infix- und Postfix-Alignments sind von entscheidender Bedeutung, da sie inkrementelle Process Discovery mit Prozessausführungsfragmenten ermöglichen.

Darüber hinaus stellen wir Cortado vor, ein Open-Source-Softwaretool für Process Mining, das die in dieser Dissertation vorgeschlagenen Algorithmen und Techniken in einer integrierten und umfassenden Weise implementiert. Das Tool Cortado demonstriert, wie die in dieser Arbeit vorgestellten Methoden und Algorithmen dem Gesamtziel der inkrementellen Prozessentdeckung dienen. Schließlich stellen wir eine Fallstudie vor, in der Cortado und damit die verschiedenen Beiträge dieser Dissertation in einem realen Szenario angewendet werden.

---

# Contents

---

List of Acronyms	xiii
------------------	------

List of Mathematical Notations	xv
--------------------------------	----

I. Opening & Fundamentals	1
---------------------------	---

1. Introduction	3
-----------------	---

1.1. Process Mining	3
1.2. Process Models	6
1.3. Process Discovery	8
1.3.1. Conventional Process Discovery	9
1.3.2. Non-Conventional Process Discovery	10
1.4. Research Goals & Contributions	13
1.5. Thesis Outline	14

2. Literature Review	19
----------------------	----

2.1. Distinguishing Features	22
2.1.1. Defining Distinguishing Features	22
2.1.2. Overview of the Distinguishing Features	22
2.1.3. Dependencies Among Characteristics and Features	28
2.2. Methodology & Design	29
2.3. Identified Approaches	31
2.3.1. Overview	31
2.3.2. Analysis & Discussion	40
2.4. Challenges & Opportunities	43
2.4.1. Challenge 1—Blending Explicit Domain Knowledge & User Feedback	43
2.4.2. Challenge 2—Advanced User Interaction	43
2.4.3. Challenge 3—Various Modes of Interactivity	43
2.4.4. Challenge 4—Scalable Conformance Checking	44
2.4.5. Challenge 5—Minimizing Representational Bias	44
2.4.6. Challenge 6—Event Data & Process Model Visualizations	44
2.4.7. Challenge 7—Domain Knowledge Specification	44
2.4.8. Challenge 8—Event Data & Domain Knowledge Fusion	45
2.4.9. Challenge 9—Software Support	45
2.4.10. Challenge 10—Discovery Beyond Control-Flow	46
2.5. Conclusion	46

<b>3. Preliminaries</b>	<b>47</b>
3.1. Basic Mathematical Concepts . . . . .	47
3.1.1. Sets & Relations . . . . .	47
3.1.2. Functions . . . . .	48
3.1.3. Multisets . . . . .	48
3.1.4. Ordered Sets . . . . .	49
3.1.5. Sequences . . . . .	52
3.1.6. Graphs & Trees . . . . .	53
3.2. Event Data & Event Logs . . . . .	59
3.3. Process Models . . . . .	62
3.3.1. Petri Nets . . . . .	62
3.3.2. Process Trees . . . . .	66
3.4. Conformance Checking Overview . . . . .	70
3.5. Alignments . . . . .	71
3.5.1. Alignments for Petri nets . . . . .	72
3.5.2. Alignments for Process Trees . . . . .	73
3.5.3. Computing Alignments . . . . .	76
<b>4. Alignments for Trace Fragments</b>	<b>81</b>
4.1. Overview . . . . .	82
4.2. Defining Prefix, Infix & Postfix Alignments . . . . .	83
4.3. Computing Infix & Postfix Alignments . . . . .	86
4.3.1. Baseline Approach . . . . .	88
4.3.2. Extended Baseline Approach Using Subsequent Filtering . . . . .	90
4.3.3. Process-Tree-Based Approach . . . . .	92
4.4. Evaluation . . . . .	101
4.4.1. Experimental Setup . . . . .	102
4.4.2. Results . . . . .	102
4.4.3. Discussion & Threats to Validity . . . . .	106
4.5. Conclusion . . . . .	106
<b>II. Incremental Process Discovery</b>	<b>107</b>
<b>5. Incremental Process Discovery Framework</b>	<b>109</b>
5.1. Introduction to the Framework . . . . .	109
5.1.1. Input-Output Perspective . . . . .	110
5.1.2. Motivation & Opportunities . . . . .	111
5.2. Naive IPDA . . . . .	112
5.3. Lowest Common Ancestor IPDA . . . . .	115
5.3.1. Running Example . . . . .	115
5.3.2. Algorithm . . . . .	118
5.3.3. Summary & Termination . . . . .	127
5.3.4. LCA Lowering . . . . .	127
5.4. Evaluation . . . . .	130
5.4.1. Experimental Setup . . . . .	130

5.4.2. Results . . . . .	131
5.4.3. Discussion & Threats to Validity . . . . .	138
5.5. Illustrative Example . . . . .	138
5.6. Trace Ordering Effects . . . . .	140
5.6.1. Framework for Recommending Trace Orderings . . . . .	141
5.6.2. Sample Instantiations of Strategy Components . . . . .	144
5.6.3. Evaluation . . . . .	148
5.7. Conclusion . . . . .	154
<b>6. Supporting Trace Fragments in Incremental Process Discovery</b>	<b>155</b>
6.1. Extended IPD Framework . . . . .	158
6.2. Trace-Fragment-Supporting IPDA . . . . .	161
6.2.1. Running Example . . . . .	161
6.2.2. Algorithm . . . . .	164
6.3. Evaluation . . . . .	172
6.3.1. Experimental Setup . . . . .	172
6.3.2. Results . . . . .	173
6.3.3. Discussion & Threats to Validity . . . . .	173
6.4. Conclusion . . . . .	175
<b>7. Freezing Process Model Parts in Incremental Process Discovery</b>	<b>177</b>
7.1. Extended IPD Framework . . . . .	180
7.2. Naive Freezing-Enabled IPDA . . . . .	181
7.3. Freezing-Enabled LCA-IPDA . . . . .	184
7.3.1. Overview . . . . .	184
7.3.2. Component (1)—Replacing Frozen Subtrees . . . . .	186
7.3.3. Component (2)—Projecting Trace to be Added Next . . . . .	187
7.3.4. Component (3)—Projecting Previously Added Traces . . . . .	191
7.3.5. Component (4)—Reinserting Frozen Subtrees . . . . .	193
7.4. Evaluation . . . . .	203
7.4.1. Experimental Setup . . . . .	203
7.4.2. Results . . . . .	204
7.4.3. Discussion & Threats to Validity . . . . .	209
7.5. Illustrative Example . . . . .	209
7.6. Conclusion . . . . .	211
<b>III. Facilitating Interaction with Event Data</b>	<b>213</b>
<b>8. Defining &amp; Visualizing Variants</b>	<b>215</b>
8.1. Overview . . . . .	218
8.2. High-Level Variants . . . . .	221
8.2.1. High-Level Case View . . . . .	221
8.2.2. Calculation & Visualization of High-Level Variants . . . . .	222
8.2.3. Limitations of the High-level Variant Visualization . . . . .	227

8.3. Low-Level Variants . . . . .	231
8.3.1. Low-Level Case View . . . . .	231
8.3.2. Calculation & Visualization of Low-Level Variants . . . . .	233
8.4. Computing High- & Low-Level Variants . . . . .	235
8.5. Time Granularity Modifier . . . . .	236
8.6. Evaluation . . . . .	241
8.6.1. Automated Experiments . . . . .	241
8.6.2. User Study . . . . .	245
8.7. Conclusion . . . . .	253
<b>9. Query Language for Variants</b>	<b>255</b>
9.1. Related Work . . . . .	256
9.2. Query Language . . . . .	257
9.2.1. Syntax . . . . .	257
9.2.2. Semantics . . . . .	260
9.2.3. Query Evaluation . . . . .	264
9.3. Illustrative Example . . . . .	264
9.4. Evaluation . . . . .	266
9.4.1. Experimental Setup . . . . .	266
9.4.2. Results . . . . .	266
9.4.3. Discussion & Threats to Validity . . . . .	268
9.5. Conclusion . . . . .	270
<b>IV. Realization &amp; Application</b>	<b>271</b>
<b>10. Tool Support: Cortado</b>	<b>273</b>
10.1. Overview . . . . .	275
10.2. Variant Handling . . . . .	278
10.2.1. Variant Explorer . . . . .	278
10.2.2. Variant Querying . . . . .	282
10.2.3. Variant Modeler . . . . .	283
10.2.4. Variant Frequent Pattern Mining . . . . .	283
10.2.5. Variant Sequentialization . . . . .	284
10.3. Incremental Process Discovery . . . . .	287
10.3.1. Visualizing & Editing Process Models . . . . .	287
10.3.2. Adding Behavior to a Process Model . . . . .	287
10.4. Temporal Performance Analysis . . . . .	291
10.4.1. Overview . . . . .	291
10.4.2. Model-Independent Performance Analysis . . . . .	292
10.4.3. Model-Based Performance Analysis . . . . .	293
10.5. Supported Data Exchange Formats . . . . .	301
10.6. Software Architecture & Distribution . . . . .	303
10.7. Conclusion . . . . .	305

<b>11. Case Study</b>	<b>307</b>
11.1. Related Work . . . . .	308
11.2. Overview . . . . .	308
11.3. Analysis Objectives & Approach . . . . .	310
11.4. Analysis Results . . . . .	311
11.4.1. Event Data Extraction & Initial Preparation . . . . .	311
11.4.2. Interactive Process Discovery . . . . .	314
11.5. Discussion . . . . .	323
11.5.1. Lessons Learned . . . . .	323
11.5.2. Practical Implications . . . . .	324
11.5.3. Limitations & Future Work . . . . .	324
11.6. Conclusion . . . . .	325
 <b>V. Closure</b>	 <b>327</b>
<b>12. Conclusion</b>	<b>329</b>
12.1. Contributions . . . . .	330
12.1.1. Review of Domain-Knowledge-Utilizing Process Discovery . . . . .	330
12.1.2. Incremental Process Discovery . . . . .	330
12.1.3. Variants for Partially Ordered Event Data . . . . .	331
12.1.4. Cortado . . . . .	331
12.2. Limitations & Remaining Challenges . . . . .	331
12.2.1. Nondeterminism of the LCA-IPDA . . . . .	331
12.2.2. Representational Bias . . . . .	332
12.2.3. Support for Partially Ordered Event Data . . . . .	332
12.2.4. Lack of Thorough User Evaluation . . . . .	333
12.2.5. Incorporating Low-Level Variants . . . . .	333
12.3. Future Research Directions . . . . .	333
12.3.1. Beyond Adding Individual Traces in IPD . . . . .	333
12.3.2. Incremental Process Reduction . . . . .	334
12.3.3. Enhanced Interaction & Assistance . . . . .	334
12.3.4. Incremental Discovery Beyond Control Flow . . . . .	335
12.3.5. Supporting Object-Centric Event Data . . . . .	335
 <b>References</b>	 <b>337</b>
 <b>List of Publications</b>	 <b>362</b>
 <b>Acknowledgment</b>	 <b>366</b>
 <b>Curriculum Vitae</b>	 <b>367</b>





---

# List of Acronyms

---

BPM	Business Process Management.
BPMN	Business Process Model and Notation.
CRM	Customer Relationship Management.
CT	Cycle Time.
DFG	Directly Follows Graph.
EPC	Event-Driven Process Chain.
ERP	Enterprise Resource Planning.
FOL	First-Order Logic.
ID	Identifier.
IM	Inductive Miner.
IPD	Interactive Process Discovery.
IPDA	Incremental Process Discovery Algorithm.
IT	Idle Time.
LCA	Lowest Common Ancestor.
NTOS	Next Trace Ordering Strategy.
PI	Performance Indicator.
RG	Research Goal.
SCM	Supply Chain Management.
SPN	Synchronous Product Net.
ST	Service Time.
STA	Semantic Tree Analysis.
TFS-IPDA	Trace-Fragment-Supporting Incremental Process Discovery Algorithm.
TGM	Time Granularity Modifier.
UI	User Interface.
UML	Unified Modeling Language.
WF-net	Workflow net.

WT	Waiting Time.
YAWL	Yet Another Workflow Language.

---

# List of Mathematical Notations

---

## Sets & Universes

$\emptyset$	Empty set
$\mathbb{N}$	Natural numbers
$\mathbb{N}_0$	Natural numbers <i>including</i> zero
$\mathbb{R}$	Real numbers
$\mathbb{R}_{\geq 0}$	Real numbers greater or equal zero
$\mathbb{B}$	Boolean values
$\mathbb{P}(X)$	Power set of a set $X$
$\mathcal{M}(X)$	Universe of multisets for a set $X$
$[]$	Empty multiset
$\mathcal{O}_{\leq}$	Universe of total orders
$\mathcal{O}_{<}$	Universe of strict total orders
$\mathcal{O}_{\preccurlyeq}$	Universe of partial orders
$\mathcal{O}_{\prec}$	Universe of strict partial orders
$\Gamma$	Universe of (full) alignments
$\Gamma^{opt}$	Universe of optimal (full) alignments
$\Gamma_{inf}$	Universe of infix alignments
$\Gamma_{pre}$	Universe of prefix alignments
$\Gamma_{inf}^{opt}$	Universe of optimal infix alignments
$\Gamma_{pre}^{opt}$	Universe of optimal prefix alignments
$\Gamma_{pos}$	Universe of postfix alignments
$\Gamma_{pos}^{opt}$	Universe of optimal postfix alignments
$\mathcal{C}$	Universe of cases
$\mathcal{E}$	Universe of events

$\mathcal{A}$	Universe of activity labels
$\mathcal{N}$	Universe of Petri nets
$\mathcal{N}_{accept}$	Universe of accepting Petri nets
$\mathcal{W}$	Universe of Workflow nets (WF-nets)
$\mathcal{T}$	Universe of labeled, rooted, ordered trees
$\mathcal{P}$	Universe of process trees
$\otimes$	Universe of process tree operators

## Frequently used variable names

$V$	Tree/graph vertices
$E$	Tree/graph edges
$\Lambda$	Tree
$v$	Tree vertex
$\gamma$	(Full) alignment
$\gamma_{pre}$	Prefix alignment
$\gamma_{inf}$	Infix alignment
$\gamma_{pos}$	Postfix alignment
$N$	Petri net
$t$	Transition
$T$	Set of transition
$p$	Place
$P$	Set of places
$F$	Arcs in a Petri net
$M$	Marking of a Petri net
$M^{init}$	Initial marking of a Petri net
$M^{final}$	Final marking of a Petri net

$p_{src}$	Source place	$\circ$	Concatenation of (sets of) sequences
$p_{sink}$	Sink place	$\diamond$	Interleaving (sets of) sequences
$L$	Event log	$true$	Boolean true
$L^s$	Simplified event log	$false$	Boolean false
$C$	Case	$\sqsubseteq$	Subtree relation
$e$	Event	$\tilde{\sqsubseteq}$	Isomorphic subtree relation
$\sigma$	Sequence	$\rightarrow$	Sequence process tree operator
<b>Other symbols</b>		$\circlearrowleft$	Loop process tree operator
$\succcurlyeq$	Partial order	$\times$	Exclusive choice process tree operator
$\prec$	Strict partial order	$\wedge$	Parallel process tree operator
$\leq$	Total order	$\mathcal{RS}$	Process tree running sequences
$<$	Strict total order	$\mathcal{S}$	Process tree running steps
$\triangleleft$	Arbitrary order	$\mathcal{RM}$	Petri net's reachable markings
$\cong$	Isomorphism of labeled ordered sets	$\otimes$	Concurrency-aware variants fall through operator
$\uplus$	Multiset union	$\lambda$	Labeling function
$\langle \rangle$	Empty sequence		

## Part I.

# Opening & Fundamentals



# Chapter 1.

## Introduction

Most organizations, ranging from businesses to government agencies, organize their operations through processes that specify, control, and coordinate the various activities and resources involved in executing a process. Many operational processes exist, ranging from production to administrative processes. The execution of operational processes (hereafter referred to as processes) is often supported and controlled by *information systems*, including Enterprise Resource Planning (ERP), Supply Chain Management (SCM), and Customer Relationship Management (CRM) systems. These systems track the execution of such processes, often in great detail. The data generated during process execution are referred to as *event data*. Event data contain valuable information about the actual execution of processes. Hence, the field of *process mining* [208, 211] has emerged, providing algorithms, techniques, and tools that allow for the analysis of *event data* to generate insights into the process. These insights include, for example, discovered process models from event data [216], conformance checking diagnostics [45], temporal performance statistics [226], process simulation results [139], and prediction models [65]. Process analysts, designers, owners, and other stakeholders use these insights to derive process modifications and refinements to optimize the process. Processes can be optimized for various aspects, such as reducing cycle times, improving resource allocations, and increasing conformity with reference models.

### 1.1. Process Mining

The goal of process mining is the data-driven optimization of processes through the analysis of event data. Overall, process mining provides unique opportunities for organizations to streamline their processes [141, 209], thus contributing to the creation of business value [17]. Several industries have successfully applied process mining, for example, in production [166, 150, 82], in auditing [106, 222], in healthcare [134, 135, 142], in finance industry [61], in software development [167], and in education [33].

Figure 1.1 illustrates various process mining artifacts and techniques, as well as their interactions.<sup>1</sup> As shown, processes that are supported and controlled by information systems are central to process mining. Executing these processes allows for information systems to track event data that record the various process executions. For example, event data contain information about which activities were performed during which process executions and which resources were involved, as well as temporal information about activity executions. *Event data extraction* [51, 66] from information systems is a chal-

<sup>1</sup>Note that Figure 1.1 does not present a complete picture of process mining; instead, it illustrates essential artifacts, techniques, and exemplary interconnections among them.

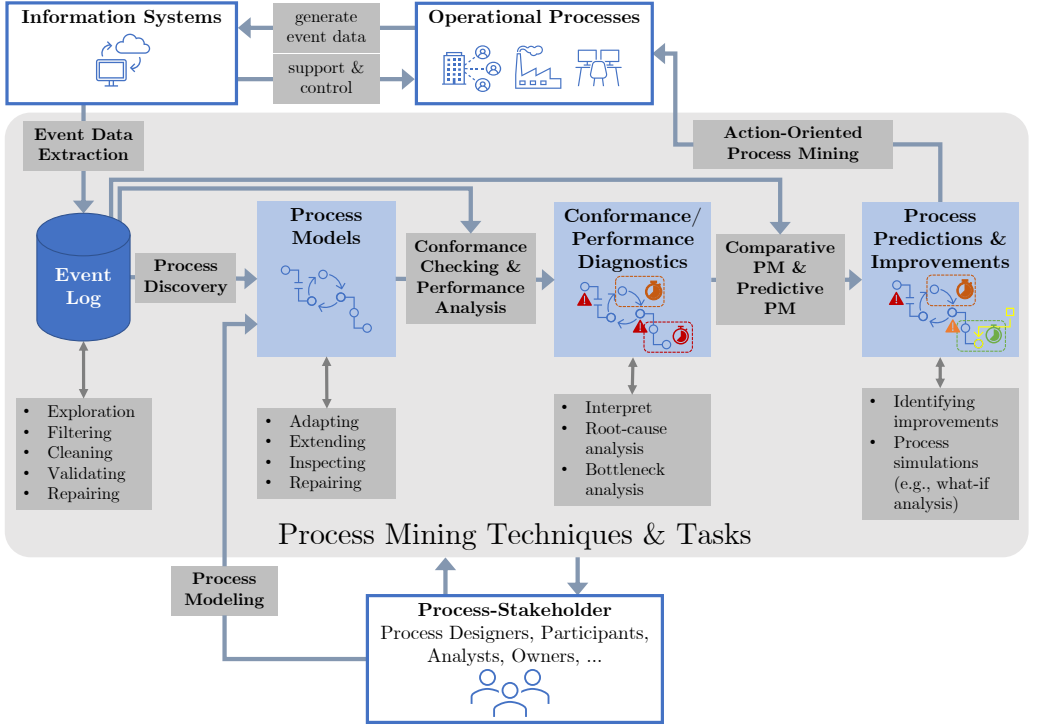


Figure 1.1: Broad overview of key process mining techniques, tasks, and artifacts

lenging task that generally involves substantial manual effort [195]. We refer to event data describing a particular process as an *event log*, cf. Figure 1.1.

As in any other data-driven domain, the preparation and preprocessing of even data are paramount for subsequent analysis. Various techniques exist, for example, exploration, filtering, and cleaning techniques. Most process mining techniques require an event log as input to gain insights into the actual process. Traditionally, three main disciplines within process mining are distinguished: *process discovery*, *conformance checking*, and *process enhancement* [211]. Since process enhancement [55] is often used as an umbrella term for various process mining types, we use a more nuanced view in this thesis. According to [215], six different types of process mining can be distinguished, as shown in Figure 1.1.

1. **Process discovery** [216] deals with learning process models from event data and potentially other information, for instance, domain knowledge as considered in this thesis. Process discovery is a central discipline within process mining, since process models are an important artifact and serve as an input for many subsequent analysis approaches, cf. Figure 1.1.
2. **Conformance checking** techniques [45] compares observed with modeled process behavior. While observed behavior refers to recorded process executions as reflected



by event data, modeled process behavior refers to process models. These techniques aim to align observed and modeled process behavior to detect mismatches between them. Therefore, these techniques are used to find deviations in process executions compared with reference process models and to assess the quality of process models concerning event data.

3. **Performance analysis** techniques derive temporal performance statistics [226]. Since a significant optimization scenario for processes often involves temporal aspects, performance analysis is essential in the practical application of process mining.
4. **Comparative process mining** compares multiple event logs. For instance, process cubes [34, 210] provide an approach to slice an event log into sub-logs that are compared with each other. That is, event data might be divided according to geographical locations and then compared, for example, to analyze location-based differences in the execution of processes.
5. **Predictive process mining** deals with providing various process-related forecasts [138]. In contrast, techniques mentioned above, such as process discovery, conformance checking, and performance analysis, are usually backward-oriented. Thus, these techniques take event data that record historical process executions and provide insights. In many cases, however, it is of great practical importance to react proactively while the processes are still being executed. Predictive process mining is a large field that includes various aspects and can be considered forward-oriented compared to the abovementioned disciplines. For instance, predictive process mining comprises approaches for predicting the remaining cycle time of running process executions [49] and the next activity in an ongoing process execution [200].
6. **Action-oriented process mining** combines backward-oriented with forward-oriented process mining. Actions influencing the actual process are generated based on insights and diagnostics into processes. A general framework for action-oriented process mining in which stakeholders define constraints representing patterns of interest in a process is presented in [148]. During the execution of the process, a constraint monitor supervises the occurrence of the defined constraints; for instance, concept drift detection techniques [44] can be used to detect specific patterns. Once a constraint is detected, corresponding actions defined by stakeholders are executed. For example, if a particular activity is executed significantly longer, more resources are attached to this activity.

Note that these six types represent only the main process mining types. Various other process mining tasks, for example, event data extraction [51] and preprocessing [35], are left out. Most techniques from these six process mining types rely on process models to some extent, making process models a crucial artifact of process mining in general. In this context, process discovery is of critical importance.

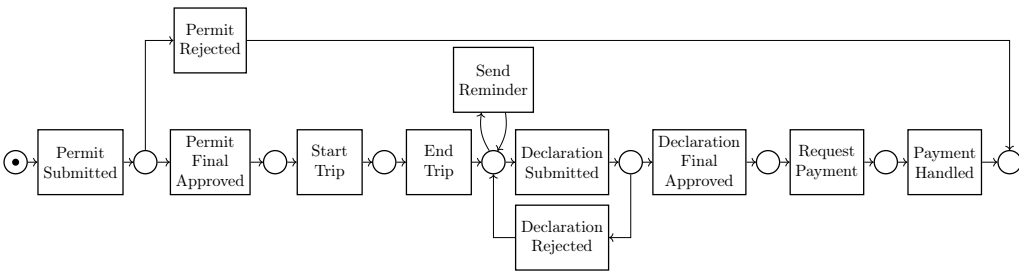
## 1.2. Process Models

Process models are essential artifacts within process mining and are used as input for many analysis techniques, cf. Figure 1.1. There are several process modeling formalisms, such as Petri nets [168], process trees [122], Business Process Model and Notation (BPMN) [48], Unified Modeling Language (UML) [81], Yet Another Workflow Language (YAWL) [220], and Event-Driven Process Chains (EPCs) [171, 205]. However, this thesis mainly focuses on *Petri nets* and *process trees*, which are an essential subclass of Petri nets often used in process mining.

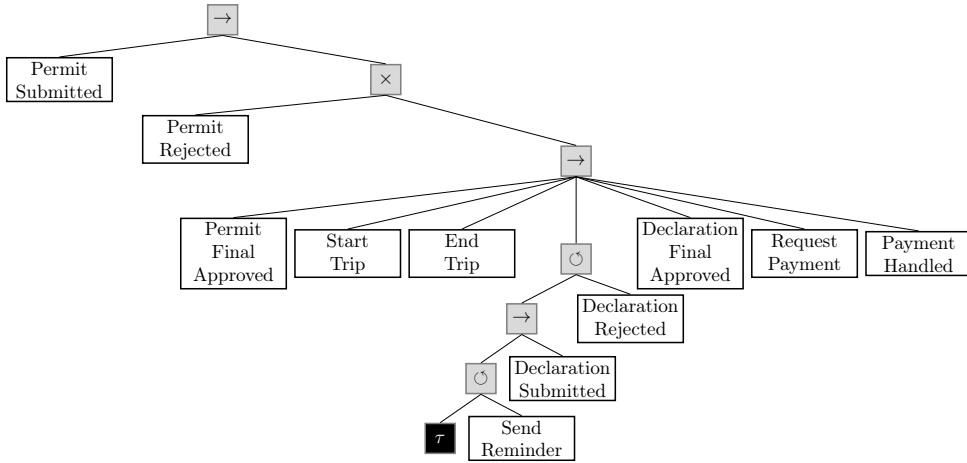
For example, Figure 1.2 depicts three process models in different formalisms, all modeling the same travel permission process. Initially, a travel permission is submitted, cf. activity ‘Permit submitted’. If it is not rejected, the trip takes place, indicated by the activities: ‘Permit Final Approval’, ‘Start Trip’, and ‘End Trip’. Next, a travel declaration is submitted. If the declaration is submitted too late or if a rejected declaration is not re-submitted in time, ‘Send Reminder’ is executed. Once approved, cf. activity ‘Declaration Final Approval’, payment-related activities ‘Request Payment’ and ‘Payment Handled’ are performed, and the process is complete. Figure 1.2a models the process as a Petri net; more precisely a workflow net [204, 168], which is a subclass of Petri nets often used to model business processes. Figure 1.2b models the process as a process tree, also referred to as block-structured process models [122]. In comparison to Petri nets, process trees have an inherent hierarchical structure due to their tree structure. Finally, Figure 1.2c models the process in BPMN.

All three models represent the same *control-flow* of activities that constitute the process. The control-flow refers to how and when activities within a process can be executed: (1) the activities are executed sequentially; (2) their execution is optional; (3) the activities are executed in parallel; (4) the activities are executed multiple times; (5) the activities are executed once specific dependencies are fulfilled. In this thesis, the control flow perspective is primarily considered with regard to process models. Note, however, that formalisms such as BPMN allow the modeling of additional perspectives such as *organizational and data* aspects [77]. For example, the BPMN model depicted in Figure 1.2c contains information about organizational aspects. This model contains one pool called ‘Example Organization’, indicating that the modeled process represents an internal process in which no external parties are involved. This pool is divided into three parts by lanes, each representing different stakeholders and systems involved in the process: ‘Travel Applicant’, ‘Supervisor’, and ‘Travelling Permit System’.

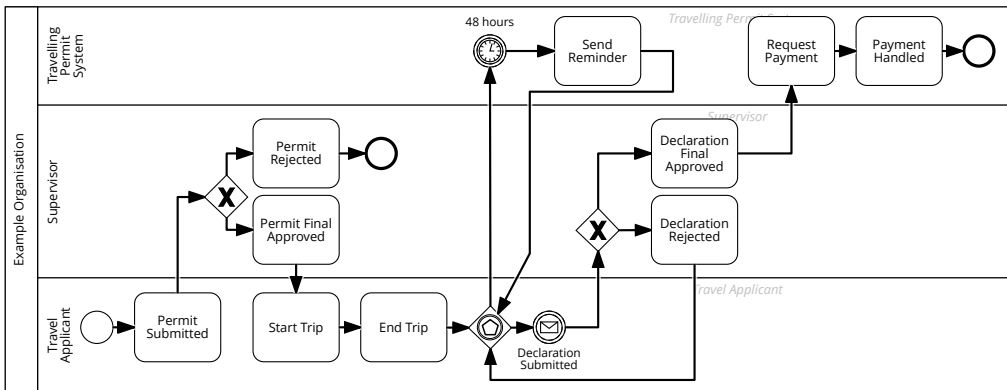
In addition to playing an essential role in process mining, process models serve as a focal point for stakeholder discussions. Furthermore, process models are used to specify process-aware information systems [76]. For instance, Business Process Management Systems (BPMSs) support the entire BPM lifecycle, ranging from design, modeling, and execution to monitoring [78]. These systems require formal process models and through an execution engine, these models are implemented. In short, process models are central artifacts used for many different use cases.



(a) A Petri net (a WF-net to be precise)



(b) A process tree



(c) A BPMN model

Figure 1.2: Process models specified in the same travel permission process using different formalisms (partly adapted from [184, Figure 4])

### 1.3. Process Discovery

This section introduces the central topic of this thesis—the field of process discovery. Process discovery is an important discipline within process mining and deals with learning process models from event data. Note that the term process discovery is also used within the field of Business Process Management (BPM) [78]. However, within BPM, the term is defined more broadly as collecting information about a process under consideration and transforming this information into an as-is model [78]. Process discovery techniques in the BPM domain consist of three primary methods: interview-based discovery, analysis of documents specifying processes, and manual observation of individual process executions. These methods are used to obtain information about the process executions to eventually create a model of the overall process. In this thesis, however, we focus on process discovery as considered in process mining, i.e., using data-driven approaches to (automatically) learn process models from event data. Furthermore, we focus on discovering control-flow structures of processes, i.e., the control-flow perspective.

Process discovery comprises data-driven approaches to (automatically) learn process models from event data, as opposed to manually modeling processes. Therefore, the problem of process discovery can be described as learning a process model that represents the process behavior from a given event log. However, different quality criteria and common challenges in handling event data pose challenges to process discovery because they have frequently quality problems [141, 254]. These data quality issues directly affect the quality of the discovered process models. Overall, four major process model quality dimensions exist that influence each other [40, 211].

- **Fitness:** The model should incorporate the process behavior as recorded in the event log.
- **Precision:** The model should not underfit the event log, i.e., the model should not allow behavior that is not recorded in the event log.
- **Generalization:** The model should not overfit the event log, i.e., it should generalize the process behavior recorded in the event log.
- **Simplicity:** The model should be easy for process stakeholders to understand. The simplest model should be used, since there are different ways to model the same behavior. Research indicates that the model size is a primary driver of perceived complexity [144].

The four quality dimensions listed above are interconnected. As a result, optimizing only one dimension generally has a negative impact on the other dimensions. For example, there is often a trade-off between precision and generalization, i.e., between underfitting and overfitting. These circumstances, i.e., interdependent quality dimensions, coupled with event data quality issues, make process discovery challenging and explain why different approaches have been developed. This section further introduces conventional process discovery (Section 1.3.1) and unconventional process discovery approaches (Section 1.3.2) on which this thesis focuses.

### 1.3.1. Conventional Process Discovery

This section presents process discovery as it is mainly considered in process mining. We use the term *conventional process discovery* to refer to approaches that solely use event data as input and discover process models in a fully automated fashion.<sup>2</sup> A large body of work exists regarding conventional process discovery. We refer to [15, 60, 235] for detailed review articles. Figure 1.3 illustrates the general operating principle of conventional process discovery approaches from a user's perspective. An event log capturing various executions of a process is provided, and the process discovery approach learns a process model without any further interaction required from users, i.e., fully automatically. Apart from the ability to configure the algorithm's configuration parameters, users cannot interact with or influence the discovery algorithm. Their influence is limited to preprocessing the input (i.e., the event log) or postprocessing the output (i.e., the discovered process model). Therefore, conventional process discovery appears as a black box to users and offers limited interaction options to guide and influence the discovery phase<sup>3</sup>.

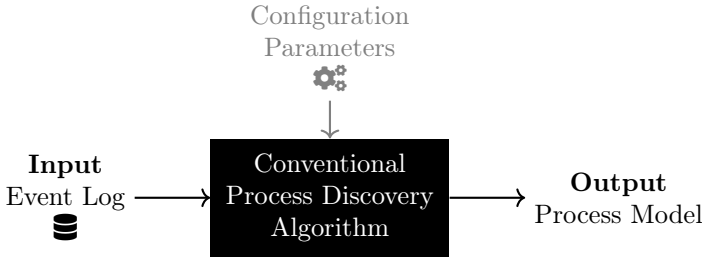


Figure 1.3: Schematic visualization of conventional process discovery algorithms

Many conventional process discovery algorithms have been proposed that focus on discovering the control flow of processes from event data. However, no clear dominating discovery approach exists that works best for all event logs. Thus, users must find a suitable process discovery algorithm for an event log to obtain satisfactory results. Furthermore, users often need to know the setting parameters and, therefore, the details of an algorithm to obtain valuable results. The plethora of conventional discovery algorithms even led to approaches recommending the best suiting discovery algorithm for a given event log [47, 163].

Various distinction criteria can be used to organize existing conventional process discovery approaches. For instance, the process model formalism used for the discovered process models, supported control-flow structures, runtime, implementation, and formal

<sup>2</sup>Note that conventional process discovery, as considered in this thesis, is also often referred to *automated process discovery* [15, 78, 119]. We use a differentiated perspective on the term *automated* in this thesis. While all conventional process discovery algorithms, which learn a process model solely from an event log, are automated because no interaction during the actual discovery is required, and the algorithm works fully automated, also non-conventional process discovery algorithms (cf. Section 1.3.2) can be *automated*.

<sup>3</sup>We use the term *discovery phase* to describe the phase in which the discovery algorithm learns a process model from the existing inputs. In conventional process discovery (cf. Section 1.3.1), this phase is fully automated and functions like a black box from the user's perspective.

guarantees on the resulting process model such as soundness [224] or replay-fitness guarantees, i.e., all process behavior from the event log is reflected by the process model. In short, many conventional process discovery algorithms exist that differ in various aspects and provide different results for the same input event log. However, from a user's perspective, these algorithms do not provide any interaction, as they solely depend on event data to discover a process model, making them function like a black box as illustrated in Figure 1.3.

### 1.3.2. Non-Conventional Process Discovery

This section introduces non-conventional process discovery [184]. The critical difference to conventional process discovery is the utilization of additional information, hereinafter referred to as *domain knowledge*, besides event data. Domain knowledge can be any information about the process to be discovered; for instance, precedence constraints about the activities in the process [99]. Further, we also consider *user interactions* with a discovery algorithm to steer and guide the discovery phase as domain knowledge for simplicity, for instance, cf. [71]. In short, domain knowledge is any form of information utilized to discover a process model other than event data or configuration parameters. Figure 1.4 illustrates non-conventional process discovery from a users perspective similar to Figure 1.3.

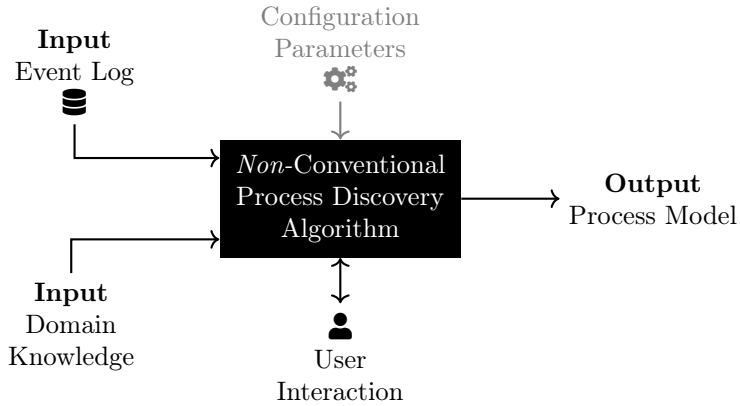


Figure 1.4: Schematic visualization of non-conventional process discovery algorithms; next to an event log domain knowledge or user interaction is used

Next to the usage of domain knowledge besides event data, the provision timing of domain knowledge may differ. We call a non-conventional process discovery approach *automated* if domain knowledge is provided as an input at the beginning next to the event log; no further possibility to provide domain knowledge exists during the discovery phase to ingest further domain knowledge, for example, user interactions. We refer to a non-conventional process discovery approach allowing the ingestion of domain knowledge, for example, user interaction, during the process discovery phase as *interactive*.

Non-conventional process discovery emerged to overcome the limitations of conventional process discovery. Event data are often affected by data quality issues posing challenges for process discovery in general; studies with process mining practitioners confirm this challenge [141, 254]. Event data quality issues comprise various aspects, below we list some common issues.

- **Incorrect event data** refers to scenarios in which event data is wrongly captured. For instance, incorrect timestamps may occur that do not reflect the timing information of the actual execution of an activity within a process execution. Since timestamps are essential for process discovery to learn the control-flow of a process, said issues may severely affect the quality of discovered process models.
- **Imprecise event data** refers to scenarios in which event data information is too inaccurate for the intended process analysis. For instance, activity names might be recorded too coarsely. As a result, multiple executions of identical labeled activities might exist within individual process executions.
- **Irrelevant event data** refers to logged activities in event logs that are irrelevant to the intended process analysis. For example, irrelevant low-level events may have been recorded that either need to be filtered or aggregated to high-level events.

Various event data per-processing techniques exist that support process analysts in mitigating these event data quality issues [35, 59, 211]. Further, automated filtering techniques have shown to improve process discovery results, for instance, consider [169, 253]. Existing conventional process discovery approaches, however, still often generate low-quality process models based on real event data. In addition, the fully automated approach of conventional process discovery may be tedious, as process models must be learned repeatedly from scratch when the input event data is filtered or adjusted due to non-satisfactory results.

Besides event data quality issues, insufficient availability of event data impacts the process model quality. Missing event data refers to scenarios in which event data are incomplete for the intended process analysis. For instance, individual events might be missing in the event log; thus, the event data represents process executions only partly. Further, event data might be, in general, incompletely capturing the actual process. Although individual process executions are recorded correctly, not all possible process executions of the actual process are recorded in the event data because they did not happen in the period the event data was extracted. Incomplete event data can result in an incomplete understanding of the process; the process discovery approach cannot fully discover the actual process. In such a scenario, the use of domain knowledge alongside event data is advantageous.

As conventional process discovery focuses exclusively on event data, it is only possible to consider process behavior if contained in the event data. However, in specific scenarios, process analysts may want to include normative process behavior in a process model alongside the behavior recorded in the event data, representing the actual execution of the process. Note that event data is assumed to record the actual execution of a process and not the intended execution. Although domain knowledge about the process being analyzed exists, it is largely ignored and not utilized within process discovery. In [19], the

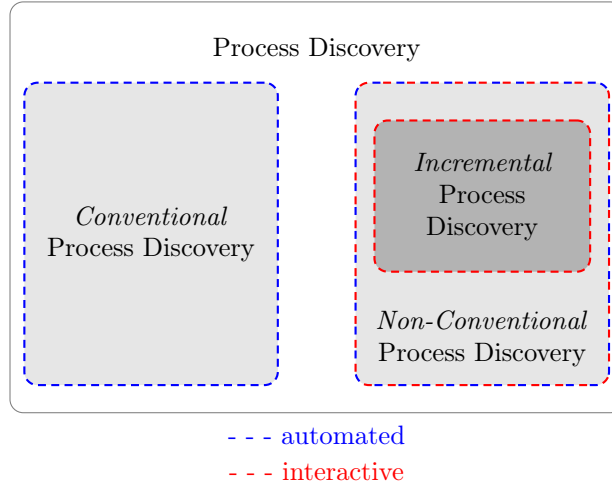


Figure 1.5: Relation between conventional, non-conventional, incremental, automated, and interactive process discovery: Conventional discovery techniques are always automated, incremental discovery techniques are considered interactive, and non-conventional discovery approaches besides incremental techniques can either be automated or interactive

authors identify nine central research problems of the BPM domain. The utilization of domain knowledge within process mining is one of these problems. Further, the authors reinforce prevalent data quality issues of event data and the potential of utilizing domain knowledge to overcome these challenges.

Finally, the lack of user interaction in conventional process discovery can have disadvantages. Recall the four process model quality dimensions that influence each other, cf. Section 1.3. Influencing these quality dimensions through parameter settings of process discovery algorithms is often not directly possible, as many parameter settings cannot be assigned to specific dimensions. Further, if a conventional process discovery algorithm produces an undesired process model, the discovery must start from scratch, i.e., the previously discovered model is discarded. In contrast, non-conventional process discovery allows users to interact and steer the discovery phase. Thus, process models are learned gradually, with users having the opportunity to intervene, correct, or guide the algorithm.

Compared to conventional process discovery, few non-conventional process discovery approaches exist [184]. This thesis contributes to the field of non-conventional process discovery by proposing novel *incremental discovery approaches* allowing the utilization of domain knowledge within process discovery. Furthermore, this thesis breaks with the prevailing approach of fully automated process discovery and enables a gradual approach to process discovery. Figure 1.5 summarizes the various terms in the context of process discovery and positions incremental process discovery within process discovery in general. Conventional process discovery is always automated. In contrast, non-conventional process discovery comprises both automated and interactive approaches. Incremental process



discovery, i.e., part of non-conventional process discovery, is considered interactive.

## 1.4. Research Goals & Contributions

This section presents the central Research Goals (RGs) of this thesis. Further, we outline how this thesis contributes to achieving these RGs.

- RG 1 A structured overview of non-conventional process discovery approaches
- RG 2 Incremental discovery of process models from event data and enabling users of incremental process discovery algorithms to influence or steer the algorithm beyond setting configuration parameters
- RG 3 Facilitate the exploration of event data and the process behavior contained therein to facilitate interactive approaches to process discovery
- RG 4 Development of a comprehensive prototype that integrates and unifies the different methods, algorithms and techniques from the field of incremental process discovery presented in this thesis

We address research goal [RG 1](#) by conducting a systematic literature review of non-conventional process discovery approaches. Further, we aim to develop a taxonomy for said approaches. Research goal [RG 2](#) aims at developing a novel process discovery approach that allows the incorporation of domain knowledge while gradually discovering a process model. We address [RG 2](#) by proposing a novel *incremental process discovery framework*. This framework allows users to discover a process model from event data incrementally. Figure [1.6](#) outlines the central idea. Within one iteration, process behavior from an event log is selected by a user. Note that technically a user is not required; instead, any automated method can also incrementally select the process behavior. The selected process behavior, potential further domain knowledge, and an (initial) process model  $M$  are fed into the incremental process discovery algorithm. The algorithm extends the process model  $M$  such that the intended process behavior is reflected by the discovered model  $M'$  and the intended domain knowledge is respected by  $M'$ . This described iteration is repeated using the discovered model  $M'$  as an input in the next iteration. Several algorithmic contributions to incremental process discovery are provided throughout this thesis, all founded in a common framework.

[RG 3](#) is considered a building block for incremental process discovery, as users play an essential role in incremental process discovery and must be supported accordingly in their decision-making. We propose new approaches for defining and visualizing process execution variants from event data. Such variants are essential in process mining as they group individual process executions with identical control-flow behavior. Thereby, variants facilitate the handling of vast amounts of event data. Variants are particularly significant in process discovery as they decrease the number of individual process executions that need to be analyzed by a process discovery approach to learn a process model. Especially in incremental process discovery, where the gradual selection of process behavior is central, variants and corresponding visualizations are paramount. Further, we propose a

## Incremental Process Discovery Approach

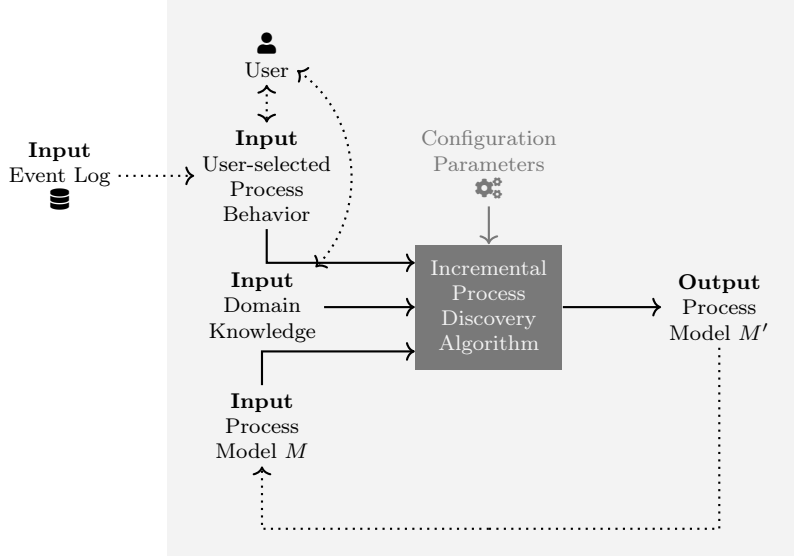


Figure 1.6: Schematic visualization of the incremental process discovery framework proposed in this thesis; one iteration of the procedure consists of selecting process behavior from the event log, which is subsequently integrated into a process model  $M$  resulting in  $M'$  that serves as input in the next iteration

new query language for process execution variants that facilitates users in exploring and selecting process behavior from event logs.

Regarding [RG 4](#), we present an open-source process mining software tool implementing the proposed contributions in an integrated fashion. Thus, the tool features among others an incremental process discovery approach, conformance checking techniques, performance analysis, and novel variant visualizations. As this tool was developed as a part of this thesis, it is considered a substantial contribution. Furthermore, we present a case study in which this tool is applied to analyze a healthcare process.

## 1.5. Thesis Outline

This thesis is divided into five parts. Subsequently, we introduce these five parts, outline the chapters they contain, and highlight the essential contributions. Figure [1.7](#) illustrates the outline.

Part [I](#) provides a broad introduction to the thesis topic incremental process discovery. In Chapter [2](#), we provide an overview of related work and background information on process mining, modeling, and discovery. Further, we present a *systematic literature review on domain-knowledge-utilizing process discovery*. To this end, we propose a taxonomy for such process discovery approaches. Chapter [3](#), outlines the necessary foundational defi-



Figure 1.7: Thesis outline

nitions, mathematical concepts, and process mining principles that we utilize throughout the thesis. The last chapter of the first part, i.e., Chapter 4, proposes an extension of the state-of-the-art conformance checking technique *alignments*; we extend alignments by defining *infix* and *postfix alignments* and presenting their computation. This extension is used in the following part to specify an incremental process discovery approach.

Part II, which comprises three chapters, is devoted to incremental process discovery. First, we propose a foundational *incremental process discovery framework* that allows us to gradually discover process models from event data in Chapter 5. We extend this framework in the two subsequent chapters. Chapter 7 proposes an extension allowing to *freeze submodels during incremental process discovery*. The incremental process discovery approach does not alter frozen process model parts, providing users with a powerful option to guide the process discovery. Chapter 6 extends the framework by allowing users to *utilize process execution fragments within incremental process discovery* next to complete process executions.

Part III deals with event data interactions in the broader context. In Chapter 8, we present *novel definitions of process execution variants for partially ordered event data*. Further, we present corresponding *variant visualizations* that are generally paramount for users during event data exploration. Proceeding from these variants, we introduce a novel *query language for process executions and variants that contain partially ordered event data*. The proposed query language facilitates users in event data exploration.

Part IV deals with implementing and applying the techniques and approaches proposed so far. We present an *open-source process mining software tool called Cortado* in Chapter 10. Cortado implements all the proposed algorithms and approaches presented earlier regarding incremental process discovery in an end-user-oriented tool. Chapter 11 presents a *case study* in which we analyze a healthcare process using Cortado and its distinct features, as outlined in this thesis.

Finally, Part V concludes this thesis. Chapter 12 summarizes the central contributions. Further, we discuss remaining challenges in the broad context of incremental process discovery and present directions for future work.

Figures 1.8 to 1.11 position the chapters in the context of the incremental process discovery framework, shown in Figure 1.6 (page 14). Figure 1.8 shows the positioning of Chapter 5, formally introducing the overall framework and presenting concrete algorithms supporting complete process executions. Figure 1.9 positions Chapter 6, which introduces an algorithm that supports process execution fragments besides complete executions, compared to Figure 1.8. Moreover, the proposed algorithm in Chapter 6 utilizes the infix and postfix alignments presented in Chapter 4. Figure 1.10 positions Chapter 7, which extends the framework by allowing for specifying frozen process model parts, i.e., a form of domain knowledge. Finally, Figure 1.11 positions Chapters 8 and 9, which deal with representing and querying process behavior from event data.

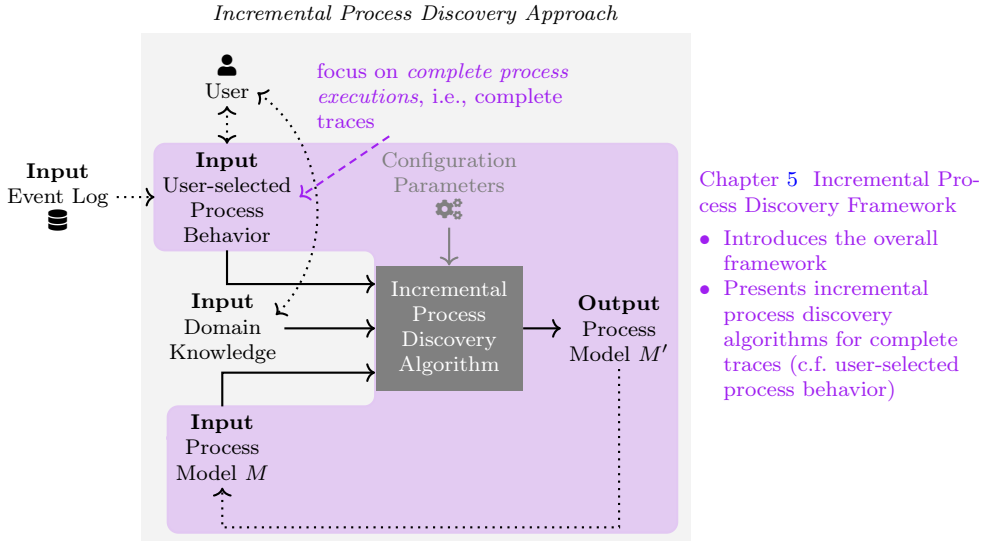


Figure 1.8: Positioning Chapter 5 Incremental Process Discovery Framework (Part II) in the context of incremental process discovery

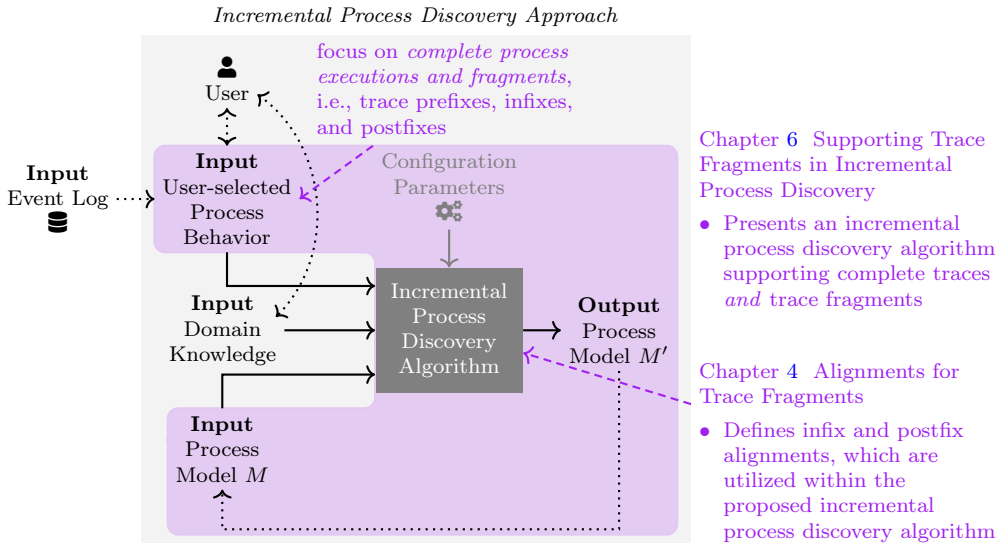


Figure 1.9: Positioning Chapter 6 (Part II) and Chapter 4 (Part I) in the context of incremental process discovery

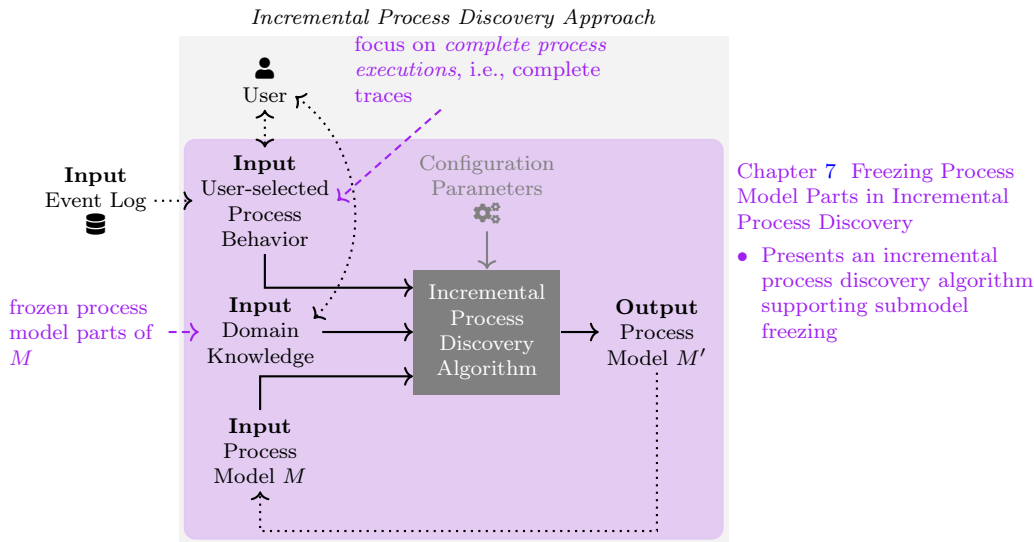


Figure 1.10: Positioning Chapter 7 in the context of incremental process discovery

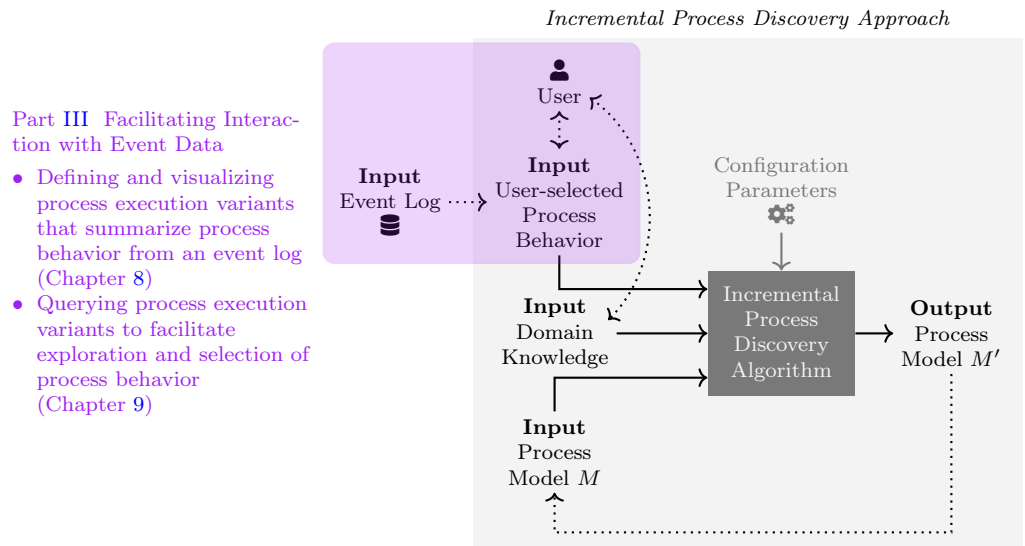


Figure 1.11: Positioning Part III, comprising Chapters 8 and 9, in the context of incremental process discovery

# Chapter 2.

## Literature Review

This chapter is largely based on the following published work.

- D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Utilizing domain knowledge in data-driven process discovery: A literature review. *Computers in Industry*, 137:103612, 2022. doi:10.1016/j.compind.2022.103612 [184].

This section presents a *systematic literature review* on non-conventional process discovery. Recall that conventional process discovery solely discovers process models from event data automatically, cf. Figure 1.3 and Section 1.3.1. In contrast, reconsider Figure 1.4, which provides a high-level overview of non-conventional process discovery approaches. These discovery approaches assume domain knowledge in addition to event data as input and may allow for user interaction with the discovery algorithm. Thus, from an input-output perspective, non-conventional process discovery differs significantly from conventional process discovery.

Figure 2.1 provides a more detailed conceptual view on non-conventional process discovery. For simplicity, we consider any form of additional information besides event data as domain knowledge; thus, user interaction with a process discovery approach is considered domain knowledge, too. As illustrated in Figure 2.1, three phases can be distinguished within the context of non-conventional process discovery:

1. the event data *preprocessing phase*,
2. the actual *process discovery phase* in which the process model is discovered based on an event log and domain knowledge, and
3. the *process model post-processing phase*.

In each phase, domain knowledge can be utilized. Note that Figure 2.1 summarizes various options for utilizing domain knowledge; however, concrete approaches use only selected options illustrated in Figure 2.1. Thus, most approaches can be easily categorized into one of three phases, i.e., event data preprocessing, process discovery, and process model post-processing approaches. Therefore, Figure 2.1 should be considered a summarizing overview of various options on domain-knowledge-utilization within the context of process discovery.

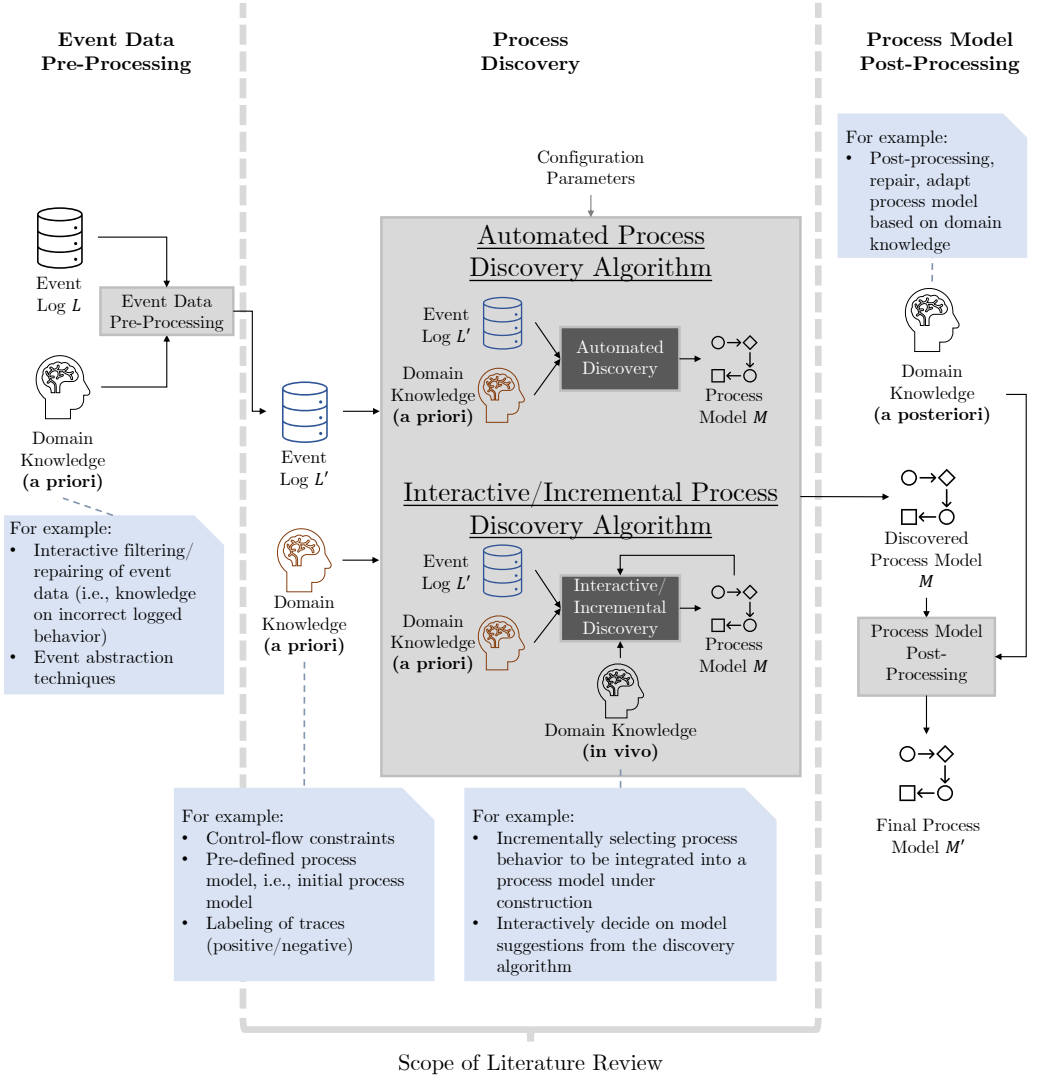


Figure 2.1: Conceptual overview of different non-conventional process discovery approaches and potential provisioning timing of domain knowledge, i.e., a priori, in vivo, and a posteriori (partly adapted from [184, Figure 2])

During the event data preprocessing phase, domain knowledge can be utilized in various ways. Domain experts can use interactive event data exploration and filtering approaches, for instance, [73, 129, 170], and apply their knowledge to increase the quality of the event log  $L$ , cf. Figure 2.1. Further, event abstraction techniques might be applied that often require domain experts to provide activity patterns that are used to generate high-level



activities from low-level activities [132]. We refer to [238] for an extensive overview of event abstraction techniques. In short, many event data preprocessing techniques exist that allow the utilization of domain knowledge to improve the quality of event logs; however, we do not consider such approaches in this review.

For the second phase, the actual process discovery phase, Figure 2.1 distinguishes automated and interactive/incremental non-conventional process discovery approaches. Automated approaches utilize a priori-provided domain knowledge and an event log  $L'$ , which is obtained after applying preprocessing techniques, cf. Figure 2.1. Based on these inputs, a process model  $M$  is fully automatically discovered. Thus, domain knowledge is considered an additional input compared to conventional process discovery, cf. Figure 1.3. For instance, a priori domain knowledge can be negative process executions that should not be incorporated into the discovered process model [192]. Control-flow constraints, i.e., precedence constraints over activities [99], are a further example of a priori domain knowledge used within the process discovery phase.

In contrast to automated non-conventional process discovery, interactive/incremental approaches (cf. Figure 2.1) utilize domain knowledge in vivo, i.e., domain knowledge is provided and utilized while the approach discovers the process model. For instance, domain knowledge provided in vivo includes incrementally selected process execution variants provided to an incremental process discovery approach [174], or user feedback on activity positioning recommendations provided by the process discovery approach [74]. Additionally, interactive/incremental process discovery algorithms may utilize a priori-provided domain knowledge. In short, automated and interactive/incremental non-conventional discovery approaches differ mainly because the domain knowledge is provided in vivo in incremental/interactive approaches, i.e., they do not operate fully automated.

Finally, process model post-processing can be applied after discovering a process model  $M$ , cf. Figure 2.1. Note that process model post-processing and, likewise, event data preprocessing are not a necessity. Nevertheless, these two phases are often performed as part of process discovery. For example, domain experts may manually edit the process model in an editor [113]. Further, (interactive) process model repair techniques [14, 85, 86, 162] may be applied to add behavior not yet reflected by the model  $M$ . Likewise, approaches to process model simplification can be applied [190]. Moreover, the discovered process model  $M$  could be merged with other process models using techniques as [116]. Note that some techniques mentioned above require further inputs than domain knowledge and a process model. However, a detailed review of process model post-processing approaches is outside the scope of this thesis.

In conclusion, Figure 2.1 demonstrates that non-conventional process discovery algorithms are more diverse compared to conventional ones, from the user's point of view. Therefore, we present a systematic literature review to organize said approaches. We focus on the actual process discovery phase as highlighted in Figure 2.1. Thus, we exclude non-conventional approaches for event data preprocessing and process model post-processing. However, we include selected approaches for repairing process models if they can be employed as incremental process discovery algorithms. Note, however, that process model repair approaches were primarily developed for the process model post-processing phase, cf. Figure 2.1. In the course of this literature review, we provide more detailed insights into how process model repair can be used for incremental process

discovery. For the sake of simplicity, we also refer to the process model repair approaches that we consider in this literature review as non-conventional process discovery.

The remainder of this chapter is structured as follows. We present distinguishing features in Section 2.1 to organize the various non-conventional process discovery approaches. Section 2.2 introduces the applied methodology and the design of this literature review. The identified and considered approaches in the context of this literature review are briefly presented in Section 2.3. Further, the approaches are systemically compared using the distinguishing features from Section 2.1. Next, open challenges in the field of non-conventional process discovery are presented and linked to this thesis and its research questions. Finally, Section 2.5 concludes this literature review.

## 2.1. Distinguishing Features

In this section, we present distinguishing features with corresponding characteristics to organize and compare various non-conventional approaches. First, we will briefly describe our approach to specifying the distinguishing features and how we have organized them. Next, we present the individual distinguishing features in detail and introduce the corresponding characteristics. Finally, we briefly discuss potential dependencies between different features and characteristics.

### 2.1.1. Defining Distinguishing Features

This section briefly covers the identification and organization of the distinguishing features for non-conventional process discovery. In [147], the authors define a method for taxonomy development for the information systems field. Note that a taxonomy is used to classify objects in a systematic manner. According to [147], a taxonomy comprises a set of  $n \in \mathbb{N}$  dimensions, referred to as  $D_1, \dots, D_n$ . Further, each dimension  $D_i \in \{D_1, \dots, D_n\}$  has at least two  $k_i \geq 2$  corresponding characteristics  $C_{i,j}$  for  $j \in \{1, \dots, k_i\}$ . Moreover, characteristics are 1) *mutually exclusive*, i.e., an individual object can have not more than one characteristic for the same dimension, and 2) *collectively exhaustive*, i.e., an individual object must have one characteristic for each dimension.

In this literature review, we omit mutual exclusiveness; however, collective exhaustiveness remains. Thus, an individual object, in this specific case a non-conventional process discovery approach, can have multiple but at least one characteristic for a dimension. To distinguish from the taxonomy concepts presented in [147], we refer to dimensions as distinguishing features. In the next section, we present the identified features.

### 2.1.2. Overview of the Distinguishing Features

Figure 2.2 summarizes the identified distinguishing features. We use gray highlighted boxes for features and light gray highlighted boxes for characteristics. The presented features are the result of analyzing identified approaches, internal discussion among the authors [184], and discussion with peers. Overall, we have identified first-level and second-level features, cf. Figure 2.2. For instance, if an approach has the characteristic ‘interactive/incremental’ for the feature ‘Degree of interactivity’, we are further interested if an

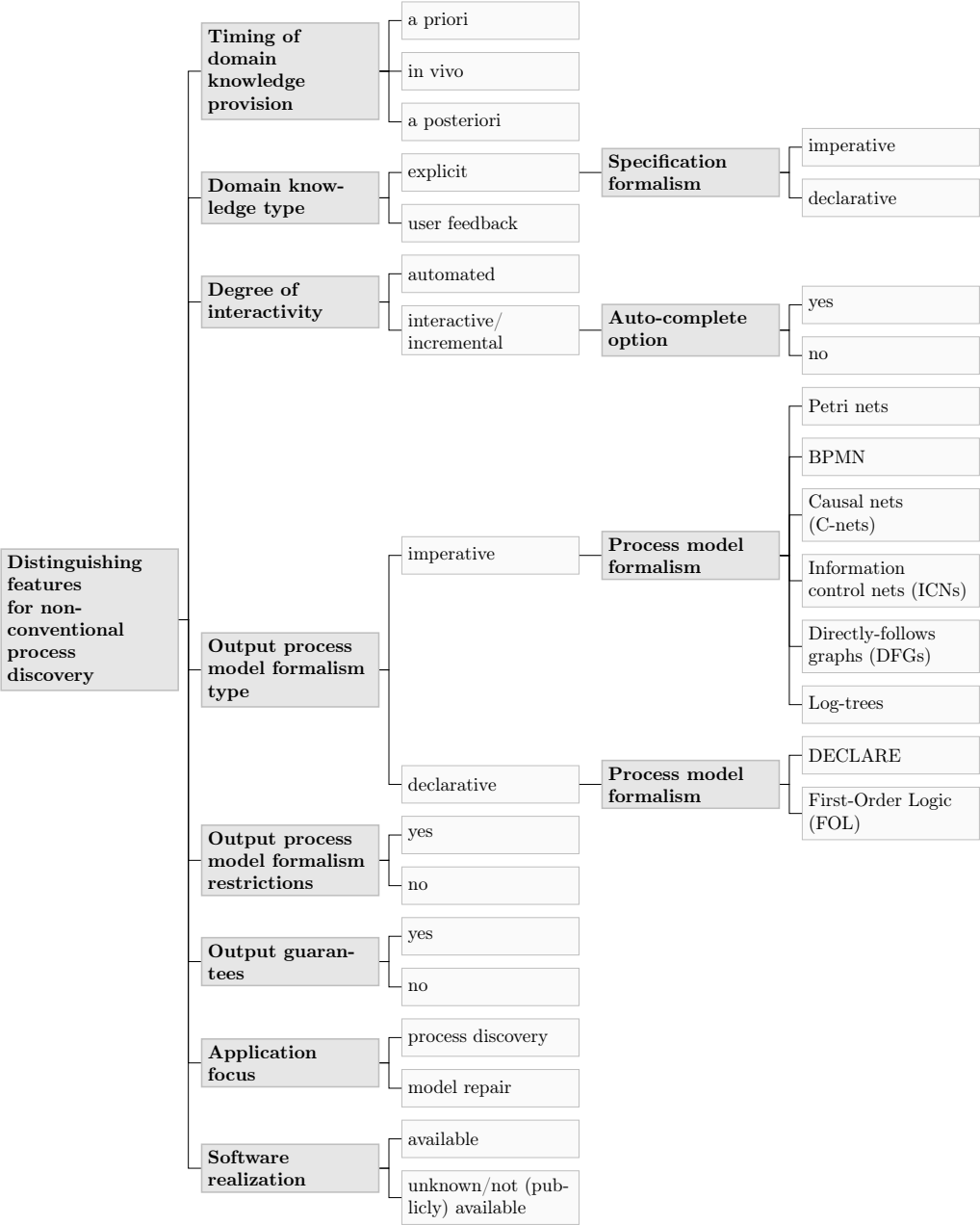


Figure 2.2: Overview of the distinguishing features for non-conventional process discovery, cf. gray filled boxes, and corresponding characteristics, cf. light gray filled boxes (partly adapted from [184, Figure 6])

‘Auto-complete option’, i.e., second-level feature exists. In the following, we introduce each distinguishing feature and corresponding characteristics from Figure 2.2.

## Timing of Domain Knowledge Provision

During the process discovery phase, as illustrated in Figure 2.1, domain knowledge can be utilized at different points in time. For instance, domain knowledge can be an additional input next to event data, domain knowledge can be utilized during an interactive discovery phase, and domain knowledge can be used after the discovery of a model to post-process it. Therefore, we are interested in comparing and classifying the ‘timing of domain knowledge provision’; we distinguish three characteristics:

1. *a priori* describes approaches in which domain knowledge is provided as an additional input alongside event data at the beginning of the discovery phase,
2. *in vivo* describes approaches where domain knowledge is provided while the process model is actively discovered,
3. *a posteriori*<sup>1</sup> describes approaches where domain knowledge is provided and used after a process model is discovered.

## Domain Knowledge Type

The concept of domain knowledge is vast and can take on various forms. Thus, we aim to classify the type of domain knowledge that is utilized by the approaches. We distinguish two primary forms of domain knowledge: *explicit* domain knowledge and domain knowledge in terms of *user feedback*.

1. *Explicit domain knowledge* is a type of domain knowledge that is formally specified and inputted into the discovery algorithm. In other words, the user needs to specify the domain knowledge formally. Examples of explicit domain knowledge are precedence constraints between process activities and initial process models, which are used as a starting point for incremental process discovery algorithms.
2. On the contrary, *user feedback* refers to the decisions made by a user from the available options presented by a discovery algorithm. An example of user feedback is the gradual selection of process behavior that an incremental process discovery incorporates into a process model. Unlike explicit domain knowledge, discovery algorithms actively request user feedback.

If an approach utilizes explicit domain knowledge, we are further interested in the *specification formalism* used to specify the explicit domain knowledge. Note that the *specification formalism* represents a second-level distinguishing feature. We distinguish two specification formalisms.

<sup>1</sup>Note that we excluded non-conventional approaches that are intended for process model post-processing, i.e., the third phase, cf. Figure 2.1. However, since there are approaches that are classified to the second phase, i.e., process discovery, and utilize domain knowledge also a posteriori, we include this characteristic for the feature ‘Timing of domain knowledge provision’.

1. *Imperative* formalisms for domain knowledge follow a closed-world assumption. Therefore, only the behavior explicitly described in the formalism is permissible, while behavior not mentioned is excluded. For instance, recall the imperative process model formalism depicted in Figure 1.2.
2. *Declarative* domain knowledge formalisms follow an open-world assumption. Therefore, declarative formalisms usually allow the specification of constraints. If the behavior of a process conforms to the specified rules, it is permissible. Thus, unlike imperative formalisms, not all permissible behavior must be explicitly specified.

Finally, note that feature ‘domain knowledge type’ and feature ‘timing of domain knowledge provision’ are independent. As an illustration, a user can provide explicit domain knowledge directly to the discovery approach *in vivo*, which means during the discovery phase or beforehand, referred to as *a priori*.

### Degree of Interactivity

During the process discovery phase, non-conventional discovery approaches differ in their course to involving users. Some algorithms are fully automated, meaning users cannot interact during the discovery phase. Others, however, involve users significantly in an interactive discovery phase that allows users to make important decisions about the process model being discovered. Consider Figure 2.1; we distinguish two main non-conventional approaches.

1. *Automated* approaches do not offer any option to provide domain knowledge during the actual discovery of the process model. Hence, by definition, approaches being classified as automated do not utilize domain knowledge *in vivo*.
2. *Interactive/incremental* approaches, on the contrary, allow to provide domain knowledge *in vivo*, i.e., during the actual process discovery phase.

We further distinguish approaches classified as ‘interactive/incremental’ by a second-level feature *auto-complete option*. An auto-complete function enables the algorithm to progress in discovering a model independently when the user opts out of providing feedback. Approaches can either have or not have an auto-complete option.

### Output Process Model Formalism Type & Restrictions

As already shown in Figure 1.2, various process model formalisms exist.<sup>2</sup> Generally, it is not possible to translate any process model into another formalism while maintaining the exact behavior specified in the target formalism. If a model contains elements or behavioral constructs from a formalism that are not part of or cannot be modeled in the target formalism, the model cannot be translated with behavioral exactness. Thus, the formalism used by a discovery approach is of great importance. Second, potential restrictions on the class of process models that a discovery algorithm can discover within a given formalism are another critical distinguishing feature. Examples of such restrictions are, for instance, if a discovered model can contain the same activity labels multiple

<sup>2</sup>Note that Figure 1.2 solely illustrates imperative process model formalisms.

times. Therefore, the process model formalism used by a discovery approach and potential restrictions regarding the class of models within a given formalism, also referred to as *representational bias* [206, 227], are critical distinguishing features.

To this end, we distinguish the class of supported process model formalism per approach, i.e., the first-level distinguishing feature *output process model formalism*. We distinguish between *imperative* and *declarative* process model formalisms; we refer to [87, 88, 153] for detailed comparisons between these two model paradigms. In brief, imperative language specifies how a process is executed by explicitly modeling what is allowed. Instead of explicitly defining what is allowed, declarative languages initially allow for all possible behavior, and every new element in such a model restricts the potential executions further. For each formalism class, we further distinguish the exact formalism used, i.e., *Petri nets*, *BPMN*, *causal nets (C-nets)*, *information control nets*, *Directly Follows Graphs (DFGs)*, *log-trees*, *DECLARE*, or *FOL*.<sup>3</sup>

Furthermore, we distinguish the approaches according to whether they constrain the chosen process model formalism. For example, specific process discovery algorithms cannot identify models that contain identical activity labels in separate locations within the model, so-called duplicate labels—although process model formalisms allow this. We have two characteristics for the distinguishing feature *constraints on the output process model formalism*, namely *yes* and *no*.

## Output Guarantees

When comparing different approaches, it's crucial to consider the guarantees provided regarding the discovered model concerning the provided event log and domain knowledge. For instance, if replay fitness is guaranteed, all the process behavior from the event log is captured in the model. Further, does the approach guarantees that the discovered model adheres to the provided explicit domain knowledge? Guarantees can also pertain to process models and their properties, such as the approach guarantees to return a sound WF-net for arbitrary input. In short, guarantees regarding the discovered process model are essential when comparing discovery approaches.

## Application Focus

Although this literature review focuses on non-conventional process discovery, approaches from *model repair* [85, 86], which is considered a dedicated research field within process mining, can also be used to discover process models incrementally. Hence, we consider model repair to a certain extent as non-conventional process discovery and, therefore, include model repair approaches. However, a complete overview of model repair is beyond the scope of this review. Instead, we list approaches that can be distinguished using the proposed distinguishing criteria and can be considered for non-conventional process discovery.

Figure 2.3 provides a high-level illustration of model repair. As input an event log  $L$  and a process model  $M$  is assumed. If  $M$  does not entirely support the process behavior recorded in  $L$ , the process model repair algorithm alters respectively extends the process

<sup>3</sup>The list of process model formalisms is derived from the identified approaches. Therefore, it is not considered complete and may need to be extended for future approaches not considered in this review.

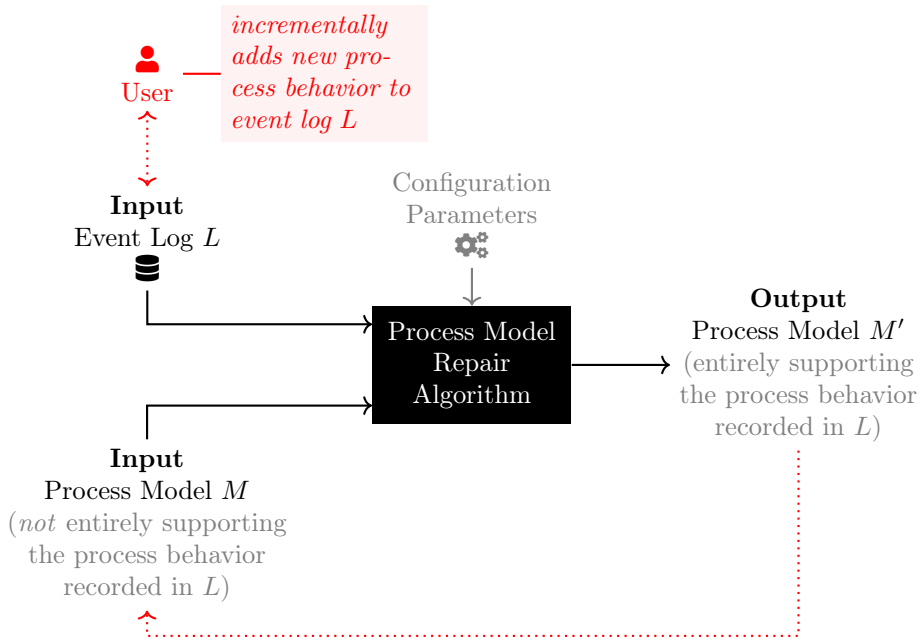


Figure 2.3: Schematic overview of process model repair approaches; elements highlighted in red illustrate the use of a model repair algorithm for incremental process discovery purposes

model  $M$  such that all process behavior from  $L$  is supported by the altered respectively extended model  $M'$ . Process model repair algorithms aim to change the model  $M$  as little as possible, i.e., model  $M$  and  $M'$  should be as close as possible [86]. The reason for this goal is the intention to *repair* the process model. If this goal did not exist, users could use instead process discovery algorithms (cf. Figure 1.3 in comparison) that are fitness-preserving to learn a new process model  $M'$  from  $L$  that entirely supports the process behavior from  $L$ . In this case, however, there is no guarantee of similarity between the provided process model  $M$  and the new model  $M'$  as the input model  $M$  is ignored.

Model repair approaches can be applied gradually and thus mimic incremental process discovery, cf. Figure 2.3. Assume the output process model  $M'$  is used in a further iteration again as input, and a user gradually extends  $L$  by adding more process behavior. In this regard, model repair approaches can be used to mimic incremental process discovery. We are therefore interested in distinguishing the intended *application focus* of algorithms that we consider in the context of this literature review. The application focus indicates whether an approach is primarily designed for *process discovery* or *model repair* use cases.

## Software Realization

Having software support is crucial, especially for process discovery approaches that utilize domain knowledge. Interactive algorithms for process discovery must communicate

intermediate results, design decisions, or questions to the user and request feedback from them. Additionally, users require software support for eliciting and specifying domain knowledge, such as constraints in a specific formalism, and verifying that the specified knowledge is free of contradictions. As a result, we are interested in comparing the software realizations of different approaches.

Aside from the software’s implementation, we also seek information about its graphical user interface (GUI) and how it facilitates user interaction. Specifically, we would like to understand how the tool assists users in defining explicit domain knowledge and how it handles the feedback loop between the user and the algorithm.

### 2.1.3. Dependencies Among Characteristics and Features

Table 2.1.: Overview of mutual exclusiveness regarding characteristics for each distinguishing feature; second-level distinguishing features are colored gray (partly adapted from [184, Table 2])

Distinguishing Feature	Mutual Exclusive Characteristics
Timing of domain knowledge provision	
Domain Knowledge Type	
Specification formalism*	
Degree of Interactivity	✓
Auto-complete option	✓
Output Process Model Formalism Type	✓
Process model formalism	✓
Output Guarantees	✓
Application Focus	✓
Software Realization	✓

\* The feature ‘specification formalism’ characteristics are mutually exclusive per individual, explicit domain knowledge input. For instance, assume an approach that requires two different domain knowledge inputs that are both explicit. Thus, each explicit domain knowledge input is either provided using imperative or declarative specification formalism. However, an approach that utilizes explicit domain knowledge can be categorized as imperative and declarative in the case of multiple explicit domain knowledge inputs.

As discussed earlier, we do not require mutual exclusiveness for the characteristics of individual distinguishing features. In Table 2.1, we report per distinguishing feature if mutual exclusiveness applies. The reason for omitting mutual exclusiveness for the feature ‘Timing of domain knowledge provision’ is, for example, that approaches may utilize domain knowledge at different points in time. Therefore, we cannot assign such approaches to a single characteristic; instead, several characteristics apply. A similar argumentation applies for the feature ‘domain knowledge type’. For example, approaches can use both explicit domain knowledge and user feedback.

Further, there exist two dependencies between certain features and characteristics. Thus, not all theoretically potential feature/characteristic combinations exist. In the



following, we explain these two dependencies.

1. If the feature ‘application focus’ of an approach is model repair, the approach requires a process model as input, i.e., explicit domain knowledge. Hence, the approach’s ‘domain knowledge type’ is explicit. Note that the approach can additionally utilize user feedback because the characteristics of the feature ‘domain knowledge type’ are not mutual exclusive.
2. If the ‘degree of interactivity’ of an approach is automated, the characteristic ‘in vivo’ is not applicable for the feature ‘timing of domain knowledge provision’.

## 2.2. Methodology & Design

This section, briefly presents the methodology applied for conducting the literature review. We follow the established methodologies and guidelines presented in [105, 242]. According to [242], five phases of a literature review can be distinguished.

1. Defining the literature review’s scope
2. Conceptualizing the topic of the literature review
3. Literature search
4. Literature analysis and synthesis
5. Research agenda

Regarding the first phase, i.e., defining the literature review’s scope, reconsider Figure 2.1, which overviews the field of non-conventional process discovery and highlights the scope of this literature review. Our focus in this literature review is on approaches incorporating domain knowledge in the process discovery phase while excluding those solely pertaining to event data preprocessing or process model post-processing, cf. Figure 2.1. As an example, we omit event data filtering techniques using domain knowledge [170] and event abstraction techniques [132, 238]. Further, we are interested in specific approaches; thus, we exclude case studies on applying non-conventional process discovery, for instance, [22, 23]. Moreover, we exclude work describing conceptual ideas and frameworks without a specific implementation, respectively instantiation. For example, in [100], the authors present requirements for an interactive workflow mining system. Similarly, in [104], the authors propose a high-level framework for mining workflows from document versioning systems. Furthermore, we exclude approaches that do not use event data as input, for instance, approaches that solely use natural language text to discover a process model [95, 96]. Lastly, we exclude approaches that are presented in this thesis.

Finally, we partly consider process model repair techniques, cf. Section 2.1.2. Although they are intended to be used as a process model post-processing technique, these techniques can also be applied incrementally to discover a process model as discussed in Section 2.1.2. Therefore, we include repair techniques; however, we do not provide a complete overview of these techniques unless they can be differentiated based on the proposed distinguishing features. Conclusively, we define the scope using four criteria (C), which are evaluated for each document found during the database search.

Table 2.2.: Executed search queries to identify relevant literature

Database	Executed query
Scopus	TITLE-ABS-KEY("process mining" AND ("process discovery" OR "model repair") AND ("interactive" OR "domain knowledge" OR "hybrid intelligence" OR "human-in-the-loop"))
ACM Digital Library	[All: "process mining"] AND [[All: "process discovery"] OR [All: "model repair"]] AND [[All: "interactive"] OR [All: "domain knowledge"] OR [All: "hybrid intelligence"] OR [All: "human-in-the-loop"]]
SpringerLink	"process mining" AND ("process discovery" OR "model repair") AND ("interactive" OR "incremental" OR "domain knowledge" OR "hybrid intelligence" OR "human-in-the-loop")

- C1 The document found is a single article, not a proceedings volume, a PhD thesis, or the like. Further, the document is written in English. After all, the document is not part of this thesis. For example, we exclude [69, 93].
- C2 The main focus of the found document is either on process discovery or on process model repair. For instance, approaches on conformance checking designed for interactive process discovery are excluded [72].
- C3 The document proposes a specific algorithm; as explained above, for example, case studies [22, 23], overviews [162], and general frameworks without concrete instantiations [111] are excluded.
- C4 The document deals with process discovery or process model repair using event data and domain knowledge as input, i.e., the approach can be considered a non-conventional process discovery approach, cf. Figure 1.4 (page 10).

The second phase, i.e., the conceptualization of the research field, is approached through the distinguishing features, cf. Section 2.1. We use the distinguishing features to systematically compare the identified approaches. In addition, we briefly summarize each identified approach to provide insight into the distinctive approaches.

For the literature search, we perform search queries in common research databases. We queried Scopus<sup>4</sup>, ACM Digital Library<sup>5</sup>, and SpringerLink<sup>6</sup> that together index a plethora of conference proceedings and journals related to computer science. We use the semantically same query for all three databases; Table 2.2 provides an overview of the executed queries. We do not further restrict the search results; for example, we do not limit the publication time or restrict the results to journal publications only.

<sup>4</sup><https://scopus.com>

<sup>5</sup><https://dl.acm.org/>

<sup>6</sup><https://link.springer.com/>

Table 2.3.: Overview of the literature search process; twelve non-conventional process discovery approaches were eventually identified

Source	Hits**	Publications satisfying		
		C1–C2	C1–C3	C1–C4
Scopus	61	26	14	7
ACM Digital Library	70	26	11	2
SpringerLink	690	96	74	8
<i>Sum of approaches*</i>				<i>11</i>
Backward Search (given the 12 identified publications)	270	71	61	7
<i>Sum of approaches*</i>				<i>12</i>

\* The total given shows the number of unique approaches, *not* the number of publications identified. As there are sometimes several publications describing the same approach, for example, a workshop paper that was extended in a journal article or a corresponding tool paper, and there are overlaps between the database results, the total shown is lower than the total of the values per database.

\*\* The queries were performed on 25.08.2021 as part of the corresponding publication [184] and again on 30.06.2023 to ensure up-to-dateness.

2.3. Identified Approaches

First, Section 2.3.1 provides an overview of the identified approaches and introduces them briefly. Subsequently, Section 2.1 systematically compares the identified approaches using the distinguishing features introduced.

2.3.1. Overview

Table 2.3 summarizes the results from the performed queries. For example, the query executed in Scopus yielded 61 results. Next, we evaluate the criteria listed in Section 2.2 for each result. For the results from Scopus, we identify seven publications that satisfy all four criteria. In total, we have identified eleven approaches from the three queried databases, i.e., Scopus, ACM, and SpringerLink. Next, we performed a backward search given the eleven identified approaches. The backward search yielded one more publication that we did not find before. Thus, we finally identified twelve approaches that are considered in this literature review.

Table 2.4 lists the twelve identified approaches. Note that we summarized approaches if they have been introduced among multiple publications; as stated in Table 2.4. In addition, Table 2.4 provides an overview of the distinguishing features and the corresponding number of approaches that can be assigned to certain characteristic values. In the following, each approach, i.e. A1 to A12, is briefly presented and key characteristics of selected features (cf. Section 2.1) discussed.

Table 2.4.: Overview of identified non-conventional process discovery approaches that meet the four criteria, i.e., C1 to C4, specified in Section 2.2

Abbr.	First Author	Title **	Reference *	Year **
A1	Goedertier et al.	Robust process discovery with artificial negative events	[98, 196]	2009
A2	Maggi et al.	User-guided discovery of declarative process model	[130]	2011
A3	Rembert et al.	Process discovery using prior knowledge	[161]	2013
A4	Yahya et al.	Process discovery by synthesizing activity proximity and user's domain knowledge	[246]	2013
A5	Dixit et al.	Using Domain Knowledge to Enhance Process Mining Results	[70]	2017
A6	Greco et al.	Process Discovery under Precedence Constraints	[97, 99]	2015
A7	Fahland et al.	Model repair — aligning process models to reality	[85, 86]	2015
A8	Armas Cervantes et al.	Interactive and incremental business process model repair	[14, 13]	2017
A9	Canensi et al.	Multi-level interactive medical process mining	[43]	2017
A10	Dixit et al.	Interactive data-driven process model construction	[74, 71]	2018
A11	Yürek et al.	Interactive process miner: a new approach for process mining	[248]	2018
A12	Ferilli et al.	Incremental declarative process mining with WoMan	[90, 91, 92]	2020

\* We summarize approaches published in different articles, for example, a workshop paper extended in a journal article.

\*\* We list the title and the year of the most recent publication.

Table 2.5.: Overview of the number of approaches assigned to the individual characteristics of the defined distinguishing features, cf. Figure 2.2

Distinguishing feature (1 <sup>st</sup> level) Distinguishing feature (2 <sup>nd</sup> level)	Characteristic	Number of Approaches**
Timing of domain knowledge*	a priori	9 (75%)
	in vivo	6 (50%)
	a posteriori	1 (8%)
Domain knowledge type*	explicit	11 (92%)
Specification formalism	imperative	6 (55%)
	declarative	5 (45%)
	user feedback	4 (33%)
Degree of interactivity	automated	5 (42%)
	interactive/incremental	7 (58%)
Auto-complete option	yes	5 (71%)
	no	2 (29%)
Output process model formalism	imperative	10 (83%)
Process model formalism	Petri nets	4 (40%)
	BPMN	1 (10%)
	Causal nets (C-nets)	1 (10%)
	Information control nets	1 (10%)
	Directly follows graphs	2 (20%)
	Log-trees	1 (10%)
	declarative	2 (17%)
Process model formalism	DECLARE	1 (50%)
	FOL	1 (50%)
Output process model formalism restrictions	yes	3 (25%)
	no	9 (75%)
Output guarantees	yes	7 (58%)
	no	5 (42%)
Application focus	process discovery	10 (83%)
	model repair	2 (17%)
Software realization	available	7 (58%)
	unknown/not (publicly) available	5 (42%)

\* Non mutual exclusive characteristics, cf. Table 2.1

\*\* For the calculation of the relative numbers of the characteristics of 2<sup>nd</sup> level distinguishing features, we have taken the absolute number of approaches classified to the corresponding characteristic of the 1<sup>st</sup> level feature as a basis. The relative numbers are rounded to integers.

**Approach A1—Goedertier et al. (2009)**

Goedertier et al. [98, 196] present the AGNEs algorithm, an *automated* discovery approach that utilizes negative events to transform process discovery into a classification problem. The approach assumes the completeness of the event log provided, i.e., the event log fully describes the behavior of the actual process under consideration. Note that the completeness assumption can be problematic if processes contain much parallel behavior. In this case, many different sequences of activities are likely due to parallelity. For instance, consider a process with 6 parallel activities; thus,  $6! = 720$  many different permutations of these activities exist. However, it is unlikely that all these permutations will also be found in an event log. Therefore, the approach allows the optional input of domain knowledge that characterizes parallel and sequential activities and dependencies between activities. If the domain knowledge conflicts with the recorded behavior in the event log, the provided domain knowledge is favored. Next, for each process execution, negative events are generated, i.e., at each position in a process execution, events that are not allowed to take place are automatically determined. From these process executions enriched with negative events, a Petri net is eventually discovered.

**Approach A2—Maggi et al. (2011)**

Maggi et al. [130] propose an *automated* approach for discovering DECLARE models, which use behavioral templates to represent relationships between process activities. The user selects *a priori* a desired subset of templates for the algorithm to use, restricting the resulting process model. The approach then learns constraints based on the selected templates and generates a process model that ensures the given event data fits the discovered constraints. This user selection is considered domain knowledge, a crucial aspect of the proposed approach, and significantly impacts the discovered process model. Further, this selection lets the user focus on specific process parts/constraints of interest. We consider this *a priori* user selection as domain knowledge because it is a fundamental part of the proposed approach and significantly influences the discovered process model.

**Approach A3—Rembert et al. (2013)**

Rembert et al. [161] propose an *automated* discovery approach that utilizes *a priori* domain knowledge in the form of an augmented Information Control Net (ICN), which is an *imperative* formalism. ICNs are directed graphs where nodes represent activities and different edge types model control flow constraints, for instance, precedence constraints, dependency relations, independency relations, and mutual exclusion constraints. In an augmented ICN, the user assigns belief values between 0 and 1 to the relationships expressed in the ICN, representing the user's belief in the given dependency between the corresponding activities. The approach learns a process model automatically from the event data and the augmented ICN; the output formalism is also an ICN. The authors emphasize that their approach is particularly useful for dealing with uncertain event data or rare process behavior.

**Approach A4—Yahya et al. (2013)**

Yahya et al. [246] proposes the proximity miner, i.e., an *automated approach* utilizing *a priori* provided domain knowledge. Users can input their knowledge about a process in terms of causal, unrelated, and parallel relationships between activities. They can also specify the start and end activities. Using the concept of activity proximity, which indicates the connections between activities as recorded in the event log, and the provided domain knowledge, the proximity miner creates a process model. This model is a directed graph that displays the directly-follows-relation between activities, i.e., a DFG [212].

**Approach A5—Dixit et al. (2017)**

Dixit et al. [70] present an *automated* discovery approach that uses, besides an event log, an initial process model and user-specified DECLARE constraints [151] as input. Note that the approach restricts to a subset of all available DECLARE templates. Process model modifications are applied to the initially provided process model to generate a set of candidate models. The authors present three techniques: 1) a brute force modification approach that randomly edits the initial model, 2) a genetic modification approach that applies edit operations guided by the standard four quality measures for process models, and 3) a constraint-specific modification that edits the initial model guided by the user-defined constraints. All three approaches yield a set of process models derived from the initial process model. Next, the resulting models are evaluated based on the event data—standard quality measures are calculated, i.e., replay fitness, precision, generalization, and simplicity—and based on the number of satisfied user-specified constraints.<sup>7</sup> These five measures are used to create a Pareto front of the best process models. In contrast, consider Figure 1.1 showing that a process discovery returns a single process model. Therefore, the obtained selection of process models is presented to the user, who can select a process model. We categorize this final selection as *a posteriori user feedback*.

**Approach A6—Greco et al. (2015)**

Greco et al. [97, 99] present an automated approach to process discovery that incorporates explicit domain knowledge in the form of precedence constraints. These precedence constraints define the relationships between activities and are provided *a priori* alongside the event log. The resulting process models are represented as extended causal nets.<sup>8</sup> The approach guarantees that the resulting extended causal net describes the behavior in the event data and fulfills the given precedence constraints; otherwise, if event data and user-defined precedence constraints contradict, no model is returned. According to the authors, this approach is beneficial for addressing the log completeness problem, which arises when event data only captures some but not all possible executions of the process being studied.

<sup>7</sup>For an introduction to the standard quality measures, we refer to [211, Chapter 6.4.3] and [38].

<sup>8</sup>Causal nets *C-nets* are introduced in [223].

**Approach A7—Fahland et al. (2015)**

Fahland et al. [85, 86] introduce the first process model repair approach, which works *automated*. As input, the proposed approach assumes an *a priori* provided process model and an event log. By modifying the input model, the approach ensures that the resulting process model accurately represents all the process behavior recorded in the provided event log. As discussed before, cf. Section 2.1.2, model repair techniques can be applied incrementally to mimic incremental process discovery, i.e., a process model is gradually repaired. Thus, a repaired process model is used again as input in the next iteration, cf. Figure 2.3 (page 27). This procedure allows users to discover the process model gradually and to control the process discover phase by selecting event data to be added incrementally. We categorize this repair approach as *interactive/incremental*, i.e., starting from an initial process model (*explicit, a priori* domain knowledge), we repair the process model by incrementally adding trace variants to it (*explicit, in vivo* domain knowledge). The repair approach generally works on Petri nets, i.e., no restriction on a specific subclass. Moreover, an *auto-complete option* is theoretically given by automatically adding all remaining non-fitting process behavior from a given log in one go.

**Approach A8—Armas Cervantes et al. (2017)**

Armas-Cervantes et al. [14] propose an interactive and incremental process model repair approach that relies on user feedback. The approach requires an *a priori* provided process model in BPMN notation as input besides an event log. In brief, mismatches between the provided model and the event log are detected and displayed to a user who manually repairs the process model. These visualizations of discrepancies between the model and the log are key feature of the approach. Next to visualizing the discrepancies, the approach also visualizes repair proposals based on modification patterns in the model. The user can manually repair the process model or apply the suggested repair based on the visual feedback. However, the authors note that resolving discrepancies between the log and model may require a significant amount of manual effort on the user's part.

**Approach A9—Canensi et al. (2017)**

Canensi et al. [43] propose an *interactive/incremental* process discovery approach that consists of two main phases. First, a process model, i.e., a log-tree, is discovered using a conventional process discovery algorithm [36]. In brief, a log-tree encodes all traces from the event log in a tree structure and thus can contain identical activity labels multiple times. The discovered log-tree describes the entire event log, i.e., perfect fitness, and additionally has perfect precision, i.e., the model allows no other behavior that is not recorded in the event log. Since the log tree has perfect fitness and precision, the model may lack generalizability and simplicity, which are important quality dimensions of process models [40]. To address this issue, the second phase of the approach involves abstraction and generalization of the process model based on user feedback and explicit domain knowledge. Users can specify subgraphs, which the approach identifies and highlights in the log-tree. Next, based on *user feedback*, i.e., the user selects identified subgraphs in the log-tree, the approach merges the selected subgraphs to obtain a simplified log-tree. While this may result in a log-tree that describes other behavior not



present in the provided event log, the model's initial perfect accuracy is balanced against generalizability and simplicity.

#### Approach A10—Dixit et al. (2018)

Dixit et al. [71, 74] propose an *interactive/incremental* process discovery approach where the user constructs the process model in an interactive editor. The approach recommends modeling options based on the provided event log to the user. The process model formalism used is free-choice [63] WF-nets, a subclass of *Petri nets*. Starting from an initial model, the user gradually constructs the model by adding new elements. The approach guarantees that the Petri net under construction remains sound and free-choice, two favorable property of Petri nets, by restricting the edit operations to applying synthesis rules [63]. The approach offers the user three rules or methods to change the net: abstraction rule (adding a new place and a new transition), place rule (adding a new place), and transition rule (adding a new transition). These three rules are called synthesis rules [63], which guarantee that the Petri net under construction remains free-choice and sound; both are favorable properties of Petri nets. In short, a user element-wise constructs a WF-net guided by recommendations that are derived from the provided event log from the approach.

#### Approach A11—Yürek et al. (2018)

Yürek et al. [248] propose the Interactive Process Miner (IPM), an approach to interactive/incremental discovery that leverages explicit domain knowledge *in vivo*. First, the approach discovers a DFG from the given event log. Then, the user can explicitly modify the model by merging multiple activities into a single one, deleting activities, and adding activities. However, the algorithm processes the actual change to the DFG, and the user only specifies one of the three above mentioned changes, for example, between process activity *a* and *b*, process activity *x* should be executed. The algorithm then updates the process model accordingly. This procedure can be repeated iteratively, cf. [248, Figure 3].

#### Approach A12—Ferilli et al. (2020)

Ferilli et al. [90, 91, 92] propose an *interactive/incremental* process discovery approach called WoMan that represents process behavior in a declarative way using FOL [18]. Thus, the resulting process model is a set of FOL formulae. Starting from an initial model, i.e., a set of FOL formulae that might also be empty, a user can incrementally add new process behaviors, i.e., individual process executions, incorporated into the model by the approach. We categorize the initial model as *a priori* provided *explicit* domain knowledge and the incremental user selections as *in vivo explicit* domain knowledge. Compared to the other approaches, the process model discovered by WoMan also contains process information beyond the control flow, including details on the resources involved in executing process activities. WoMan also offers an *auto-complete option* by adding all behavior from an event log at once. Thus, if the user starts from the empty process model and adds the entire event log at once, WoMan functions like a conventional discovery approach.

Table 2.6.: 1<sup>st</sup> part of the overview of the identified approaches' characteristics regarding the distinguishing features (partly adapted from [184, Table 5])

Abbr.*	Degree of Interactivity	Domain Knowledge (Input)	
		Timing of Provision	Type
A1	automated	a priori	explicit (declarative) (parallel executed process activities)
A2	automated	a priori	explicit (declarative) (allowed DECLARE templates)
A3	automated	a priori	explicit (imperative) (augmented ICN)
A4	automated	a priori	explicit (declarative) (declarative constraints specifying the relationship between activities and specification of potential start and end activities)
A5	interactive/ incremental	(1) a priori & (2) a posteriori	(1) explicit (declarative) (DECLARE constraints) & (2) user feedback (selecting finally a process model from a set of result candidates)
A6	automated	a priori	explicit (declarative) (precedence constraints over process activities)
A7	interactive/ incremental (auto-complete option)	(1) a priori & (2) in vivo	(1) explicit (imperative) (initial process model) & (2) explicit (imperative) (traces incrementally selected by the user)
A8	interactive/ incremental (auto-complete option)	(1) a priori & (2) in vivo	(1) explicit (imperative) (initial process model) & (2) explicit (imperative) (traces incrementally selected by the user) & (2) user feedback (accepting/modifying proposed repair)
A9	interactive/ incremental	in vivo	explicit (imperative) (subgraphs contained in the log-tree) & user feedback (selection of detected subgraphs in the process model)
A10	interactive/ incremental (auto-complete option)	in vivo	user feedback (supported by suggestions from the algorithm, a user creates the process model in an editor)
A11	interactive/ incremental (auto-complete option)	in vivo	explicit (imperative) (merging, deletion, adding requests of activities from the user)
A12	interactive/ incremental (auto-complete option)	(1) a priori & (2) in vivo	(1) explicit (declarative) (initial process model, specified with FOL formulae) & (2) explicit (imperative) (traces incrementally selected by the user)

\* Consider Table 2.4 for an overview of approach abbreviations and their references.

Table 2.7.: 2<sup>nd</sup> part of the overview of the identified approaches' characteristics regarding the distinguishing features (partly adapted from [184, Table 5])

Abbr.*	Process Model (Output)			Application focus	Software realization
	Formalism	Formalism Restrictions	Guarantees		
A1	imperative (Petri nets)	no	no	process discovery	available (ProM plugin)
A2	declarative (DECLARE)	no	yes (event data fits the discovered DECLARE model)	process discovery	available (ProM plugin)
A3	imperative (ICN)	no	no (resulting model contains only statistically significant behavior)	process discovery	unknown/not (publicly) available
A4	imperative (DFG)	no	no	process discovery	available (ProM plugin)
A5	imperative (Petri nets)	yes (process trees)	no (resulting process models candidates are randomly generated)	process discovery	unknown/not (publicly) available
A6	imperative (C-nets)	no	yes (event data and the discovered process model satisfy the precedence constraints)	process discovery	available (ProM plugin)
A7	imperative (Petri nets)	no	yes (incrementally added traces fit the resulting model)	model repair	available (ProM plugin)
A8	imperative (BPMN)	no (only control flow perspective covered)	(yes) (incrementally added traces are accepted in the resulting model if the user follows the proposed changes)	model repair	available (stand-alone tool <i>Apromore</i> [117])
A9	imperative (log-tree)	no	yes (discovered model describes all behavior from the log)	process discovery	unknown/not (publicly) available
A10	imperative (Petri nets)	yes (free-choice & sound WF-nets)	yes (discovered model is always sound)	process discovery	available (ProM plugin)
A11	imperative (DFG)	no	no	process discovery	unknown/not (publicly) available
A12	declarative (FOL formulae)	yes (Data-log Horn clauses)	yes (incrementally added traces fit the resulting model)	process discovery	unknown/not (publicly) available

\* Consider Table 2.4 for an overview of the approach abbreviations and their references.

### 2.3.2. Analysis & Discussion

This section analyzes and discusses the presented approaches based on the distinguishing features, cf. Figure 2.2. Table 2.5 presents an overview of the distribution of the frequency of the various characteristics of the corresponding distinguishing features. Tables 2.6 and 2.7 provide a detailed overview of the twelve identified approaches and their characteristics for each distinguishing feature. Subsequently, we organize the discussion along the first-level distinguishing features; thus, we discuss key findings per distinguishing feature.

#### Degree of Interactivity

We observe a balanced number of automated and interactive/incremental approaches; five approaches are automated and seven approaches are classified interactive/incremental. Reconsider Figure 2.1 illustrating that automated approaches do not allow for in vivo provided domain knowledge while interactive/incremental approaches require in vivo provided domain knowledge. Moreover, five of seven interactive/incremental approaches include an auto-complete option. Such an option ensures that approaches that utilize domain knowledge in vivo can still discover a process model if the in vivo domain knowledge is absent, or the user stops providing in vivo domain knowledge at some point.

#### Timing of Domain Knowledge Provision

Comparing the timing of domain knowledge provision, we observe that 75% of the approaches assume domain knowledge being provided a priori, 50% assume domain knowledge in vivo, and only one approach assumes domain knowledge a posteriori. Recall that the characteristics of this distinguishing feature are not mutually exclusive, cf. Table 2.1; thus, an approach can assume domain knowledge being provided a priori, in vivo, as well as a posteriori. Moreover, recall that we exclude approaches solely utilizing domain knowledge provided a posteriori.

Of the twelve identified approaches, only seven approaches are interactive/incremental. Thus, seven approaches assume domain knowledge is being provided in vivo. In contrast, approaches purely utilizing domain knowledge provided a priori—thus, these approaches are automated—do not allow intervention during the discovery phase to correct respectively steer the discovery phase. The identified incremental/interactive approaches

#### Domain Knowledge Type

Eleven of the twelve approaches utilize explicit domain knowledge, while only four utilize user feedback. Focusing on the approaches utilizing explicit domain knowledge, we observe that imperative and declarative formalism to specify domain knowledge is nearly equally used, cf. Table 2.5. Most approaches using declarative explicit domain knowledge consider constraints, such as precedence, over the activities contained in a log, cf. A1, A4, and A6. The two model repair techniques identified, i.e., A7 and A8, both use an initial imperative process model as explicit domain knowledge and traces that are incrementally added as a second explicit domain knowledge. Similarly, approach A12 uses an initial model provided as FOL formulae, i.e., considered a declarative formalism, and

incrementally added traces as explicit domain knowledge. Interestingly, only A12 uses both domain knowledge specified in declarative and imperative formalisms.

In total, four approaches utilize user feedback. Approaches A8–10 propose changes to a process model during the discovery phase and leave the final decision to the user on whether to apply the proposed change to the model. Approach A5 discovers multiple process models and leaves the final decision which one to choose to the user.

### Output Process Model Formalism: Type, Restrictions & Guarantees

Examining the process model formalisms used to specify the discovered process model, we find that a wide variety of formalisms are employed. Only two approaches discover a declarative process model, while the others use imperative process model formalisms. However, many process model formalisms utilized are not typically seen in industrial process mining software; such uncommon formalisms include Petri nets, process trees, log-tree, C-nets, and ICNs.

Most process discovery approaches, i.e., four, use Petri nets as process model formalism. However, three approaches that use Petri nets focus on a subclass, process trees, or free-choice workflow nets. Process trees, representing block-structured and sound Workflow nets, are widely used in conventional process discovery approaches like Inductive Mining [121] and Evolutionary Tree Miner [39]. Process trees ensure favorable behavioral characteristics, such as being deadlock-free and sound by construction. However, it's important to note that process trees have limited expressiveness and cannot, for example, model long-term dependencies. Free-choice WF-nets are more expressive than process trees; in return, such nets are not sound by construction compared to process trees. Further, also free-choice WF-net We provide a more detailed introduction to different classes of Petri nets in Section 3.3.1.

Approaches A4 and A11 employ DFG as process model formalism. DFGs have the most limited expressiveness compared to all identified model formalisms. They cannot model control flow operators such as parallelism and choices, which are fundamental control flow patterns [221]. Moreover, using DFGs can lead to inaccurate diagnoses, as exemplified in [212], despite their frequent usage in industrial process mining applications due to the simplicity of both their interpretation and discovery.

In general, note that many process models represented in a particular formalism can be translated into other formalism. For instance, any process tree can be easily translated into BPMN models or Petri nets [121]. However, for example, not every Petri net can be transformed into a BPMN model and vice versa [125].

### Output guarantees

Comparing the output guarantees concerning the discovered process model, we find that seven approaches provide guarantees. In contrast, the other approaches do not provide guarantees or are unknown, respectively not evident from the corresponding publications. Fitness is one of the most common output guarantees, i.e., the discovered process model fully reflects process behavior recorded in the provided event data. In this regard, we distinguish automated and interactive/incremental approaches, cf. Figure 2.1. Automated approaches guarantee that the entire provided event log is fitting the discovered process

model (for instance, A2 and A6), while interactive/incremental approaches guarantee to fit the incremental added process behavior (for instance, A7, A8, A9, and A12).

Output guarantees are especially important in automated approaches, since users have no way to provide their domain knowledge *in vivo* to steer and correct the algorithm; thus, they are at the mercy of the algorithm. Further, data quality is paramount in automated approaches, especially for approaches guaranteeing that the discovered process model has perfect fitness regarding the provided log. In case the event log has quality issues, for example, incomplete captured process behavior and incorrect activity orderings due to incorrect timestamps, these issues will very likely be present in the process model if these quality issues are not resolved in the data preparation phase. Thus, the quality of the input is critical to the output model quality.

In addition to the guarantees for the discovered process model concerning the event log, guarantees may also concern the domain knowledge provided. Only approaches A2 and A6 also provide guarantees regarding the provided domain knowledge. Approach A2 allows the user *a priori* to restrict the process model to be discovered, i.e., users can specify DECLARE templates that should be incorporated in the process model. A2 guarantees that the learned model only consists of the allowed templates and that the learned constraints fit the process behavior in the provided event log. Approach A6 guarantees that the discovered process model fits the event log and the *a priori* provided domain knowledge, i.e., precedence constraints among activities. Therefore, no process model is returned if the *a priori* domain knowledge and the event log conflict. This observation leads to an interesting algorithmic challenge, i.e., how to combine conflicting domain knowledge with event data. Similarly, as automated approaches, interactive/incremental approaches benefit from output guarantees. Imagine that an interactive approach offers no guarantees; for example, the algorithm could ignore domain knowledge supplied *in vivo*. Therefore, the user would no longer have guaranteed control over the process discovery algorithm.

## Application Focus

As elaborated in Section 2.1.2, model repair techniques can be utilized to discover process models in an incremental fashion—although these techniques were not designed for such purpose. In total, we identified two approaches primarily intended for model repair, i.e., A7 and A8, while all other approaches are intended for process discovery. Recall that we do not provide a complete review of model repair techniques as elaborated in Section 2.2.

## Software Realization

Mots approaches, i.e., six of twelve, have been implemented as a plugin within the process mining software ProM [234]. ProM is an open-source process mining software that provides a plugin infrastructure for process mining algorithms and approaches. A large number of plugins have been developed for ProM making it one of the most common software solutions within process mining research. Approach A8 has been implemented in Apromore [117], a commercial process mining solution. For the other approaches, no (publicly) available software implementation could be found. All available software implementations come with a graphical user interface.

## 2.4. Challenges & Opportunities

From the identified approaches, this section derives research challenges and opportunities for the area of non-conventional process discovery. In total, we present ten research challenges.

### 2.4.1. Challenge 1—Blending Explicit Domain Knowledge & User Feedback

As the review has shown (cf. Tables 2.6 and 2.7), most approaches utilize explicit domain knowledge and only a few approaches truly incorporate user feedback. Nevertheless, individual evaluations of the identified approaches indicate that both types of domain knowledge are advantageous. Therefore, incorporating both forms of domain knowledge would offer users more flexibility and options. Nevertheless, the incorporation of more and different domain knowledge also poses challenges on fusing these different sources of information, as explained in a later challenge, cf. *Challenge 8—Event Data & Domain Knowledge Fusion*.

### 2.4.2. Challenge 2—Advanced User Interaction

The identified approaches that leverage user feedback often merely delegate a specific task to users. For example, Approach A5 suggests a repair to the user, who can either accept or repair it manually. Similarly, approach A10 recommends where to place a transition in a WF-net; the user can follow the suggestion or manually place the transition. These examples highlight that user interaction usually centers on a single task. In addition, user interaction often consists only of a one-way question-answer pair; a dialogue between the discovery algorithm and the user often does not occur. Therefore, we derive the challenge of enhancing user feedback beyond a single task by gathering feedback on multiple aspects.

### 2.4.3. Challenge 3—Various Modes of Interactivity

The twelve identified approaches can be clearly categorized into automated and incremental/interactive approaches, cf. Table 2.6. Five of seven interactive/incremental approaches offer an auto-complete option allowing users to skip providing in vivo domain knowledge and automatically discover a process model. An interesting direction for future work is to provide different levels of interactivity that users can freely switch between as needed. For example, a user may run an interactive/incremental discovery approach in an automated mode where only some intermediate process models are displayed to the user. However, as soon as the user notices that an intermediate process model is developing in an undesirable direction, for example, activities become optional that should not be optional, the user intervenes and switches to a more interactive mode to steer or influence the algorithm and thus the process discovery phase by providing in vivo domain knowledge. After the observed issue in the intermediate process model is solved, the user can switch back to a less interactive mode of the discovery approach and resume the supervisory position.

#### 2.4.4. Challenge 4—Scalable Conformance Checking

Conformance checking techniques are a central component of several approaches, for instance, approach A7. Moreover, in incremental/interactive process discovery approaches, it is essential to compare intermediate process models with the provided event data to make informed decisions on how to continue. However, state-of-the-art conformance checking techniques like *alignments* [6, 226] suffer from the state-space explosion problem [45]. Thus, calculating alignments may be an exponential problem depending on the complexity of the process model and the process behavior. Therefore, it is crucial for interactive/incremental approaches to have fast, reliable, and interpretable conformance checking results.

#### 2.4.5. Challenge 5—Minimizing Representational Bias

Each process discovery approach discovers models in a specific formalism. The discover approach might further restrict the model class; for instance, approach A10 discovers free-choice WF-nets, i.e., a subclass of Petri nets. This upfront predetermination on a model class is referred to as *representational bias* since the discovery approach assumes that the process to be discovered can be represented in the given formalism.

For instance, approaches A4 and A11 discover DFGs. Although DFGs are widely used in commercial process mining solutions and easy to interpret, they lack expressiveness compared to, for example, Petri nets or BPMN. DFGs cannot explicitly model central control-flow constructs such as concurrency and choices. Further, DFGs can lead to wrong conclusion to their simplicity [212].

#### 2.4.6. Challenge 6—Event Data & Process Model Visualizations

The identified approaches use various different process model formalisms, cf. Table 2.7. As elaborated before, many model formalisms used are hardly used in organizations. Also the authors in [141] found that ‘incomprehensible outcomes’, i.e., non-standard visualizations of process models lead to challenges when applying process mining in organizations. Likewise, the authors in [225] list the improvement of output representations for non-process mining experts as a general challenge. To this end, it is important to visualize discovered process models and especially intermediate process models in an interactive/incremental process discovery setting in well-known formalisms. Note that the formalism used to visualize a process model and the formalism used during process discovery internally within a algorithm may differ. For instance, an automated process discovery algorithm might learn a process tree that is, however, visualized as a BPMN model to users. Similarly, as many approaches require user interaction and user decisions regarding the event log, adequate visualizations of process behavior in an event log are paramount to facilitate users’ decision-making.

#### 2.4.7. Challenge 7—Domain Knowledge Specification

Most of the identified approaches use explicit domain knowledge, for instance, control-flow constraints among activities. Since explicit domain knowledge is specified in a particular



formalism, it is clear to support users in this specification task. The identified approaches, however, presume that explicit domain knowledge is specified in the required formalism and ready for use. Nevertheless, it is equally important to consider where this explicit domain knowledge comes from and how tools, techniques, and approaches can assist users in specifying explicit domain knowledge. Furthermore, questions like what happens when conflicting information exists must also be addressed since domain knowledge may originate from various stakeholders.

#### 2.4.8. Challenge 8—Event Data & Domain Knowledge Fusion

Many approaches utilize explicit domain knowledge that specifies relations among activities, cf. Section 2.3. Moreover, also the recorded process behavior in the provided event log indirectly specifies relations among activities. As long as these relationships are not contradictory, both sources of information can be used complimentary within the process discovery. However, in case of conflicting information, a decision must be made.

For instance, imagine the situation in which the user provides a priori the precedence constraint that activity  $a_1$  is always executed before  $a_2$ . Further, assume that the provided event log contains process executions in which  $a_2$  is executed before  $a_1$ . Obviously, the provided domain knowledge contradicts the observations recorded in the event log. This contradiction leads to the question: “Which source to prioritize?” Several options may exist for how a non-conventional process discovery approach could resolve the conflict. The most naive option could always prioritize the event log or the domain knowledge when conflicts arise. Another option could be to examine how often the domain knowledge, i.e., a precedence constraint, occurs in the event log and how often it conflicts. Next, statistics could be calculated, and only significant domain knowledge would be considered. For the example, we would check how often  $a_1$  is executed before  $a_2$  and vice versa. If only a few examples contradict the precedence constraint, we favor the domain knowledge, i.e., the precedence constraint, over the observations in the event log. Alternatively, the algorithm interactively presents them to the user upon detection, who has to decide how to proceed in conflicting situations.

The above example illustrates the ample solution space for potential process discovery approaches using domain knowledge and the importance of carefully designing these approaches. From the observations in this review, disagreements between domain knowledge and event data are often not or only partially considered. Therefore, we highlight this issue, *domain knowledge fusion with event data* [80], as an important challenge.

#### 2.4.9. Challenge 9—Software Support

Having adequate software support is central for process discovery approaches utilizing domain knowledge. In cases where the approach utilizes user feedback or explicit domain knowledge in vivo, a sophisticated implementation is required as users need to interact with the algorithm during the actual process discovery phase. Since many approaches require users to make decisions during the process discovery phase, adequate representation of all information needed to decide is paramount. For instance, in incremental process discovery (cf. A7, A8, and A12), users decide on process behavior to be added

to a process model. This decision requires that users are equipped with adequate representations of the vast amount of observed process executions in the event log to make informed decisions; consider *Challenge 6—Event Data & Process Model Visualizations*. Further, as elaborated in *Challenge 7—Domain Knowledge Specification*, the specification of explicit domain knowledge is currently often ignored but also requires decent software support. Moreover, the authors in [141] found that insufficient analytical skills are a common challenge when process mining is applied in organizations. Although the software design can only partially address this problem, the software realization is critical in how easily users can use an approach. In short, software support comprises more than a basic implementation of the approach but also requires the adequate representation of the user's information needs to facilitate design making and includes support in specifying explicit domain knowledge.

#### 2.4.10. Challenge 10—Discovery Beyond Control-Flow

Most identified approaches focus on discovering process models that reflect the control flow perspective of processes. Moreover, timing information of activities are often ignored by process discovery approaches; instead, timestamp information is only used to deduce the order relationships between activities. As briefly elaborated in Section 1.2, process model formalisms like BPMN allow to model further aspects of a process beyond the control flow perspective. Especially in non-conventional process discovery, it is reasonable to discover from the provided inputs further perspectives such as organizational structures including resource allocations and process rules, such as, maximal waiting times between two activities.

### 2.5. Conclusion

This section concludes the presented literature review. Twelve non-conventional process discovery approaches<sup>9</sup> were identified and systematically compared along the proposed distinguishing features. Compared to the number of algorithms and approaches in conventional process discovery [15, 60, 211, 235], the field of non-conventional process discovery is still relatively small. Further, the various challenges and opportunities indicate the need for further development. To this end, this thesis's incremental process discovery approach contributes to this area by providing a novel approach to interactive/incremental non-conventional process discovery.

---

<sup>9</sup>For the sake of simplicity, we also refer to the identified process model repair approaches as non-conventional process discovery.

# Chapter 3.

## Preliminaries

This chapter introduces concepts, notations, and definitions used throughout this thesis. First, basic mathematical concepts are presented in Section 3.1. Next, Section 3.2 introduces event data and defines corresponding concepts such as events, cases, event logs, and traces. Section 3.3 introduces process model formalisms. Finally, Section 3.4 introduces conformance checking, and Section 3.5 introduces alignments, i.e., a state-of-the-art conformance checking technique.

### 3.1. Basic Mathematical Concepts

This section introduces sets and multisets in Section 3.1.1. Section 3.1.2 introduces functions. Multisets are introduced in Section 3.1.3. Further, Section 3.1.4 introduces ordered sets. Sequences, corresponding notations, and operators are introduced in Section 3.1.5. Finally, we introduce trees in Section 3.1.6.

#### 3.1.1. Sets & Relations

We denote the natural numbers by  $\mathbb{N} = \{1, 2, 3, \dots\}$ ;  $\mathbb{N}_0$  denotes the natural numbers including 0, i.e.,  $\mathbb{N}_0 = \{0, 1, 2, 3, \dots\}$ . Analogously, we denote the real numbers by  $\mathbb{R}$  and real numbers that are greater or equal zero by  $\mathbb{R}_{\geq 0}$ . The Boolean values are denoted by  $\mathbb{B} = \{true, false\}$ . We denote the empty set by  $\emptyset$ . We write  $X \subseteq Y$  to denote that  $X$  is a *subset* of  $Y$ , and we write  $X \subset Y$  to denote that  $X$  is a *strict subset* of  $Y$ , i.e.,  $X \subset Y \Leftrightarrow X \subseteq Y \wedge X \neq Y$ . We denote the power set of an arbitrary set  $X$  as  $\mathbb{P}(X) = \{X' \subseteq X\}$ . For instance, let  $X = \{x_1, x_2, x_3\}$ , the power set  $\mathbb{P}(X) = \{\emptyset, \{x_1\}, \{x_2\}, \{x_3\}, \{x_1, x_2\}, \{x_1, x_3\}, \{x_2, x_3\}, \{x_1, x_2, x_3\}\}$ .

Let  $X_1, \dots, X_n$  be arbitrary sets (for  $n \geq 2$ ). We define the Cartesian product of these  $n$  sets as  $X_1 \times \dots \times X_n = \{(x_1, \dots, x_n) \mid x_1 \in X_1, \dots, x_n \in X_n\}$ . For instance, let  $X = \{a, b, c\}$  and  $Y = \{d, e, f\}$ . The Cartesian product  $X \times Y$  contains the following tuples  $\{(a, d), (a, e), \dots, (c, e), (c, f)\} = X \times Y$ . Given a Cartesian product of  $n$  arbitrary sets, i.e.,  $X_1, \dots, X_n$ , we refer to a set  $R \subseteq X_1, \dots, X_n$  containing  $n$ -tuples as *relation*. If  $n = 2$ , we refer to  $R \subseteq X_1 \times X_2$  as a *binary relation*.

Further, we define projection functions that extract a specific element from a tuple. Let  $(x_1, \dots, x_n) \in X_1 \times \dots \times X_n$  for  $n \in \mathbb{N}$  be an arbitrary  $n$ -tuple. We define the projection functions  $\pi_i : X_1 \times \dots \times X_n \rightarrow X_i$  for all  $1 \leq i \leq n$  that extract the  $i$ -th element from the tuple. For instance,  $\pi_3((a, b, c, b, e, d)) = c$ .

### 3.1.2. Functions

In this section, functions and corresponding properties and classifications of functions are introduced. Let  $X, Y$  be sets and  $f : X \rightarrow Y$  be a *function* that assigns to each element in  $X$  *at most one* element from  $Y$ . Thus, function  $f$  represents a binary relation  $R_f \subseteq X \times Y$  such that  $\forall x \in X \left( \exists y \in Y ((x, y) \in R_f) \Rightarrow \nexists y' \in Y (y' \neq y \wedge (x, y') \in R_f) \right)$ . Set  $X$  is referred to as the *domain* of  $f$ . We write  $\text{dom}(f) = X$ . Analogously, we denote the codomain of  $f$  as  $\text{codom}(f) = Y$ . Further, let  $X'$  be a subset of  $X$ , i.e.,  $X' \subseteq X$ . We denote the *restriction* of  $f$ 's domain to  $X'$  as  $f|_{X'} : X' \rightarrow Y$  with  $f|_{X'}(x) = f(x)$  for  $x \in X'$ .

A *partial function* from  $X$  to  $Y$ , denoted as  $f : X \rightharpoonup Y$ , is a function from  $X' \subseteq X$  to  $Y$ . Thus, a partial function  $f$  is only defined on a subset  $X' \subseteq X$ .

Given a function  $f : X \rightarrow Y$ , we call  $f$  a *bijective-function* if the following properties are satisfied.

1. Function  $f$  is surjective; thus, each element from  $Y$  is assigned at least one element from  $X$ , i.e.,  $\forall y \in Y \left( \exists x \in X (y = f(x)) \right)$
2. Function  $f$  is injective; thus, no two different elements from  $X$  are assigned the same element from  $Y$ , i.e.,  $\forall x_1, x_2 \in X ((x_1 \neq x_2) \Rightarrow f(x_1) \neq f(x_2))$

### 3.1.3. Multisets

Multi sets generalize the notion of sets by allowing the occurrence of an element multiple times. For instance, assume the set  $X = \{a, b, c\}$ . The multiset  $M$  over the set  $X$  with  $M = [a, c^3]$  contains once element  $a$ , three times  $c$ , and no  $b$ . Note that we omit the superscript if an element is contained only once in a multiset. We denote the empty multiset as  $[\ ]$ . We formally define multisets as follows.

**Definition 3.1** (Multiset)

Let  $X$  be an arbitrary set. A multiset  $M$  is a function assigning every element from  $X$  a cardinality, i.e.,  $M : X \rightarrow \mathbb{N}_0$ . We denote the universe of multisets over  $X$  as  $\mathcal{M}(X)$ .

Given a multiset  $M$ , we write  $x \in M$  if element  $x$  is contained at least once in  $M$ ; for example,  $a, c \in [a, c^3]$ . Further, we refer to  $x$ 's *cardinality* in  $M$  as  $M(x) \in \mathbb{N}_0$ . Reconsider the example above;  $M(c) = 3$  and  $M(b) = 0$ . We denote the *union of two multisets*  $M_1, M_2 \in \mathcal{M}(X)$  by  $M_1 \uplus M_2$ . For instance,  $[a^2, b] \uplus [a^4, b, c] = [a^6, b^2, c]$ .

Note that any multiset over a set  $X'$  can be trivially extended to be a multiset over a set  $X$  that is a superset of  $X'$ , i.e.,  $X' \subset X$ , by assigning all  $x \in X \setminus X'$  to 0. Further, any set  $X$  can be easily transformed into a multiset  $M \in \mathcal{M}(X)$  by assigning each element from  $X$  the cardinality 1. For instance, the set  $\{a, b, c\}$  is represented by the equivalent multiset  $[a, b, c]$ .

Given a multiset  $M \in \mathcal{M}(X)$ , we denote its *unique elements* by  $\overline{M}$  as defined below.

$$\overline{M} = \{x \in M\} \subseteq X$$

For instance, for multiset  $M = [a^6, b^2, c, d^{11}]$ , the set  $\overline{M} = \{a, b, c, d\}$ .

Finally, given  $n$  sets containing multisets, i.e.,  $B_1, \dots, B_n \subseteq \mathcal{M}(X)$ , we define the Cartesian product over  $B_1$  until  $B_n$  by  $B_1 \times \dots \times B_n = \{b_1 \uplus \dots \uplus b_n \mid b_1 \in B_1 \wedge \dots \wedge b_n \in B_n\}$ . For instance,  $\{[a^2, b], [c]\} \times \{[d^3]\} = \{[a^2, b, d^3], [c, d^3]\}$ .

### 3.1.4. Ordered Sets

This section introduces orders over elements of a given set. We generally distinguish partial and total orders that define ordering relations over a set of elements. A total order makes any two elements from a set comparable, while a partial order may not make any two elements comparable, i.e., there might be incomparable elements.

Figure 3.1 shows an example of a strict partial order  $\prec$  over the set  $X = \{a, b, c, \dots, j\}$  visualized as a graph. Vertices represent elements of the set  $X$  and arcs indicate relations among elements according to  $\prec$ . For instance,  $a \prec b$ ,  $a \prec c$ , and  $a \prec j$ ; however,  $a \not\prec a$ ,  $c \not\prec d$ , and  $f \not\prec i$ . Note that dotted arcs represent transitive relations. As (strict) total orders are a refinement of partial orders, we introduce partial orders first.

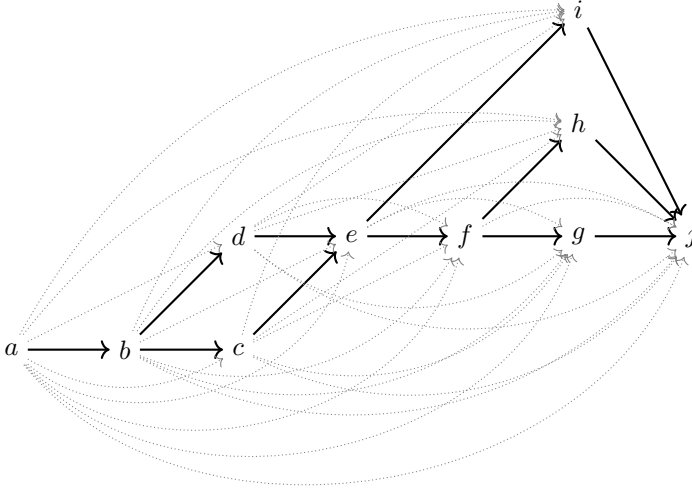


Figure 3.1: Graph representation of a strict partial order  $\prec$  over the set  $X = \{a, b, c, \dots, j\}$ ; vertices represent the elements of  $X$ , arcs represent the relations among elements (for example,  $a \prec b$ ,  $b \prec c$ ,  $a \prec c$ ,  $b \prec d$ ), and dotted arcs represent transitive relations

**Definition 3.2** ((Strict) Partial Order)

Let  $X$  be an arbitrary set. A partial order over  $X$  is a binary relation, i.e.,  $\preceq \subseteq X \times X$ , that satisfies the following conditions for all  $a, b, c \in X$ .

1.  $a \preceq a$  (reflexive)
2. If  $a \preceq b$  and  $b \preceq a$ , then  $a = b$  (antisymmetric)
3. If  $a \preceq b$  and  $b \preceq c$ , then  $a \preceq c$  (transitive)

A strict partial order, denoted as  $\prec \subseteq X \times X$ , is defined as follows.

1.  $a \not\prec a$  (irreflexive)
2. If  $a \prec b$ , then  $b \not\prec a$  (asymmetric)
3. If  $a \prec b$  and  $b \prec c$ , then  $a \prec c$  (transitive)

Reconsider the strict partial order depicted in Figure 3.1. Next, we define (strict) total orders that further restrict (strict) partial orders (cf. Definition 3.2). In a total order, any two elements are related to one-another. For instance, the  $\leq$  operator defines a total order on the set of natural numbers  $\mathbb{N}$ . Any two elements are related to each other;  $x \leq y$  or  $y \leq x$  hold for arbitrary  $x, y \in \mathbb{N}$ . Similarly,  $<$  defines a strict total order on  $\mathbb{N}$ . Next, we define (strict) total orders.

**Definition 3.3** ((Strict) Total Order)

A total order over  $X$  is a binary relation, i.e.,  $\leq \subseteq X \times X$ , that satisfies the following conditions for all  $a, b \in X$ .

1.  $\leq$  is a partial order (cf. Definition 3.2)
2.  $a \leq b$  or  $b \leq a$  (strongly connected)

A strict total order, denoted as  $< \subseteq X \times X$ , is defined as follows.

1.  $<$  is a strict partial order (cf. Definition 3.2)
2. If  $a \neq b$ , then either  $a < b$  or  $b < a$  (connected)

In the remainder of this thesis, we simply refer to an *order*, if we define/discuss concepts that are generally applicable for (strict) total or (strict) partial orders. Assume a set  $X$  and an order  $\triangleleft \subseteq X \times X$ . We refer to  $X$  as an ordered set, written as  $(X, \triangleleft)$ .

Next, we define the transitive reduction of an ordered set. Consider Figure 3.1. All solid arcs together represent the *transitive reduction*, while all solid and dashed arcs together represent the *transitive closure*.<sup>1</sup> Thus, the transitive reduction excludes all relations that emerge from the transitivity property of orders, cf. Definition 3.2 and Definition 3.3. Subsequently, we define the transitive reduction.

<sup>1</sup>Note that the transitive reduction of a finite, directed, acyclic graph is unique. In contrast, for directed graphs with cycles, the transitive reduction might not be unique [9]. However, the ordered sets considered in this thesis can all be represented by an acyclic, directed graph; thus, their transitive reduction is unique.

**Definition 3.4** (Transitive Reduction)

Let  $(X, \triangleleft)$  be an ordered set. Further, we assume that the order is acyclic, i.e., the order can be represented as an acyclic, ordered graph. The transitive reduction of  $\triangleleft$ , denoted by  $\triangleleft^R$ , is defined as  $\triangleleft^R = \{(a, b) \mid a \triangleleft b \wedge \nexists x \in X(a \triangleleft x \wedge x \triangleleft b)\}$ .

Reconsider the ordered set  $(X, \prec)$  in Figure 3.1. Let  $X' = \{b, c, d\} \subset X$  be a subset. We define the restricted partial order  $\prec|_{X'} \in X' \times X'$  that orders the elements in  $X' \subset X$  identical to  $\prec \in X \times X$ . Hence,  $b \prec|_{X'} c$  and  $b \prec|_{X'} d$ . In general, order restrictions allow us to generate from an ordered set any ordered subset such that elements from the subset are identically ordered as in the superset.

**Definition 3.5** (Order Restriction)

Let  $(X, \triangleleft)$  be an ordered set, and let  $X' \subset X$ . We define the order restriction of  $\triangleleft$  to set  $X'$  as  $\triangleleft|_{X'} \in X' \times X'$  with  $(x_1 \triangleleft|_{X'} x_2) \Leftrightarrow (x_1 \triangleleft x_2)$  for all  $x_1, x_2 \in X'$ .

Next, we define labeled orders. The difference to orders respectively ordered sets as introduced so far is that the elements of the set over which the order is defined are labeled. For instance, consider the set  $X$  in the left part of Figure 3.2. The elements of  $X$  are assigned labels, for example, element  $x_1$  is labeled  $a$  and  $x_2$  is labeled  $d$ .

**Definition 3.6** (Labeled Ordered Set)

Let  $(X, \triangleleft)$  be a ordered set,  $\Sigma$  be a set of labels, and  $\lambda : X \rightarrow \Sigma$  be a label function. We refer to  $(X, \triangleleft, \Sigma, \lambda)$  as a labeled ordered set.

Finally, we introduce *isomorphism for labeled ordered sets*. For instance, consider the two labeled ordered sets  $X$  and  $Y$  depicted in Figure 3.2. We call  $X$  and  $Y$  isomorphic because we can map each element from  $X$  to an element in  $Y$  such that the mapping preserves the label and the ordering relations. For example,  $x_5 \in X$  is mapped to  $y_1 \in Y$  because both have the same label  $a$  and neither element is related to any other element of the respective set. Below, we define isomorphism of labeled ordered sets.

**Definition 3.7** (Isomorphism of Labeled Ordered Sets)

Let  $(X, \triangleleft^X, \Sigma^X, \lambda^X)$  and  $(Y, \triangleleft^Y, \Sigma^Y, \lambda^Y)$  be two arbitrary labeled ordered sets. The labeled ordered set  $(X, \triangleleft^X, \Sigma^X, \lambda^X)$  is isomorphic to  $(Y, \triangleleft^Y, \Sigma^Y, \lambda^Y)$ , denoted as

$$(X, \triangleleft^X, \Sigma^X, \lambda^X) \cong (Y, \triangleleft^Y, \Sigma^Y, \lambda^Y),$$

iff a bijective function  $f : X \rightarrow Y$  exists such that:

1.  $\forall x \in X \left( \lambda^X(x) = \lambda^Y(f(x)) \right)$  and
2.  $\forall x_1, x_2 \in X \left( x_1 \triangleleft^X x_2 \Leftrightarrow f(x_1) \triangleleft^Y f(x_2) \right)$ .

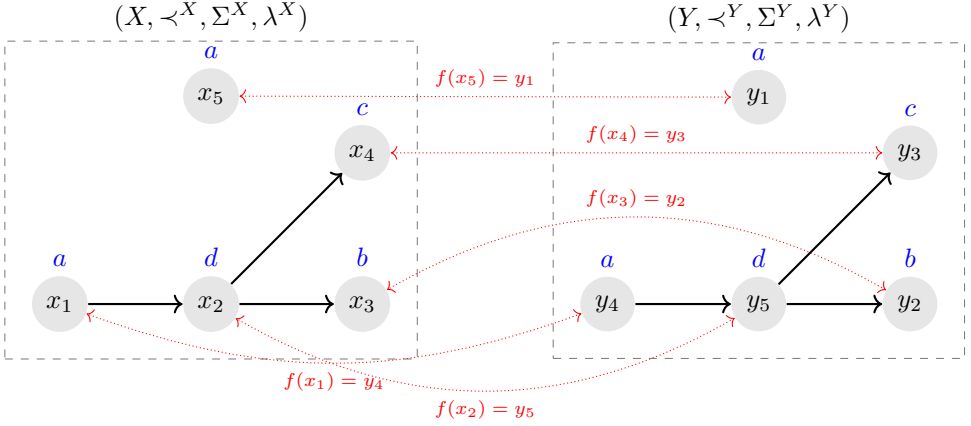


Figure 3.2: Two isomorphic labeled ordered sets  $(X, \prec^X, \Sigma^X, \lambda^X)$  and  $(Y, \prec^Y, \Sigma^Y, \lambda^Y)$ ; for simplicity and readability, black arcs indicate the transitive closures and red dotted arcs correspond to the isomorphic mapping, i.e., a bijective function  $f : X \rightarrow Y$

### 3.1.5. Sequences

Assume the set  $X = \{a, b, c, d\}$ . A sequence over  $X$  is an enumerated collection consisting of elements from  $X$ . For instance,  $\sigma = \langle d, a, a, b, d \rangle$  is a sequence with length five over  $X$ . Below, we define sequences.

**Definition 3.8** (Sequence)

Let  $X$  be an arbitrary set. A sequence  $\sigma$  of length  $n \in \mathbb{N}$  over set  $X$  is a function assigning each index an element from  $X$ , i.e.,  $\sigma : \{1, \dots, n\} \rightarrow X$ .<sup>a</sup>

<sup>a</sup>Note that we can easily construct a labeled ordered set  $(\{1, \dots, n\}, <)$  that adheres to the ordering defined by  $\sigma$ . Elements from  $\{1, \dots, n\}$  are assigned a label from  $X$  according to  $\sigma$ . The strict total order  $<$  orders the elements such that  $1 < \dots < n$ .

For an arbitrary set  $X$ , we denote the set of all sequences over an arbitrary set  $X$  as  $X^*$ . We denote the length of a sequence  $\sigma$  as  $|\sigma| \in \mathbb{N}_0$ . The empty sequence is denoted as  $\langle \rangle$ . For sequence  $\sigma \in X^*$  with length  $n = |\sigma|$ , we write  $\sigma$  as  $\langle \sigma(1), \dots, \sigma(n) \rangle$ . Thus, for a sequence  $\sigma$  with length  $n$  and  $i \in \{1, \dots, n\}$ ,  $\sigma(i)$  denotes the  $i$ -th element of  $\sigma$ . We overload the notation of element inclusion for sequences; for a sequence  $\sigma \in X^*$  and  $x \in X$ , we write  $x \in \sigma$  if  $\exists 1 \leq i \leq |\sigma| \ (\sigma(i) = x)$ .

Given two sequences  $\sigma_1, \sigma_2 \in X^*$ , we refer to their *concatenation* as  $\sigma_1 \circ \sigma_2 \in X^*$ . For instance,  $\langle a, a \rangle \circ \langle d, a \rangle = \langle a, a, d, a \rangle$ . The concatenation operator is trivially extended to sets of sequences. Therefore, we overload the concatenation operator  $\circ$ . Let  $S_1, S_2 \subseteq X^*$  be two sets of sequences, their concatenation is defined as  $S_1 \circ S_2 = \{\sigma_1 \circ \sigma_2 \mid \sigma_1 \in S_1 \wedge \sigma_2 \in S_2\}$ . For example, let  $S_1 = \{\langle d, d \rangle\}$  and  $S_2 = \{\langle a, b \rangle, \langle b, d, d \rangle\}$ ; their concatenation  $S_1 \circ S_2 = \{\langle d, d, a, b \rangle, \langle d, d, b, d, d \rangle\}$ .



For two sequences  $\sigma_1, \sigma_2 \in X^*$ , we refer to all interleavings of the two sequences as  $\sigma_1 \diamond \sigma_2 \subseteq X^*$ . Note that all interleaved sequences have length  $n + m$ , i.e.,  $\forall \sigma \in \sigma_1 \diamond \sigma_2 (|\sigma| = n + m)$ . Let  $n = |\sigma_1|$  and  $m = |\sigma_2|$ , the set of interleaved sequences

$$\sigma_1 \diamond \sigma_2 = \left\{ \sigma \mid \sigma \in X^* \wedge \exists i_1, \dots, i_n, j_1, \dots, j_m \in \{1, \dots, n + m\} \left[ \begin{aligned} &i_1 < \dots < i_n \wedge j_1 < \dots < j_m \wedge \{i_1, \dots, i_n, j_1, \dots, j_m\} = \{1, \dots, n + m\} \wedge \\ &\forall 1 \leq k \leq n (\sigma(i_k) = \sigma_1(k)) \wedge \forall 1 \leq k \leq m (\sigma(j_k) = \sigma_2(k)) \right] \right\}.$$

For example, let  $\sigma_1 = \langle b, a \rangle$  and  $\sigma_2 = \langle d \rangle$ . The set of interleaved sequences  $\sigma_1 \diamond \sigma_2 = \{\langle d, b, a \rangle, \langle b, d, a \rangle, \langle b, a, d \rangle\}$ . We extend the interleaving operator  $\diamond$  to sets of sequences. Let  $S_1, S_2 \subseteq X^*$ , we define  $S_1 \diamond S_2 = \bigcup_{\sigma_1 \in S_1, \sigma_2 \in S_2} (\sigma_1 \diamond \sigma_2)$ .

Given a sequence  $\sigma \in X^*$  and a subset  $X' \subseteq X$ , we introduce the projection function  $\downarrow$  that removes all elements from  $\sigma$  that are in  $X \setminus X'$ . For instance, let  $X = \{a, b, c, d, e, f\}$ ,  $X' = \{b, e, f\} \subset X$ , and  $\sigma = \langle b, b, c, f, d, e, f \rangle \in X^*$ . Applying the projection function to  $\sigma$ , denoted as  $\sigma \downarrow_{X'}$ , results in  $\sigma \downarrow_{X'} = \langle b, b, f, e, f \rangle \in (X')^*$ .

### Definition 3.9 (Sequence Projection Function)

Let  $X, X'$  be arbitrary sets with  $X' \subset X$  and  $\sigma \in X$ . The projection function  $\downarrow_{X'}: X^* \rightarrow (X')^*$  is recursively defined. We write  $\sigma \downarrow_{X'}$  instead of  $\downarrow_{X'}(\sigma)$ . For  $\sigma \in X^*$ :

$$\sigma \downarrow_{X'} = \begin{cases} \langle \rangle & \text{if } \sigma = \langle \rangle \\ \langle x \rangle \circ \bar{\sigma} \downarrow_{X'} & \text{if } \sigma = \langle x \rangle \circ \bar{\sigma} \text{ with } x \in X' \\ \bar{\sigma} \downarrow_{X'} & \text{if } \sigma = \langle x \rangle \circ \bar{\sigma} \text{ with } x \notin X' \end{cases}$$

We define for sequences  $\sigma$  containing  $n$ -tuples projection functions  $\pi_i^*$  that extract a sequence containing the  $i$ -th element of each tuple. Let

$$\sigma = \langle (x_1^1, \dots, x_n^1), \dots, (x_1^m, \dots, x_n^m) \rangle \in (X_1 \times \dots \times X_n)^*$$

be a sequence of length  $m$  containing  $n$ -tuples. For all  $1 \leq i \leq n$ , we define the projection function  $\pi_i^*: (X_1 \times \dots \times X_n)^* \rightarrow X_i^*$  with  $\pi_i^*(\sigma) = \langle x_i^1, \dots, x_i^m \rangle$ . For example,  $\pi_2^*(\langle (b, d), (a, c), (d, d) \rangle) = \langle d, c, d \rangle$ .

### 3.1.6. Graphs & Trees

This section introduces graphs and trees. A *graph* consists of vertices and (un)directed edges connecting vertices. For example, Figure 3.1 shows a directed graph  $G = (V, E)$  with vertices  $V = \{a, \dots, j\}$  and directed edges  $E = \{(a, b), \dots, (i, j)\} \subseteq E \times E$ . Below, we define directed graphs.

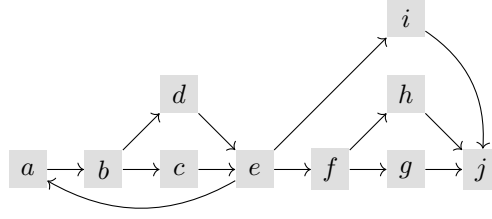


Figure 3.3: Exemplary directed graph  $G = (V, E)$  with vertices  $V = \{a, \dots, j\}$  and directed edges  $E = \{(a, b), \dots, (i, j)\}$

**Definition 3.10** (Directed Graph)

Let  $V$  be a set of vertices, and  $E \subseteq V \times V$  be a set of directed edges. We call  $G = (V, E)$  a directed graph.

A *directed path* in a directed graph is a sequence of distinct edges—a path contains an edge at most once—such that the end vertex of each edge in the sequence is identical to the start vertex of the subsequent edge in the path. Consider the directed graph depicted in Figure 3.3. For instance, the directed path  $\sigma_1 = \langle (a, b), (b, c), (c, e) \rangle$  leads from vertex  $a$  to vertex  $e$ . Similarly, an *undirected path* in a directed tree is a sequence of distinct edges such that consecutive edges always have a vertex in common. Thus, we ignore the orientation of the directed edges and assume any edge exists in both directions. For instance, the undirected path  $\sigma_2 = \langle (f, h), (f, g), (g, j), (i, j) \rangle$  is an undirected path leading from vertex  $h$  to  $i$  in the directed graph depicted in Figure 3.3. For an undirected path, we refer to the vertex that is part of the first edge of the path but not part of the second edge as the start vertex. Reconsider the undirected path  $\sigma_2$ . Vertex  $h$  is the start vertex as  $h$  is not part of the second edge in  $\sigma_2$ . Similarly, we refer to the vertex of the last edge not present in the previous edge as the end vertex.

An (un)directed path forms a cycle if the start vertex of the path's first edge equals the end vertex of the path's last edge. The directed graph depicted in Figure 3.3 contains a cycle, for instance, the path  $\sigma_3 = \langle (a, b), (b, c), (c, e), (e, a) \rangle$  forms a cycle starting/ending at vertex  $a$ . We call directed graphs without cycles *acyclic*.

Further, we call a directed graph *weakly connected* if there exists for any two vertices an undirected path connecting these two vertices. Analogously, we call a directed graph *strongly connected* if there exists for any two vertices a directed path connecting these vertices. The graph depicted in Figure 3.3 is weakly connected but *not* strongly connected, for instance, there exists no directed path from vertex  $v_6$  to  $v_1$ .

Next, we introduce *labeled, ordered, rooted trees* that represent a subclass of directed graphs. Said trees are directed graphs that are *acyclic* and *weakly-connected*. Figure 3.4 depicts an example tree  $\Lambda_1$  that consists of eleven labeled vertices  $V = \{v_1, \dots, v_{11}\}$  that are totally ordered with  $v_1 < v_2 < \dots < v_{11}$ , ten directed edges  $E = \{(v_1, v_2), (v_1, v_3), (v_3, v_4), (v_3, v_5), (v_1, v_6), (v_1, v_7), (v_7, v_8), (v_7, v_9), (v_9, v_{10}), (v_9, v_{11})\}$ , labels  $\Sigma = \{A, R, Q, Z, D, I, O, K, T\}$ , labeling function  $\lambda : V \rightarrow L$  with  $\lambda(v_1) = A, \dots, \lambda(v_{11}) = D$ , and the unique root vertex  $v_1$ . When considering the directed edges as undirected edges, between

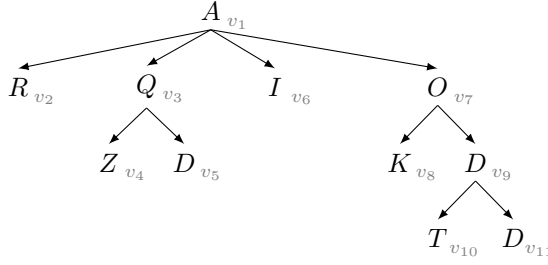


Figure 3.4: Example of a labeled, ordered, rooted tree  $\Lambda_1$  with vertices  $V = \{v_1, \dots, v_{11}\}$  and labels  $\Sigma = \{A, D, I, K, O, Q, R, T, Z\}$ ; each vertex is labeled, for example,  $\lambda(v_2) = R$

each two vertices exactly one path exists. Note that all edges point away from the root vertex. Such trees are also called an *out-tree*, cf. [62, page 207]. Below, we define labeled, ordered, rooted trees.

**Definition 3.11** (Labeled, Ordered, Rooted Tree)

A labeled, ordered, rooted tree  $\Lambda$  is a 6-tuple  $\Lambda = (V, E, \Sigma, \lambda, r, <)$  consisting of:

- a set of vertices  $V$ ,
- a set of edges  $E \subseteq V \times V$ ,
- a set of labels  $\Sigma$ ,
- a labeling function  $\lambda : V \rightarrow \Sigma$  that assigns each vertex a label,
- an unique root vertex  $r \in V$ , and
- a strict total order  $< \in V \times V$ .

Further,  $\Lambda$  satisfies the following constraints.

1.  $(V, E)$  represents a directed graph that is acyclic and weakly connected.
2. There exists exactly one directed path from the root vertex  $r$  to any vertex  $v \in V \setminus \{r\}$ .

We denote the universe of labeled, ordered, rooted trees as  $\mathcal{T}$ .

In the remainder of this thesis, we refer to labeled, ordered, rooted trees simply as trees. Furthermore, since all edges point away from the root node, we refrain from representing this in illustrations of trees hereinafter.

For a given tree  $\Lambda = (V, E, \Sigma, \lambda, r, <) \in \mathcal{T}$  and a vertex  $v \in V$ , we refer to the child vertices of  $v$  as  $child_\Lambda(v)$ . For instance, consider the tree  $\Lambda$  shown in Figure 3.4. The child vertices of  $v_7$  are  $child_\Lambda(v_7) = \{v_8, v_9\}$  with  $v_8 < v_9$  according to the ordering of  $V$ . Similar, we refer to the parent of vertex  $v$  as  $parent_\Lambda(v)$ ; for example,  $parent_\Lambda(v_3) = v_1$ . Further, we refer to the descendants of a vertex  $v$  as  $desc_\Lambda(v)$ . For instance, consider Figure 3.4,  $desc_\Lambda(v_7) = \{v_8, v_9, v_{10}, v_{11}\}$ . Likewise, we refer to the ancestors of a vertex  $v$  as  $anc_\Lambda(v)$ . For example,  $anc_\Lambda(v_8) = \{v_7, v_1\}$ . Below, we define these functions.

**Definition 3.12** (Tree Child/Parent/Descendants/Ancestors Function)

Let  $\Lambda = (V, E, \Sigma, \lambda, r, <) \in \mathcal{T}$  and  $v \in V$ . We define the following functions:

- Child function:  $child_\Lambda : V \rightarrow \mathbb{P}(V)$  with  $child_\Lambda(v) = \{v' \in V \mid (v, v') \in E\}$
- Parent function:  $parent_\Lambda : V \rightarrow V$  with

$$parent_\Lambda(v) = v' \text{ such that } (v', v) \in E$$

$$parent_\Lambda(r) \text{ is undefined}$$

- Descendants function:  $desc_\Lambda : V \rightarrow \mathbb{P}(V)$  with

$$desc_\Lambda(v) = child_\Lambda(v) \cup \left( \bigcup_{v_i \in child_\Lambda(v)} desc_\Lambda(v_i) \right)$$

- Ancestors function:  $anc_\Lambda : V \rightarrow \mathbb{P}(V)$  with

$$anc_\Lambda(v) = \begin{cases} \emptyset, & \text{if } v = r, \\ \{parent_\Lambda(v)\} \cup anc_\Lambda(parent_\Lambda(v)), & \text{otherwise} \end{cases}$$

Given the vertices of a tree, we distinguish *inner vertices*, i.e., vertices that have children, and *leaf vertices*, i.e., vertices without children. For instance, consider  $\Lambda_1$  depicted in Figure 3.4. Vertices  $v_2, v_4, v_5, v_6, v_8, v_{10}$ , and  $v_{11}$  are leaf vertices; the other vertices are inner vertices. Below, we define functions to retain the inner leaf vertices of a given tree.

**Definition 3.13** (Inner/Leaf Vertices)

Let  $\Lambda = (V, E, \Sigma, \lambda, r, <) \in \mathcal{T}$ . We define the following functions:

- Leaf vertices  $leaves : \mathcal{T} \rightarrow \mathbb{P}(V)$  with  $leaves(\Lambda) = \{v \in V \mid desc_\Lambda(v) = \emptyset\}$
- Inner vertices  $inner : \mathcal{T} \rightarrow \mathbb{P}(V)$  is defined as  $inner(\Lambda) = V \setminus leaves_\Lambda(\Lambda)$

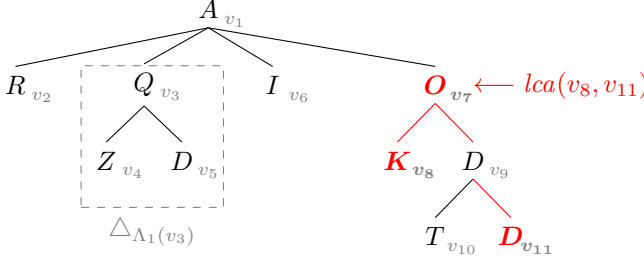
We introduce the *simplified textual representation* of trees. Since vertices are strictly totally ordered, a given tree's textual representation is unique. Consider the tree depicted in Figure 3.4. The textual representation of the tree is:

$$\Lambda_1 \hat{=} v_1 \left( v_2, v_3 \left( v_4, v_5 \right), v_6, v_7 \left( v_8, v_9 \left( v_{10}, v_{11} \right) \right) \right).$$

Similarly, we introduce the textual label-projected representation of a tree:

$$\Lambda_1 \hat{=}_{\Sigma} A \left( R, Q \left( Z, D \right), I, O \left( K, D \left( T, D \right) \right) \right).$$

For two vertices  $v_1, v_2 \in V$  of a given tree, the Lowest Common Ancestor (LCA) defines the lowest vertex  $v_{LCA} \in V$  in the tree that contains  $v_1$  and  $v_2$  as descendants.


 Figure 3.5: Example of a subtree and a LCA in tree  $\Lambda_1$ 

For example, consider  $v_{11}$  and  $v_8$  in tree  $\Lambda_1$  depicted in Figure 3.4. The Lowest Common Ancestor (LCA) of  $v_{11}$  and  $v_8$  is  $lca_{\Lambda}(v_{11}, v_8) = v_7$ , cf. Figure 3.5.

**Definition 3.14** (Lowest Common Ancestor (LCA))

Let  $\Lambda = (V, E, \Sigma, \lambda, r, <) \in \mathcal{T}$  and  $v_1, v_2 \in V$ . We define the LCA as a function  $lca_{\Lambda} : V \times V \rightarrow V$  with  $lca_{\Lambda}(v_1, v_2) = v_{LCA}$  such that:

1.  $v_1, v_2 \in desc_{\Lambda}(v_{LCA})$  and
2.  $\nexists v' \in desc_{\Lambda}(v_{LCA}) [v_1, v_2 \in desc_{\Lambda}(v')]$ .

For a given tree  $\Lambda = (V, E, \Sigma, \lambda, r, <)$  and a set of vertices  $V' = \{v_1, v_2, \dots, v_{n-1}, v_n\} \subseteq V$ , we define the LCA of all vertices contained in  $V'$  as follows.

$$lca_{\Lambda}(V') := lca_{\Lambda} \left( v_1, lca_{\Lambda} \left( v_2, lca_{\Lambda} \left( \dots lca_{\Lambda}(v_{n-1}, v_n) \right) \right) \right)$$

Note that for the empty set  $\emptyset \subseteq V$ , function  $lca_{\Lambda}(\emptyset)$  is undefined. For example, recall tree  $\Lambda_1$  (cf. Figure 3.5). The LCA of the vertices  $v_4, v_5$ , and  $v_8$  is defined as follows.

$$lca_{\Lambda_1}(\{v_4, v_5, v_8\}) := lca_{\Lambda_1}(v_4, lca_{\Lambda_1}(v_5, v_8)) = v_1$$

For a given tree  $\Lambda = (V, E, \Sigma, \lambda, r, <)$  and a vertex  $v \in V$ , we refer to the subtree of  $\Lambda$  that is rooted at  $v$  as  $\Delta_{\Lambda}(v) \in \mathcal{T}$ . For instance, consider the tree depicted in Figure 3.5. The subtree rooted at  $v_3$  is  $\Delta_{\Lambda}(v_3) = \left( \{v_3, v_4, v_5\}, \{(v_3, v_4), (v_3, v_5)\}, \{Q, Z, D\}, \lambda', v_3 \right)$  with  $\lambda' = \lambda|_{\{v_3, v_4, v_5\}}$ , cf. Figure 3.5. Below, we define the subtree function that returns for a vertex  $v$  from a tree, the corresponding subtree rooted at  $v$ .

**Definition 3.15** (Subtree Function)

Let  $\Lambda = (V, E, \Sigma, \lambda, r, <) \in \mathcal{T}$  and  $v \in V$ . The subtree  $\Delta_\Lambda(v) = (V_{sub}, E_{sub}, \Sigma_{sub}, \lambda_{sub}, r_{sub}, <_{sub})$  is defined as follows.

- $V_{sub} = (\text{desc}_\Lambda(v) \cup \{v\}) \subseteq V$
- $E_{sub} = \{(v_1, v_2) \in E \mid v_1, v_2 \in V_{sub}\} \subseteq E$
- $\Sigma_{sub} = \{\lambda(v') \mid v' \in V_{sub}\} \subseteq \Sigma$
- $\lambda_{sub} = \lambda|_{V_{sub}}$
- $r_{sub} = v$
- $<_{sub} = <|_{V_{sub}}$

For a given tree  $\Lambda \in \mathcal{T}$  and a subtree  $\Lambda'$  of  $\Lambda$ , we write  $\Lambda' \sqsubseteq \Lambda$ . For instance, consider  $\Lambda$ , depicted in Figure 3.4. The subtree  $\Delta_{\Lambda_1}(v_7) \sqsubseteq \Lambda_1$ . Note that the vertices' identifiers have to match.

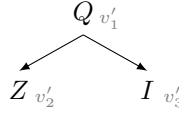


Figure 3.6: Tree  $\Lambda_2 \in \mathcal{T}$ , which is an isomorphic subtree of  $\Lambda_1$  (cf. Figure 3.4) but *not* a subtree of  $\Lambda_1$  (cf. Figure 3.4)

For given trees  $\Lambda_1 = (V_1, E_1, \Sigma_1, \lambda_1, r_1, <_1), \Lambda_2 = (V_2, E_2, \Sigma_2, \lambda_2, r_2, <_2) \in \mathcal{T}$ , we write  $\Lambda_1 \tilde{\sqsubseteq} \Lambda_2$  if  $\Lambda_1$  is an *isomorphic subtree* of  $\Lambda_2$ . Consider tree  $\Lambda_2$  depicted in Figure 3.6. Tree  $\Lambda_2$  is an isomorphic subtree of  $\Lambda_1$  (cf. Figure 3.4) because we can map the vertices from  $\Lambda_2$  to vertices in  $\Lambda_1$  such that the mapping preserves the labels and connections between vertices. However,  $\Lambda_2 \not\sqsubseteq \Lambda_1$  since both trees have different vertices.

**Definition 3.16** (Subtree Isomorphism)

Let  $\Lambda_1 = (V_1, E_1, \Sigma_1, \lambda_1, r_1, <_1), \Lambda_2 = (V_2, E_2, \Sigma_2, \lambda_2, r_2, <_2) \in \mathcal{T}$ . Tree  $\Lambda_1$  is an isomorphic subtree of  $\Lambda_2$ , denoted as  $\Lambda_1 \tilde{\sqsubseteq} \Lambda_2$ , if there is an injective function  $f : V_1 \rightarrow V_2$  such that the following properties hold.

- $\forall v \in V_1 \left( \lambda_1(v) = \lambda_2(f(v)) \right)$
- $\forall v, v' \in V_1 \left( (v, v') \in E_1 \Leftrightarrow (f(v), f(v')) \in E_2 \right)$

Note that by definition, every subtree is also an isomorphic subtree. Thus, for arbitrary  $\Lambda_1, \Lambda_2 \in \mathcal{T}$  it holds  $\Lambda_1 \sqsubseteq \Lambda_2 \Rightarrow \Lambda_1 \tilde{\sqsubseteq} \Lambda_2$ . However, the other direction generally does not hold.

Table 3.1.: Example of an event log representing the execution of a mortgage application process. Each row represents an individual event. Events are sorted based on event Identifier (ID) in this table.

ID		Activity label (activity label abbreviation)	Temporal information		
Event	Case		Timestamp	Duration *	...
...	...	...	...	...	...
8245	134	credit request received (CRR)	16.06.21 12:43:35	—	...
8246	134	document check (DC)	17.06.21 08:32:23	1d, 3h, 28m, 48s	...
8247	134	request information from applicant (RIP)	19.06.21 09:34:00	2d, 23h, 38m, 0s	...
8248	134	request information from third parties (RIT)	19.06.21 14:54:00	5d, 18h, 3m, 12s	...
8249	134	document check (DC)	28.06.21 14:23:59	—	...
8250	134	credit assessment (CA)	30.06.21 13:02:11	3d, 19h, m9, 21s	...
8251	134	security risk assessment (SRA)	01.07.21 17:23:11	5d, 1h, 28m, 32s	...
8252	134	property inspection (PI)	05.07.21 00:00:00	—	...
8253	134	loan-to-value ratio determined (LTV)	05.07.21 00:00:00	—	...
8254	134	decision made (DM)	08.07.21 14:13:18	—	...
8255	135	credit request received (CRR)	17.06.21 23:21:31	—	...
8256	135	document check (DC)	18.06.21 11:34:12	3d, 21h, 8m, 32s	...
...	...	...	...	...	...

\* If the duration of an event  $e$  is  $e^d = 0$ , we display — within the duration column.

## 3.2. Event Data & Event Logs

This section introduces event data and related concepts such as cases and traces. In general, event data describe the historical execution of processes. Table 3.1 provides an example event log that describes multiple process executions of a mortgage application process. Each row corresponds to an individual event. For instance, the first event with ID 8245 describes that for the case 134, the activity ‘credit request received,’ abbreviated by ‘CRR,’ has been executed. Note that a case refers to a single execution of a process, i.e., in the specific example, an individual mortgagee application. The activity ‘CRR’ was executed on 16.06.2021 at 12:43:35; duration information is unavailable for this activity. The three dots ... indicate that each event may have many more attributes; for example, the resource involved, costs of performing the activity, and various other features specific to the activity at hand cf. [59]. However, we focus on five attributes that are present per event: event ID, case ID, activity label, timestamp, and optional duration information. For the sake of simplicity, we assume that time points and durations are represented as non-negative real numbers.<sup>2</sup>

<sup>2</sup>For example, consider Unix time to represent date and time in computing. Timestamps are defined based on the seconds that have elapsed since January 1, 1970. Thus, the above-made assumption that real values represent timestamps is, therefore, feasible.

**Definition 3.17** (Event)

Let  $\mathcal{A}$  denote the universe of activity labels. An event  $e$  is a 5-tuple  $e = (i, c, a, t, d) \in \mathbb{N} \times \mathbb{N} \times \mathcal{A} \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0}$  consisting of an event ID  $i \in \mathbb{N}$ , a case ID  $c \in \mathbb{N}$ , an activity label  $a \in \mathcal{A}$ , a timestamp  $t \in \mathbb{R}_{\geq 0}$ , and a duration  $d \in \mathbb{R}_{\geq 0}$ .

We denote the universe of events as  $\mathcal{E}$ ; we assume that every event is uniquely identifiable, as specified below.

$$\forall e_1=(i_1, c_1, a_1, t_1, d_1), e_2=(i_2, c_2, a_2, t_2, d_2) \in \mathcal{E} (e_1 \neq e_2 \Rightarrow i_1 \neq i_2)$$

Note that the definition above allows both atomic and non-atomic events, i.e., events that span a period. With Definition 3.17, we can, therefore, represent the execution of an activity, which can extend over some time, with a single event. Note that in process mining literature, events are often considered atomic, for example, cf. [211, Definition 5.1] and [215, Definition 2].<sup>3</sup> Activities that span a period of time are usually described by two separate atomic events, one of which indicates the start and the other the completion of the period. Definition 3.17 allows us to avoid this split of an activity into two atomic events. Moreover, activities can have a life cycle, for instance, an activity can be planned, executed, paused, executed again and finally completed [59]. However, such a fine-grained distinction is often not made in most process mining approaches and techniques. In this thesis, life cycle information is also not considered. Therefore, Definition 3.17 is sufficient, i.e., an event describing an activity is either atomic or describes a time period.

For an event  $e = (i, c, a, t, d) \in \mathcal{E}$ , we introduce shortcuts for the specific components:  $e^i = \pi_1(e)$ ,  $e^c = \pi_2(e)$ ,  $e^a = \pi_3(e)$ ,  $e^t = \pi_4(e)$ , and  $e^d = \pi_5(e)$ . Next, we define an event log as exemplified in Table 3.1.

**Definition 3.18** (Event Log)

An event log  $L \subseteq \mathcal{E}$  is a set of events.

Next, we define cases that group events from an event log with identical case IDs. Cases describe individual process executions and are a key concept in process mining.

**Definition 3.19** (Case)

Let  $L$  be an event log. Let  $c \in \mathbb{N}$  be a case ID and  $C \subseteq L$  is a set containing all events from  $L$  having  $c$  as case ID, i.e.,  $C = \{e \in L \mid e^c = c\}$ .

Note that cases may also have attributes that affect the case itself, in addition to events having attributes, cf. [211, Definition 5.3]. However, we do not consider case attributes in this thesis.

Next, we introduce *traces* that order events within a case into a sequence. Thus, since we can easily convert sequences into strictly totally ordered sets (Definition 3.8), traces can also be represented as a strict total order over the events in a case. Below, we present an example of such a strict total ordering over events of a sequence. Consider Table 3.1

<sup>3</sup>However, note that both referenced definitions allow having additional attributes like the duration attribute explicitly specified in Definition 3.17.



and let  $e_{8245}$  denote the first depicted event; we refer to the other events accordingly. For example, the trace for case 134 is  $\sigma_{134} = \langle e_{8245}, e_{8246}, \dots, e_{8252}, e_{8253}, e_{8254} \rangle$ . Note that all events are ordered based on their timestamp, starting from the earliest to the latest event. For events having the same timestamp, i.e.,  $e_{8252}$  and  $e_{8253}$ , we use the event IDs  $8252 < 8253$  as a second order criterion. Hence,  $e_{8252}$  occurs before  $e_{8253}$  in the trace. Sequentializing events of a case into a trace is common in process mining [59, 211]. Since events of a case may have identical timestamps, a second-order criterion is needed; we use the event IDs in this thesis as exemplified above.

**Definition 3.20** (Trace)

Let  $L$  be an event log and  $C = \{e_1, \dots, e_n\} \subseteq L$  be a case with  $n \in \mathbb{N}$  events. The trace representing case  $C$  is a sequence of its events, i.e., a strict total order. We define the trace of  $C$  as  $\sigma \in C^*$  with:

- $|\sigma| = |C| = n$ ,
- $\forall e \in C \ (e \in \sigma)$ , and
- $\forall 1 \leq i < j \leq n \left( \sigma(i)^t < \sigma(j)^t \vee \left( \sigma(i)^t = \sigma(j)^t \wedge \sigma(i)^i < \sigma(j)^i \right) \right)$ .

Consider the trace  $\sigma_{134}$  shown above. When projecting  $\sigma_{134}$  to the sequence of activity labels<sup>4</sup>, we obtain  $\pi_3^*(\sigma_{134}) = \langle CRR, DC, RIP, RIT, DC, CA, SRA, PI, LTV, DM \rangle$ . We refer to this sequence as *simplified trace* that only shows the sequence of executed activities for a given case. Below, we define a *simplified trace*.

**Definition 3.21** (Simplified Trace)

Let  $\mathcal{A}$  be the universe of activity labels,  $L \subseteq \mathcal{A}^*$  be an event log,  $C \subseteq L$  be a case, and  $\sigma \in C^*$  be the corresponding trace. We define the simplified trace  $\sigma' \in \mathcal{A}^*$  as  $\sigma' = \pi_3^*(\sigma)$ .

Note that multiple traces may correspond to the same simplified trace because we only consider the sequence of activity labels. Accordingly, we define a *simplified event log* consisting of a multiset of simplified traces.

**Definition 3.22** (Simplified Event Log)

Let  $\mathcal{A}$  be the universe of activity labels. A simplified event log  $L^s$  is a multiset of simplified traces, i.e.,  $L^s \in \mathcal{M}(\mathcal{A}^*)$ .

Unless otherwise noted, *simplified traces* are referred to as *traces* in the remainder of this thesis for ease of reading. For instance, the simplified event log

$$L^s = \left[ \langle CRR, DC, RIP, RIT, DC, CA, SRA, PI, LTV, DM \rangle^5, \right. \\ \left. \langle CRR, DC, RIP, RIT, DC, DM \rangle^2, \langle CRR, DC, DM \rangle^2 \right] \in \mathcal{M}(\mathcal{A}^*)$$

<sup>4</sup>We only use the activity label abbreviations in the following.

contains nine traces. Moreover, the simplified event log  $L^s$  contains the following three unique traces.

$$\overline{L^s} = \left\{ \langle CRR, DC, RIP, RIT, DC, CA, SRA, PI, LTV, DM \rangle, \right. \\ \left. \langle CRR, DC, RIP, RIT, DC, DM \rangle, \langle CRR, DC, DM \rangle \right\} \subseteq \mathcal{A}^*$$

In short, an event log  $L \subseteq \mathcal{E}$  (cf. Definition 3.18) is a set of events. Given an event log  $L$ , the corresponding simplified event log  $L^s \in \mathcal{M}(\mathcal{A}^*)$  (cf. Definition 3.22) contains the simplified traces (cf. Definition 3.21) from  $L$ . Finally, since  $L^s$  is a multiset, we can refer to the unique simplified traces as  $\overline{L^s} \subseteq \mathcal{A}^*$ .

In Chapters 4 to 7, we assume simplified event logs and simplified traces, cf. Definitions 3.21 and 3.22. Later, in Part III, comprising Chapters 8 and 9, we assume events and event logs as specified in Definitions 3.17 and 3.18.

### 3.3. Process Models

This section introduces process model formalisms, which allow specifying process behavior. Most formalisms focus primarily on the control flow of activities, for example, where are branches or decisions between activities, where are repetitions, in which order are activities executed, and which activities are executed in parallel. In the remainder of this section, we introduce two process model formalisms: Petri nets in Section 3.3.1 and process trees in Section 3.3.2. Note that many more formalisms exist, for example, BPMN [48], event-driven process chains (EPCs) [205], yet another workflow language (YAWL) [220], and UML statecharts [50]. However, most formalisms can be translated to Petri nets, for instance, [67] describes a transformation from BPMN to Petri nets.

#### 3.3.1. Petri Nets

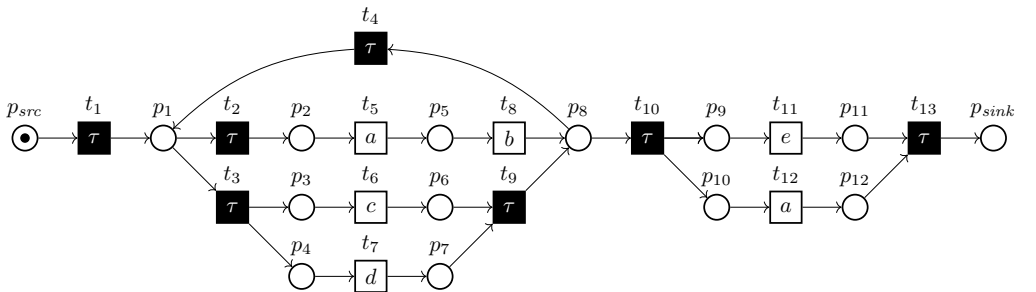


Figure 3.7: Example Petri net  $N_1$  consisting of 14 places  $p_{src}, p_1, \dots, p_{12}, p_{sink}$  and 13 labeled transitions  $t_1, \dots, t_{13}$ ; the visualized initial marking contains one token in place  $p_{src}$  and the final marking contains one token in place  $p_{sink}$

Petri nets [152] allow to model process behavior and are a frequently used formalism in process mining [204]. Figure 3.7 depicts an example Petri net  $N_1$ . Petri nets generally consist of places visually represented as circles, transitions visually represented as squares, and directed edges that connect places and transitions. Note that directed edges never directly connect two places or transitions. Petri net  $N_1$  consists of twelve places and ten labeled transitions; for instance, transition  $t_5$  is labeled with  $a$ . Next, we define labeled Petri nets.

**Definition 3.23** (Labeled Petri net)

Let  $\mathcal{A}$  be the universe of activity labels with  $\tau \notin \mathcal{A}$ . A labeled Petri net  $N$  is a 4-tuple  $N = (P, T, F, \lambda)$  where  $P$  is a set of places,  $T$  a set of transitions with  $P \cap T = \emptyset$ ,  $F \subseteq (P \times T) \cup (T \times P)$  a set of arcs, and a labeling function  $\lambda: T \rightarrow (\mathcal{A} \cup \{\tau\})$ .

We denote the universe of Petri nets as  $\mathcal{N}$ .

We refer to transitions labeled  $\tau$  as *silent transitions* we assume that these cannot be observed, i.e., they do not represent a process activity. Let  $N = (P, T, F, \lambda) \in \mathcal{N}$  be a Petri net. Given a node  $x \in P \cup T$ , we define the set of nodes having an arc pointing to  $x$  as  $\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$ . Similarly, we define the nodes with an incoming arc from  $x$  as  $x \bullet = \{y \in P \cup T \mid (x, y) \in F\}$ . For example, consider Petri net  $N_1$  depicted in Figure 3.7,  $\bullet p_8 = \{t_8, t_9\}$  and  $t_3 \bullet = \{p_3, p_4\}$ .

The state of a Petri net  $N = (P, T, F, \lambda)$  can be described by a *marking*. Formally, a marking  $M$  is a multiset of places, i.e.,  $M \in \mathcal{M}(P)$ . We illustrate a marking by drawing dots, i.e., tokens, into the corresponding places contained in the multiset according to their occurrence. As an example, consider  $N_1$ . The illustrated marking  $[p_{src}]$  is shown by drawing one token in place  $p_{src}$ , cf. Figure 3.7. Given a Petri net  $N$  and a marking  $M$ , we write  $(N, M)$  to refer to the *marked* Petri net. For instance, the Petri net depicted in Figure 3.7 can be written as a marked Petri net  $(N_1, [p_{src}])$ .

The transitions of a Petri net allow to change its state. For a Petri net  $N = (P, T, F, \lambda)$ , a marking  $M \in \mathcal{M}(P)$ , and a transition  $t \in T$ , we call  $t$  *enabled* if  $\forall p \in \bullet t \ [(M(p) > 0)]$ . We write  $(N, M)[t]$  if  $t$  is enabled in  $M$ . For instance,  $(N_1, [p_{src}])[t_1]$ , cf. Figure 3.7. Enabled transitions can be *fired*; firing a transition may lead to a state change. Upon firing  $t$  in marking  $M \in \mathcal{M}(P)$ , we obtain  $M' \in \mathcal{M}(P)$  with for all  $p \in P$  the following holds.

$$\begin{aligned} M'(p) &= M(p) + 1 && \text{if } p \notin \bullet t \wedge p \in t \bullet \\ M'(p) &= M(p) - 1 && \text{if } p \in \bullet t \wedge p \notin t \bullet \\ M'(p) &= M(p) && \text{if } p \in \bullet t \wedge p \in t \bullet \text{ or } p \notin \bullet t \wedge p \notin t \bullet \end{aligned}$$

We write  $(N, M) \xrightarrow{t} (N, M')$  to denote that firing transition  $t$  in marking  $M$  leads to marking  $M'$ . For example,  $(N_1, [p_{src}]) \xrightarrow{t_1} (N_1, [p_1])$ , cf. Figure 3.7. Likewise, we write  $(N, M) \xrightarrow{\sigma} (N, M')$  to denote that a sequence of transitions  $\sigma \in T^*$  leads from marking  $M$  to  $M'$ . For example,  $(N_1, [p_{src}]) \xrightarrow{\langle t_1, t_2, t_3, t_7 \rangle} (N_1, [p_2, p_3, p_7])$ , cf. Figure 3.7. Further, we write  $(N, M) \rightsquigarrow (N, M')$  iff  $\exists \sigma \in T^* \left( (N, M) \xrightarrow{\sigma} (N, M') \right)$ . Finally, we define the

reachable markings of a Petri net  $N$  for a given marking by  $M$  by

$$\mathcal{RM}(N, M) = \left\{ M' \in \mathcal{M}(P) \mid \exists \sigma \in T^* \left[ (N, M) \xrightarrow{\sigma} (N, M') \right] \right\}.$$

For instance,  $\mathcal{RM}(N_1, [p_9, p_{10}]) = \{[p_9, p_{10}], [p_{10}, p_{11}], [p_9, p_{12}], [p_{11}, p_{12}], [p_{sink}]\}$ , cf. Figure 3.7.<sup>5</sup> Next, we define accepting Petri nets that combine a Petri net as defined in Definition 3.23 with an initial and a final marking.

**Definition 3.24** (Accepting Petri Net)

Let  $(P, T, F, \lambda) \in \mathcal{N}$  be a Petri net. An accepting Petri net  $N$  is a 6-tuple  $N = (P, T, F, \lambda, M^{init}, M^{final})$  with  $M^{init} \in \mathcal{M}(P)$  being the initial marking and  $M^{final} \in \mathcal{M}(P)$  being the final marking.

We denote the universe of accepting Petri nets as  $\mathcal{N}_{accept}$ .

To define the language of accepting Petri nets, we first generalize the label function  $\lambda : T \rightarrow \mathcal{A} \cup \{\tau\}$  to the function  $\lambda^* : T^* \rightarrow (\mathcal{A} \cup \{\tau\})^*$  with

$$\lambda^*(\sigma) = \begin{cases} \langle \rangle, & \text{if } \sigma = \langle \rangle, \\ \langle \lambda^*(t) \rangle & \text{if } \sigma = \langle t \rangle, \\ \lambda^*(\sigma') \circ \langle \lambda^*(t) \rangle & \text{if } \sigma = \sigma' \circ \langle t \rangle \end{cases}$$

For instance, consider  $N_1$  depicted in Figure 3.7,  $\lambda^*(\langle t_1, t_2, t_5, t_8, t_{10}, t_{11}, t_{12}, t_{13} \rangle) = \langle \tau, \tau, a, b, \tau, e, a, \tau \rangle$  and  $\lambda^*(\langle t_1, t_2, t_5, t_8, t_{10}, t_{11}, t_{12}, t_{13} \rangle) \downarrow_{\mathcal{A}} = \langle a, b, e, a \rangle$ . Next, we define for accepting Petri nets their language in terms of accepted traces of activity labels.

**Definition 3.25** (Language of an Accepting Petri Net)

Let  $N = (P, T, F, \lambda, M^{init}, M^{final}) \in \mathcal{N}_{accept}$  be an accepting Petri net. We define the language of  $N$  as  $\mathbb{L}(N) = \left\{ \lambda^*(\sigma) \downarrow_{\mathcal{A}} \mid \sigma \in T^* \wedge (N, M^{init}) \xrightarrow{\sigma} (N, M^{final}) \right\} \subseteq \mathcal{A}^*$ .

Next, we define Workflow nets (WF-nets), which are a subclass of accepting Petri nets. WF-nets have a unique source place representing the initial marking and a sink place representing the final marking. Further, when connecting the sink place with an arc to the source place, between any pair of nodes—places and transitions—there exists a directed path.

**Definition 3.26** (Workflow Net)

Let  $N = (P, T, F, \lambda, M^{init}, M^{final}) \in \mathcal{N}_{accept}$  be an accepting Petri net.  $N$  is a WF-net if it satisfies the following requirements.

- $M^{init} = [p_{src}]$  for  $p_{src} \in P$  with  $\bullet p_{src} = \emptyset$

<sup>5</sup>Note that the provided marking, in this case,  $[p_9, p_{10}]$ , is also part of the set of reachable markings since the empty sequence of transitions exists.

- $M^{final} = [p_{sink}]$  for some  $p_{sink} \in P$  with  $p_{sink} \bullet = \emptyset$
- $\bar{N} = \left( P, T \cup \{\bar{t}\}, F \cup \{(p_{sink}, \bar{t}), (\bar{t}, p_{src})\}, \bar{\lambda} \right) \in \mathcal{N}$  with arbitrary labeling function  $\bar{\lambda}$  is strongly connected, i.e., a directed path exists from any pair of nodes (transitions and places are considered nodes) in  $\bar{N}$ .

We denote the universe of WF-nets as  $\mathcal{W} \subset \mathcal{N}_{accept}$ .

WF-nets are often used when modeling business processes [204, 211]. An important subclass of WF-nets are *sound* WF-nets, which have favorable behavioral properties. We define soundness below.

**Definition 3.27** (Soundness of WF-net)

Let  $N = (P, T, F, \lambda, M^{init}, M^{final}) \in \mathcal{W}$  with  $M^{init} = [p_{src}]$  and  $M^{final} = [p_{sink}]$ .  $N$  is sound iff the following behavioral properties are fulfilled [224].

1. From any reachable marking there is always the option to complete, i.e., the final marking is reachable. Thus,  $\forall M \in \mathcal{RM}(N, M^{init}) [M^{final} \in \mathcal{RM}(N, M)]$
2. If the sink place  $p_{sink}$  is marked, no other place is marked, i.e., proper completion. Thus,  $\forall M \in \mathcal{RM}(N, M^{init}) [p_{sink} \in M \Rightarrow [p_{sink}] = M]$ .
3.  $N$  contains no dead transitions, i.e., all transitions can be fired. Thus,  $\forall t \in T \exists M \in \mathcal{RM}(N, M^{init}) [(N, M)[t]]$ .

Consider  $N_1$ , the WF-net depicted in Figure 3.7. All three behavioral properties are fulfilled by  $N_1$ ; thus,  $N_1$  is *sound*.

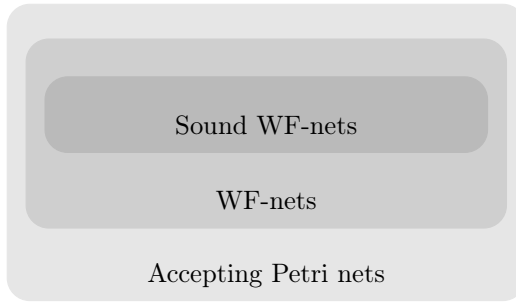


Figure 3.8: Overview of different Petri net classes

In this thesis, we always assume sound WF-nets when referring to WF-nets if not otherwise specified. Figure 3.8 provides an overview of different Petri net classes. The next section introduces process trees, which represent a subclass of sound WF-nets.

### 3.3.2. Process Trees

Process trees are a process model formalism that represent processes as a hierarchical composition of activities; they have been introduced in [120]. Figure 3.9 depicts an example process tree  $\Lambda_{example}$  that models the same behavior as the Petri net  $N_1$ , i.e., a sound WF-net, shown in Figure 3.7. Note that any process trees can be translated into a language-equivalent, sound WF-net, cf. Figure 3.8. Table 3.2 provides a complete overview on translating process trees into sound WF-nets. Note that process trees are also often referred to as *block-structured* WF-nets [120, 211].<sup>6</sup>

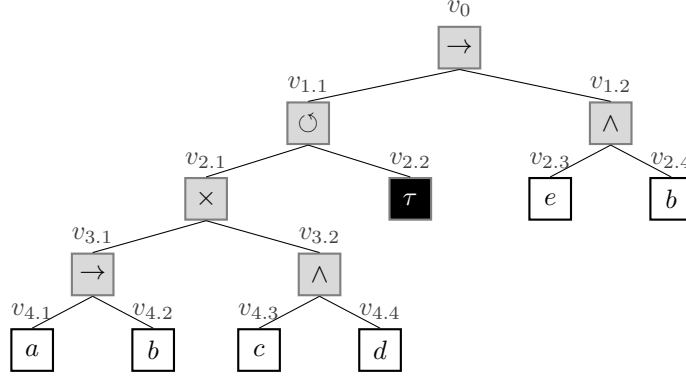


Figure 3.9: Example of a process tree that models the same language as the sound WF-net depicted in Figure 3.7


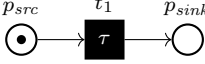

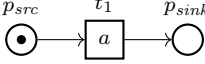
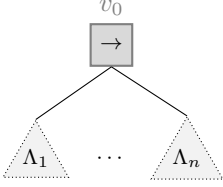
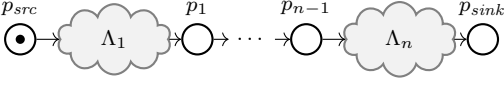
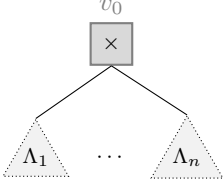
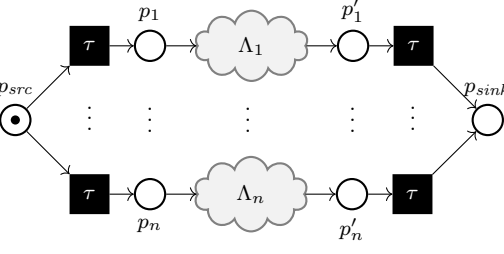
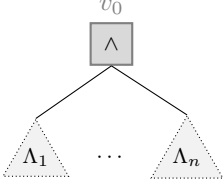
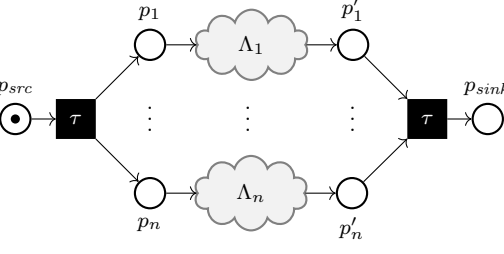
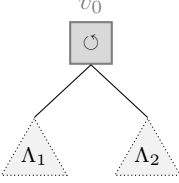
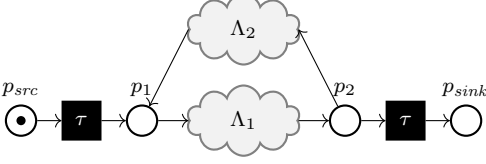
In general, a process tree is a labeled, rooted ordered tree as introduced in Definition 3.11 that satisfies certain properties. Inner vertices of a process tree represent control flow operators. We distinguish the four standard operators [216] in this thesis.

- The *sequence* operator ( $\rightarrow$ ) specifies that all its children must be executed in the given order.
- The *exclusive choice* operator ( $\times$ ) specifies that precisely one child must be executed.
- The *parallel* operator ( $\wedge$ ) specifies that its children can be executed in any order including interleaving execution.
- The *loop* operator ( $\odot$ ) specifies that its first child has to be executed once. After each execution of the first child, the second child can be optionally executed; if the second one is executed, the first must be subsequently re-executed.

Leaf vertices of a process tree represent activity labels from  $\mathcal{A}$  or the unobservable activity  $\tau$ . Next, we define the syntax of process trees.

<sup>6</sup>Note that to the best of our knowledge a formal definition of *block-structuredness* for WF-nets does not exist.

Table 3.2.: Conversion of process trees into language-equivalent, sound WF-nets

Activity/ operator	Process tree	WF-net
Invisible activity		
Visible activity		
Sequence		
Exclusive- choice		
Parallel		
Loop		

**Definition 3.28** (Process Tree Syntax)

Let  $\otimes = \{\rightarrow, \times, \circ, \wedge\}$  be the universe of process tree operators. A process tree  $\Lambda = (V, E, \Sigma, \lambda, r, <)$   $\in \mathcal{T}$  is a tree (cf. Definition 3.11) satisfying the following restrictions:

- $\Sigma = \mathcal{A} \cup \{\tau\} \cup \otimes$
- $\forall v \in \text{child}(\Lambda) [\lambda(v) \in \mathcal{A}]$
- $\forall v \in \text{inner}(\Lambda) [\lambda(v) \in \otimes]$
- $\forall v \in V [(\lambda(v) = \circ) \Rightarrow (|\text{child}_\Lambda(v)| = 2)]$

We denote the universe of process trees as  $\mathcal{P}$ .

Next, we specify the semantics of process trees. Therefore, we define *running steps* and *running sequences* of a process tree, i.e., a sequence of executed process tree leafs.

**Definition 3.29** (Process Tree Running Steps & Running Sequences)

Let  $\tau, \text{open}, \text{close} \notin \mathcal{A}$  and  $\Lambda = (V, E, \Sigma, \lambda, r, <) \in \mathcal{P}$  be a process tree. If  $\text{child}_\Lambda(r) \neq \emptyset$ , we refer to  $r$ 's children as  $v_1, \dots, v_n$  with  $v_1 < \dots < v_n$  according to  $V$ .

We define  $\mathcal{S}(\Lambda) = V \times (\mathcal{A} \cup \{\tau, \text{open}, \text{close}\})$  as the set of potential running steps.

We define the running sequences of  $\Lambda$  as  $\mathcal{RS}(\Lambda) \subseteq (\mathcal{S}(\Lambda))^*$ .

$$\mathcal{RS}(\Lambda) = \begin{cases} \{ \langle (r, \lambda(r)) \rangle \} & \text{if } \lambda(r) \in \mathcal{A} \cup \{\tau\}^a \\ \{ \langle (r, \text{open}) \rangle \} \circ \mathcal{RS}(\Delta_\Lambda(v_1)) \circ \dots \circ \mathcal{RS}(\Delta_\Lambda(v_n)) \circ \{ \langle (r, \text{close}) \rangle \} & \text{if } \lambda(r) = \rightarrow, n \geq 1 \\ \{ \langle (r, \text{open}) \rangle \} \circ \{ \mathcal{RS}(\Delta_\Lambda(v_1)) \cup \dots \cup \mathcal{RS}(\Delta_\Lambda(v_n)) \} \circ \{ \langle (r, \text{close}) \rangle \} & \text{if } \lambda(r) = \times, n \geq 1 \\ \{ \langle (r, \text{open}) \rangle \} \circ \{ \mathcal{RS}(\Delta_\Lambda(v_1)) \diamond \dots \diamond \mathcal{RS}(\Delta_\Lambda(v_n)) \} \circ \{ \langle (r, \text{close}) \rangle \} & \text{if } \lambda(r) = \wedge, n \geq 1 \\ \left\{ \langle (r, \text{open}) \rangle \circ \sigma_1 \circ \sigma'_1 \circ \sigma_2 \circ \sigma'_2 \circ \dots \circ \sigma'_{m-1} \circ \sigma_m \circ \langle (r, \text{close}) \rangle \mid \right. \\ \quad m \geq 1 \wedge \forall 1 \leq i \leq m [\sigma_i \in \mathcal{RS}(\text{child}_\Lambda(v_1))] \wedge \\ \quad \left. \forall 1 \leq i < m [\sigma'_i \in \mathcal{RS}(\text{child}_\Lambda(v_2))] \right\} & \text{if } \lambda(r) = \circ, n = 2 \end{cases}$$

<sup>a</sup>In this case, the tree consists of a single vertex.

Reconsider the process tree  $\Lambda_{\text{example}}$  depicted in Figure 3.9. This tree has infinitely many running sequences because it contains a loop ( $\circ$ ). Below, we list three exemplary running sequences  $\sigma_1, \sigma_2, \sigma_3 \in \mathcal{RS}(\Lambda_{\text{example}})$ . Further, we project each running sequence onto the sequence of executed leaf nodes representing an activity.

- $\sigma_1 = \langle (v_0, \text{open}), (v_{1.1}, \text{open}), (v_{2.1}, \text{open}), (v_{3.1}, \text{open}), (v_{4.1}, a), (v_{4.2}, b), (v_{3.1}, \text{close}), (v_{2.1}, \text{close}), (v_{1.1}, \text{close}), (v_{1.2}, \text{open}), (v_{2.3}, e), (v_{2.4}, a), (v_0, \text{close}) \rangle$   
 $\pi_2^*(\sigma_1) \downarrow_{\mathcal{A}} = \langle a, b, e, a \rangle$



- $\sigma_2 = \langle (v_0, \text{open}), (v_{1.1}, \text{open}), (v_{2.1}, \text{open}), (v_{3.2}, \text{open}), (v_{4.4}, d), (v_{4.3}, c), (v_{3.2}, \text{close}), (v_{2.1}, \text{close}), (v_{1.1}, \text{close}), (v_{1.2}, \text{open}), (v_{2.3}, e), (v_{2.4}, a), (v_0, \text{close}) \rangle$   
 $\pi_2^*(\sigma_2) \downarrow_{\mathcal{A}} = \langle d, c, e, a \rangle$
- $\sigma_3 = \langle (v_0, \text{open}), (v_{1.1}, \text{open}), (v_{2.1}, \text{open}), (v_{3.2}, \text{open}), (v_{4.4}, d), (v_{4.3}, c), (v_{3.2}, \text{close}), (v_{2.1}, \text{close}), (v_{2.2}, \tau), (v_{2.1}, \text{open}), (v_{3.2}, \text{open}), (v_{4.3}, c), (v_{4.4}, d), (v_{3.2}, \text{close}), (v_{2.1}, \text{close}), (v_{1.1}, \text{close}), (v_{1.2}, \text{open}), (v_{2.4}, a), (v_{2.3}, e), (v_0, \text{close}) \rangle$   
 $\pi_2^*(\sigma_3) \downarrow_{\mathcal{A}} = \langle d, c, c, d, a, e \rangle$

Next, we define the language of a process tree as a set of sequences over activity labels, similar to the language definition of Petri nets presented in Definition 3.25.

**Definition 3.30** (Process Tree Language)

Let  $\Lambda \in \mathcal{P}$ . We define its language  $\mathbb{L}(\Lambda) = \{(\pi_2^*(\sigma)) \downarrow_{\mathcal{A}} \mid \sigma \in \mathcal{RS}(\Lambda)\} \subseteq \mathcal{A}^*$ .

Reconsider the example process tree  $\Lambda_{\text{example}}$  depicted in Figure 3.9 and the three above shown running sequences;  $\{\langle a, b, e, a \rangle, \langle d, c, e, a \rangle, \langle d, c, c, d, a, e \rangle\} \subset \mathbb{L}(\Lambda_{\text{example}})$ . Eventually, we define *language equivalence* of two process trees if they define the same language.

**Definition 3.31** (Process Tree Language Equivalence)

Let  $\Lambda_1, \Lambda_2 \in \mathcal{P}$ . Process tree  $\Lambda_1$  is language equivalent to  $\Lambda_2$ , denoted as  $\Lambda_1 \sim \Lambda_2$ , if  $\mathbb{L}(\Lambda_1) = \mathbb{L}(\Lambda_2)$ .

Finally, we define the function *discovery* that discovers from a given event log a process tree such that the tree fully supports the provided event log, i.e., function *discovery* is *fitness-preserving*.

**Definition 3.32** (Fitness-preserving Process Discovery Function *discovery*)

The function *discovery* :  $\mathcal{M}(\mathcal{A}^*) \rightarrow \mathcal{P}$  maps an arbitrary event log  $L^s \in \mathcal{M}(\mathcal{A}^*)$  onto a process tree  $\Lambda \in \mathcal{P}$ . The function *discovery* is *fitness-preserving* if  $L^s \subseteq \mathbb{L}(\text{discovery}(L^s))$ .

An example of the above-defined fitness-preserving process discovery function is the Inductive Miner algorithm [122]. Note that an entire family of IM algorithms exists; however, not all are fitness-preserving.

This thesis focuses mainly on process trees as the primary model formalism. Process trees are widely used in process mining research [121, 211, 198, 39, 191, 216, 38], have unambiguous semantics, and represent an essential class of process models; each process tree can be converted into a sound WF-net and a BPMN model. Especially the fact that process trees are *sound by construction* makes them a valuable formalism. In addition, the hierarchical structure of process trees offers unique opportunities for manipulating them in the context of incremental process discovery.

### 3.4. Conformance Checking Overview

This section provides a brief introduction into the field of conformance checking [45, 46, 79]. Overall, conformance checking techniques relate observed process behavior, as reflected by the event data, with modeled behavior, as specified by process models. The aim is to provide statistics for conformity between event data and a process model. Further, conformance checking techniques are also used to assess the quality of process models with respect to a given event log [40]. Here, we focus on three central approaches: rule-based conformance checking, token-based replay, and alignments [46]. Figure 3.10 illustrates these techniques and their key differences. What these techniques have in common is that they all assume a process execution from an event log.

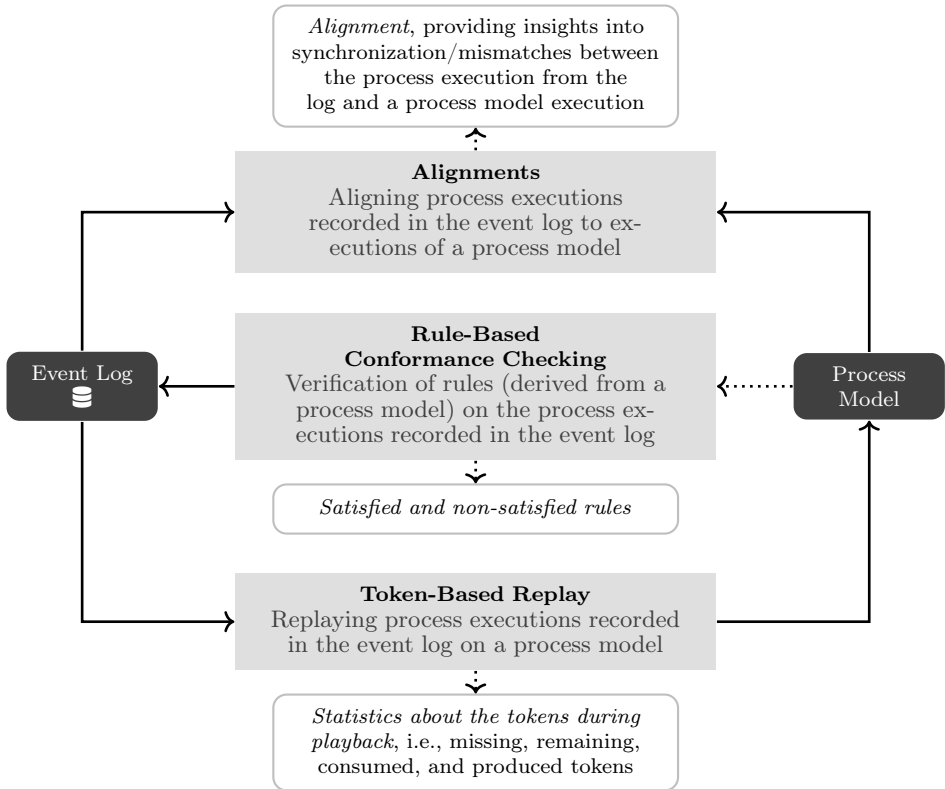


Figure 3.10: Overview of three major conformance checking approaches according to [46]

Rule-based conformance checking verifies a set of rules that might be derived from a process model on process executions from an event log. Compared to alignments and token-based replay, cf. Figure 3.10, the process model is of little importance as it is only used to derive rules. Process executions are assumed to fit if the set of rules is satisfied. According to [45] four main rule types can be distinguished. *Cardinality rules* set upper and lower limits for the execution of individual activities. *Precedence and response rules*

define relations between two activities. A precedence rule specifies that an activity always precedes another activity. In contrast, a response rule defines that an activity is always eventually followed by another one. *Ordering rules* specify if two activities occur both in individual process executions, their execution is ordered; for instance, one activity is executed after the other one. Finally, *exclusiveness rules* define pairs of activities that should never be executed both in individual process executions. In short, rule-based conformance checking does not necessarily require a process model as input and evaluates the conformance of process executions based on satisfied rules. However, covering all constraints of a process model with rules may lead to an exponential number of rules and in certain cases it may even be that impossible to specify the behavior of a model completely with rules [45].

Token-based replay [165] is a technique that replays process executions from the event log onto a process model. Process models are assumed to be Petri nets. During replaying a process execution, statistics about consumed and produced tokens are recorded. In case a part of a process execution cannot be replayed in the model, missing tokens are added to the Petri net to ensure that the process execution can be replayed. After finishing replaying the process executions, remaining tokens within the Petri net are counted. Based on these numbers regarding tokens, conformance statistics can be computed. In short, token-based replay takes a process execution and enforces its replay on the model. However, token-based replay also has limitations. First, activities that are present in the process execution but not in the model cannot be replayed by the technique. Further, token-based replay results might be non-deterministic in case the same activity occurs multiple times in a process model, i.e., also referred to as *duplicate labels*. Finally, silent transitions in Petri nets are problematic for token-based replay as it is not upfront clear which one to take in case multiple silent transitions are executable. To this end, extensions of token-based replay have been proposed [25].

Alignments [6, 226] are a state-of-the-art conformance checking technique, relating a process execution from the event log to a process execution from the process model. Thus, a valid process execution allowed by the process model is aligned with a process execution from the provided event log. An alignment, therefore, allows for identifying missing behavior, which refers to behavior that should have occurred according to the model but was not observed in the recorded process execution, and additional behavior, which refers to behavior that was recorded but should not have taken place according to the model. Compared to token-based replay, where a process execution from the event log is forcibly replayed in the process model, alignments take a symmetric approach concerning the process model and process execution [45].

### 3.5. Alignments

This section introduces alignments, which we will use in later chapters in the context of incremental process discovery. Alignments match a given trace from event data to a execution of a given process model. Alignments are initially introduced in [6] for Petri nets. Below, we show the structure of alignments, followed by an example. Subsequently, we formally define alignments for Petri nets (cf. Section 3.5.1) and process trees (cf. Section 3.5.2). Finally, Section 3.5.3 briefly elaborates on computing alignments.

An alignment represents a sequence of *alignment moves*; we distinguish four different move types as depicted below.

- **Synchronous moves** indicate a *match*, i.e., a synchronization, between the model and the current activity from the trace.
- **Log moves** indicate a *mismatch*, i.e., the current activity from the trace cannot be replayed in the model.
- **Visible model moves** indicate a *mismatch* between model and trace, i.e., the model is executing an activity that is not observed in the trace at that time.
- **Invisible model moves** indicates *no true mismatch*, yet the model performs a  $\tau$  activity that by definition cannot be observed in the trace.

Subsequently, we formally introduce alignments for Petri nets (Section 3.5.1) and process trees (Section 3.5.2).

### 3.5.1. Alignments for Petri nets

$\gamma_1 =$

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.
$\gg$	$\gg$	$a$	$b$	$\gg$	$\gg$	$d$	$d$	$\gg$	$\gg$	$\gg$	$a$	$e$	$\gg$	$f$
$t_1$	$t_2$	$t_5$	$t_8$	$t_4$	$t_3$	$t_7$	$\gg$	$t_6$	$t_9$	$t_{10}$	$t_{12}$	$t_{11}$	$t_{13}$	$\gg$
$(\tau)$	$(\tau)$	$(a)$	$(b)$	$(\tau)$	$(\tau)$	$(d)$	$\gg$	$(c)$	$(\tau)$	$(\tau)$	$(a)$	$(e)$	$(\tau)$	$\gg$

Figure 3.11: Exemplary *optimal* alignment  $\gamma_1 = \langle (\gg, t_1), (\gg, t_2), \dots, (f, \gg) \rangle$  consisting of 15 alignment moves for the trace  $\sigma = \langle a, b, d, d, a, e, f \rangle$  and the Petri net  $N_1$  (cf. Figure 3.7); under each transition, the assigned label is shown

Reconsider Petri net  $N_1$  depicted in Figure 3.7. Further, let  $\sigma = \langle a, b, d, d, a, e, f \rangle$  be a trace. Figure 3.11 illustrates an exemplary optimal alignment. Note that the first row of an alignment corresponds to the given trace when ignoring the skip symbol  $\gg$ . The second row corresponds to a sequence of transitions leading from  $(N_1, [p_{src}])$  to  $(N_1, [p_{sink}])$ . The alignment consists of a total of 15 alignment moves. The first two moves are invisible model moves and, thus, do not indicate a deviation. After that,  $t_5$  and  $t_8$  are the first executed transitions whose label represents an activity. The labels of these two transitions correspond to the first two activity labels of the trace  $\sigma$ ; thus, the 2. and 3. move are synchronous moves. Synchronous moves indicate that the model in its current state agrees with the execution of the activities occurring in the trace. Moves 5. and 6. are invisible model moves as the label of the corresponding transitions are labeled with  $\tau$ . Move 7. is again a synchronous move. Move 8. is a log move, i.e., the activity  $d$  cannot be replayed in the model at the current marking; thus, activity  $d$  is considered an unexpected/additional activity. The next activity after the second  $d$  observed in the trace is an  $a$  activity. However, the model requires to execute a  $c$  activity before, as indicated by move 9., i.e., a visible model move. After the missing  $c$  activity, the alignment indicates that the observed  $a$  and  $e$  activity from the trace

can be replayed in the model, cf. synchronous moves 12. and 13. The last move is a log move indicating unexpected/additional behavior according to the model, i.e., a  $f$  activity. In conclusion, the alignment indicates three deviations: (1) an additional  $d$  activity (8. move), (2) a missing  $c$  activity (9. move), an additional  $f$  activity (15. move). Below, we define alignments for a given trace and a Petri net.

**Definition 3.33** (Complete Alignment for Petri Nets)

Let  $(P, T, F, \lambda, M^{init}, M^{final}) \in \mathcal{W}$  be a sound WF-net and  $\sigma \in \mathcal{A}^*$  be a trace. A sequence  $\gamma \in \left( (\mathcal{A} \cup \{\gg\}) \times (T \times \{\gg\}) \right)^*$  is an alignment if:

1.  $\sigma = \pi_1^*(\gamma) \downarrow_{\mathcal{A}}$
2.  $(N, M^{init}) \xrightarrow{\pi_2^*(\gamma) \downarrow_T} (N, M^{final})$
3.  $\forall a \in \mathcal{A} \forall t \in T [(a, t) \in \gamma \Rightarrow a = \lambda(t)]$
4.  $(\gg, \gg) \notin \gamma$

We denote the universe of complete alignments for  $N$  and  $\sigma$  by  $\Gamma(N, \sigma)$ . We denote the universe of optimal complete alignments for  $N$  and  $\sigma$  by  $\Gamma^{opt}(N, \sigma)$ .

$\gamma_2 =$

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.
$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$a$	$b$	$d$	$d$	$a$	$e$	$f$
$t_1$	$t_2$	$t_5$	$t_8$	$t_{10}$	$t_{11}$	$t_{12}$	$t_{13}$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$
$(\tau)$	$(\tau)$	$(a)$	$(b)$	$(\tau)$	$(e)$	$(a)$	$(\tau)$							

Figure 3.12: Exemplary *non-optimal* alignment  $\gamma_2 = \langle (\gg, t_1), (\gg, t_2), \dots, (f, \gg) \rangle$  consisting of 15 alignment moves for the trace  $\sigma = \langle a, b, d, d, a, e, f \rangle$  and the Petri net  $N_1$  (cf. Figure 3.7); under each transition, the assigned label is shown

Many different alignments may exist for a given trace and a Petri net. For instance, Figure 3.12 shows another alignment for  $N_1$  and  $\sigma = \langle a, b, d, d, a, e, f \rangle$ . Still, the first row corresponds to the given trace when ignoring the skip symbol  $\gg$ , and the second row corresponds to a sequence of transitions leading from the initial to the final marking when ignoring  $\gg$ . However, alignment  $\gamma_2$  contains no synchronous moves compared to  $\gamma_1$ , cf. Figure 3.11. The interpretation of this alignment means that there are deviations everywhere. To this end, the notion of *optimality* exists. An alignment is optimal if the number of visible model and log moves is minimal compared to other alignments. Thus, alignment  $\gamma_2$  is *not* optimal because there exists an alignment, for instance,  $\gamma_1$ , with a lower number of log and visible model moves. In fact, alignment  $\gamma_1$  is optimal.

### 3.5.2. Alignments for Process Trees

This section introduces alignments for process trees. Note that the fundamental concept of alignments remains unchanged; compared to complete alignments on Petri nets, the

second row of an alignment is a running sequence of the tree rather than a sequence of transitions. Consider the alignment  $\gamma_3$  depicted in Figure 3.13 for process tree  $\Lambda_{example}$  and the same trace as before  $\sigma = \langle a, b, d, d, a, e, f \rangle$ . The second row represents now a running sequence of the process tree  $\Lambda_{example}$  when ignoring the skip symbol ( $\gg$ ). Below, we define alignments for process trees.

**Definition 3.34** (Complete Alignment for Process Trees)

Let  $\Lambda \in \mathcal{P}$  be a process tree and  $\sigma \in \mathcal{A}^*$  be a trace. Let  $\gg$  be the skip symbol with  $\gg \notin \mathcal{A}$ . A sequence  $\gamma \in \left( (\mathcal{A} \cup \{\gg\}) \times (\mathcal{S}(\Lambda) \cup \{\gg\}) \right)^*$  is a complete alignment iff:

1.  $\sigma = \pi_1^*(\gamma) \downarrow_{\mathcal{A}}$
2.  $\pi_2^*(\gamma) \downarrow_{\mathcal{S}(\Lambda)} \in \mathcal{RS}(\Lambda)$
3.  $\forall a \in \mathcal{A} \forall s \in \mathcal{RS}(\Lambda) \left[ (a, s) \in \gamma \Rightarrow a = \lambda(\pi_2(s)) \right]$
4.  $(\gg, \gg) \notin \gamma$

We denote the universe of complete alignments for  $\Lambda$  and  $\sigma$  by  $\Gamma(\Lambda, \sigma)$ . We denote the universe of optimal complete alignments for  $\Lambda$  and  $\sigma$  by  $\Gamma^{opt}(\Lambda, \sigma)$ .

$\gamma_3 =$

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	
$\gg$	$\gg$	$\gg$	$\gg$	$a$	$b$	$\gg$	$\gg$	$\gg$	$\gg$	...
$(v_0, open)$	$(v_{1.1}, open)$	$(v_{2.1}, open)$	$(v_{3.1}, open)$	$(v_{4.1}, a)$	$(v_{4.2}, b)$	$(v_{3.1}, close)$	$(v_{2.1}, close)$	$(v_{2.2}, \tau)$	$(v_{2.1}, open)$	

11.	12.	13.	14.	15.	16.	17.	18.	19.	20.	
$\gg$	$d$	$d$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$a$	$e$	...
$(v_{3.2}, open)$	$(v_{4.4}, d)$	$\gg$	$(v_{4.3}, c)$	$(v_{3.2}, close)$	$(v_{2.1}, close)$	$(v_{1.1}, close)$	$(v_{1.2}, open)$	$(v_{2.4}, a)$	$(v_{2.3}, e)$	

21.	22.	23.
$\gg$	$\gg$	$f$
$(v_{1.2}, close)$	$(v_0, close)$	$\gg$

...

Figure 3.13: Exemplary *optimal* alignment  $\gamma_3 = \langle (\gg, (v_0, open)), \dots, (f, \gg) \rangle$  for the trace  $\langle a, b, d, d, a, e, f \rangle$  and the process tree  $\Lambda_{example}$ , depicted in cf. Figure 3.9

In the remainder of this thesis, we refer to complete alignments simplistically as alignments. Further, we refer to the overall universe of all alignments—independent of a specific process model and trace—simply as  $\Gamma$ . Subsequently, we introduce various auxiliary functions for alignments. We use the above-specified auxiliary functions primarily for defining incremental process discovery approaches in Part II of this thesis.

Assume an arbitrary process tree  $\Lambda = (V, E, \Sigma, \lambda, r, <) \in \mathcal{P}$  and trace  $\sigma \in \mathcal{A}^*$ . Let  $\gamma \in \Gamma(\Lambda, \sigma)$  be an alignment and  $\gamma(i)$  for  $1 \leq i \leq |\gamma|$  be an arbitrary alignment move of  $\gamma$ . For ease of reading, we define three auxiliary functions to extract specific components of an alignment move.

- $traceLabel : \left( (\mathcal{A} \cup \{\gg\}) \times (\mathcal{S}(\Lambda) \cup \{\gg\}) \right) \rightarrow \mathcal{A} \cup \{\gg\}$  with  
 $traceLabel(\gamma(i)) = \pi_1(\gamma(i))$
- $modelVertex : \left( (\mathcal{A} \cup \{\gg\}) \times (\mathcal{S}(\Lambda) \cup \{\gg\}) \right) \rightarrow V \cup \{\gg\}$  with  
 $modelVertex(\gamma(i)) = \begin{cases} \pi_1(\pi_2(\gamma(i))) & \text{if } \pi_2(\gamma(i)) \neq \gg \\ \gg & \text{otherwise} \end{cases}$
- $modelLabel : \left( (\mathcal{A} \cup \{\gg\}) \times (\mathcal{S}(\Lambda) \cup \{\gg\}) \right) \rightarrow \mathcal{A} \cup \{\tau, \gg\}$  with  
 $modelLabel(\gamma(i)) = \begin{cases} \pi_2(\pi_2(\gamma(i))) & \text{if } \pi_2(\gamma(i)) \neq \gg \\ \gg & \text{otherwise} \end{cases}$

For example, consider alignment  $\gamma_3$  depicted in Figure 3.13. The following applies:

- $traceLabel(\gamma_3(5)) = a$ ,
- $traceLabel(\gamma_3(1)) = \gg$ ,
- $modelLabel(\gamma_3(5)) = a$ ,
- $modelLabel(\gamma_3(13)) = \gg$ ,
- $modelLabel(\gamma_3(9)) = \tau$ ,
- $modelLabel(\gamma_3(1)) = open$ ,
- $modelLabel(\gamma_3(7)) = close$ ,
- $modelVertex(\gamma_3(15)) = v_{3.2}$ , and
- $modelVertex(\gamma_3(13)) = \gg$ .

Moreover, we define four auxiliary functions to assess if an arbitrary alignment move is either a log, a visible model, an invisible model, or a synchronous move.

- $logMv : \left( (\mathcal{A} \cup \{\gg\}) \times (\mathcal{S}(\Lambda) \cup \{\gg\}) \right) \rightarrow \mathbb{B}$  with  
 $logMv(\gamma(i)) = \begin{cases} true & \text{if } modelLabel(\gamma(i)) = \gg \\ false & \text{otherwise} \end{cases}$
- $invModelMv : \left( (\mathcal{A} \cup \{\gg\}) \times (\mathcal{S}(\Lambda) \cup \{\gg\}) \right) \rightarrow \mathbb{B}$  with  
 $invModelMv(\gamma(i)) = \begin{cases} true & \text{if } modelLabel(\gamma(i)) = \tau \\ false & \text{otherwise} \end{cases}$
- $visModelMv : \left( (\mathcal{A} \cup \{\gg\}) \times (\mathcal{S}(\Lambda) \cup \{\gg\}) \right) \rightarrow \mathbb{B}$  with  
 $visModelMv(\gamma(i)) = \begin{cases} true & \text{if } traceLabel(\gamma(i)) = \gg \wedge modelLabel(\gamma(i)) \in \mathcal{A} \\ false & \text{otherwise} \end{cases}$

- $syncMv : ((\mathcal{A} \cup \{\gg\}) \times (\mathcal{S}(\Lambda) \cup \{\gg\})) \rightarrow \mathbb{B}$  with
 
$$syncMv(\gamma(i)) = \begin{cases} true & \text{if } traceLabel(\gamma(i)) = modelLabel(\gamma(i)) \\ false & \text{otherwise} \end{cases}$$

Further, we introduce  $invModelMvTau : ((\mathcal{A} \cup \{\gg\}) \times (\mathcal{S}(\Lambda) \cup \{\gg\})) \rightarrow \mathbb{B}$  that returns *true* if the provided alignment is an invisible model move and the executed vertex in the tree represents a leaf node; thus, the vertex is labeled  $\tau$ .

$$invModelMvTau(\gamma(i)) = \begin{cases} true & invModelMv(\gamma(i)) \wedge modelLabel(\gamma(i)) = \tau \\ false & \text{otherwise} \end{cases}$$

We say that alignment  $\gamma$  *indicates a deviation* between the corresponding process tree and the trace if  $\gamma$  contains at least one visible model move or log move. For instance, the optimal alignment  $\gamma_3$  depicted in Figure 3.13 indicates a deviation as it contains two log moves, i.e., the 13<sup>th</sup> and 23<sup>rd</sup> move, and one visible model move, i.e., the 14<sup>th</sup> move. Analogously, we say that alignment move  $\gamma(i)$  for  $1 \leq i \leq |\gamma|$  *indicates a deviation* if  $\gamma(i)$  is a log or visible model move.

We define the auxiliary function  $deviationMv : ((\mathcal{A} \cup \{\gg\}) \times (\mathcal{S}(\Lambda) \cup \{\gg\})) \rightarrow \mathbb{B}$  that returns *true* if the given alignment move is either a log move or a visible model move.

$$deviationMv(\gamma(i)) = \begin{cases} true & logMv(\gamma(i)) \vee visModelMv(\gamma(i)) \\ false & \text{otherwise} \end{cases}$$

Subsequently, we define the auxiliary function  $deviation : \Gamma \rightarrow \mathbb{B}$  that returns *true* if an alignment indicates a deviation, i.e., there exists at least one log move or visible model move.

$$deviation(\gamma) = \begin{cases} true & \text{if } \exists 1 \leq i \leq |\gamma| (deviationMv(\gamma(i))) \\ false & \text{otherwise} \end{cases}$$

Finally, we define the partial function  $firstDeviationMvIndex : \Gamma \dashrightarrow \mathbb{N}$  that returns the index of the first deviation-indicating alignment move from a given alignment if such move exists.

$$firstDeviationMvIndex(\gamma) = i \in \{1, \dots, |\gamma|\} \text{ such that}$$

$$deviationMv(\gamma(i)) \wedge \nexists 0 \leq j < i (deviationMv(\gamma(j)))$$

### 3.5.3. Computing Alignments

The computation of alignments can be reduced to a shortest path problem [6, 45, 230]. This section introduces the definition of the Synchronous Product Net (SPN), a WF-net defined for a given process model and a trace for which an alignment should be computed. The SPN's state space defines the search space of the shortest path problem. The SPN itself is composed of a *trace net*, encoding the given trace as a WF-net, and a process



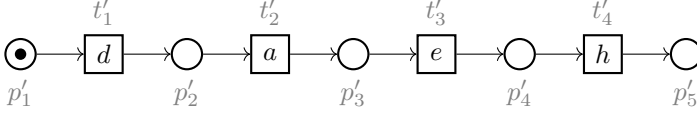


Figure 3.14: Example trace net  $N_{\sigma_1}$  for the trace  $\sigma_1 = \langle d, a, e, h \rangle$  (adapted from [180, Figure 5])

model provided as WF-net.<sup>7</sup> Further, we assume in this chapter that all WF-nets are sound; note that trace nets are sound by definition Definition 3.35. Consider the trace  $\sigma_1 = \langle d, a, e, h \rangle$ . Figure 3.14 depicts the corresponding trace net. A trace net encodes a given trace in a Petri net; thus, each transition represents an activity from the trace. Below, we formally specify a trace net for a given trace.

**Definition 3.35** (Trace net)

Let  $\sigma \in \mathcal{A}^*$  with length  $n = |\sigma|$ . We define the corresponding trace net  $N_\sigma = (P_\sigma, T_\sigma, F_\sigma, \lambda_\sigma, M_\sigma^{init}, M_\sigma^{final})$  with:

- $P_\sigma = \{p_i \mid 1 \leq i \leq n+1\}$ ,
- $T_\sigma = \{t_i \mid 1 \leq i \leq n\}$ ,
- $F_\sigma = \{(p_i, t_i) \mid 1 \leq i \leq n\} \cup \{(t_i, p_{i+1}) \mid 1 \leq i \leq n\}$ ,
- $\lambda(p_i) = \sigma(i)$  for  $1 \leq i \leq n$ ,
- $M^{init} = [p_1]$ , and
- $M^{final} = [p_{n+1}]$ .

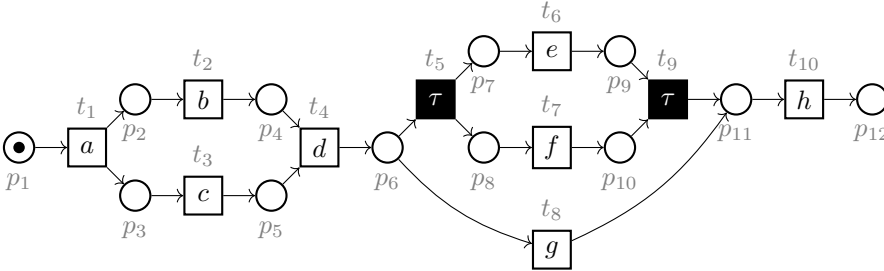


Figure 3.15: Example WF-net  $N_1$  with  $M^{init} = [p_1]$  and  $M^{final} = [p_{12}]$  (partly adapted from [180, Figure 2])

Next to the example trace  $\sigma_1 = \langle d, a, e, h \rangle$  and the corresponding trace net, consider the example WF-net shown in Figure 3.15. Given the trace net (cf. Figure 3.14) and the process model (cf. Figure 3.15), we can construct the SPN, which is illustrated in

<sup>7</sup>Note that process trees can be easily translated into WF-nets, cf. Table 3.2 (page 67). Therefore, we present the computation of alignments for WF-nets.

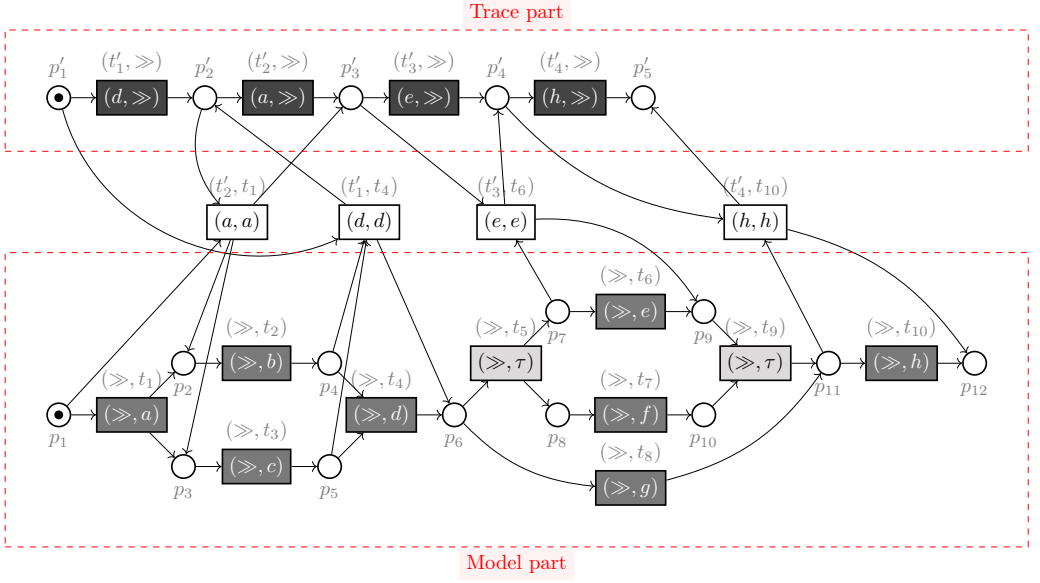


Figure 3.16: Example SPN using the trace net depicted in Figure 3.14 and the process model shown in Figure 3.15; each transition corresponds to an alignment move (partly adapted from [180, Figure 2])

Figure 3.16. Each transition in the SPN corresponds to an alignment move. We highlight the transitions according to the alignment move they represent—we use the same colors as in Section 3.5 (page 71 ff.). Two parts within an SPN can be distinguished.

1. The *trace part* (cf. Figure 3.16) is constructed from the provided trace net and contains solely transitions representing log moves.
2. The *model part* (cf. Figure 3.16) is constructed from the provided WF-net and contains solely visible and invisible model moves.

Between the trace and model part, transitions solely representing synchronous moves are placed. These transitions are, in general, the only elements connecting the trace and model part.

A complete alignment for the trace  $\sigma_1 = \langle d, a, e, h \rangle$  and the WF-net depicted in Figure 3.15 corresponds to a firing sequence from the initial marking  $[p'_1, p_1]$  to the final marking  $[p'_5, p_{12}]$  in the SPN depicted in Figure 3.16. Using a *cost function* that assigns cost to each alignment move respectively each transition in an SPN, we can compute an *optimal* complete alignment. In this thesis, we assume the *standard cost function* that assigns cost 1 to deviation-indicating alignment moves (i.e., log moves and visible model moves) and cost 0 to non-deviation-indicating alignment moves (i.e., invisible

model moves and synchronous moves) [45].<sup>8</sup> The actual computation of a complete alignment reduces to a shortest path problem on the state space of the SPN. Since moving from one state to another—which can only be done by firing a transition—always has associated costs assigned by the provided cost function, the problem of computing optimal alignments is a shortest path problem. Various established search algorithms exist, like Dijkstra’s algorithm [68] or the A\* algorithm [101]; the latter is often used for alignment computation [6, 45]. Below, we define the SPN for a given trace net and WF-net.

**Definition 3.36** (Synchronous product net)

For a given trace  $\sigma \in \mathcal{A}^*$ , the corresponding trace net  $N_\sigma = (P_\sigma, T_\sigma, F_\sigma, \lambda_\sigma, M_\sigma^{init}, M_\sigma^{final})$  and a WF-net  $N = (P, T, F, \lambda, M^{init}, M^{final})$  such that  $P^\sigma \cap P = \emptyset$  and  $T^\sigma \cap T = \emptyset$ , we define the SPN  $N_S = (P_S, T_S, F_S, \lambda_S, M_S^{init}, M_S^{final})$  with:

- $P_S = P_\sigma \cup P$ ,
- $T_S = (T_\sigma \times \{\gg\}) \cup (\{\gg\} \times T) \cup \{(t', t) \in T_\sigma \times T \mid \lambda(t) = \lambda_\sigma(t') \neq \tau\}$ ,
- $F_S = \left\{ (p, (t', t)) \in P_S \times T_S \mid (p, t') \in F_\sigma \vee (p, t) \in F \right\} \cup$   
 $\left\{ ((t', t), p) \in T_S \times P^S \mid (t', p) \in F_\sigma \vee (t, p) \in F \right\}$
- $\lambda^S : T^S \rightarrow (\mathcal{A} \cup \{\tau, \gg\}) \times (\mathcal{A} \cup \{\tau, \gg\})$  (assuming  $\gg \notin \mathcal{A} \cup \{\tau\}$ ) s.t.:
  - $\lambda^S((t', \gg)) = (\lambda^\sigma(t'), \gg)$  for  $t' \in T_\sigma$
  - $\lambda^S((\gg, t)) = (\gg, \lambda(t))$  for  $t \in T$
  - $\lambda^S((t', t)) = (\lambda^\sigma(t'), \lambda(t))$  for  $t' \in T_\sigma, t \in T$ ,
- $M_S^{init} = M_\sigma^{init} \uplus M^{init}$ , and
- $M_S^{final} = M_\sigma^{final} \uplus M^{final}$ .

In short, alignments are a state-of-the-art conformance-checking technique. They provide detailed information on how and where observed and modeled process behavior diverge. We introduce alignments since they are integral for incremental process discovery. For example, they can help determine if an intermediate process model already covers the user-selected process behavior and which parts of a process model need to be changed. Moreover, in the subsequent chapter, we extend alignments as introduced in this section for trace fragments, i.e., trace prefixes, infixes, and suffixes. Alignments for trace fragments are required to support trace fragments in incremental process discovery.

<sup>8</sup>Note that when implementing alignment computation, invisible model moves are often assigned very low costs (i.e.,  $0 + \epsilon$  for some  $0 < \epsilon \ll 1$ ) to avoid infinite alignments. This can happen if the process model allows loops on  $\tau$ -labeled activities.



---

## Chapter 4.

# Alignments for Trace Fragments

---

This chapter is largely based on the following published work.

- D. Schuster, N. Föcking, S. J. van Zelst, and W. M. P. van der Aalst. Conformance checking for trace fragments using infix and postfix alignments. In M. Sellami, P. Ceravolo, H. A. Reijers, W. Gaaloul, and H. Panetto, editors, *Cooperative Information Systems*, volume 13591 of *Lecture Notes in Computer Science*, pages 299–310. Springer, 2022. doi:10.1007/978-3-031-17834-4\_18 [180]

Alignments, as introduced in Section 3.5, are usually defined for complete traces, i.e., traces that span the process from start to completion. This chapter extends the widely used conformance checking technique alignments [6, 45, 226] for *trace fragments*, which comprises trace prefixes, infixes, and postfixes. Prefix alignments for comparing a trace prefix with a process model were introduced in [6]. Prefix alignments are, for instance, used in *online conformance checking* [41] where event streams rather than event logs are considered [173, 237, 250, 251]. However, alignments for trace infixes and postfixes do not exist. Therefore, this chapter introduces infix and postfix alignments to support trace fragments fully.

Computing alignments for trace fragments is paramount for the proposed incremental process discovery approach that will be introduced in Chapter 6. Trace fragments are an essential artifact in incremental process discovery as they give users more flexibility in discovering processes than the exclusive focus on full traces, as often assumed in process discovery. Thus, conformance checking techniques supporting trace fragments are essential in the context of this thesis.

Even though we motivate and leverage infix and postfix alignments in this thesis in the context of incremental process discovery, the proposed alignments for trace fragments can be used in a broader context for general conformance-checking purposes. For example, processes can often be divided into stages, each representing different logical or temporal phases. Consequently, conformance requirements might differ per stage; conformance-critical and conformance-uncritical stages might exist. Thus, conformance checking for trace fragments covering conformance-critical process stages is very important. Furthermore, event data must often be extracted and combined from various data sources to analyze a process holistically because multiple information systems are involved in the

execution of a process. In such scenarios, conformance-checking techniques for trace fragments are valuable as these do not require complete traces as input and can be directly applied to trace fragments. In short, alignments for trace fragments are a valuable extension and may be applied not only in incremental process discovery.

## 4.1. Overview

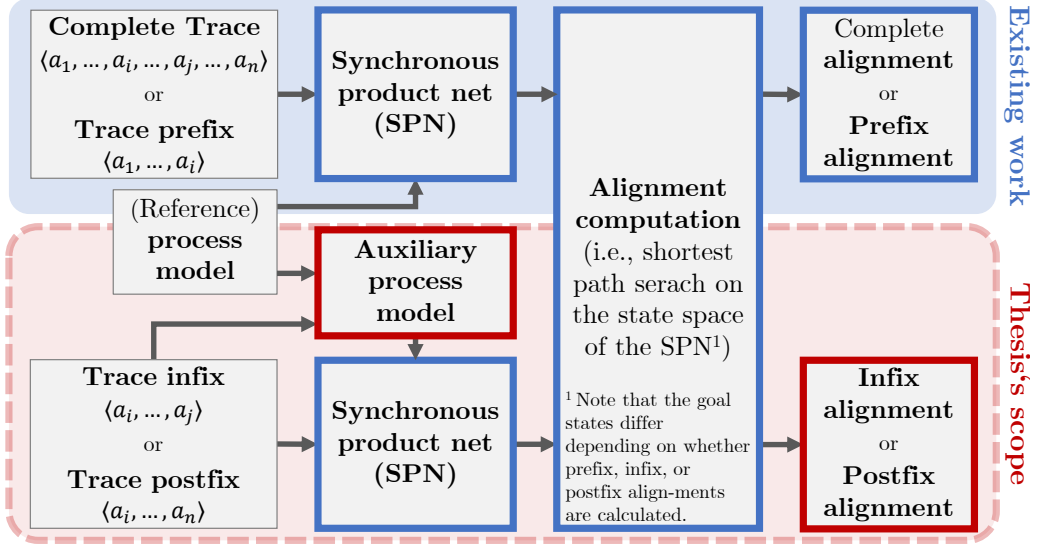


Figure 4.1: Overview of alignment computation and this chapter's contributions regarding infix and postfix alignments (partly adapted from [180, Figure 1])

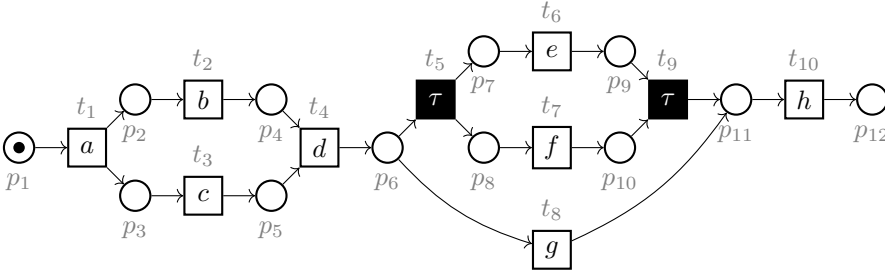
Figure 4.1 provides a high-level overview of alignment computation. As presented in Section 3.5, a SPN is calculated for a given trace and process model. The state-space defined by the SPN represents the search space of the alignment computation for the given trace and model. This described procedure is applied for complete as well as for prefix alignments. However, when computing infix and postfix alignments, we cannot use the SPN directly. Therefore, we modify the SPN to adapt it for infix and postfix alignment computation by creating an *auxiliary process model* from the (reference) process model. We use this auxiliary process model for the SPN generation instead of the provided (reference) process model. The SPN generated from the auxiliary process model allows us to rely on the established alignment computation machinery without adapting further established concepts and techniques, cf. Figure 4.1. Thus, the search for the shortest path on the state space of the SPN remains unchanged compared to the computation of the complete and prefix alignment except for modifying goal states. Therefore, the main contribution of this chapter is the creation of said auxiliary process model for computing infix and postfix alignments. We propose two approaches to derive an *auxiliary process*

model from a given reference process model. The first approach assumes WF-nets as a (reference) process model, while the second assumes process trees. We evaluate these two approaches on publicly available event data capturing real-life processes.

This chapter's remainder is organized as follows. Section 4.2 defines prefix, infix and postfix alignments. Section 4.3 presents the computation of infix and postfix alignments. We evaluate infix and postfix alignments in Section 4.4. Finally, Section 4.5 concludes this section.

## 4.2. Defining Prefix, Infix & Postfix Alignments

This section formally defines alignments for trace fragments, i.e., prefix, infix and postfix alignments. Note that prefix alignments have already been defined in [6]. First, we provide an informal introduction to prefix, infix, and postfix alignments and present examples. Subsequently, we present corresponding definitions.



(a) WF-net  $N_1$  with  $M^{init} = [p_1]$  and  $M^{final} = [p_{12}]$  (partly adapted from [180, Figure 2])

$$\gamma_1 = \begin{array}{|c|c|} \hline d & g \\ \hline t_4 & t_8 \\ \hline (d) & (g) \\ \hline \end{array}$$

(b) An optimal infix alignment  $\gamma_1$  for trace infix  $\sigma_1 = \langle d, g \rangle$

$$\gamma_2 = \begin{array}{|c|c|c|c|c|} \hline b & d & \gg & f & g \\ \hline t_2 & t_4 & t_5 & t_7 & \\ \hline (b) & (d) & (\tau) & (f) & \gg \\ \hline \end{array}$$

(c) An optimal infix alignment  $\gamma_2$  for trace infix  $\sigma_2 = \langle b, d, f, g \rangle$

Figure 4.2: Optimal infix alignments (Figures 4.2b and 4.2c) for WF-net  $N_1$  (Figure 4.2a)

The conceptual idea of an infix alignment is to align a given trace infix against an infix of the WF-net's language. The model part of an infix alignment starts at some marking that is reachable from the initial marking of the given WF-net and ends at an arbitrary marking of the WF-net. Consequently, every complete alignment is also an infix alignment, but not vice versa. Figure 4.2 depicts two infix alignments for the WF-net shown in Figure 3.15 and two different trace infixes. As for complete alignments (cf. Definition 3.33), the first row of an infix alignment corresponds to the given trace infix when we ignore the skip symbol  $\gg$ . The second row of an infix alignment corresponds to a firing sequence (ignoring  $\gg$ ) starting from a marking that is reachable from the initial

marking of the provided WF-net. Recall infix alignment  $\gamma_1$  depicted in Figure 4.2b. Infix alignment  $\gamma_1$  specifies the firing sequence  $\langle t_4, t_8 \rangle$ . Given the initial marking  $M^{init} = [p_1]$  of the WF-net depicted in Figure 3.15, only one marking exists in the set of reachable markings that enables transition  $t_4$ , i.e.,  $[p_4, p_5] \in \mathcal{RM}(N_1, [p_1])$ . Hence, infix alignment  $\gamma_1$  indicates in its model part to start at marking  $[p_4, p_5]$  and execute transitions  $t_4$  and  $t_8$ ; thus,

$$(N_1, [p_1]) \rightsquigarrow (N_1, [p_4, p_5]) \xrightarrow{\langle t_4, t_8 \rangle} (N_1, [p_{11}]) \rightsquigarrow (N_1, [p_{12}]).$$

Similarly, the model part of infix alignment  $\gamma_2$  (cf. Figure 4.2c) indicates that:

$$(N_1, [p_1]) \rightsquigarrow (N_1, [p_2, p_5]) \xrightarrow{\langle t_2, t_4, t_5, t_7 \rangle} (N_1, [p_7, p_{10}]) \rightsquigarrow (N_1, [p_{12}]).$$

$$\gamma_3 =$$

$d$	$g$	$\gg$
$t_4$	$t_8$	$t_{10}$
$(d)$	$(g)$	$(h)$

(a) An optimal postfix alignment  $\gamma_3$  for trace postfix  $\sigma_3 = \langle d, g \rangle$

$$\gamma_4 =$$

$a$	$d$	$g$	$\gg$
$\gg$	$t_4$	$t_8$	$t_{10}$
$\gg$	$(d)$	$(g)$	$(h)$

(b) An optimal postfix alignment  $\gamma_4$  for trace postfix  $\sigma_4 = \langle a, d, g \rangle$

Figure 4.3: Optimal postfix alignments for WF-net  $N_1$  (Figure 4.2a)

The definition of postfix alignments is based on the same concept as that of infix alignments. The model part of a postfix alignment starts at a reachable marking from the given WF-net's initial marking; however, it must end in its final marking. Figure 4.3 shows examples of optimal postfix alignments for the WF-net  $N_1$  depicted in Figure 3.15. For instance, the model part of postfix alignment  $\gamma_3$  indicates that:

$$(N_1, [p_1]) \rightsquigarrow (N_1, [p_4, p_5]) \xrightarrow{\langle t_4, t_8, t_{10} \rangle} (N_1, [p_{12}]).$$

The model part of  $\gamma_4$  (cf. Figure 4.3b) indicates that:

$$(N_1, [p_1]) \rightsquigarrow (N_1, [p_2, p_5]) \xrightarrow{\langle t_2, t_4, t_5, t_7 \rangle} (N_1, [p_{12}]).$$

Subsequently, we define prefix, infix, and postfix alignments for Petri nets.<sup>1</sup> Compared to the definition of complete alignments (cf. Definition 3.33 on page 73), the definition for prefix, infix, and postfix alignments differs in the second requirement.

<sup>1</sup> Note that the prefix alignments have already been defined in [6]. However, the definitions for infix and postfix alignments are considered a contribution of this thesis.



**Definition 4.1** (Prefix, Infix, and Postfix Alignment for Petri Nets)

Let  $N = (P, T, F, \lambda, M^{init}, M^{final}) \in \mathcal{W}$  be a sound WF-net and  $\sigma \in \mathcal{A}^*$  be a trace. A sequence  $\gamma \in \left( (\mathcal{A} \cup \{\gg\}) \times (T \times \{\gg\}) \right)^*$  is a prefix/infix/postfix alignment if the subsequent constraints are satisfied.

1.  $\sigma = \pi_1^*(\gamma) \downarrow_{\mathcal{A}}$
2.
  - It is a **prefix alignment** if there exists a  $M' \in \mathcal{RM}(N, M^{init})$  such that  $(N, M^{init}) \xrightarrow{\pi_2^*(\gamma) \downarrow_T} (N, M') \rightsquigarrow (N, M^{final})$ .
  - It is an **infix alignment** if there exists a  $M', M'' \in \mathcal{RM}(N, M^{init})$  such that  $(N, M^{init}) \rightsquigarrow (N, M') \xrightarrow{\pi_2^*(\gamma) \downarrow_T} (N, M'') \rightsquigarrow (N, M^{final})$ .
  - It is a **postfix alignment** if there exists a  $M' \in \mathcal{RM}(N, M^{init})$  such that  $(N, M^{init}) \rightsquigarrow (N, M') \xrightarrow{\pi_2^*(\gamma) \downarrow_T} (N, M^{final})$ .
3.  $\forall a \in \mathcal{A} \forall t \in T [(a, t) \in \gamma \Rightarrow a = \lambda(t)]$
4.  $(\gg, \gg) \notin \gamma$

For  $N$  and  $\sigma$ , we denote the set of

- prefix alignments by  $\Gamma_{pre}(N, \sigma)$ , optimal prefix alignments by  $\Gamma_{pre}^{opt}(N, \sigma)$ ,
- infix alignments by  $\Gamma_{inf}(N, \sigma)$ , optimal infix alignments by  $\Gamma_{inf}^{opt}(N, \sigma)$ ,
- postfix alignments by  $\Gamma_{pos}(N, \sigma)$ , and optimal postfix alignments by  $\Gamma_{pos}^{opt}(N, \sigma)$ .

Moreover, we denote the universe of

- prefix alignments by  $\Gamma_{pre}$ , optimal prefix alignments by  $\Gamma_{pre}^{opt}$ ,
- infix alignments by  $\Gamma_{inf}$ , optimal infix alignments by  $\Gamma_{inf}^{opt}$ ,
- postfix alignments by  $\Gamma_{pos}$ , and optimal postfix alignments by  $\Gamma_{pos}^{opt}$ .

As for complete alignments, *optimality* applies equally to prefix, infix, and postfix alignments. In general, a complete/prefix/infix/postfix alignment is optimal if no other complete/prefix/infix/postfix alignment exists with fewer log and visible model moves, i.e., an optimal complete/prefix/infix/postfix alignment is cost-minimal regarding the standard cost function, cf. Section 3.5.3.

### 4.3. Computing Infix & Postfix Alignments

Computing complete and prefix alignments is done by solving the shortest path problem on the corresponding SPN as defined in Section 3.5.3. For a detailed introduction, we refer to [6, 45]. However, it is not possible to calculate infix and postfix alignments in the same manner. Recall Definition 4.1. For both infix and postfix alignments, the transition sequence from the second row of the alignment starts in some marking  $M'$  that is reachable from the initial marking  $M^{init}$ . For instance, reconsider the discussed examples of infix and postfix alignments in Figure 4.3. Therefore, we cannot use the SPN as defined in Definition 3.36 (page 79) because the SPN's initial marking requires starting in its model part at the initial marking of the provided model. However, according to Definition 4.1, we must start at a marking  $M'$  that is reachable from  $M^{init}$  in the model part.

We solve this issue by modifying the SPN. Reconsider Figure 4.1 (page 82) outlining our approach to compute infix and postfix alignments. We achieve the SPN modification indirectly by using an *auxiliary process model* instead of the given process model for constructing the SPN. The SPN constructed using the auxiliary process model allows to jump to a marking in the model part that is reachable from the initial model marking and serves as the starting point for the infix/postfix alignment to be computed, i.e., marking  $M'$  in Definition 4.1. In the remainder of this section, we refer to candidate markings for  $M'$  (cf. Definition 4.1) as *relevant markings*.

**Definition 4.2** (Relevant markings for infix/postfix alignment computation)

Let  $N = (P, T, F, \lambda, M^{init}, M^{final})$  be a sound WF-net and  $\sigma \in \mathcal{A}^*$  be a trace infix/postfix. We define relevant markings  $\{M_1, \dots, M_m\} \subseteq \mathcal{RM}(N, M^{init})$  that are candidate markings for  $M'$  for optimal infix  $\gamma \in \Gamma_{inf}^{opt}(N, \sigma)$  or postfix alignments  $\gamma \in \Gamma_{pos}^{opt}(N, \sigma)$ , cf. Definition 4.1. Thus,  $\{M_1, \dots, M_m\} \subseteq \mathcal{RM}(N, M^{init})$  are relevant markings if  $M' \in \{M_1, \dots, M_m\}$ .

The central research question is how to (efficiently) determine relevant markings from a given WF-net. We summarize the overall approach for computing infix/postfix alignments below and in Figure 4.4.

1. Create an *auxiliary* WF-net from the provided reference WF-net.
  - a) Calculate *relevant markings* in the given WF-net that may represent the start of an infix/postfix alignment in the model part, cf. marking  $M'$  in Definition 4.1.
  - b) Create the auxiliary WF-net using the relevant markings
2. Construct the SPN, according to Definition 3.36, using the *auxiliary* WF-net constructed in the previous step and the given trace infix/postfix.
3. Perform a shortest path search on the SPN's state space with corresponding final markings, i.e., goal states regarding the shortest path search.<sup>2</sup>

<sup>2</sup>The actual shortest path search on the state space of the SPN remains unchanged compared to computing complete/prefix alignments, cf. Figure 4.1 (page 82); however, the goal states differ.

- **Infix alignment:** all markings of the SPN that mark the last place of its trace net part are goal markings for the shortest path search
  - **Postfix alignment:** the goal marking is the SPN's final marking as specified in Definition 3.36
4. Post-process the calculated infix/postfix alignment because the infix/postfix alignment aligns the given trace infix/postfix with the auxiliary WF-net and not the original WF-net. Thus, we remove the alignment moves that result from using the auxiliary WF-net instead of the provided WF-net.

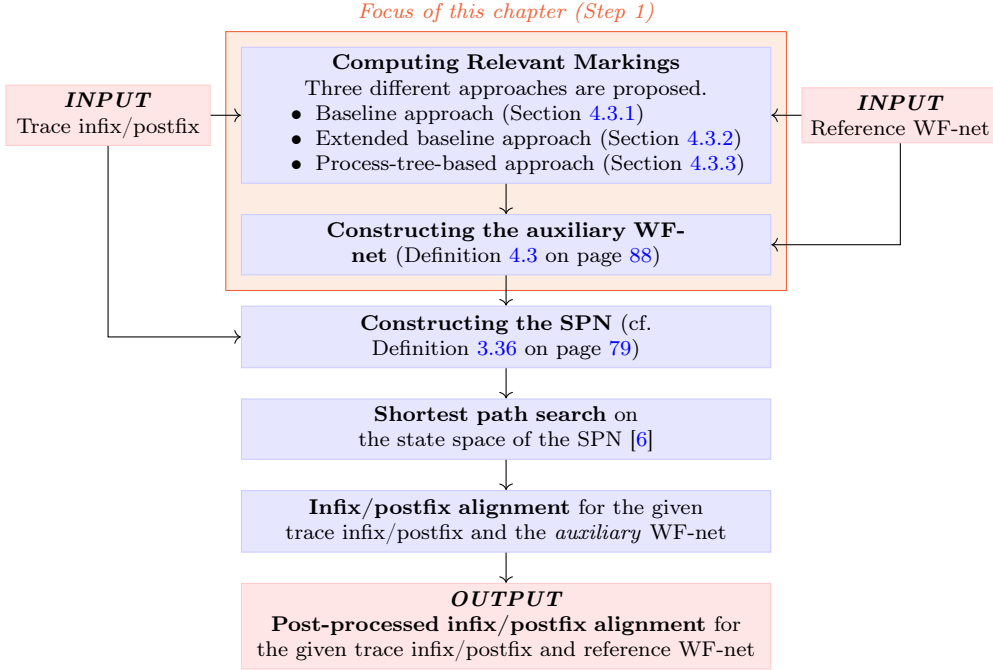


Figure 4.4: Overview of computing infix and postfix alignments

As described above, the first step is essential, i.e., the creation of the auxiliary WF-net. The subsequent SPN generation using the auxiliary WF-net and the trace infix/postfix, i.e., Step 2, is performed as specified in Definition 3.36. Next, the shortest path search is executed, i.e., Step 3. Note that compared to complete/infix alignments, the goal marking(s) differ as described above. However, apart from different goal markings, the shortest path problem remains and can be solved similarly to complete/prefix alignments. Finally, the computed infix-/postfix alignment is post-processed.

In the remainder of this section, we focus on constructing the auxiliary process model and present two approaches for determining relevant markings. We present a naive approach in Section 4.3.1 that we improve in Section 4.3.2. Finally, Section 4.3.3 proposes a process-tree-based approach tailored to process trees for computing relevant markings needed to create the auxiliary process model.

### 4.3.1. Baseline Approach

This section proposes a baseline approach to compute an auxiliary WF-net. The proposed baseline approach assumes an arbitrary, sound WF-net  $N = (P, T, F, \lambda, M^{init}, M^{final})$  as input. Since sound WF-nets are *bounded* by definition [224], their state space is finite. Hence, the following equation holds.

$$|\mathcal{RM}(N, M^{init})| = n \in \mathbb{N}$$

Further, we can list all markings of  $N$  that are reachable from the initial marking, i.e.,  $\mathcal{RM}(N, M^{init}) = \{M_1, \dots, M_n\}$ . For instance, for WF-net  $N_1$  depicted in Figure 3.15 (page 77), the set of markings reachable from the initial one

$$\begin{aligned} \mathcal{RM}(N, M^{init}) = \{ & M_1=[p_1], \quad M_2=[p_2, p_3], \quad M_3=[p_2, p_5], \quad M_4=[p_3, p_4], \quad M_5=[p_4, p_5], \\ & M_6=[p_6], \quad M_7=[p_7, p_8], \quad M_8=[p_8, p_9], \quad M_9=[p_7, p_{10}], \quad M_{10}=[p_9, p_{10}], \\ & M_{11}=[p_{11}], \quad M_{12}=[p_{12}]\}. \end{aligned}$$

Thus, twelve reachable markings including the initial marking  $M^{init} = [p_1]$  exist, i.e.,  $|\mathcal{RM}(N, M^{init})| = 12$ . The *baseline approach* considers all these reachable markings as relevant markings, i.e., candidates for marking  $M'$  when computing infix/postfix alignments, cf. Definition 4.1 (page 85). Obviously, considering all reachable markings as relevant markings is feasible according to Definition 4.2 because it is guaranteed that one marking from the set of all reachable markings is a candidate marking for  $M'$ .

The corresponding auxiliary process model, referred to as  $N_{aux}$ , comprises the provided WF-net  $N_1$ , a new place  $p'_0$ , and twelve transitions, i.e., for each relevant marking, that connect place  $p'_0$  with the places specified in the corresponding relevant marking. Further, the initial marking of the auxiliary WF-net is updated to  $M_{aux}^{init} = [p'_0]$ . Consider Figure 4.5 showing the auxiliary WF-net  $N_{1_{aux}}$  for the WF-net  $N_1$  depicted in Figure 3.15 (page 77). Blue highlighted elements, comprising the new place  $p'_0$ , transitions, and arcs, have been added to the provided WF-net. For each relevant marking, a transition is added, i.e.,  $t'_1, \dots, t'_{12}$ . Further, each added transition is enabled in the initial marking  $M_{aux}^{init} = [p'_0]$  and, upon firing, yields a relevant marking. No further transitions are enabled by the initial marking. For instance, transition  $t'_4$  is enabled and yields upon firing the relevant marking  $M_4 = [p_3, p_4]$ , cf. Figure 4.5. Adding a silent transition for each relevant marking allows the auxiliary WF-net to reach one of the relevant markings from the initial marking. In short, the auxiliary WF-net allows once, i.e., from its initial marking  $M_{aux}^{init} = [p'_0]$ , to yield any relevant marking by firing one of the added silent transitions, i.e.,  $t'_1, \dots, t'_{12}$ . After firing one of the added transitions, the auxiliary WF-net behaves as the provided WF-net. Below, a formal definition of an auxiliary WF-net is provided that assumes a WF-net and corresponding relevant markings as input.

**Definition 4.3** (Auxiliary WF-net)

Let  $N = (P, T, F, \lambda, M^{init}, M^{final})$  be a WF-net and  $\{M_1, \dots, M_n\} \subseteq \mathcal{RM}(N, M^{init})$  be the set of relevant markings. We define the auxiliary WF-net  $N_{aux} = (P_{aux}, T_{aux}, F_{aux}, \lambda_{aux}, M_{aux}^{init}, M_{aux}^{final})$  with:

- $P_{aux} = P \cup \{p'_0\}$  (assuming that  $p'_0 \notin P$ )
- $T_{aux} = T \cup \{t'_i | 1 \leq i \leq n\}$  (assuming that  $t'_i \notin T$  for all  $1 \leq i \leq n$ )
- $F_{aux} = F \cup \{(p'_0, t'_i) | 1 \leq i \leq n\} \cup \{(t'_i, p) | 1 \leq i \leq n \wedge p \in M_i\}$
- $\lambda_{aux}(t) = \begin{cases} \lambda(t) & \text{if } t \in T \\ \tau & \text{otherwise (i.e., } t \in T_{aux} \setminus T) \end{cases}$
- $M_{aux}^{init} = [p'_0]$
- $M_{aux}^{final} = M^{final}$

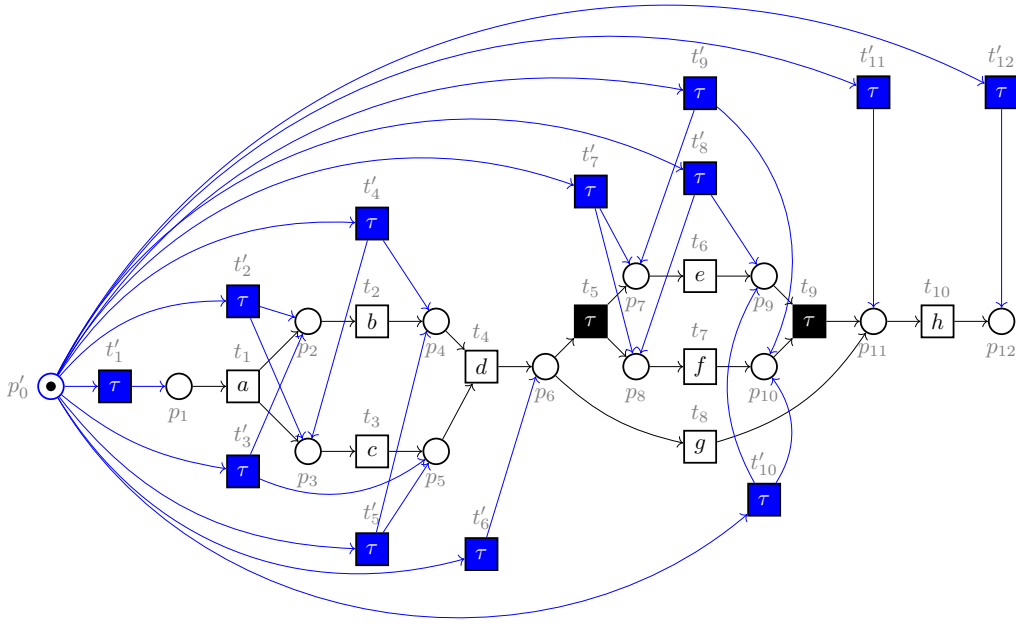


Figure 4.5: Auxiliary WF-net  $N_{1_{aux}}$  generated by the *baseline approach* for WF-net  $N_1$  (Figure 4.2a); blue highlighted elements (i.e., places, transitions, arcs) were added to the provided WF-net depicted in Figure 3.15 (partly adapted from [180, Figure 8])

When using the auxiliary WF-net depicted in Figure 4.5 for the construction of an SPN, the added silent transitions, i.e.,  $t'_1, \dots, t'_{12}$ , turn into invisible model moves, which do not indicate a deviation and are therefore assigned no cost according to the standard cost function (Section 3.5.3 page 76 ff.). Thus, in the model part of the corresponding SPN, we must first fire one of the transitions representing invisible model moves on the added transitions to proceed. As a result of firing one of the added silent transitions, we

yield a relevant marking in the model part. Since the auxiliary WF-net has a fixed initial marking and allows to reach any relevant marking, the computation for infix/postfix alignments can be executed as for complete/prefix alignments, cf. Section 3.5.3.

### 4.3.2. Extended Baseline Approach Using Subsequent Filtering

The proposed baseline approach, cf. Section 4.3.1, can be further improved for computing relevant markings used eventually for constructing the auxiliary WF-net. This section extends the previously presented baseline approach by a subsequently applied filtering function. First, all reachable markings are computed as before. Subsequently, the actual trace infix/postfix is considered for which an infix/postfix alignment should be computed. Each reachable marking is considered and checked, which transitions the marking enables. If all transitions enabled are not labeled with any label from the given trace infix/postfix, we can remove this marking from the set of reachable markings. Equation (4.1) formally specifies the set of relevant markings using the described filtering.

$$\left\{ M \in \mathcal{RM}(N, M^{init}) \mid \exists t \in T \left( (N, M)[t] \wedge (\lambda(t) \in \sigma \vee \lambda(t) = \tau) \right) \right\} \cup \left\{ M^{final} \right\} \quad (4.1)$$

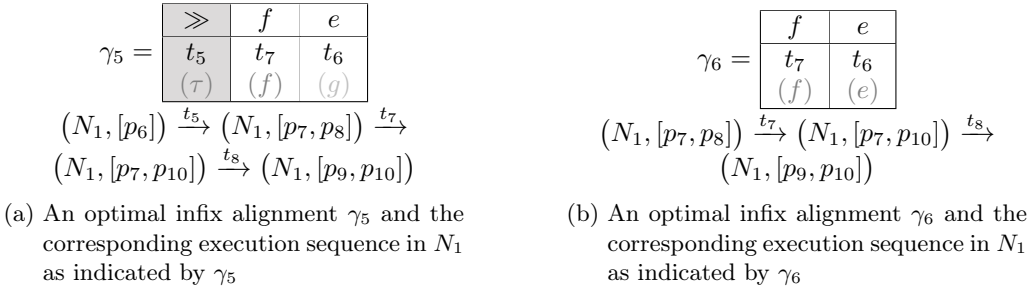
It is permissible to ignore all markings that solely enable visible transitions whose label is not contained in the trace infix/postfix because we aim to compute only optimal infix/postfix alignments. However, as long as a marking enables a transition whose label is contained in the provided trace infix/postfix or whose label is  $\tau$ , we consider the marking relevant. Any reachable marking that enables solely visible transitions whose label is not contained in the trace infix/postfix cannot be the start of an optimal infix/postfix alignment. Starting in a marking that enables only visible transitions whose labels are not contained in the trace infix/postfix leads to a visible model move to proceed in the SPN; hence, such markings cannot be relevant starting markings for optimal infix/postfix alignment computation.

In the context of incremental process discovery, we are not interested in all potential optimal prefix/infix/postfix alignments for a given trace fragment and process model. Instead, a single optimal prefix/infix/postfix alignment for a given trace fragment and model is sufficient. To this end, we can further reduce the number of relevant markings as specified above. We exclude all transitions that do not enable any visible transition whose label is contained in the provided trace fragment  $\sigma$ . Thus, we ignore if a marking enables a silent transition. Equation (4.2) specifies the modified set of relevant markings compared to Equation (4.1).

$$\left\{ M \in \mathcal{RM}(N, M^{init}) \mid \exists t \in T \left( (N, M)[t] \wedge (\lambda(t) \in \sigma) \right) \right\} \cup \left\{ M^{final} \right\} \quad (4.2)$$

Note that when using the above-specified set of relevant markings (cf. Equation (4.2)), not all optimal prefix/infix/postfix alignment might be found for a given trace fragment and process model. Since markings are removed that, for example, only enable silent transitions, not all optimal prefix/infix/postfix alignments can be found.<sup>3</sup>

<sup>3</sup> Recall that this thesis assumes the standard cost function, cf. Section 3.5.3. Thus, invisible model moves have cost zero.


 Figure 4.6: Optimal infix alignments for WF-net  $N_1$  (Figure 4.2a) and trace infix  $\langle f, e \rangle$ 

For example, reconsider  $N_1$  (cf. Figure 4.2a) and the trace infix  $\sigma = \langle f, e \rangle$ . Figure 4.6 depicts two corresponding optimal infix alignments; both alignments have cost zero according to the standard cost function. Alignment  $\gamma_5$  contains one invisible model move and two synchronous moves, while  $\gamma_6$  contains solely synchronous moves. When using the relevant markings as defined in Equation (4.2), we cannot compute  $\gamma_5$  because the corresponding set of relevant markings does not include a marking enabling  $t_5$ , which is executed first in  $\gamma_5$ . Note that only one reachable marking exists that enables  $t_5$ , i.e.,  $[p_6] \in \mathcal{RM}(N_1, [p_1])$  with  $(N_1, [p_6]) [t_5]$ . Marking  $[p_6]$  is contained in the set of relevant markings as specified in Equation (4.1); however, it is not contained in the set specified by Equation (4.2). Hence, using the relevant markings as specified by Equation (4.1), we can find both shown alignments, and when using the relevant markings as specified by Equation (4.2), we can only find  $\gamma_6$  for the given example. In short, when using the relevant markings as specified in Equation (4.2), the optimal alignments found do not start with invisible model moves. However, as mentioned above, we are only interested in one optimal alignment for a given trace fragment and process model in the context of this thesis, which is why we assume Equation (4.2) in the following when referring to the extended baseline approach.

Consider the same WF-net as before, i.e.,  $N_1$  (Figure 3.15 on page 77), and the trace infix/postfix  $\sigma = \langle b, d, f \rangle$ . The relevant markings using the extended baseline approach are as follows. For the sake of comparability with the baseline approach introduced in Section 4.3.1, we cancel non-relevant markings according to the extended baseline approach.

$$\begin{aligned} \mathcal{RM}(N_1, M^{init}) = \{ & \overline{M_1=[p_1]}, M_2=[p_2, p_3], M_3=[p_2, p_5], \overline{M_4=[p_3, p_4]}, M_5=[p_4, p_5], \\ & \overline{M_6=[p_6]}, M_7=[p_7, p_8], M_8=[p_8, p_9], \overline{M_9=[p_7, p_{10}]}, \overline{M_{10}=[p_9, p_{10}]}, \\ & \overline{M_{11}=[p_{11}]}, M_{12}=[p_{12}] \} \end{aligned}$$

Figure 4.7 shows the auxiliary WF-net when using the relevant markings, as defined above. We mark out transitions that are not relevant to the extended baseline approach for comparability. Observe that the auxiliary WF-net using the extended baseline approach contains only six additional transitions. For instance, the transition  $t'_4$  is not relevant as the marking  $[p_3, p_4]$  enables only transition  $t_3$ , which is labeled  $c$ . Since a  $c$  is





Sections 4.3.1 and 4.3.2) is that the process-tree-based approach calculates relevant markings *directly* from the tree structure. In contrast, the baseline approaches exhaustively calculate all relevant markings on which the extended baseline approach subsequently applies filtering.

### Motivating Example

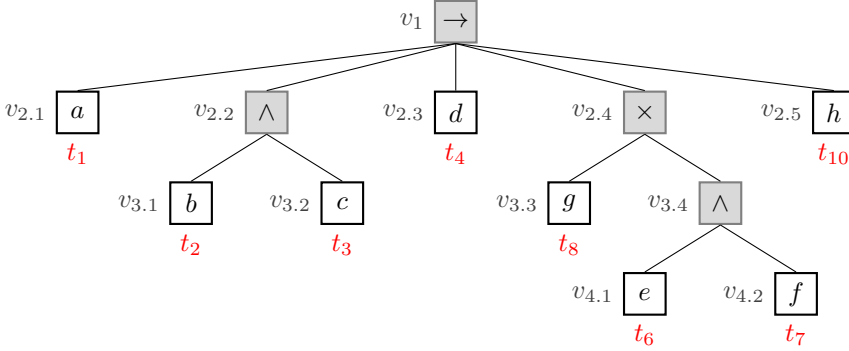


Figure 4.8: Process tree  $\Lambda_{N_1}$  corresponding to WF-net  $N_1$  (Figure 4.2a); below each leaf vertex the corresponding transition from  $N_1$  is indicated

Reconsider the WF-net  $N_1$  depicted in Figure 4.2a (page 83) and the trace postfix  $\sigma = \langle b, d, f \rangle$ . Further, reconsider the set of reachable markings  $\mathcal{RM}(N_1, M^{init})$  depicted in Section 4.3.1. Moreover, note that  $N_1$  can also be represented as a process tree, as shown in Figure 4.8. The extended baseline approach (cf. Section 4.3.2) considers marking  $M_2 = [p_2, p_3]$  as relevant because it enables transition  $t_2$  with  $\lambda(t_2) = b \in \sigma$  (cf. Figure 4.7); further,  $M_2$  enables  $t_3$  with  $\lambda(t_3) = c \notin \sigma$ . Consider the auxiliary WF-net  $N'_{aux}$  constructed using the relevant markings from the extended baseline approach, cf. Figure 4.7. Assume we fire  $t'_2$  to reach the relevant marking  $M_2 = [p_2, p_3]$  and next transition  $t_2$  with  $\lambda(t_2) = b \in \sigma$ .

$$(N'_{aux}, p'_0) \xrightarrow{t'_2} (N'_{aux}, [p_2, p_3]) \xrightarrow{t_2} (N'_{aux}, [p_3, p_4])$$

Starting in relevant marking  $M_2 = [p_2, p_3]$  works well initially because we can directly align  $\sigma$ 's first activity  $b$  by executing  $t_2$  with  $\lambda(t_2) = b$ . However, upon executing transition  $t_2$ , we reach marking  $[p_3, p_4]$ . In this marking, only transition  $t_3$  is enabled with  $\lambda(t_3) = c \notin \sigma$ . Thus, when computing an optimal postfix alignment, we need to execute next a visible model move on transition  $t_3$  to proceed in the model part of the corresponding SPN. The execution of this visible model move would increase the alignment cost by one, according to the standard cost function.

Alternatively, consider the relevant marking  $M_3 = [p_3, p_5]$ . From marking  $M_3$  we can align activity  $b$  as well by executing  $t_2$  with  $\lambda(t_2) = b = \sigma(1)$ . Further, we can, after firing  $t_2$ , immediately execute transition  $t_4$  with  $\lambda(t_4) = d = \sigma(2)$ . Thus, we avoid the invisible

4

4



4

4

## Overall Algorithm

Figure 4.10 provides an example of the process-tree-based approach applied to process tree  $\Lambda_{N_1}$  (cf. Figure 4.8) and trace infix/postfix  $\langle b, d, f \rangle$ . The process-tree-based approach generates relevant markings bottom-up. A relevant marking is generated for each leaf vertex whose label is contained in the provided infix/postfix trace. Figure 4.10 exemplifies this approach for the leaf vertex labeled  $b$  that represents transition  $t_2$  in the corresponding WF-net  $N_1$  (cf. Figure 3.15). Since the label  $\lambda(t_2) = b \in \langle b, d, f \rangle$ , we start constructing the relevant marking by considering the preset of transition  $t_2$ , i.e.,  $\bullet t_2$  (cf. Figure 4.10a). Next, we recursively consider the parent, cf. Figure 4.10b. As this is a parallel operator, we must also consider the other child vertices to obtain a valid marking that is reachable from the WF-net's initial marking. The only child vertex not considered is labeled  $c$ , cf. Figure 4.10c. Since  $c \notin \langle b, d, f \rangle$ , we only consider the postset of  $t_3$  because we want to avoid executing  $t_3$ . Next, we recursively go up the hierarchy and create a marking for the subtree rooted at the parallel operator, i.e.,  $\bullet t_2 \cup t_3 \bullet$ , cf. Figure 4.10d. Finally, we reach the root operator, cf. Figure 4.10e. Since the root vertex represents a sequence operator, the calculated marking for the subtree considered before is also a marking of the entire tree, and we store the relevant marking  $[p_2, p_5]$ . Subsequently, we start all over from the next leaf vertex labeled  $b$ ,  $d$ , or  $f$ .

---

**Algorithm 4.1:** Process-tree-based approach for calculating relevant markings for process trees, i.e., block-structured WF-nets

---

```

input  :  $\Lambda = (V, E, \Sigma, \lambda, r, <) \in \mathcal{P}$ , // process tree
 $N_\Lambda = (P_\Lambda, T_\Lambda, F_\Lambda, \lambda_\Lambda, M_\Lambda^{init}, M_\Lambda^{final})$ , // corresponding WF-net for process tree  $\Lambda$ 
 $\sigma \in \mathcal{A}^*$  // trace infix/postfix
output:  $R \subseteq \mathcal{RM}(N_\Lambda, M_\Lambda^{init})$  // relevant markings
begin
1   $R \leftarrow \{\}$  // initialize the set of relevant markings
2   $A \leftarrow \{a \in \mathcal{A} \mid a \in \sigma\}$  // store all activity labels from  $\sigma$  in set  $A$ 
3  forall  $v \in \{v \in \text{leaves}(\Lambda) \mid \lambda(v) \in A\}$  do // iterate over leaf vertices  $v$  whose
    label is contained in  $\sigma$ 
4       $R \leftarrow R \cup \text{BuMG}(\Lambda, v, \text{undefined}, N_\Lambda, \emptyset, A)$  // BuMG returns relevant
        markings  $R \in \mathcal{RM}(N_\Lambda, M_\Lambda^{init})$  such that markings in  $R$  enable the transition
        representing  $v$ 
5  return  $R \cup \{M_\Lambda^{final}\}$  // adding the final marking  $M_\Lambda^{final}$  is necessary for
    postfix alignments to enable skipping the entire model part

```

---

Algorithm 4.1 formally presents the proposed process-tree-based approach in detail. As input, a process tree  $\Lambda$ , a corresponding WF-net representing the tree  $\Lambda$ , and the trace infix/postfix  $\sigma$ . The algorithm returns the set of relevant markings  $R$ . First, the algorithm initializes the set  $R$  (line 1) and extracts the activity labels from the provided trace infix/postfix  $\sigma$  into the set  $A$  (line 2). In lines 3 and 4, the actual computation takes place. For each leaf vertex whose label is contained in the provided trace infix/postfix (line 3), the algorithm *BuMG* is called. This algorithm calculates in a bottom-up fashion

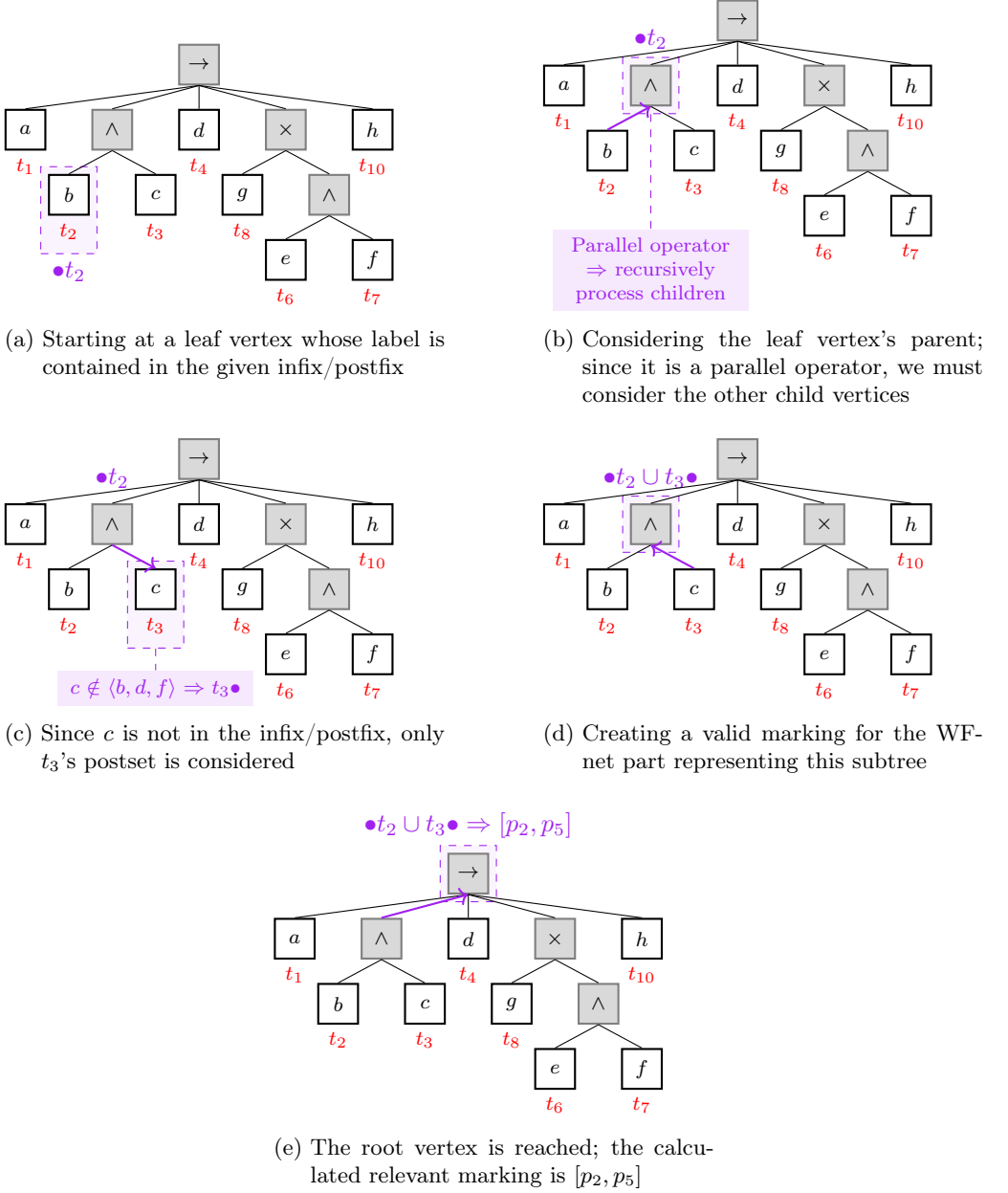


Figure 4.10: Example of the process-tree-based-approach calculating the relevant marking  $[p_2, p_5]$  for process tree  $\Lambda_{N_1}$  and trace infix/postfix  $\langle b, d, f \rangle$

for the provided leaf vertex a marking from the corresponding WF-net that enables the transition representing the leaf vertex. We introduce the *BuMG* algorithm in detail below. Finally, the calculated set of relevant markings is returned (line 5). Note that adding the final marking is needed only for postfix alignments because according to Definition 4.1 (85), postfix alignments must end in the final marking of the provided WF-net.

### Bottom-up Marking Generation Algorithm *BuMG*

Algorithm 4.2 presents the recursively defined *bottom-up marking generation algorithm BuMG* that is called in Algorithm 4.1 line 4. *BuMG* calculates for a given leaf vertex when called from Algorithm 4.1, relevant markings that enable the transition representing the provided leaf vertex  $v$ . In general, Algorithm 4.2 assumes as input:

1. the process tree  $\Lambda$  as considered in Algorithm 4.1,
2. a currently considered vertex  $v$  of  $\Lambda$  as determined in Algorithm 4.1 line 3,
3. a child vertex  $v'$  of vertex  $v$  (note that  $v'$  might be undefined),
4. a WF-net  $N_\Lambda$  representing the tree  $\Lambda$  as considered in Algorithm 4.1,
5. a set of markings  $R$  that might not necessarily be reachable markings, i.e.,  $R \subseteq \mathcal{RM}(N_\Lambda, M_\Lambda^{init})$  does *not* hold in general, because *BuMG* is a recursively defined algorithm, and
6. the set of activity labels  $A$  as calculated in Algorithm 4.1 line 2.

Starting from a leaf vertex  $v$  whose label is contained in the trace infix/postfix, when being called in Algorithm 4.1 line 4, *BuMG* recursively creates relevant markings that enable the corresponding transition in  $N_\Lambda$ . To this end, the algorithm recursively moves up the tree hierarchy from the initially provided leaf vertex  $v$  until the root vertex  $r$  is reached (line 9). First, all places having an arc towards transition  $t_v$ , which represents the leaf vertex  $v$  (line 2), are added to a marking, which is added to the set of relevant markings (line 3). Note that this marking, although it enables transition  $t_v$  is not necessarily a reachable marking from the initial marking of WF-net  $N_\Lambda$ . Thus, the marking added must be further enriched by other places.

Assume  $v$  has a parent vertex. Next, we recursively call Algorithm 4.2 on  $v$ 's parent, i.e.,  $parent_\Lambda(v)$  (line 10). We keep going up the tree hierarchy (line 10) until either (1) we reach the root vertex (line 9) or (2) we reach an inner vertex labeled with a parallel operator  $\wedge$  (line 4). In case (1), we return  $R$ . Further, at this point, all markings in  $R$  are reachable markings from  $N_\Lambda$ 's initial marking. In case (2) (line 4), i.e., we reach an inner vertex labeled with the parallel operator, we know that the markings generated so far in  $R$  are not complete, i.e., these markings cannot be reached from  $N_\Lambda$ 's initial marking. Thus, we have to consider all subtrees beneath except the one from which we reached  $v$  with (line 5).<sup>4</sup> To this end, we call the *Top-down marking generation algorithm TdMG* (line 6), which we present later in Algorithm 4.3. As input, algorithm *TdMG* is provided

<sup>4</sup>If we ignored the inner vertex labeled with the parallel operator, we would not get any markings reachable from the initial marking of  $N_\Lambda$  because the semantics of the parallel operator defines that any subtree under it can be executed concurrently, respectively interleaving. The markings generated so far only consider one subtree beneath the reached vertex labeled with the parallel operator. Thus, we must extend the markings to include tokens from places representing parts of the other subtrees that we have ignored so far.

**Algorithm 4.2:** Bottom-up marking generation (*BuMG*)

---

```

input :  $\Lambda = (V, E, \Sigma, \lambda, r, <) \in \mathcal{P}$ ,
           $v \in V$ , // currently considered vertex
           $v' \in V \cup \{\text{undefined}\}$ , // vertex to be ignored with  $v' \in \text{child}_\Lambda(v)$ 
           $N_\Lambda = (P_\Lambda, T_\Lambda, F_\Lambda, \lambda_\Lambda, M_\Lambda^{\text{init}}, M_\Lambda^{\text{final}})$ , // WF-net representing  $\Lambda$ 
           $R \subseteq \mathcal{M}(P_\Lambda)$ , // (partially complete) relevant markings
           $A \subseteq \mathcal{A}$  // activity labels contained in trace infix/postfix  $\sigma$ 
output:  $R \subseteq \mathcal{M}(P_\Lambda)$  // (partially complete) relevant markings
begin
1  if  $\lambda(v) \in \mathcal{A}$  then //  $v$  is a leaf vertex of  $\Lambda$  with a visible label
2    let  $t_v \in T_\Lambda$  be the transition representing  $v \in V$ 
3     $R \leftarrow \{[p \in \bullet t_v]\}$  // initialize  $R$  with a marking enabling  $t_v$ 
4  else if  $\lambda(v) = \wedge$  then //  $v$  represents a parallel operator
5    forall  $v_j \in \text{child}_\Lambda(v)$  with  $v_j \neq v'$  do // iterate over the children of  $v$ 
6      without  $v'$ ; thus, the siblings of  $v'$ 
7       $R_{v_j} \leftarrow \text{TdMG}(\Lambda, N_\Lambda, v_j, A, \text{true})$  // cf. Algorithm 4.3
8       $R \leftarrow R \times R_{v_1} \times \dots \times R_{v_k}$  // Cartesian product because  $\lambda(v) = \wedge$ 
9  if  $v = r$  then // vertex  $v$  is the root node of  $\Lambda$ 
10   return  $R$ 
11 else // vertex  $v$  is not the root of  $\Lambda$ 
12   return  $\text{BuMG}(\Lambda, \text{parent}_\Lambda(v), v, N_\Lambda, R, A)$  // call BuMG on  $v$ 's parent

```

---

the tree  $\Lambda$ , the corresponding WF-net  $N_\Lambda$ , the vertex  $v_j$ , the set of activity labels from the trace infix/postfix  $A$ , and a Boolean flag indicating if the marking reached after executing the part in WF-net  $N_\Lambda$  representing the subtree should be included. Algorithm *TdMG* returns a set of markings, which we have to assemble using the Cartesian product (line 7).

In summary, Algorithm 4.1 calls *BuMG* for any leaf vertex labeled with a label in the provided trace infix/postfix. The algorithm *BuMG* is responsible for creating a relevant marking that enables at least the transition representing the provided vertex by Algorithm 4.1 (cf. line 4 in Algorithm 4.1). To this end, *BuMG* creates a marking containing all places having an arc towards the determined transition. Next, starting from the provided leaf vertex, *BuMG* recursively goes up the hierarchy in the tree until the root vertex is reached and the marking is finally returned. However, if on the bottom-up walk in the tree hierarchy a vertex labeled with the parallel operator is detected, *BuMG* invokes the *Top-down marking generation algorithm TdMG* on the subtrees not yet covered. This step is required to ensure that the markings eventually returned are reachable from  $N_\Lambda$ . When reaching the root vertex, *BuMG* returns the set of relevant markings for the vertex provided by Algorithm 4.1.

**Algorithm 4.3:** Top-down marking generation (*TdMG*)

---

```

input :  $\Lambda = (V, E, \Sigma, \lambda, r, <) \in \mathcal{P}$ ,
         $N_\Lambda = (P_\Lambda, T_\Lambda, F_\Lambda, \lambda_\Lambda, M_\Lambda^{init}, M_\Lambda^{final})$ ,           // WF-net representing  $\Lambda$ 
         $v \in V$ ,                                           // currently considered node
         $A \subseteq \mathcal{A}$                                      // activity labels contained in trace infix/postfix  $\sigma$ 
         $addFinalMarking \in \mathbb{B}$                                // Boolean flag

output:  $R \subseteq \mathcal{M}(P_\Lambda)$ 

begin
1  if  $v \in \text{leaves}(\Lambda)$  then
2      let  $t_v \in T_\Lambda$  be the transition representing vertex  $v$ 
3       $R \leftarrow \{ \}$ 
4      if  $\lambda(v) \in A$  then
5           $R \leftarrow R \cup \{p \in \bullet t_v\}$            //  $t_v$ 's label is in the trace infix/postfix
6      if  $addFinalMarking$  then
7           $R \leftarrow R \cup \{p \in t_v \bullet\}$ 
8      return  $R$ 
9  else                                     //  $r$  represents an operator, i.e.,  $\lambda(r) \in \{\rightarrow, \wedge, \odot, \times\}$ 
10     let  $\langle v_1, \dots, v_k \rangle = \text{child}_\Lambda(r)$ 
11     if  $\lambda(v) = \rightarrow$  then
12          $\text{return } TdMG(\Lambda, N, v_1, A, false) \cup \dots \cup$ 
13          $TdMG(\Lambda, N, v_{k-1}, A, false) \cup TdMG(\Lambda, N, v_k, A, addFinalMarking)$ 
14     if  $\lambda(v) = \wedge$  then
15          $\text{return } TdMG(\Lambda, N, v_1, A, true) \times \dots \times TdMG(\Lambda, N, v_k, A, true)$ 
16     if  $\lambda(v) \in \{\odot, \times\}$  then
17          $\text{return } TdMG(\Lambda, N, v_1, A, addFinalMarking) \cup$ 
18          $TdMG(\Lambda, N, v_2, A, false) \cup \dots \cup TdMG(\Lambda, N, v_k, A, false)$ 

```

---

**Top-down Marking Generation Algorithm *TdMG***

Algorithm 4.3 presents the recursively specified *Top-down marking generation algorithm* *TdMG*, which is called in Algorithm 4.2 line 6. As the name indicates, algorithm *TdMG* moves from a given vertex  $v$  down the tree hierarchy until leaf vertices are reached. As input, *TdMG* is provided the process tree  $\Lambda$ , the corresponding WF-net  $N_\Lambda$ , a starting vertex  $v \in V$ , the set of activity labels  $A$  that are contained in given trace infix/postfix, and a Boolean flag *addFinalMarking* indicating if the final marking from the WF-net part representing the subtree rooted at  $v$  should be incorporated.

When *TdMG* eventually reaches a leaf vertex (line 1), *TdMG* only returns a marking if either the transition representing the vertex  $v$  is labeled with a label from  $A$  (line 5) or if the Boolean flag *addFinalMarking* is *true* (line 7). In all other cases, *TdMG* returns the empty set, i.e.,  $R = \emptyset$ .

When *TdMG* is called by Algorithm 4.2 (line 6), vertex  $v$  is not a leaf. Thus, we enter

the else part starting at line 9. First, we retrieve the child vertices of  $v$  (line 10). Next, we make a case distinction based on the operator of  $v$ , i.e., the label of  $v$ .

1. If  $v$  is labeled with a sequence operator (line 11), we recursively execute *TdMG* on the child vertices of  $v$  (line 12). Note that we set the Boolean flag *addFinalMarking* to *false* for vertices  $v_1, \dots, v_{k-1}$ . Only for the last child vertex  $v_k$ , we set the flag according to the input, i.e., *addFinalMarking*. Note that when *TdMG* is called from *BuMG* (cf. Algorithm 4.2 line 6), the flag *addFinalMarking* = *true*. Thus, if non of the subtrees below vertex  $v$  contains leaf vertices that are labeled with labels from  $A$ , only the final marking of the WF-net part that represents the subtree rooted at  $v_k$  is returned.
2. If  $v$  is labeled with a parallel operator (line 13), we recursively execute *TdMG* on the child vertices of  $v$  (line 14). Note that we set the Boolean flag *addFinalMarking* to *true* for all child vertices. This is required to ensure skipping the parallel branch. Thus, if non of the subtrees below vertex  $v$  contains leaf vertices that are labeled with labels from  $A$ , the final marking contains places from  $N_\Lambda$  representing the end of each parallel branch.
3. If  $v$  is labeled with a exclusive-choice or loop operator (line 15), we recursively execute *TdMG* on the child vertices of  $v$  (line 16). In case we deal with a loop operator, we know that  $k = 1$  because a vertex labeled with a loop operator has always two child vertices, according to Definition 3.28 (page 68).<sup>5</sup> Thus, we set the Boolean flag *addFinalMarking* for the first subtree to *true* while for the other subtree to *false*. In case neither the first nor the second subtree below  $v$  contains a transition labeled with a label from  $A$ , we start in the final marking of the WF-net part that represents the first subtree, i.e., do part.

If we deal with a exclusive-choice operator, we have in general  $k$  child vertices to consider. Since we can freely choose which subtree to execute, we must ensure to include at least the corresponding final marking of the WF-net part representing one of the  $k$  subtrees. For the sake of simplicity, we always choose the subtree rooted at  $v_1$ .<sup>6</sup>

In conclusion, *TdMG* is only called from *BuMG* (cf. Algorithm 4.2 line 6) when a parallel operator is reached during the recursive traversing from a leaf vertex to the root vertex. If a parallel operator is hit, *BuMG* calls *TdMG* for each subtree beneath the vertex labeled with the parallel operator individually. While *BuMG* recursively goes up in the tree hierarchy, *TdMG* goes down the hierarchy starting from a provided vertex until leaf vertices are reached. In general, *TdMG* is required to ensure the markings eventually returned by *BuMG* are reachable markings from the initial marking  $M_\Lambda^{init}$  of  $N_\Lambda$ , which represents the provided tree  $\Lambda$ .

<sup>5</sup>Recall that the first subtree represents the do part and the second subtree the redo part. Further, executing the redo part requires another execution of the do part, cf. Definition 3.29.

<sup>6</sup>Note that any but at least for one vertex the Boolean flag *baddFinalMarking* must be set to *true*. The decision for the first subtree was made for the simple reason that we can handle the cases for the loop and the exclusive-choice operator in the same way.



### Example

Here, we present an extensive example of the process-tree-based approach. Reconsider the example provided at the beginning of Section 4.3.3. The WF-net depicted in Figure 3.15 (page 77) represents the process tree  $\Lambda_{N_1}$  shown in Figure 4.8. Below each leaf vertex, the corresponding transition in WF-net  $N_1$  is displayed. Further, reconsider the trace infix/postfix  $\sigma = \langle b, d, f \rangle$ .

First, we call Algorithm 4.1 (page 95). As input, we provide tree  $\Lambda_{N_1}$ ,  $N_1$ , and  $\sigma$ . First, the set of labels from  $\sigma$  is calculated, i.e.,  $A = \{b, d, f\}$ . Next, for transitions  $t_2$ ,  $t_4$ , and  $t_7$ , algorithm *BuMG* (cf. Algorithm 4.2) is called because  $\lambda(t_2), \lambda(t_4), \lambda(t_7) \in A$  (cf. Algorithm 4.1 line 3). Subsequently, we exemplify the call of *BuMG* for transition  $t_7$ .

1. **Call:**  $BuMG(\Lambda_{N_1}, v_{4.2}, \text{undefined}, N_1, \emptyset, A\sigma)$   
**Calling algorithm:** Algorithm 4.1 (line 4)  
**Intermediate result:**  $R = \{[p_8]\}$
2. **Call:**  $BuMG(\Lambda_{N_1}, v_{3.4}, v_{4.2}, N_1, \{[p_8]\}, A)$   
**Calling algorithm:** Algorithm 4.2 (line 10)
  - a) **Call:**  $TdMG(\Lambda_{N_1}, N_1, v_{4.1}, A, \text{true})$   
**Calling algorithm:** Algorithm 4.2 (line 6)  
**Intermediate result:**  $R = \{[p_9]\}$**Intermediate result:**  $R = \{[p_8, p_9]\} = \{[p_8]\} \times \{[p_9]\}$  (at line 7)
3. **Call:**  $BuMG(\Lambda_{N_1}, v_{2.4}, v_{3.4}, N_1, \{[p_8, p_9]\}, A\sigma)$   
**Calling algorithm:** Algorithm 4.2 line 10  
**Intermediate result:**  $R = \{[p_8, p_9]\}$
4. **Call:**  $BuMG(\Lambda_{N_1}, v_1, v_{2.4}, N_1, \{[p_8, p_9]\}, A\sigma)$   
**Calling algorithm:** Algorithm 4.2 line 10  
**Final result:**  $R = \{[p_8, p_9]\}$

Above, we have shown the call of *BuMG* for transition  $t_7$ . Algorithm *BuMG* returns one relevant marking, i.e.,  $[p_8, p_9]$ . Calling *BuMG* also for transition  $t_2$  would yield the marking  $[p_2, p_5]$  and for transition  $t_4$  would yield marking  $[p_4, p_5]$ . Thus, we obtain three relevant markings. When eventually adding the final marking to the set of relevant markings (cf. Algorithm 4.1), we end up with four relevant markings that are used to create the auxiliary WF-net shown earlier in Figure 4.9 (page 94).<sup>7</sup>

## 4.4. Evaluation

This section presents an experimental evaluation of the proposed computation of infix and postfix alignments. Aim of the evaluation is demonstrate the applicability of the proposed

<sup>7</sup>Recall that the final marking of the provided WF-net is only considered a relevant marking if a postfix alignment is calculated.

approaches on real-life event logs. Further, we aim at comparing the three proposed approaches for determining relevant markings, i.e., the baseline approach (cf. Section 4.3.1), the extended baseline approach utilizing subsequent filtering (cf. Section 4.3.2), and the process-tree-based approach (cf. Section 4.3.3). Section 4.4.1 outlines the experimental setup, followed by the presentation of results in Section 4.4.2, and finally, a discussion of the findings and threats to validity in Section 4.4.3.

#### 4.4.1. Experimental Setup

We use publicly available event logs capturing real business processes: the BPI Challenge logs from 2020—Prepaid Travel Cost log (BPI Ch. 2020) [232], 2012 (BPI Ch. 2012) [228], 2019 (BPI Ch. 2019) [231], and the Road Traffic Fine Management log (RTFM) [56]. We sampled up to 10,000 trace infixes (if possible) from the full traces for each log. Further, we discovered a process model for each log using the Inductive Miner infrequent algorithm [120] from the original event log using a noise threshold of 0.9. We choose the Inductive Miner infrequent to obtain a process tree, i.e., a block-structure WF-net, and thus being able to compare all three approaches. Further we made use of the noise threshold to obtain a process model not fully supporting the provided event log and thus to obtain alignments indicating deviations.

#### 4.4.2. Results

This section presents the findings. Figure 4.11 shows the computation time of the relevant markings using the three proposed techniques (cf. Sections 4.3.1 to 4.3.3). Recall that the baseline approach (cf. Section 4.3.1) calculates relevant markings only from the provided WF-net. Thus, the time spent by the baseline approach is independent of the infix and, hence, the infix length. Therefore, a horizontal blue line can be seen in the plots shown in Figure 4.11. For the two other approaches, i.e., the extended baseline approach (cf. Section 4.3.2) and the process-tree-based approach (cf. Section 4.3.3), the time spent differs for each considered trace infix. We grouped these time values based on the infix length for better visibility. We observe for both approaches, i.e., the extended baseline approach and the process-tree-based approach, that the infix length is relatively independent of the time spent determining the relevant markings. This observation is, however, not surprising since, for the extended baseline approach, all reachable markings are first calculated, which is independent of the provided infix, and then the subsequent filtering is applied. Since the extended baseline approach performs subsequent filtering, the approach takes longer on average than the baseline approach. This phenomenon can be observed for all four logs. Further, observe that the whiskers of the box plots for the extended baseline approach go lower than those for the baseline approach. Theoretically, this observation should not occur since the extended baseline approach performs identical computations plus additional filtering. This phenomenon can only be explained by uncontrollable load fluctuations or other influences on the computer system in which we performed the experiments. Nevertheless, the interquartile range of the extended baseline approach is always above the time spent by the baseline approach. Overall, the three approaches have similar times spent in the millisecond range. Further, we observe that

the process-tree-based approach is the fastest on average since it does not exhaustively calculate all reachable markings compared to the baseline approaches.

Figure 4.12 shows the time spent for computing optimal infix alignments for each proposed approach. The time values shown cover all four steps as introduced in Section 4.3 and Figure 4.4; thus, ranging from determining relevant markings, i.e., the times indicated in Figure 4.11 (step 1a), generating the auxiliary WF-net (step 1b), creating the SPN (step 2), solving the shortest path problem (step 3) until the postprocessing (step 4). In most cases, the process-tree-based approach outperforms the other approaches. However as we observed in Figure 4.11, the differences among the three approaches in computing the relevant markings and thus also the generation of the auxiliary WF-net

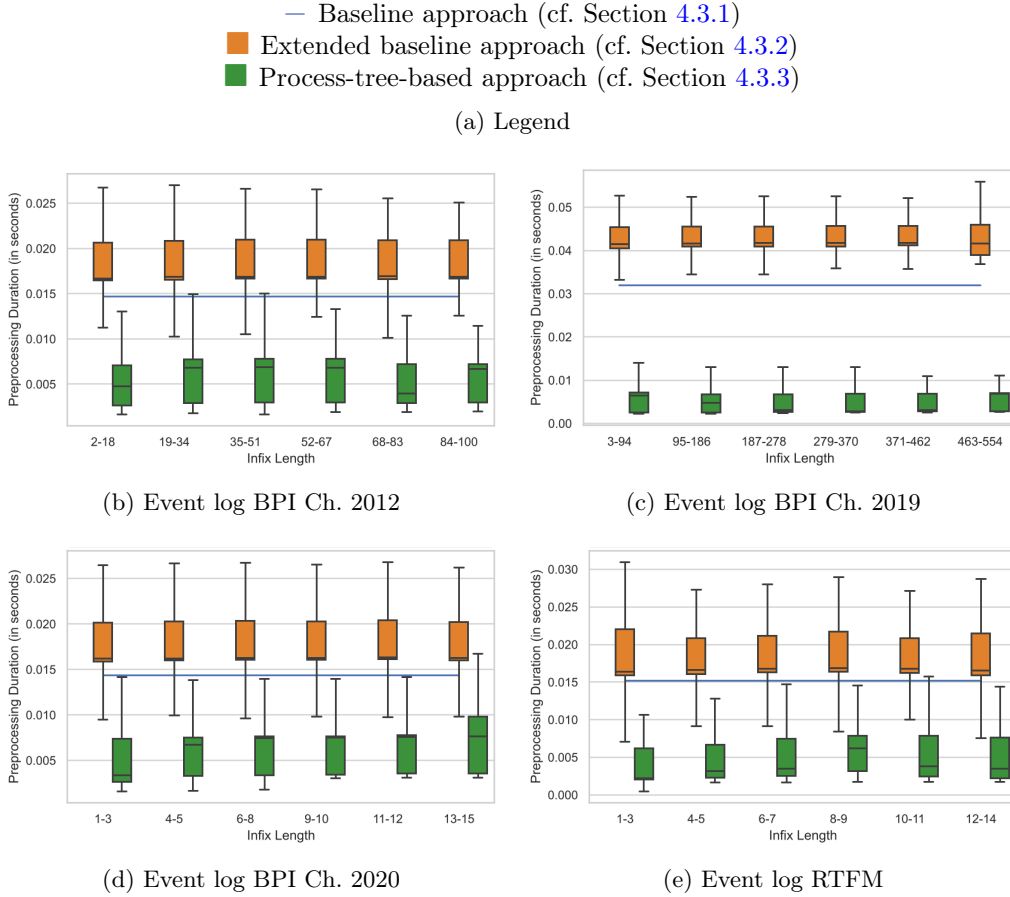


Figure 4.11: Time spent in seconds for determining relevant markings, i.e., step 1a (cf. Section 4.3 on page 86), using the three proposed approaches (cf. Sections 4.3.1 to 4.3.3)

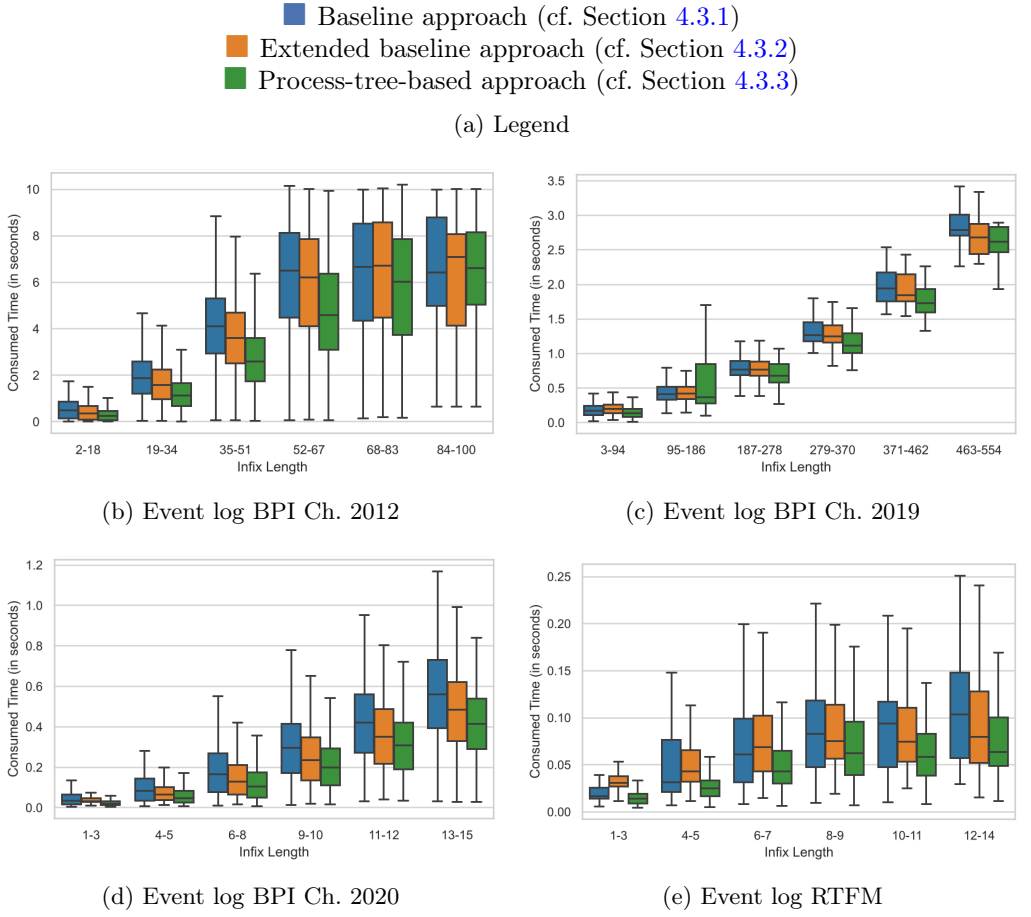


Figure 4.12: Time spent in seconds for calculating optimal infix alignments, i.e., steps 1–4 (cf. Section 4.3), using the three proposed approaches for calculating relevant markings (cf. Sections 4.3.1 to 4.3.3) (partly adapted from [180, Figure 8])

and the SPN are marginal. Therefore, we conclude that the state space of the SPN when using the auxiliary WF-net constructed with the relevant markings computed by the process-tree-based approach has the smallest state space and hence the overall computation time is lowest. Further, we observe that the longer the infix size, the longer it takes to compute a corresponding optimal infix alignments.

Figure 4.13 shows the number of relevant markings computed per approach. We observe that the number of relevant markings computed by the process-tree-based approach is, on average, lowest compared to the baseline approach. Further, we observe a significant difference regarding the number of relevant markings between the baseline and extended

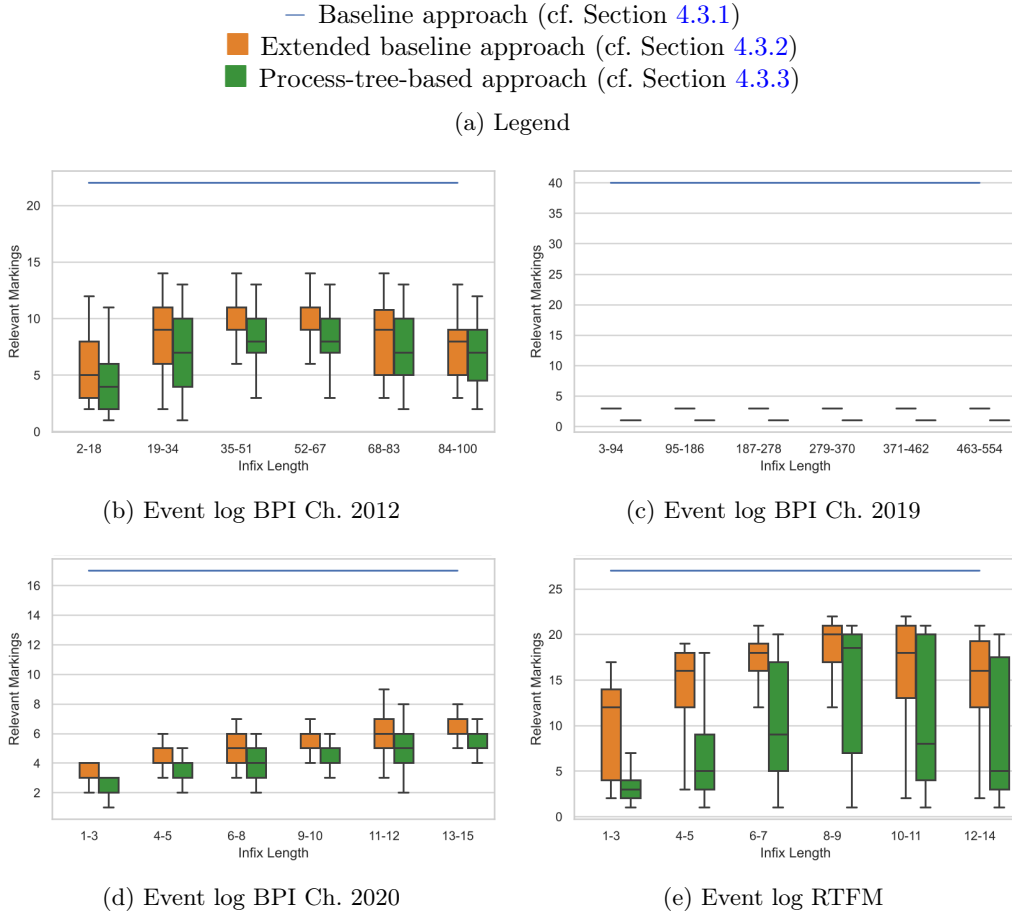


Figure 4.13: Number of calculated relevant markings using the three proposed approaches (cf. Sections 4.3.1 to 4.3.3)

baseline approaches. In short, the number of relevant markings and the associated complexity of the auxiliary WF-nets used to construct the SPNs is the main reason for the performance differences in computation time between the approaches.

Regarding the correctness of the proposed approaches determining relevant markings, we calculated in the conducted experiments the costs of each infix alignment. We note that for each trace infix the computed optimal infix alignment has the same cost regardless of which of the three approaches we use. Since, the baseline approach is guaranteed to be correct since it considers all reachable markings as relevant markings, the made comparison regarding the costs is feasible although no general conclusions can be drawn beyond the event logs used.

### 4.4.3. Discussion & Threats to Validity

The conducted experiments indicate the applicability of infix and postfix alignments to real-world event logs. We primarily focused on the computation time, i.e., for determining relevant markings and for computing optimal infix alignments overall. As stated in Section 4.4.2, the data points for the extended baseline approach should theoretically never be below the baseline approach. However, we observe this phenomenon in some instances. These observations indicate that there are slight fluctuations regarding the calculation period when computing relevant markings. As mentioned in Section 4.4.2, these fluctuations cannot be controlled and must be considered measurement inaccuracies.

We solely used process trees as input models for all approaches. Recall that only the process-tree-based approach requires process trees. Further experiments could analyze the differences between the baseline and the extended baseline approach when using WF-nets that cannot be represented as a process tree. Moreover, we use one process model per event log in the conducted experiments. In particular, the degree of parallelism of the process model provided is a significant factor for the computing time as the state space grows exponentially. The baseline approaches, in particular, would have a significant disadvantage compared to the process-tree-based approach in the case of a high degree of parallelism in the process model provided, as they have to calculate all reachable markings. Since we only use one process model per event log, this effect can only be observed to a limited extent in the experiments presented. Synthetic experiments, in which the degree of parallelism in the given process model is systematically changed, would be an option to show that the baseline approaches perform increasingly worse than the process-tree-based approach as the degree of parallelism increases.

## 4.5. Conclusion

Alignments are a state-of-the-art conformance checking technique [45, 46]. So far, alignments were defined for complete traces—most process mining approaches solely focus on complete traces—and trace prefixes [6]. This chapter extended alignments by infix and postfix alignments. We provided a formal definition and approaches to calculate infix/postfix alignments. The proposed calculation of infix/postfix alignments builds upon established approaches in alignment computation. Thus, we utilize the SPN and the reduction of computing optimal alignments to a shortest path problem; reconsider Figure 4.1 (page 82). While for complete and prefix alignment, the starting marking in the provided WF-net is clear (i.e., the WF-net’s initial marking), we must find the starting marking when computing infix and postfix alignments. To this end, we create an auxiliary process model that allows starting from its initial marking to reach via silent transitions potential marking representing the start of the model part in an optimal infix/postfix alignment. We presented three approaches: a baseline approach, an extended baseline approach, and an process-tree-based approach specifically for block-structured WF-nets respectively process trees. The conducted experiments indicate that the process-tree-based approach outperforms the baseline approaches since it creates overall a lower number of relevant markings. However, recall that the process-tree-based approach solely supports process trees, while the baseline approaches support arbitrary WF-nets.

## Part II.

# Incremental Process Discovery





## Chapter 5.

# Incremental Process Discovery Framework

Parts of this chapter are based on published work.

- Sections 5.1 to 5.4 are largely based on *D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Incremental discovery of hierarchical process models. In F. Dalpiaz, J. Zdravkovic, and P. Loucopoulos, editors, Research Challenges in Information Science, volume 385 of Lecture Notes in Business Information Processing, pages 417–433. Springer, 2020. doi:10.1007/978-3-030-50316-1\_25 [174].*
- Section 5.6 is largely based on *D. Schuster, E. Domnitsch, S. J. van Zelst, and W. M. P. van der Aalst. A generic trace ordering framework for incremental process discovery. In T. Bouadi, E. Fromont, and E. Hüllermeier, editors, Advances in Intelligent Data Analysis XX, volume 13205 of Lecture Notes in Computer Science, pages 264–277. Springer, 2022. doi:10.1007/978-3-031-01333-1\_21 [179].*

This chapter introduces an incremental process discovery framework that allows the gradual discovery of a process model from event data. The incremental aspect refers to the step-by-step addition of traces to a process model that is considered under construction. Section 5.1 introduces the overall framework and details the inputs and outputs. The two subsequent chapters introduce instantiations of this framework; Section 5.2 presents a naive baseline instantiation while Section 5.3 presents a more advanced approach. Subsequently, Section 5.4 evaluates the proposed instantiations. Section 5.5 provides an illustrative example showcasing the proposed incremental approach’s advantage over a conventional process discovery algorithm. Finally, Section 5.6 deals with trace ordering effects in incremental process discovery.

## 5.1. Introduction to the Framework

This section introduces the incremental process discovery framework that provides the foundation of the subsequent chapters. Section 5.1.1 presents its inputs and outputs,

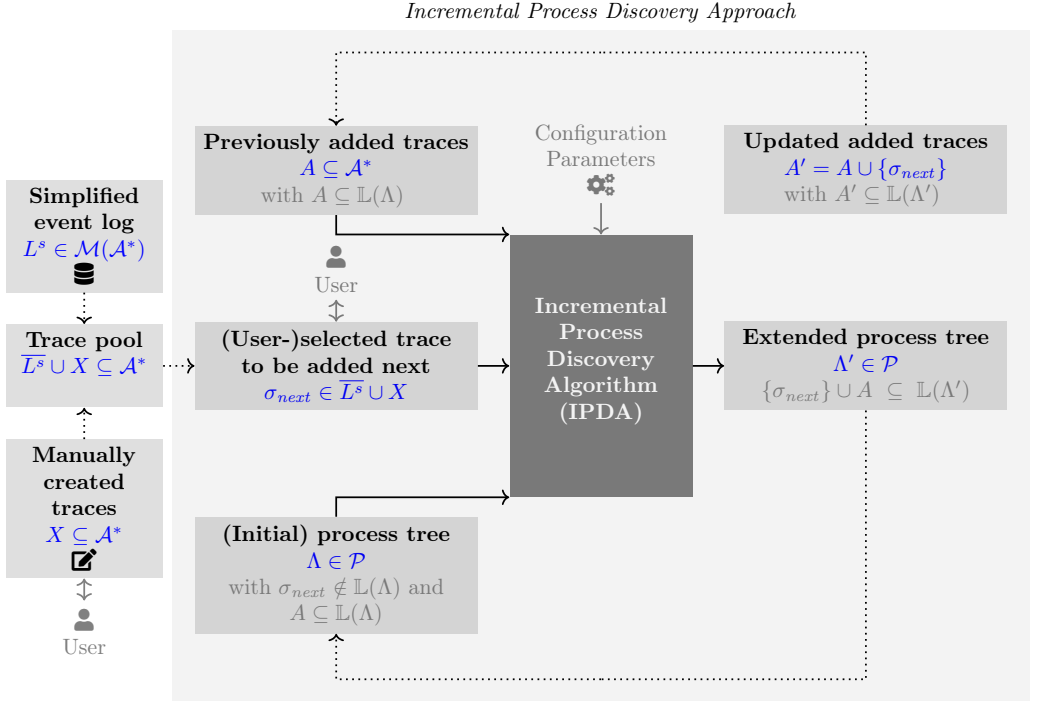


Figure 5.1: Input-output perspective of the incremental process discovery framework

while Section 5.1.2 discusses the underlying motivation and the resulting opportunities.

### 5.1.1. Input-Output Perspective

Figure 5.1 illustrates the incremental process discovery framework focusing on the input-output perspective. Central to this framework is an Incremental Process Discovery Algorithm (IPDA), which assumes three inputs:

1. an (initial) process tree  $\Lambda \in \mathcal{P}$  that is gradually extended during execution of the incremental process discovery approach<sup>1</sup>,
2. previously added traces  $A \subseteq \mathcal{A}^*$  that have been added to the process tree in previous iterations, and
3. a trace to be added next  $\sigma_{next} \in \mathcal{A}^*$  to process tree  $\Lambda$ .

The trace to be added next  $\sigma_{next}$  can be selected by a user or by an algorithmic approach from the trace pool. The trace pool originates from a provided simplified event log  $L^s \in \mathcal{M}(\mathcal{A}^*)$  and manually created traces  $X \subseteq \mathcal{A}^*$ , cf. Section 5.1. Further, all

<sup>1</sup>The initial process tree in the first iteration can be as simple as containing only a single labeled or a silent activity.

previously added traces contained in  $A$  are assumed to fit the current process model. Thus,  $A \subseteq \mathbb{L}(\Lambda)$ .

The IPDA takes the three described input and modifies the provided process tree  $\Lambda$  into  $\Lambda'$  such that previously added traces  $A$  and the selected trace  $\sigma_{next}$  are fitting  $\Lambda'$ .

$$\{\sigma_{next}\} \cup A \subseteq \mathbb{L}(\Lambda')$$

Process tree  $\Lambda'$  is then used within the subsequent incremental execution as an input. Further, trace  $\sigma_{next}$  is added to the set of previously added traces. The extended set  $A'$  is used as an input in the subsequent incremental execution, too. Finally, note that the IPDA might offer additional configuration parameters that a user can specify. However, these configuration parameters are specific to the particular IPDA employed and will not be discussed further here. Subsequently, we formally define an IPDA.

**Definition 5.1** (Incremental Process Discovery Algorithm (IPDA))

*The function*

$$ipda : \mathcal{P} \times \mathcal{A}^* \times \mathbb{P}(\mathcal{A}^*) \rightarrow \mathcal{P}$$

*is an IPDA if for any (initial) process tree  $\Lambda \in \mathcal{P}$ , trace  $\sigma_{next} \in \mathcal{A}^*$ , and previously added traces  $A \subseteq \mathcal{A}^*$  with  $A \subseteq \mathbb{L}(\Lambda)$  it holds that*

$$A \cup \{\sigma_{next}\} \subseteq \mathbb{L}(ipda(\Lambda, \sigma_{next}, A)).$$

*If  $A \not\subseteq \mathbb{L}(\Lambda)$ , function  $ipda$  is undefined.*

### 5.1.2. Motivation & Opportunities

Providing the option to discover process models in an incremental fashion enables new opportunities for users. First, users can observe how the discovered process model evolves during the overall process discovery phase.<sup>2</sup> Thus, users see intermediate discovered process models compared to automated process discovery, which comprises all conventional process discovery algorithms and also automated non-conventional approaches (cf. Figure 1.5). Observing intermediate discovered process models may help users better understand why the discovery approaches develop a specific model. Thus, incremental process discovery may help to increase trust in the finally discovered process model, as process discovery is not perceived as a black box. In [141], the authors identified the lack of trust in process mining insights as a critical challenge when applying process mining in organizations. Additionally, the authors found that the absence of trust stems from a limited comprehension of the utilized methods and the perception of process mining techniques as black boxes.

Access to intermediate discovered process models allows users to intervene and correct the discovery algorithm. For instance, a user might change an intermediate process model before it is used in the next incremental iteration. Consider Figure 5.1. If a user changes the process tree  $\Lambda'$  into  $\Lambda''$ , a check is required if the traces in  $A'$  are still supported by

<sup>2</sup>We refer to the *process discovery phase* as the procedure leading up to the discovery of a final process model.

the user-modified process model  $\Lambda''$ . If traces from  $A'$  are not covered anymore by  $\Lambda''$  they have to be removed such that the assumptions regarding the inputs made by the IPDA are satisfied. In short, users can direct and control the discovery phase by not only selecting traces but also making manual adjustments to the intermediate discovered process models.

Moreover, the incremental process discovery approach allows to utilize a priori domain knowledge by incorporating an initial process model. Recall that this domain knowledge is virtually optional since one can start with a model containing only a visible or invisible activity label. Moreover, the gradual interaction with the event data results in users paying more attention to event data quality aspects when incrementally selecting traces to be added to the process model. While automated or manual filtering of event data may also be performed in conventional process discovery, direct feedback on what effect, for example, an incorrect trace from the event log has on the discovered process model is much more apparent due to the incremental approach and the possibility to observe intermediate discovered learned process models after adding a trace.

In the remainder of this section, we focus on algorithmic aspects of IPDAs, i.e., we propose two concrete IPDA instantiations that comply with the introduced framework depicted in Figure 5.1. Section 5.2 proposes a naive IPDA, while Section 5.3 proposes a more advanced IPDA.

## 5.2. Naive IPDA

This section introduces a naive IPDA that can be employed in the proposed incremental process discovery framework, cf. Figure 5.1. The core idea of this naive approach is to first calculate an optimal alignment for  $\sigma_{next}$  and process tree  $\Lambda$  to assess if  $\sigma_{next}$  fits  $\Lambda$ . If the optimal alignment indicates deviations, the naive IPDA processes each alignment move that indicates a deviation separately. For each deviation-indicating alignment move, i.e., log moves and visible model moves (cf. Section 3.5), process tree modifications are applied that resolve this particular deviation-indicating alignment move. After processing all the deviation-indicating alignment moves, the baseline IPDA guarantees that  $\sigma_{next}$  fits the resulting process tree  $\Lambda'$ , cf. Figure 5.1.

The naive IPDA resolves alignment moves that indicate a deviation from left to right until eventually the provided trace fits the altered process tree. Figure 5.2 illustrates four resolution rules that cover different scenarios regarding the occurrence of a deviation move in an alignment.<sup>3</sup> The presented resolution rules are complete; a resolution rule covers any deviation move in an alignment. The 1<sup>st</sup> case covers visible model moves, while the other three cases, the 2<sup>nd</sup> to the 4<sup>th</sup> case, cover log moves.

- The 1<sup>st</sup> rule resolves a visible model move on activity  $a$ . The rule adds a new vertex representing an exclusive choice operator to the tree that has two child vertices: a new vertex labeled  $\tau$  and the existing activity  $a$ , cf. Figure 5.2b. This construct allows skipping the execution of vertex  $v_i$  labeled  $a$ ; thus, when recalculating an alignment, the visible model move on vertex  $v_i$  no longer occurs.

<sup>3</sup>Note that the activity labels used in Figure 5.2, i.e.,  $a$  and  $b$ , are only representatives and can be replaced by any other label from  $\mathcal{A}$ .

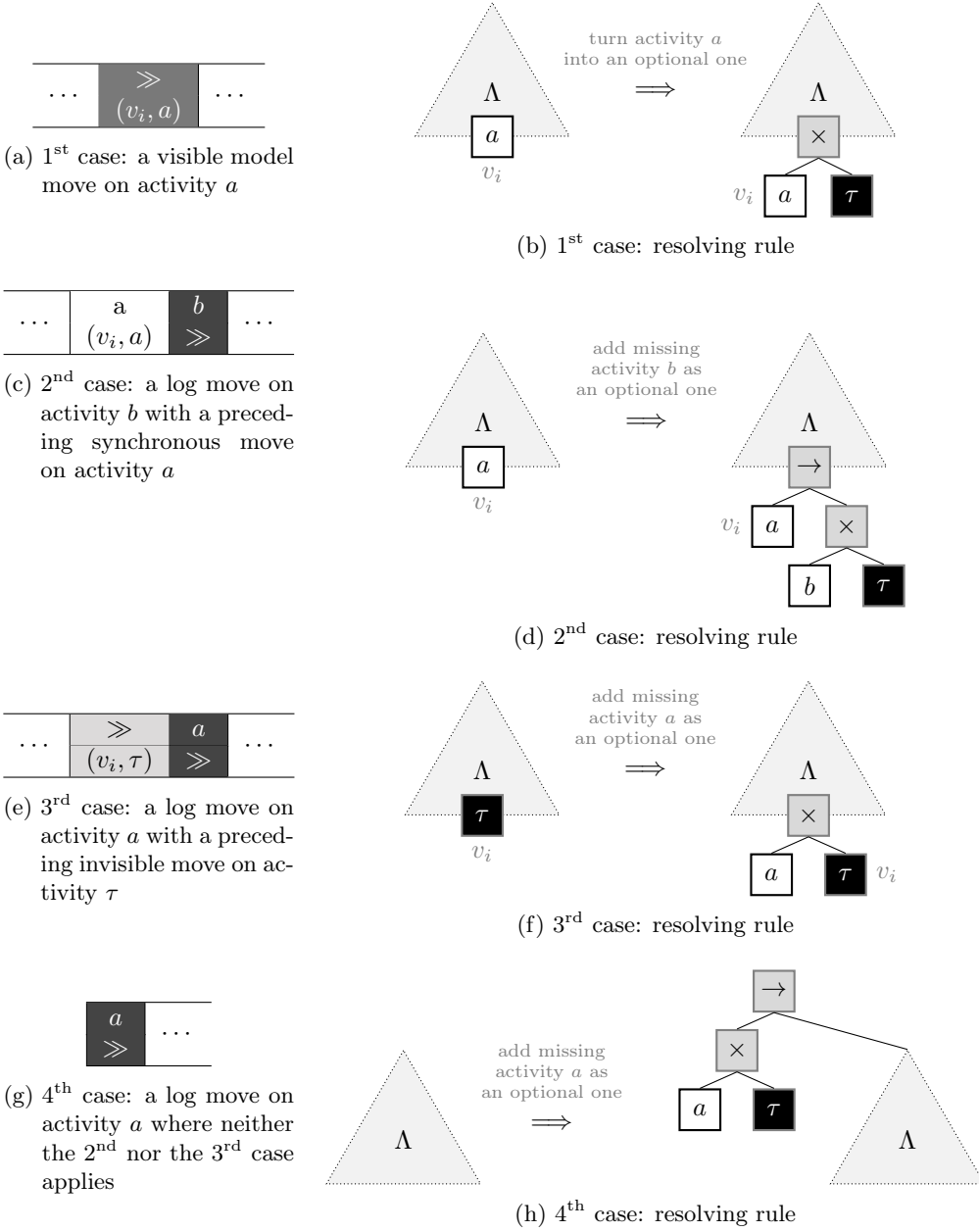


Figure 5.2: Baseline IPDA's resolving rules for alignment moves that indicate a deviation—four case distinctions (partly adapted from [174])

- The 2<sup>nd</sup> resolution rule resolves a log move on activity  $a$  with a directly preceding synchronous move on activity  $b$ , cf. Figure 5.2c. The rule modifies the tree such that after executing the activity  $a$ , activity  $b$  can be optionally executed, cf. Figure 5.2c. Thus, the log move on activity  $b$  after the synchronous move on activity  $a$  is resolved.
- The 3<sup>rd</sup> rule resolves a log move on activity  $a$  with a preceding invisible model move, cf. Figure 5.2e. The rule modifies the tree such that the  $\tau$  activity will be placed under an exclusive-choice together with an  $a$  activity, cf. Figure 5.2f. Thus, activity  $a$  can be optionally executed and the log move is resolved.
- The 4<sup>th</sup> rule resolves a log move that has neither a preceding invisible model move nor a preceding synchronous move; hence, generally only log or visible model moves are preceding. However, since the naive IPDA resolves deviations from left to right it follows that this rule only applies if a log move is the first move of an alignment. The rule modifies the tree  $\Lambda$  such that before executing the original tree  $\Lambda$ , the activity  $a$  can be optionally executed, cf. Figure 5.2h.

Algorithm 5.1 presents the naive IPDA. As input, the naive IPDA assumes the trace to be added next  $\sigma_{next} \in \mathcal{A}^*$  and the process tree  $\Lambda \in \mathcal{P}$ . Compared to the introduced incremental process discovery framework depicted in Figure 5.1, the naive IPDA does not make use of the set of previously added traces  $A \subseteq \mathcal{A}^*$ . Ignoring  $A$  is valid because the four resolution rules (cf. Figure 5.2) ensure that all previously fitting traces still fit the altered process tree  $\Lambda'$ , i.e.,  $\mathbb{L}(\Lambda) \subseteq \mathbb{L}(\Lambda')$ . All resolving rules presented in Figure 5.2 extend the resulting process tree's language.

---

**Algorithm 5.1:** Naive IPDA (N-IPDA)

---

**Input:**  $\sigma_{next} \in \mathcal{A}^*, \Lambda \in \mathcal{P}$

**Output:**  $\Lambda' \in \mathcal{P}$

**begin**

```

1  let  $\gamma \in \Gamma^{opt}(\Lambda, \sigma_{next})$                                 // calculate an optimal alignment  $\gamma$ 
2   $\Lambda' \leftarrow \Lambda$ 
3  while  $deviation(\gamma)$  do                                    //  $\sigma_{next} \notin \mathbb{L}(\Lambda')$ 
4      let  $i \in \mathbb{N}$  be the index of the first deviation move in  $\gamma$ 
          //  $deviationMv(\gamma(i)) \wedge \nexists 1 \leq j < i (deviationMv(\gamma(j)))$ 
5       $\Lambda' \leftarrow$  apply suitable resolving rule for  $\gamma(i)$  and  $\Lambda'$     // cf. Figure 5.2
6      let  $\gamma \in \Gamma^{opt}(\Lambda', \sigma_{next})$                     // calculate an optimal alignment  $\gamma$ 
7  return  $\Lambda'$                                                 //  $\sigma_{next} \in \mathbb{L}(\Lambda')$ 

```

---

First, an optimal alignment is calculated to assess if  $\sigma_{next}$  fits process tree  $\Lambda'$ . If not, the first alignment move that indicates a deviation is determined, i.e.,  $\gamma(i)$  (cf. line 4). Next, one of the four resolution rules (cf. Figure 5.2) is applied to resolve the determined alignment move. This procedure, i.e., calculating an optimal alignment and resolving the first move indicating a deviation, is repeated until the optimal alignment does not indicate a deviation anymore; hence,  $\sigma_{next} \in \mathbb{L}(\Lambda')$  and  $\Lambda'$  is returned.

The termination of Algorithm 5.1 is guaranteed as all four resolution rules (cf. Figure 5.2) resolve an alignment move that indicates a deviation. Thus, after each iteration, i.e., lines 3 to 6, the optimal alignment  $\gamma$  contains one deviation-indicating move less and eventually trace  $\sigma_{next}$  fits process tree  $\Lambda'$ .

Finally, note that none of the presented resolution rules (cf. Figure 5.2) adds a loop operator nor a parallel operator to the tree. Thus, if the initial model from which the incremental process discovery starts does not contain any loop or parallel operator, the resulting process tree does not contain such operators either. Thus, we subsequently present a more advanced approach that fully exploits the process tree formalism.

### 5.3. Lowest Common Ancestor IPDA

The previously presented naive IPDA resolves each alignment move that indicates a deviation individually. Further, the resolution rules shown are simple and may lead to large process models because every deviation-indicating alignment move leads to a process tree that contains more vertices, cf. Figure 5.2. This section introduces an alternative IPDA called Lowest Common Ancestor IPDA (LCA-IPDA) that can handle entire blocks of deviation-indicating alignment moves, compared to the naive IPDA. Moreover, LCA-IPDA utilizes all three inputs illustrated in Figure 5.1. Finally, it utilizes all process tree operators, compared to the naive IPDA, which can only add sequence and exclusive choice operators.

The core idea of LCA-IPDA involves detecting subtrees in process tree  $\Lambda$  that cause  $\sigma_{next}$  non-fitting. For these subtrees, *sublogs* are computed that represent trace fragments that the corresponding subtree should support. All detected subtrees causing  $\sigma_{next}$  non-fitting are then rediscovered from their corresponding sublog using a *fitness-preserving* conventional process discovery algorithm. Eventually, the rediscovered subtrees replace the detected subtrees and the final tree  $\Lambda'$  is returned, cf. Figure 5.1.

#### 5.3.1. Running Example

In the following, we introduce the LCA-IPDA by a running example. Assume the following input according to Figure 5.1:

- previously added traces  $A = \{\langle a, b, c, d, a, b, e, f \rangle, \langle c, d, d, c, c, d, f, e \rangle\} \subseteq \mathcal{A}^*$ ,
- trace to be added next  $\sigma_{next} = \langle a, b, b, b, f, e \rangle \in \mathcal{A}^*$ , and
- process tree  $\Lambda \in \mathcal{P}$  as depicted in Figure 5.3.

The previously added traces contained in  $A$  are replayed on process tree  $\Lambda$  to determine sublogs for each subtree, cf. Figure 5.3. A sublog contains trace fragments from traces in  $A$  that the corresponding subtree supports. Therefore, the sublog for the entire tree, i.e., the subtree rooted at  $v_0$ , always equals  $A$ . For instance, the sublog for subtree  $\Delta_\Lambda(v_{2.1})$  contains twice the trace fragment  $\langle a, b \rangle$ , three times  $\langle c, d \rangle$ , and once  $\langle d, c \rangle$ . All three trace fragments origin from traces in  $A$ .

Next, LCA-IPDA computes an optimal alignment to assess if  $\sigma_{next}$  fits  $\Lambda$ ; Figure 5.4 shows an optimal alignment. The 6<sup>th</sup> and 7<sup>th</sup> alignment moves indicate a deviation,

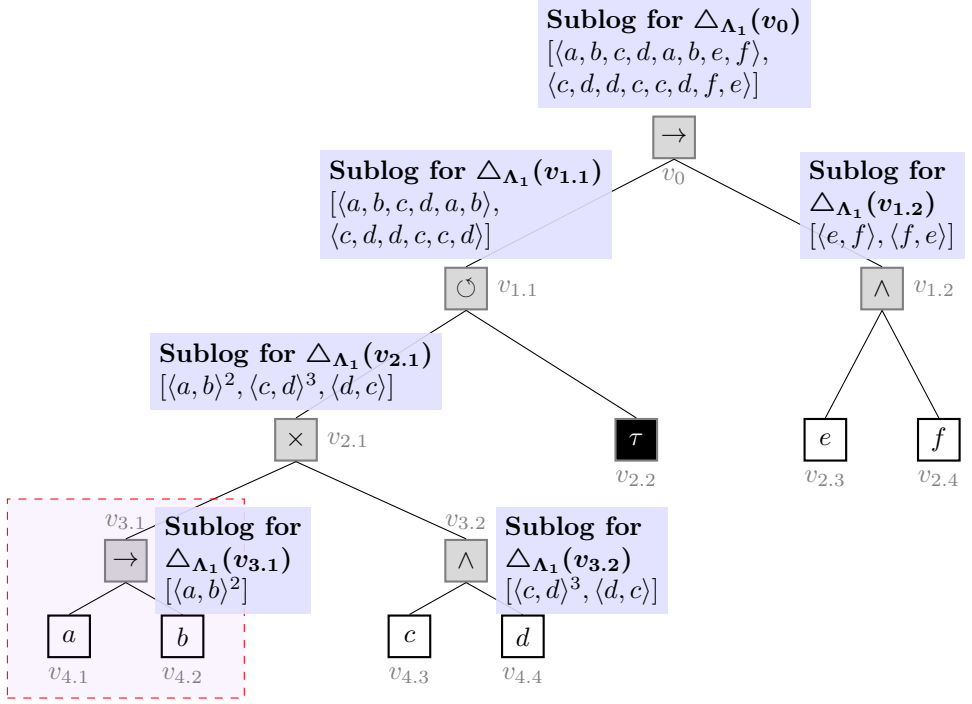


Figure 5.3: Sublogs for the process tree  $\Lambda$  and the previously added traces  $A$ ; the red highlighted subtree has been identified to be responsible for the first block of deviations in the alignment for  $\sigma_{next}$  and  $\Lambda$  (cf. Figure 5.4)

i.e., log moves on activity  $b$ . The two deviation-indicating alignment moves are next to each other and thus considered one *deviation block*. The deviation block is encompassed by two synchronous moves, i.e., a synchronous move on vertex  $v_{4.1}$  labeled  $a$  and on  $v_{4.2}$  labeled  $b$ . Next, the LCA-IPDA computes an LCA from the vertices of these two synchronous moves, i.e.,  $lca(v_{4.1}, v_{4.2}) = v_{3.1}$ . The subtree rooted at  $v_{3.1}$  is  $\Delta_{\Lambda}(v_{3.1})$ , cf. the red marked subtree in Figure 5.3. According to the computed optimal alignment, subtree  $\Delta_{\Lambda}(v_{3.1})$  causes the non-fitting of  $\sigma_{next}$ .

Next, the sublog of  $\Delta_{\Lambda}(v_{3.1})$  is extended. Therefore, the already-computed alignment is considered and the opening and closing of vertex  $v_{3.1}$ , i.e., the root vertex of the determined subtree, are searched. In the example, the subtree's root node is opened in the 4<sup>th</sup> alignment move and closed in the 9<sup>th</sup>, cf. Figure 5.3. In between, i.e., 5<sup>th</sup>–8<sup>th</sup> move, the trace fragment  $\langle a, b, b, b \rangle$  should have been executed; however, only  $\langle a, b \rangle$  is supported by the subtree. Therefore, the LCA-IPDA extends the sublog to  $[\langle a, b \rangle^2, \langle a, b, b, b \rangle]$ . The extended sublog is used as input to any fitness-preserving conventional process discovery algorithm (cf. Definition 3.32) to discover an updated subtree. The discovered subtree replaces the identified subtree in  $\Lambda$ . Since the alignment contains only one deviation block, no further adjustments are needed. Finally, LCA-IPDA returns  $\Lambda'$ , cf. Figure 5.5.



1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	...
$\gg$	$\gg$	$\gg$	$\gg$	$a$	$b$	$b$	$b$	$\gg$	$\gg$	
$(v_0, \text{open})$	$(v_{1.1}, \text{open})$	$(v_{2.1}, \text{open})$	$(v_{3.1}, \text{open})$	$(v_{4.1}, a)$	$\gg$	$\gg$	$(v_{4.2}, b)$	$(v_{3.1}, \text{close})$	$(v_{2.1}, \text{close})$	

11.	12.	13.	14.	16.	17.
$\gg$	$f$	$e$	$\gg$	$\gg$	$\gg$
$(v_{1.1}, \text{close})$	$(v_{1.2}, \text{open})$	$(v_{2.4}, f)$	$(v_{2.3}, e)$	$(v_{1.2}, \text{close})$	$(v_0, \text{close})$

Figure 5.4: An optimal alignment for process tree  $\Lambda$  (cf. Figure 5.3) and trace  $\sigma_{next} = \langle a, b, b, b, f, e \rangle$ ; the 6<sup>th</sup> and 7<sup>th</sup> move indicate a deviation

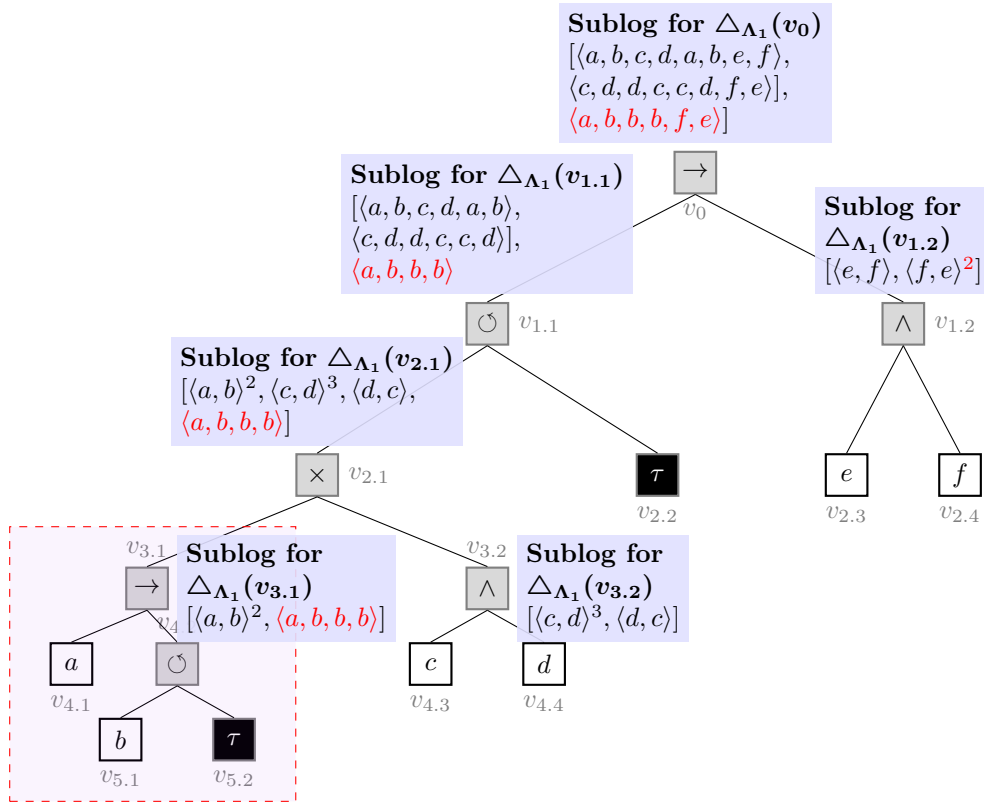


Figure 5.5: Resulting process tree  $\Lambda'$  and updated sublogs after adding previously deviating trace  $\sigma_{next} = \langle a, b, b, b, f, e \rangle$

### 5.3.2. Algorithm

Algorithm 5.2 formally presents the LCA-IPDA, which consists of three phases. In the *input preprocessing phase*, the algorithm adds artificial start  $\blacktriangleright$  and end  $\blacksquare$  activities to the tree as well as to the provided traces, cf. lines 1 to 3. We assume that the two symbols are unique, i.e.,  $\blacktriangleright, \blacksquare \notin \mathcal{A}$ . Figure 5.6 schematically depicts the extension, which is applied in line 1, of an arbitrary input tree by start and end activities. In short, a new sequence operator is created with three children: the start activity, the original tree, and the end activity. Adding artificial start and end activities to the tree and to the traces, i.e.,  $\sigma_{next}$  and traces in  $A$ , is needed to ensure that an LCA can always be found. The added start and end activities ensure that every process tree running sequence contains the execution of vertices representing the artificial start and end, i.e.,  $v_{0.1}$  and  $v_{0.2}$  in Figure 5.6. Thus, every optimal alignment for an arbitrary extended trace, i.e.,  $\sigma_{next}$  and traces in  $A$  after executing , and the extended tree contains two synchronous moves, one on the start activity ( $\blacktriangleright$ ) and one on the end activity ( $\blacksquare$ ).

---

**Algorithm 5.2: LCA-IPDA**


---

**Input:**  $A \subseteq \mathcal{A}^*, \sigma_{next} \in \mathcal{A}^*, \Lambda \in \mathcal{P}$   
**Output:**  $\Lambda \in \mathcal{P}$   
**begin**

```

1  /* input preprocessing phase */
2   $\Lambda \leftarrow \text{extend } \Lambda \text{ by artificial } \blacktriangleright \text{ and } \blacksquare \text{ activities}$  // cf. Figure 5.6
3   $A \leftarrow A \cup \{\sigma_{next}\}$  // adding  $\sigma_{next}$  to  $A$ 
4   $A \leftarrow \{\langle \blacktriangleright \rangle \circ \sigma \circ \langle \blacksquare \rangle \mid \sigma \in A\}$  // extend traces by start & end activities
5  /* main phase */
6  let  $\gamma \in \Gamma^{opt}(\Lambda, \sigma_{next})$  // calculate an optimal alignment  $\gamma$ 
7  while deviation( $\gamma$ ) do //  $\sigma_{next} \notin \mathbb{L}(\Lambda)$ 
8     $i \leftarrow \text{firstDeviationMvIndex}(\gamma)$  // first deviation-indicating move
9     $\Lambda_{LCA} \leftarrow \text{subtree}(\Lambda, \gamma, i)$  // Definition 5.2 (page 122)
10    $L_{LCA} \leftarrow \text{computeSublog}(\Lambda, \Lambda_{LCA}, A)$  // Algorithm 5.3 (page 124)
11    $\Lambda \leftarrow \text{replace } \Lambda_{LCA} \sqsubseteq \Lambda \text{ by } \text{discovery}(L_{LCA})$ 
12   let  $\gamma \in \Gamma^{opt}(\Lambda, \sigma_{next})$  // calculate an optimal alignment  $\gamma$ 
13  /* output postprocessing phase */
14   $\Lambda \leftarrow \text{remove artificial } \blacktriangleright \text{ and } \blacksquare \text{ activities from } \Lambda$  // added in line 1
15   $\Lambda \leftarrow \text{apply process tree reduction rules to } \Lambda$  // cf. [120, Chapter 5]
16  return  $\Lambda$ 

```

---

After the input preprocessing phase (cf. lines 1 to 3), the *main phase* of the LCA-IPDA starts (cf. lines 4 to 10). First, an optimal alignment  $\gamma$  is calculated for tree  $\Lambda$  and  $\sigma_{next}$  (cf. line 4). If  $\gamma$  does not indicate a deviation, we know  $\sigma_{next} \in \mathbb{L}(\Lambda)$  and thus move to the output postprocessing phase. Otherwise, the following steps are executed.

1. First, we determine the index of the first deviation-indicating alignment move in  $\gamma$  (cf. line 6). Note that such index  $i \in \{1, \dots, |\gamma|\}$  always exists because alignment  $\gamma$  contains a deviation (cf. line 5).

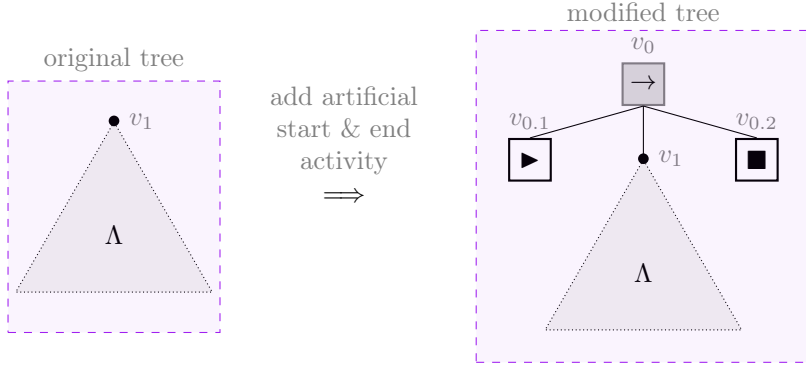


Figure 5.6: Schematic overview of extending a tree  $\Lambda$  by artificial start ( $\blacktriangleright$ ) and end ( $\blacksquare$ ) activities; overall, three new vertices  $v_0, v_{0.1}, v_{0.2}$  and corresponding edges are added

2. Function *subtree* identifies the subtree that causes the first (block of) deviation(s), cf. line 7. We present *subtree* in Definition 5.2 (page 122) in detail. In short, *subtree* identifies the subtree that causes the first (block of) deviation(s) (starting) at position  $i$  in the alignment  $\gamma$ . Note that although *subtree* is a partial function, the function always returns a tree with the inputs used in line 7.
3. The function *computeSublog* calculates the sublog for the previously determined subtree  $\Lambda_{LCA}$ , cf. line 8. We present *computeSublog* in Algorithm 5.3 in detail. In short, *computeSublog* calculates the corresponding sublog for the previously determined subtree  $\Lambda_{LCA}$ . Sublog  $L_{LCA}$  represents all relevant trace fragments from the set of previously added traces  $A$  and trace  $\sigma_{next}$  that the subtree  $\Lambda_{LCA}$  must support to resolve the first .
4. Next, a fitness-preserving conventional process discovery algorithm is invoked to discover a process tree from the previously calculated sublog. This discovered tree replaces the determined subtree, cf. line 9.
5. Finally, we recalculate an optimal alignment for the modified tree  $\Lambda$  and  $\sigma_{next}$  (cf. line 10). If the alignment indicates further deviations, we re-execute the while block (cf. lines 4 to 10). Otherwise, we exit the while block, cf. lines 5 to 10.

Eventually, the *output post-processing phase* starts (cf. lines 11 to 13). First, the in the beginning added artificial start  $\blacktriangleright$  and end  $\blacksquare$  activities are removed (cf. line 11). Next, language-preserving reduction rules are applied to simplify if applicable certain structures in the process tree (cf. line 12). Finally, the modified tree  $\Lambda$  with  $\{\sigma_{next}\} \cup A \subseteq L(\Lambda)$  is returned (cf. line 13).







## Subtree Determination

This section defines the *subtree* function that is used within Algorithm 5.2 line 7. The function  $subtree : \mathcal{P} \times \Gamma \times \mathbb{N} \rightarrow \mathcal{P}$  assumes a process tree  $\Lambda \in \mathcal{P}$ , an alignment  $\gamma \in \Gamma(\Lambda, \sigma)$  for some  $\sigma \in \mathcal{A}^*$ , and an index  $i \in \{1, \dots, |\gamma|\} \subseteq \mathbb{N}$  that represents the index of the first deviation-indicating alignment move. In short, *subtree* finds (if possible) the closest alignment moves before and after the alignment move  $\gamma(i)$ , each of which is either a synchronous move or an invisible model move; thus, alignment moves that do not indicate a deviation and can be associated to a leaf vertex in the process tree. Since the function *subtree* is called for a deviation-indicating alignment move  $\gamma(i)$  (cf. Algorithm 5.2), the two alignment moves that *subtree* aims to determine enclose the deviation-indicating alignment move  $\gamma(i)$ .

Recall the running example presented in Section 5.3.1. Figure 5.4 (page 117) shows an alignment for  $\sigma_{next}$  and the process tree  $\Lambda$  shown in Figure 5.3 (page 116). Calling *subtree* for this process tree, alignment, and the index 6 that is the index of the first deviation-indicating alignment move, *subtree* finds the 5<sup>th</sup> and 8<sup>th</sup> alignment move that are the closest alignment moves enclosing the 6<sup>th</sup> move and are both synchronous moves. Next, the two corresponding vertices  $v_{4,1}$  and  $v_{4,2}$  are derived from the 5<sup>th</sup> and 8<sup>th</sup> alignment move. Finally,  $\Delta_\Lambda(lca(v_{4,1}, v_{4,2})) = \Delta_\Lambda(v_{3,1})$  is returned, cf. red highlighted subtree in Figure 5.3.

In the following,  $i_{before}$  denotes the index of the alignment move that is before  $\gamma(i)$  and represents a synchronous or invisible model move on a leaf vertex. Likewise,  $i_{after}$  denotes the index of the alignment move that is after  $\gamma(i)$  and represents a synchronous or invisible model move on a leaf vertex. In the following, we simplify by saying that  $i_{before}$  or  $i_{after}$  exists or does not exist, respectively. Since both  $i_{before}$  and  $i_{after}$  could not exist, four cases can be distinguished. Figure 5.7 visualizes these four cases. The first case applies if  $i_{before}$  and  $i_{after}$  exist; thus, the first (block of) deviation-indicating alignment move(s) (starting) at index  $i$  is enclosed by synchronous or invisible model moves on a leaf vertex. The second case applies if  $i_{after}$  does not exist; thus, all alignment moves from index  $i$  indicate a deviation or are invisible model moves on inner vertices. The third case applies if  $i_{before}$  does not exist; thus, all alignment moves before  $i$  are invisible model moves on inner vertices. Finally, the fourth case applies if neither  $i_{before}$  nor  $i_{after}$  exists; thus, no subtree can be determined. Note that the fourth case cannot apply if *subtree* is called in Algorithm 5.2 because any optimal alignment for which *subtree* is invoked contains at least two synchronous moves, i.e., one on the artificial start (►) and one on the end (■) activity. Thus, at least  $i_{before}$  or  $i_{after}$  exists.<sup>4</sup> Subsequently, we formally define *subtree*.

<sup>4</sup>In the subsequent chapter, however, we deal with trace fixes for which case 4 can occur. Therefore, and for the sake of completeness, we present all cases here.

Color	Interpretation	Included Alignment Moves				
		Log	Sync.	Vis. Model	Inv. Model	
					on inner vertices	on leaf vertices
	no deviation-indicating moves	—	✓	—	✓	✓
	no deviation-indicating moves excluding inv. model moves on inner vertices	—	✓	—	—	✓
	deviation-indicating moves	✓	—	✓	—	—
	deviation-indicating moves and invisible model moves on inner vertices	✓	—	✓	✓	—
	invisible model moves on inner vertices	—	—	—	✓	—
	any move	✓	✓	✓	✓	✓

(a) Legend for Figures 5.7b to 5.7e; each color represents certain types of alignment moves

$$\gamma = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & \cdots & i_{\text{before}}-1 & i_{\text{before}} & \cdots & i & \cdots & i_{\text{after}} & i_{\text{after}}+1 & \cdots & |\gamma| \\ \hline \text{dark blue} & \text{cyan} & \text{light green} & \text{red} & \text{orange} & \text{cyan} & \text{yellow} & \text{yellow} & \text{yellow} & \text{yellow} & \text{yellow} \\ \hline \end{array}$$

(b) Case 1: before ( $i_{\text{before}}$ ) and after ( $i_{\text{after}}$ ) the first deviation-indicating alignment move  $i$  exists a synchronous move or an invisible model move on a leaf vertex

$$\gamma = \begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline 1 & \cdots & i_{\text{before}}-1 & i_{\text{before}} & \cdots & i & i+1 & \cdots & |\gamma| \\ \hline \text{dark blue} & \text{cyan} & \text{light green} & \text{red} & \text{orange} & \text{orange} & \text{orange} & \text{orange} & \text{orange} \\ \hline \end{array}$$

(c) Case 2: only before ( $i_{\text{before}}$ ) the first deviation-indicating alignment move  $i$  exists a synchronous move or an invisible model move on a leaf vertex

$$\gamma = \begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline 1 & \cdots & i-1 & i & \cdots & i_{\text{after}} & i_{\text{after}}+1 & \cdots & |\gamma| \\ \hline \text{light green} & \text{light green} & \text{light green} & \text{red} & \text{orange} & \text{cyan} & \text{yellow} & \text{yellow} & \text{yellow} \\ \hline \end{array}$$

(d) Case 3: only after ( $i_{\text{after}}$ ) the first deviation-indicating alignment move  $i$  exists a synchronous move or an invisible model move on a leaf vertex

$$\gamma = \begin{array}{|c|c|c|c|c|c|c|} \hline 1 \cdots i-1 & i & i+1 & \cdots & |\gamma| \\ \hline \text{light green} & \text{red} & \text{orange} & \text{orange} & \text{orange} \\ \hline \end{array}$$

(e) Case 4: neither before nor after the first deviation-indicating alignment move  $i$  exists a synchronous move or an invisible model move on a leaf vertex

Figure 5.7: Illustration of the four cases distinguished by *subtree* (cf. Definition 5.2) regarding the composition of the alignment  $\gamma$

**Definition 5.2** (Subtree Detection Function)

The partial function  $\text{subtree} : \mathcal{P} \times \Gamma \times \mathbb{N} \rightarrow \mathcal{P}$  takes as input a process tree  $\Lambda \in \mathcal{P}$ , an alignment  $\gamma \in \Gamma(\Lambda, \sigma)$  for some  $\sigma \in \mathcal{A}^*$ , and an index  $i \in \{1, \dots, |\gamma|\} \subseteq \mathbb{N}$ . The function is undefined for  $i \notin \{1, \dots, |\gamma|\}$  and for  $\gamma \in \Gamma(\Lambda', \sigma)$  with  $\Lambda \neq \Lambda' \in \mathcal{P}$ . We distinguish four cases.

- **Case 1:** Before and after the alignment move  $\gamma(i)$  indicating a deviation there is respectively a synchronous move or an invisible model move that executes a leaf vertex, i.e., a vertex labeled  $\tau$ . Thus, if

$$\begin{aligned} \exists i_{\text{before}} \in \{1, \dots, i-1\} \left[ \left( \text{syncMv}(\gamma(i_{\text{before}})) \vee \text{invModelMvTau}(\gamma(i_{\text{before}}))) \wedge \right. \right. \\ \left. \forall j \in \{i_{\text{before}}+1, \dots, i-1\} \left( \neg \text{syncMv}(\gamma(i_{\text{before}})) \wedge \right. \right. \\ \left. \left. \neg \text{invModelMvTau}(\gamma(i_{\text{before}}))) \right) \right] \end{aligned} \quad (5.1)$$

and

$$\begin{aligned} \exists i_{\text{after}} \in \{i+1, \dots, |\gamma|\} \left[ \left( \text{syncMv}(\gamma(i_{\text{after}})) \vee \text{invModelMvTau}(\gamma(i_{\text{after}}))) \wedge \right. \right. \\ \left. \forall j \in \{i+1, \dots, i_{\text{after}}-1\} \left( \neg \text{syncMv}(\gamma(i_{\text{after}})) \wedge \right. \right. \\ \left. \left. \neg \text{invModelMvTau}(\gamma(i_{\text{after}}))) \right) \right] \end{aligned} \quad (5.2)$$

the following subtree is returned.

$$\text{subtree}(\gamma, i, \Lambda) = \Delta_{\Lambda} \left( \text{lca}_{\Lambda} \left( \text{modelVertex}(\gamma(i_{\text{before}})), \text{modelVertex}(\gamma(i_{\text{after}})) \right) \right)$$

- **Case 2:** Before the alignment move  $\gamma(i)$  indicating a deviation there is a synchronous move or an invisible model move that executes a leaf vertex. However, there is neither a synchronous move nor an invisible model move that executes a leaf vertex after alignment move  $\gamma(i)$ . Thus, if Equation (5.1) holds and

$$\forall i_{\text{after}} \in \{i+1, \dots, |\gamma|\} \left[ \neg \text{syncMv}(\gamma(i_{\text{after}})) \wedge \neg \text{invModelMvTau}(\gamma(i_{\text{after}})) \right] \quad (5.3)$$

the following subtree is returned.

$$\text{subtree}(\gamma, i, \Lambda) = \Delta_{\Lambda} \left( \text{modelVertex}(\gamma(i_{\text{before}})) \right)$$

- **Case 3:** After the alignment move  $\gamma(i)$  indicating a deviation there is a synchronous move or an invisible model move that executes a leaf vertex; however,

there is neither asynchronous move nor an invisible model move that executes a leaf vertex before alignment move  $\gamma(i)$ . Thus, if Equation (5.2) holds and

$$\forall i_{\text{before}} \in \{1, \dots, i-1\} \left[ \neg \text{syncMv}(\gamma(i_{\text{before}})) \wedge \neg \text{invModelMvTau}(\gamma(i_{\text{before}})) \right] \quad (5.4)$$

the following subtree is returned.

$$\text{subtree}(\gamma, i, \Lambda) = \Delta_{\Lambda}(\text{modelVertex}(\gamma(i_{\text{after}})))$$

- **Case 4:** Neither before nor after the alignment move  $\gamma(i)$  exists a synchronous move or an invisible model move that executes a leaf vertex.<sup>a</sup> Thus, if Equations (5.2) and (5.3) hold, the function  $\text{subtree}(\gamma, i, \Lambda)$  is undefined, i.e., nothing is returned.

<sup>a</sup>Note that when invoking function  $\text{subtree}$  in Algorithm 5.2, case 4 never applies because the alignments for which  $\text{subtree}$  is invoked always contain a synchronous move on the start activity  $\blacktriangleright$  and a synchronous move on the end activity  $\blacksquare$ . Thus, for each alignment move, at least one synchronous move exists either before or after it. Hence,  $\text{subtree}$  always returns a subtree when invoked by Algorithm 5.2.

## Sublog Calculation

This section introduces the algorithm  $\text{computeSublog}$ , which is called in Algorithm 5.2 line 8 (page 118), for calculating the sublog corresponding to the determined subtree causing the first (block of) deviation(s). Algorithm  $\text{computeSublog}$  takes three inputs:

1. the entire process tree  $\Lambda \in \mathcal{P}$ ,
2. the determined subtree  $\Lambda_{LCA} \sqsubseteq \Lambda$  causing the first (block of) deviation(s), and
3. the set of traces  $A \subseteq \mathcal{A}^*$  that contains the set of previously added traces and the trace to be added next  $\sigma_{\text{next}}$ .

Algorithm  $\text{computeSublog}$  returns an event log that specifies traces that the determined subtree  $\Lambda_{LCA}$  should support. We refer to this returned event log as *sublog*.

**Introductory Example** Before we formally introduce  $\text{computeSublog}$ , we provide an example.<sup>5</sup> Recall the running example introduced in Section 5.3.1. The entire process tree  $\Lambda$  is shown Figure 5.3 (page 116) including the detected subtree  $\Lambda_{LCA}$ , which is highlighted in red. Further, assume the below-specified set of previously added traces.

$$A = \left\{ \langle a, b, c, d, a, b, e, f \rangle, \langle c, d, d, c, c, d, f, e \rangle, \langle a, b, b, b, f, e \rangle \right\}$$

Note that when  $\text{computeSublog}$  is called from Algorithm 5.2 in line 8, the set  $A$  contains all previously added traces and the trace to be added next, i.e.,  $\sigma_{\text{next}} = \langle a, b, b, b, f, e \rangle$

<sup>5</sup>For the sake of simplicity, we omit the input preprocessing phase of Algorithm 5.2; thus, we do not extend all traces and the tree by an artificial start  $\blacktriangleright$  and end  $\blacksquare$  activity. In the example we will present, these artificial activities have no effect. Note, however, that these artificial start and end activities are generally necessary for the algorithm to work correctly.

in the example.<sup>6</sup> Next, for each trace in  $A$  an optimal alignment is computed. Since except of  $\sigma_{next} \in A$  all other traces fit the process tree  $\Lambda$ , the corresponding alignments contain only synchronous moves and invisible model moves, i.e., these alignments do not indicate a deviation. For non-deviation-indicating alignments, we search for open and closing invisible model moves on the root vertex of the determined subtree  $\Lambda_{LCA}$ . From all synchronous moves between opening and closing of the root vertex that belong to the subtree  $\Lambda_{LCA}$ , the trace for the sublog of  $\Lambda_{LCA}$  is extracted.

**Algorithm *computeSublog*** Algorithm 5.3 introduces *computeSublog*, which we exemplified and informally introduced above. First, the eventually returned sublog  $L_{LCA}$  for the subtree  $\Lambda_{LCA}$  is initialized, cf. line 1. For each trace in  $A$  (cf. line 2), *computeSublog* computes an optimal alignment on the tree  $\Lambda$ , cf. line 3. Next, the computed alignment  $\gamma$  and the subtree  $\Lambda_{LCA}$  are provided to the algorithm *extractSubTraces*, which extracts the relevant sub-traces for  $\Lambda_{LCA}$  from the provided alignment, cf. line 4. After an alignment has been computed and corresponding sub-traces have been extracted for all traces in  $A$ , the sublog  $L_{LCA}$  is finally returned, cf. line 5.

---

**Algorithm 5.3:** *computeSublog* (called in Algorithm 5.2 line 8)

---

```

input :  $\Lambda \in \mathcal{P}$ , // entire process tree
         $\Lambda_{LCA} \sqsubseteq \Lambda$ , // subtree causing deviation(s)
         $A \subseteq A^*$  // contains previously added traces and  $\sigma_{next}$ 
output:  $L_{LCA} \in \mathcal{M}(A^*)$  // sub-log for  $\Lambda_{LCA}$ 
begin
1    $L_{LCA} \leftarrow []$  // initialize sub-log for  $\Lambda_{LCA}$ 
2   forall  $\sigma \in A$  do
3       let  $\gamma \in \Gamma^{opt}(\Lambda, \sigma)$  // calculate an optimal alignment  $\gamma$ 
4        $L_{LCA} \leftarrow L_{LCA} \uplus \text{extractSubTraces}(\Lambda_{LCA}, \gamma)$  // Algorithm 5.4
5   return  $L_{LCA}$ 

```

---

**Algorithm *extractSubTraces*** Algorithm 5.4 introduces *extractSubTraces* that is exclusively called in Algorithm 5.3 line 4. The inputs of *extractSubTraces* are the determined subtree  $\Lambda_{LCA} \sqsubseteq \Lambda$  and an optimal alignment  $\gamma \in \Gamma^{opt}(\Lambda, \sigma)$  for some  $\sigma \in A$ , cf. Algorithm 5.3. Overall, *extractSubTraces* iterates over the alignment  $\gamma$  (cf. line 2) and creates thereby trace(s)  $\sigma'$  that are stored in the sublog  $L_{LCA}$ . Generally, two case distinctions based on the subtree  $\Lambda_{LCA}$  are made by *extractSubTraces*. The first case applies if the subtree  $\Lambda_{LCA}$  contains only one vertex; hence,  $\Lambda_{LCA}$  represents a leaf vertex in  $\Lambda$  (cf. line 4). The second case applies if the subtree  $\Lambda_{LCA}$  contains more than one vertex; hence,  $\Lambda_{LCA}$ 's root vertex represents a process tree operator (cf. line 15). Note that per call of *extractSubTraces*, only one of the two cases applies since the subtree  $\Lambda_{LCA}$  does not change during executing *extractSubTraces*. Subsequently, we present the two cases in detail.

---

<sup>6</sup>Algorithm 5.2 adds  $\sigma_{next}$  to  $A$  in line 2.



**Algorithm 5.4:** *extractSubTraces* (called in Algorithm 5.3 line 4)

---

```

input :  $\Lambda_{LCA} = (V_{LCA}, E_{LCA}, \Sigma_{LCA}, \lambda_{LCA}, r_{LCA}, <_{LCA}) \sqsubseteq \Lambda \in \mathcal{P}$ ,
         $\gamma \in \Gamma^{opt}(\Lambda, \sigma)$  // opt. alignment  $\gamma$  for some  $\sigma \in \mathcal{A}^*$  and a tree  $\Lambda$ 
output:  $L_{LCA} \in \mathcal{M}(\mathcal{A}^*)$  // sublog for  $\Lambda_{LCA}$ 
begin
1   $L_{LCA} \leftarrow []$  // initialize sublog for  $\Lambda_{LCA}$ 
2  forall  $1 \leq i \leq |\gamma|$  do // iterate over alignment moves
3       $\sigma' \leftarrow \langle \rangle$  // initialize trace eventually added to  $L_{LCA}$ 
4      if  $V_{LCA} = \{r_{LCA}\}$  then // Case 1:  $\Lambda_{LCA}$  contains only one vertex  $r_{LCA}$ 
5          while  $modelVertex(\gamma(i)) \neq r_{LCA}$  do
6              if  $logMv(\gamma(i))$  then
7                   $\sigma' \leftarrow \sigma' \circ \langle traceLabel(\gamma(i)) \rangle$  // add log moves
8                   $i \leftarrow i+1$ 
9              if  $modelVertex(\gamma(i)) = r_{LCA}$  then
10                  $\sigma' \leftarrow \sigma' \circ \langle modelLabel(\gamma(i)) \rangle$  //  $modelLabel(\gamma(i)) = \lambda_{LCA}(r_{LCA})$ 
11                  $i \leftarrow i+1$ 
12                 if  $\forall i \leq j \leq |\gamma| \left( \neg syncMv(\gamma(j)) \wedge \neg invModelMv(\gamma(j)) \right)$  then
13                     // no more synchronous move or invisible model move that would
14                     // determine a new LCA in the next iteration of Algorithm 5.2; thus,
15                     // we add the remaining trace labels (ignoring  $\gg$ ) to  $\sigma'$ 
16                      $\sigma' \leftarrow \sigma' \circ \langle traceLabel(\gamma(j)), \dots, traceLabel(\gamma(|\gamma|)) \rangle_{\downarrow \mathcal{A}}$ 
17                  $L_{LCA} \leftarrow L_{LCA} \uplus [\sigma']$  // add trace  $\sigma'$  to the sublog of  $\Lambda_{LCA}$ 
18             else // Case 2:  $T_{LCA}$  contains more than one vertex
19                 if  $modelVertex(\gamma(i)) = r_{LCA} \wedge modelLabel(\gamma(i)) = open$  then
20                     // current move  $i$  represents an opening of  $\Lambda_{LCA}$ 's root  $r_{LCA}$ 
21                     while  $modelVertex(\gamma(i)) \neq r_{LCA} \vee modelLabel(\gamma(i)) \neq close$  do
22                         // consider all subsequent moves until  $r_{LCA}$  is closed
23                         if  $modelVertex(\gamma(i)) \in V_{LCA} \wedge syncMv(\gamma(i))$  then
24                              $\sigma' \leftarrow \sigma' \circ \langle modelLabel(\gamma(i)) \rangle$ 
25                         else if  $traceLabel(\gamma(i)) \in \mathcal{A}$  then //  $traceLabel(\gamma(i)) \neq \gg$ 
26                              $\sigma' \leftarrow \sigma' \circ \langle traceLabel(\gamma(i)) \rangle$ 
27                          $i \leftarrow i+1$ 
28                      $L_{LCA} \leftarrow L_{LCA} \uplus [\sigma']$  // add trace  $\sigma'$  to the sublog of  $\Lambda_{LCA}$ 
29 return  $L_{LCA}$ 

```

---

In the first case (cf. line 4), subtree  $\Lambda_{LCA}$  contains only one vertex, i.e., the root vertex  $r_{LCA}$ . Hence,  $\Lambda_{LCA}$  represents a leaf vertex of  $\Lambda$ . In this case, *extractSubTraces* looks for executions of  $r_{LCA}$ , cf. line 9. In case log moves before the execution of  $r_{LCA}$  exist, we add these to the sub-trace  $\sigma'$  (cf. lines 5 to 7). Adding these log moves to  $\sigma'$  is required since the eventually returned sublog  $L_{LCA}$  containing  $\sigma'$  represents all traces that tree  $\Lambda_{LCA}$  should support. Next, we add the label of the root vertex  $r_{LCA}$  to  $\sigma'$  (cf. line 10). Finally, we check if all subsequent alignment moves after  $r_{LCA}$  was executed are neither synchronous moves or invisible model moves (cf. line 12); thus, only log or visible model moves follow. If this is the case, we know that in the next iteration of Algorithm 5.2 no other subtree  $\Lambda_{LCA}$  can be found; thus, we add the remaining trace labels to  $\sigma'$  (cf. line 13). Finally, we add  $\sigma'$  to  $L_{LCA}$  (cf. line 14).

Note that *extractSubTraces* is called for alignments indicating a deviation and for alignments indicating no deviation. Recall Algorithm 5.3 that calls *extractSubTraces* for each  $\sigma \in A$ . Further, recall that all traces except  $\sigma_{next}$  in  $A$  fit the tree  $\Lambda$ . Thus, per execution of Algorithm 5.3, *extractSubTraces* is called once with a deviation-indicating alignment and  $|A| - 1$  times with alignments that indicate no alignment. For alignments that indicate no deviation, i.e., neither log moves nor visible model moves exist, only traces of length one containing the label of  $r_{LCA}$  are added to  $L_{LCA}$ . Thus, only lines 9 to 11 and 14 are executed.

In the second case (cf. line 15), subtree  $\Lambda_{LCA}$  contains more than one vertex. Thus, its root vertex  $r_{LCA}$  represents a process tree operator. Thus, we look for the invisible model move representing the opening of  $r_{LCA}$  (cf. line 16). Next, we iterate over the alignment until  $r_{LCA}$  is closed (cf. line 17).<sup>7</sup> Between the opening and closing of  $r_{LCA}$ , we add the activity label of 1) synchronous moves that belong to  $\Lambda_{LCA}$  and 2) log moves (cf. line 19). Finally, we add trace  $\sigma'$  to the sublog  $L_{LCA}$  and process the alignment  $\gamma$  further if applicable, i.e., we look for the next opening and closing of  $r_{LCA}$ .

**Detailed Example** Recall the running example presented in Section 5.3.1. When applying *extractSubTraces* to the alignment shown in Figure 5.4 (page 117) and the determined subtree  $\Lambda_{LCA}$  highlighted red in Figure 5.3 (page 116) with root vertex  $v_{3.1}$ , *extractSubTraces* returns the sublog  $L_{LCA} = [\langle a, b, b, b \rangle]$ . In detail, the second case of *extractSubTraces* applies because  $\Lambda_{LCA}$  comprises multiple vertices, cf. line 15. Next, the (first) opening of  $\Lambda_{LCA}$ 's root vertex  $v_{3.1}$  is found, i.e., the 4<sup>th</sup> alignment move, cf. Figure 5.4 (page 117). The next move, i.e., the 5<sup>th</sup> one, is a synchronous move on activity  $a$  and the corresponding executed vertex  $v_{4.1} \in \Lambda_{LCA}$ ; thus, we add activity  $a$  to trace  $\sigma' = \langle a \rangle$  (cf. lines 18 and 19 in Algorithm 5.4). The 6<sup>th</sup> and 7<sup>th</sup> move represent log moves on activity  $b$ . Thus, we add two times activity  $b$  to the trace  $\sigma' = \langle a, b, b \rangle$  (cf. lines 18 and 19). The 8<sup>th</sup> move is a synchronous move on activity  $b$  and the corresponding vertex  $v_{4.2} \in \Lambda_{LCA}$ . Thus, we add activity  $b$  to  $\sigma' = \langle a, b, b, b \rangle$  (cf. lines 18 and 19). The 9<sup>th</sup> move closes the root vertex of  $\Lambda_{LCA}$ ; hence, we add  $\sigma' = \langle a, b, b, b \rangle$  to  $L_{LCA}$  (cf. line 23). Since the root vertex is after the 9<sup>th</sup> move never opened again, we finally return  $L_{LCA} = [\langle a, b, b, b \rangle]$  (cf. line 24).

<sup>7</sup>Note that a corresponding invisible model move representing the closing of  $r_{LCA}$  must exist in  $\gamma$  because the model part of alignment  $\gamma$  represents a running sequence of  $\Lambda$ , cf. Definition 3.34

### 5.3.3. Summary & Termination

This section summarizes the proposed LCA-IPDA, cf. Algorithm 5.2. In short, the LCA-IPDA utilizes alignments to assess if the trace to be added next  $\sigma_{next}$  fits the provided tree  $\Lambda$ . Note that the previously added traces contained in  $A$  are assumed to fit the tree. If  $\sigma_{next}$  does not fit, the subtree  $\Lambda_{LCA} \subseteq \Lambda$  that is responsible for the first (block of) deviation(s) is determined, cf. Definition 5.2. The objective is to replace this determined subtree  $\Lambda_{LCA}$  with another one that resolves the first (block of) deviation(s). To this end, we calculate a sublog that contains all trace fragments that the new subtree should support such that all previously added traces still fit the altered overall tree and the first (block of) deviation(s) in the alignment for  $\sigma_{next}$  is resolved. Therefore, optimal alignments for all previously added traces and the trace to be added next are aligned with the entire tree  $\Lambda$ . From these alignments, a sublog  $L_{LCA}$  for the determined subtree  $\Lambda_{LCA}$  is calculated. This sublog reflects the trace fragments that the subtree  $\Lambda_{LCA}$  must support. Note that from the alignments for the previously added traces and  $\Lambda$ , only trace fragments that are already fitting the subtree  $\Lambda_{LCA}$  are extracted. Only from the alignment for the trace to be added next  $\sigma_{next}$  and  $\Lambda$ , the corresponding sublog is extended by trace(s) that do not fit the determined subtree.

The presented LCA-IPDA in Algorithm 5.2 guarantees termination because in each iteration (cf. lines 4 to 10) the first (block of) deviation(s) is resolved. Thus, eventually all deviations between  $\Lambda$  and  $\sigma_{next}$  are resolved. As elaborated in Section 5.3.2, function *subtree*, although a partial function, always returns a subtree  $\Lambda_{LCA}$  when invoked in Algorithm 5.2 because any alignment provided contains at least two synchronous moves on the artificial start and end activity. Further, the corresponding computed sublog  $L_{LCA}$  for the determined subtree  $\Lambda_{LCA}$  contains all trace fragments that are required for replaying traces from  $A$  and corresponding trace fragments from  $\sigma_{next}$  to ensure that the first (block of) deviation(s) is resolved. By invoking a fitness-preserving discovery algorithm in Algorithm 5.2 line 9 the new subtree that replaces  $\Lambda_{LCA}$  is guaranteed to support all trace fragments from the computed sublog  $L_{LCA}$ . Therefore, in each iteration the respective first (block of) deviation(s) is resolved until eventually  $\sigma_{next} \in \mathbb{L}(\Lambda)$ .

### 5.3.4. LCA Lowering

This section describes an extension to the above presented LCA-IPDA. The extension denoted as *LCA lowering* applies to the subtree determination function *subtree*, cf. Definition 5.2 (page 122). Recall the first case of the function *subtree*, i.e., synchronous moves or invisible model moves on leaf vertices exist at position  $i_{before} < i$  and  $i_{after} > i$  in the alignment.<sup>8</sup> Figure 5.7b (page 121) visualizes the described first case. Thus, the (block of) deviation(s) (starting) at position  $i$  in the alignment is enclosed by two alignment moves at positions  $i_{before}$  and  $i_{after}$  that do not indicate a deviation and execute a leaf vertices in the corresponding process tree. In the following, we refer to these two leaf vertices as  $v_{i_{before}}$  and  $v_{i_{after}}$ . Next, vertices  $v_{i_{before}}$  and  $v_{i_{after}}$  are used to determine the LCA (cf. case 1 of Definition 5.2). This LCA represents the root vertex of the subtree that is eventually altered by Algorithm 5.2 in line 9.

<sup>8</sup>Recall that at position  $i$  the first deviation-indicating alignment move exists.

The objective of the *LCA lowering* extension is to reduce the size of the subtree returned by *subtree* that will eventually be altered by Algorithm 5.2. To this end, *LCA lowering* alters the process tree such that the function *subtree* determines a smaller subtree. This altering of the process tree  $\Lambda$  is language-preserving; thus, *LCA lowering* only applies structural modifications to the process tree that do not change the language of the tree. According to the process tree operator of the original LCA detected by *subtree*, two distinctions can be made. Figure 5.8 illustrates these two cases. On the left side, the original tree  $\Lambda$  is depicted, and on the right side, the modified tree  $\Lambda'$  is shown, which leads to a smaller subtree.

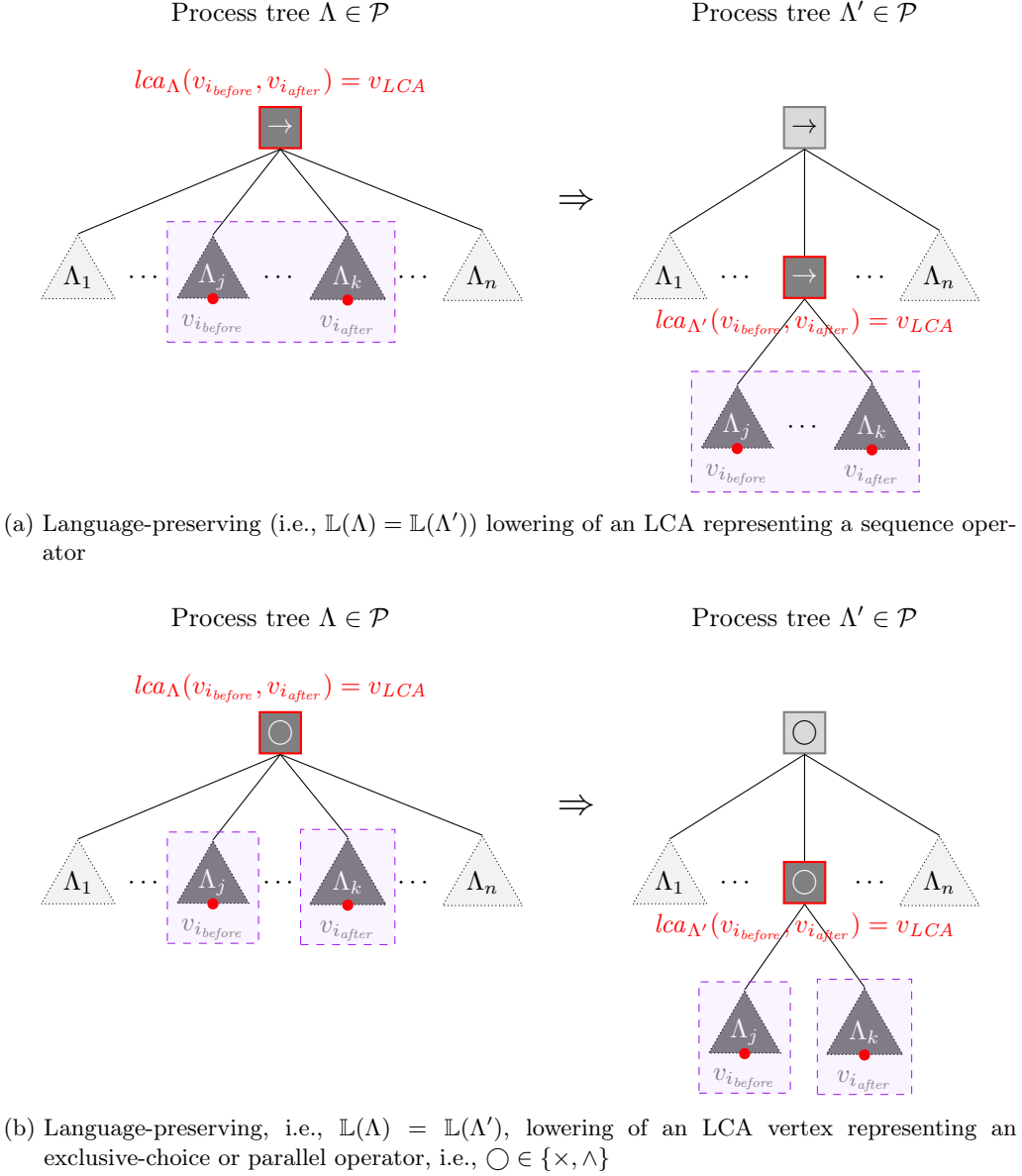
- **Case 1: LCA vertex  $lca_{\Lambda}(v_{i_{before}}, v_{i_{after}}) = v_{LCA}$  represents a sequence operator, i.e.,  $\lambda(v_{LCA}) = \rightarrow$**

In this case, subtree  $\Lambda_j$  containing vertex  $v_{i_{before}}$ , subtree  $\Lambda_k$  containing  $v_{i_{after}}$  and all subtrees in between, i.e.,  $\Lambda_{j+1}$  to  $\Lambda_{k-1}$ , are lowered. Lowering implies that the root vertices of these subtrees are moved one hierarchy level down and are connected to a new vertex representing a sequence operator. Figure 5.8a illustrates the described process tree transformation yielding  $\Lambda'$ . When determining the subtree from  $\Lambda'$  instead of  $\Lambda$  (cf. Figure 5.8a), the function *subtree* returns the subtree rooted at the newly added vertex representing a sequence operator, cf. the red highlighted vertex in  $\Lambda'$  in Figure 5.8a. Thus, the subtree that is eventually altered by Algorithm 5.2, i.e., the subtree rooted at the lowered  $v_{LCA}$  vertex, is smaller. In detail, subtrees  $\Lambda_1, \dots, \Lambda_{j-1}, \Lambda_{k+1}, \dots, \Lambda_n$  are not altered.

- **Case 2: LCA vertex  $lca_{\Lambda}(v_{i_{before}}, v_{i_{after}}) = v_{LCA}$  represents a parallel or exclusive-choice operator, i.e.,  $\lambda(v_{LCA}) \in \{\rightarrow, \wedge\}$**

In this case, subtree  $\Lambda_j$  containing vertex  $v_{i_{before}}$  and subtree  $\Lambda_k$  containing  $v_{i_{after}}$  are lowered. Note that the subtrees in between, i.e.,  $\Lambda_{j+1}$  to  $\Lambda_{k-1}$ , are *not* lowered. Lowering implies that the root vertices of these two subtrees are moved one hierarchy level down and are connected to a new vertex representing either an exclusive-choice or parallel operator respectively. Figure 5.8b illustrates the described process tree transformation yielding  $\Lambda'$ . When determining the subtree from  $\Lambda'$  instead of  $\Lambda$  (cf. Figure 5.8a), the function *subtree* returns the subtree rooted at the newly added vertex representing an exclusive-choice or parallel operator, cf. the red highlighted vertex in  $\Lambda'$  in Figure 5.8b. Thus, the subtree that is eventually altered by Algorithm 5.2, i.e., the subtree rooted at the lowered  $v_{LCA}$  vertex, is smaller. In detail, subtrees  $\Lambda_1, \dots, \Lambda_{j-1}, \Lambda_{j+1}, \dots, \Lambda_{k-1}, \Lambda_{k+1}, \dots, \Lambda_n$  are not altered.

In conclusion, *LCA lowering* alters the subtree  $\Lambda$  such that the determined subtree by *subtree* that is eventually changed by Algorithm 5.2 in line 9 is smaller. As a result, less already discovered parts of the process tree are subject to be altered. The presented lowering rules in Figure 5.8 only apply to the first case of the function *subtree*; thus, there exists  $i_{before}$  and  $i_{after}$ , cf. Figure 5.7b. In the other cases, i.e., the second and third ones (cf. Definition 5.2 on page 122), the determined subtree comprises only a leaf vertex and, thus, cannot be smaller.


 Figure 5.8: Language-preserving (i.e.,  $\mathbb{L}(\Lambda) = \mathbb{L}(\Lambda')$ ) lowering of an LCA vertex

## 5.4. Evaluation

This section presents an evaluation of the presented IPDAs. Section 5.4.1 presents the experimental setup while Section 5.4.2 presents the results.

### 5.4.1. Experimental Setup

In the experiments, we compare the proposed LCA-IPDA against the Inductive Miner (IM) [122], which discovers a process tree that accepts the given event log, and the model repair approach presented in [86]. Note that the repair algorithm does not guarantee to return a hierarchical process model. Since both the LCA-IPDA and the IM algorithm guarantee the above mentioned properties for the returned process tree, we use the IM algorithm as a comparison algorithm. Furthermore, we use the IM algorithm inside the proposed LCA-IPDA as an instantiation of the *discovery* function, cf. Algorithm 5.2.

As input, we use a publicly available event log that contains data about a road fine management process [56]. We use the complete event log, for example, we do not filter outliers. We sorted the event log based on variant frequencies in descending order, i.e., the most occurring variant first. We chose this sorting since in real applications it is common to consider first the most frequent behavior and filter out infrequent behavior. Note that the order of traces influences the resulting process model in our approach and in the model repair approach.

We use the F-measure regarding the whole event log to compare the obtained process models. The F-measure takes the *harmonic mean* of a process model's *precision* and *fitness* concerning a given event log. Fitness reflects how well a process model can replay a given event log. In contrast, precision reflects how much additional behavior not present in a given event log is supported by the process model. The aim is that both the fitness and the precision and, thus, the F-measure are close to one. This thesis uses alignment-based approaches for fitness [226] and precision calculations [8]. To this end, we transform all process trees into WF-nets and apply the alignment-based approaches to these WF-nets. In addition to the F-measure, we also report the size of the incrementally discovered WF-nets, i.e., the sum of places and transitions. The size provides an indication of the complexity or simplicity of the model.<sup>9</sup>

The procedure of the conducted experiments is described below. First, we discover a process tree on the:

- first variant,
- top 1% variants,
- top 2% variants,
- top 5% variants, and
- top 10% variants

with the IM algorithm since the LCA-IPDA and the model repair algorithm require an initial process model. Note that the LCA-IPDA can be used with any initial model. Afterward, we add variant by variant to the initially given process model with the LCA-IPDA. Analogously, we repair the initially given process model trace by trace with the

<sup>9</sup>Note that a large number of complexity measures exists [154], each focusing on different aspects. However, most complexity measures have in common that they incorporate the number of graphical symbols.

model repair algorithm. In addition, we iteratively apply the IM algorithm on the 1<sup>st</sup> variant, the 1<sup>st</sup>+2<sup>nd</sup> variant, etc.

### 5.4.2. Results

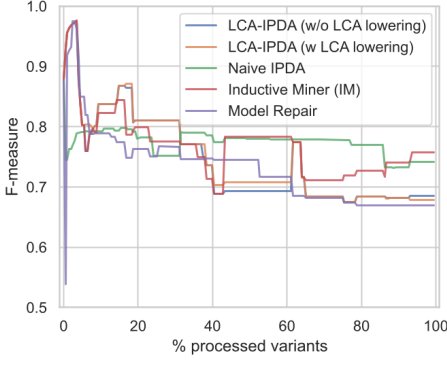
Figures 5.9 to 5.11 show the results regarding the F-measure for the road traffic fine management [56], the receipt [110], and the hospital billing event log [133].<sup>10</sup> In each plot, the approaches started from a different initial process model. Overall, there is no dominant approach that clearly outperforms the other approaches. Note that the goal is not to outperform at 100% of processed variants. Instead, it depends on the user's needs and how much behavior the process model should describe. Moreover, the three event logs exhibit a power law distribution [214], i.e., a few trace variants have a very high frequency, while most trace variants have a low frequency. Since we sort by trace frequency, after adding the first trace variants, the fitness value of the models quickly approaches values close to 1. Therefore, the dynamics we see in the F-measure are mainly due to changes in the precision values.

Surprisingly, the naive IPDA yields models with high F-measure values. Similarly, also the model repair approach sometimes yields models with F-measure values. However, when investigating the corresponding models, these models are significantly larger compared to the models discovered by the other approaches. Figure 5.13 plots the number of places and transitions per incrementally discovered WF-net. Note that we transformed the process trees into WF-nets to compare these models with the models returned by the model repair approach, which operates on Petri nets. Since the models discovered by the naive IPDA and the model repair approach quickly become very large, they are of little use as they are hardly readable for users. Furthermore, recall that the naive approach only adds vertices to an existing tree, i.e., does not add parallel-labeled and loop-labeled vertices, cf. Section 5.2. Therefore, if the initial process tree provided to the naive IPDA does not contain any vertex labeled with a loop or parallel operator, all incrementally discovered process trees also do not contain such vertices. In short, the naive IPDA discovers only a very restricted class of models, i.e., the possibilities for specifying process behavior provided by the process tree formalism are only used to a minimal extent.

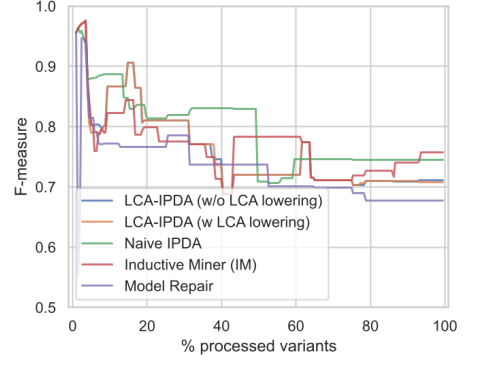
Comparing the LCA-IPDA variants with/without LCA lowering, we observe that the variant with lowering yields slightly larger process models than the version without lowering. This observation is reasonable since when LCA lowering is used, the subtree being rediscovered is potentially smaller compared to the subtree that would be rediscovered when not using LCA lowering. As a result, the chance to introduce duplicate labels increases when rediscovering a smaller subtree increases within the LCA-IPDA.

In conclusion, the results show that a clear dominant approach cannot be determined in these automated experiments. However, the proposed LCA-IPDA approach can keep up with established approaches and offers the benefits and opportunities of incremental process discovery. Furthermore, the experiments also show that the model repair and the naive approach generate huge models, which are of little practical use in the context of Interactive Process Discovery (IPD).

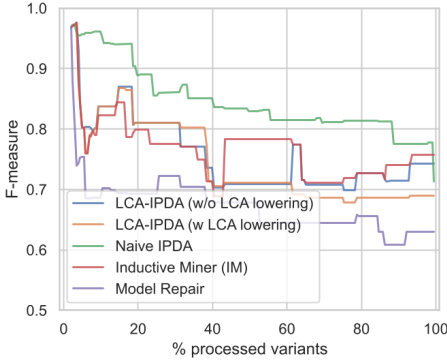
<sup>10</sup>We used a sample of the hospital billing event log because the experimental setup is too computationally intensive. We filtered infrequent traces; the sampled log contains 98,948 traces (209 trace variants) compared to the original one containing 100,000 traces (1,028 trace variants).



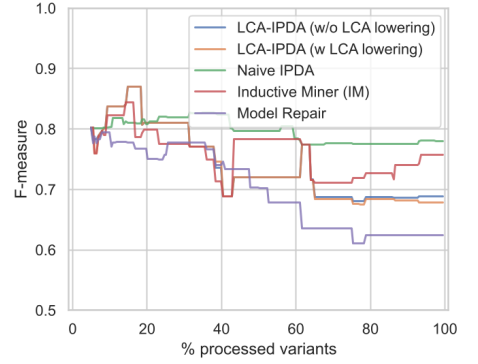
(a) Initial model covers first variant



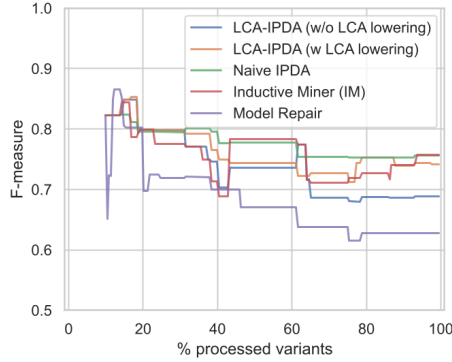
(b) Initial model covers top 1% variants



(c) Initial model covers top 2% variants



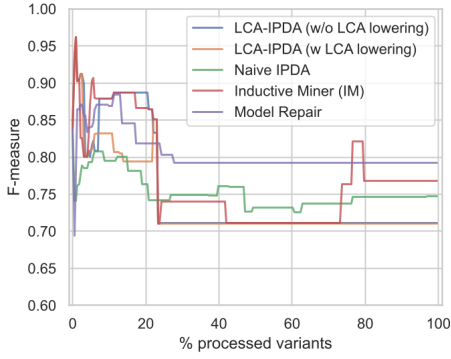
(d) Initial model covers top 5% variants



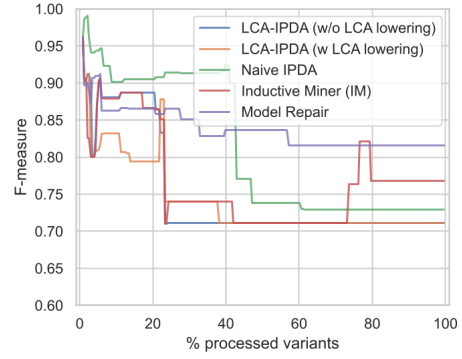
(e) Initial model covers top 10% variants

Figure 5.9: Results regarding the F-measure for the *hospital billing event log* [133] using different initial process models

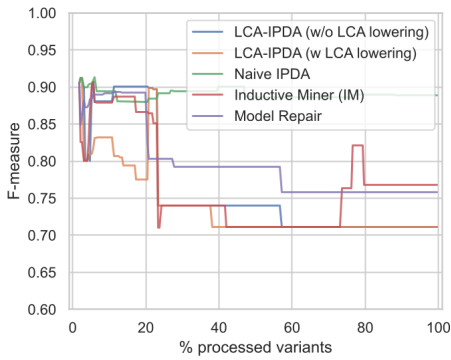




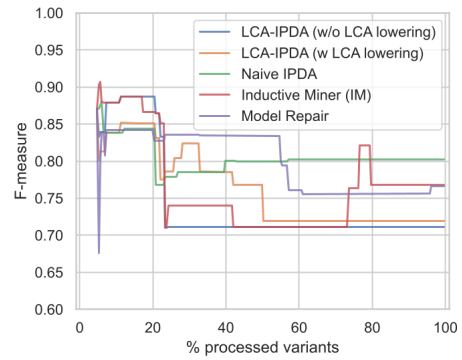
(a) Initial model covers first variant



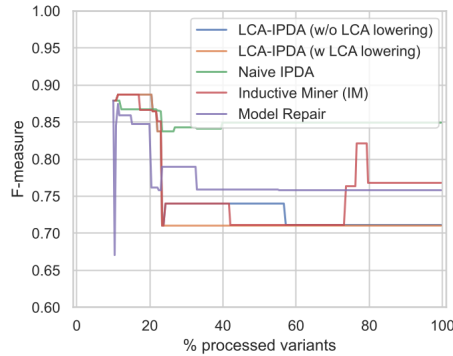
(b) Initial model covers top 1% variants



(c) Initial model covers top 2% variants

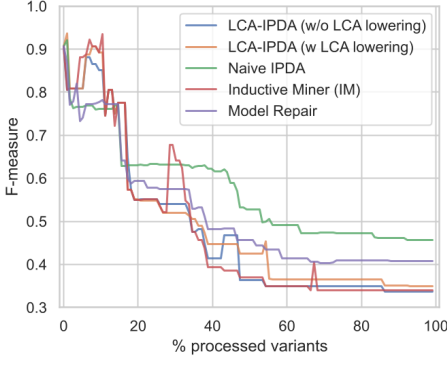


(d) Initial model covers top 5% variants

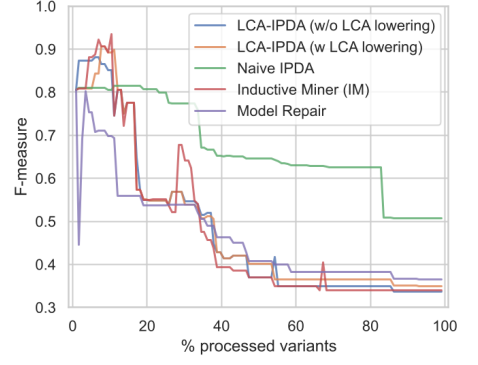


(e) Initial model covers top 10% variants

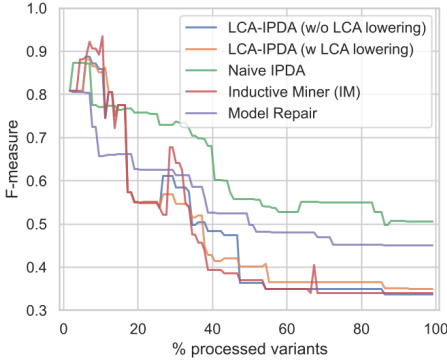
Figure 5.10: Results regarding the F-measure for the *road traffic fine management event log* [56] using different initial process models



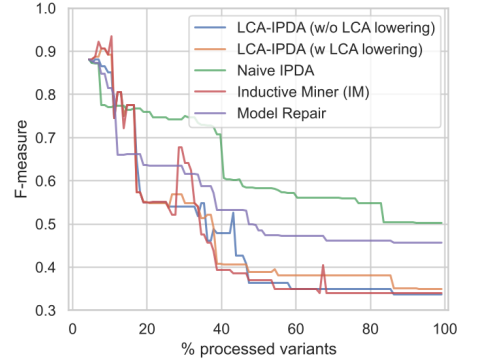
(a) Initial model covers first variant



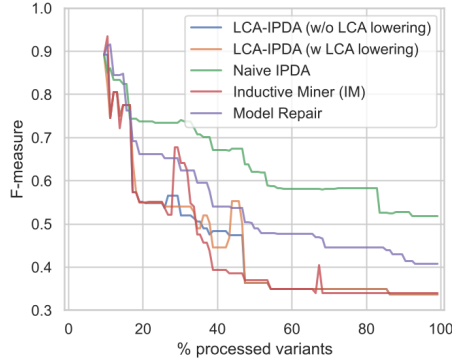
(b) Initial model covers top 1% variants



(c) Initial model covers top 2% variants

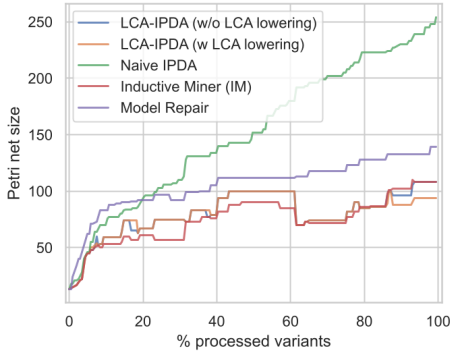


(d) Initial model covers top 5% variants

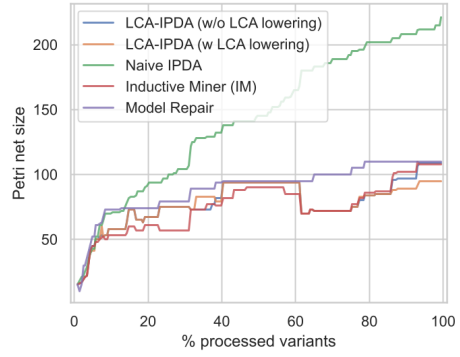


(e) Initial model covers top 10% variants

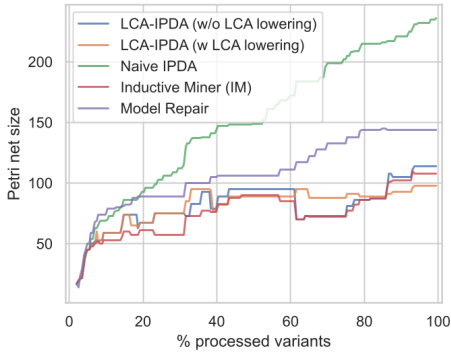
Figure 5.11: Results regarding the F-measure for the *receipt event log* [110] using different initial process models



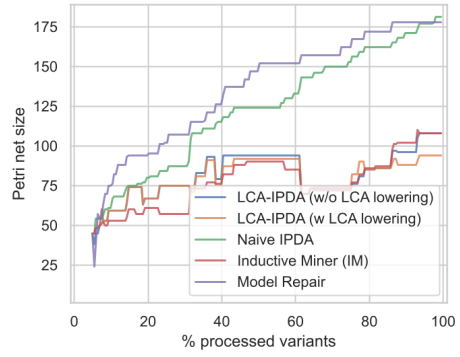
(a) Initial model covers first variant



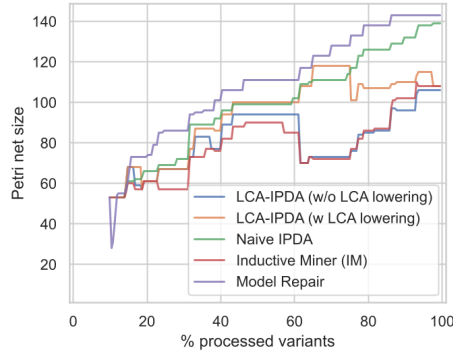
(b) Initial model covers top 1% variants



(c) Initial model covers top 2% variants

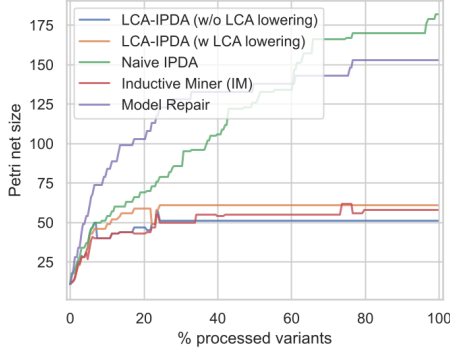


(d) Initial model covers top 5% variants

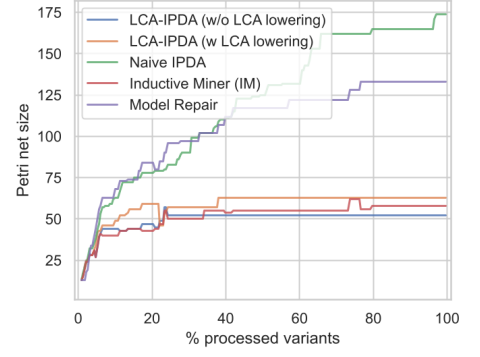


(e) Initial model covers top 10% variants

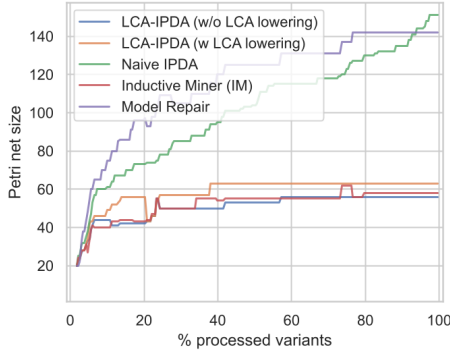
Figure 5.12: Results regarding the *size of the WF-nets*, i.e., the number of transitions and places, for the *hospital billing event log* [133] using different initial process models



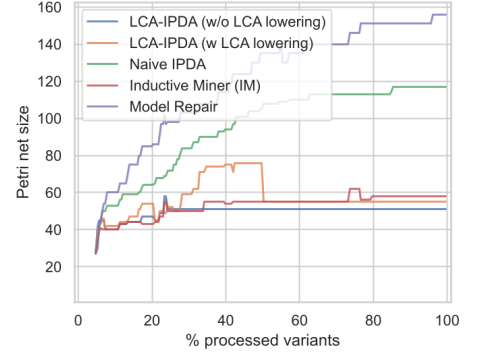
(a) Initial model covers first variant



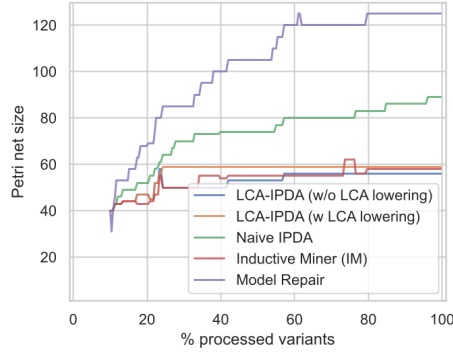
(b) Initial model covers top 1% variants



(c) Initial model covers top 2% variants

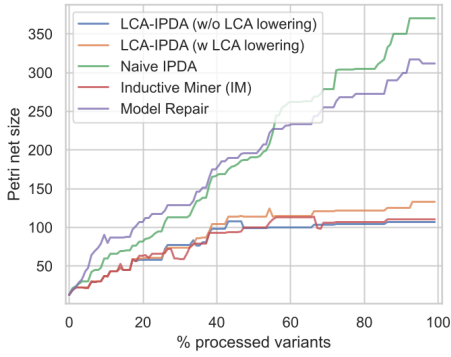


(d) Initial model covers top 5% variants

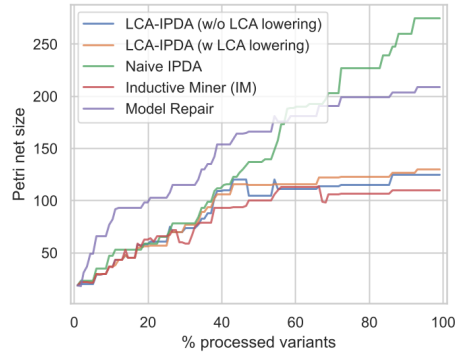


(e) Initial model covers top 10% variants

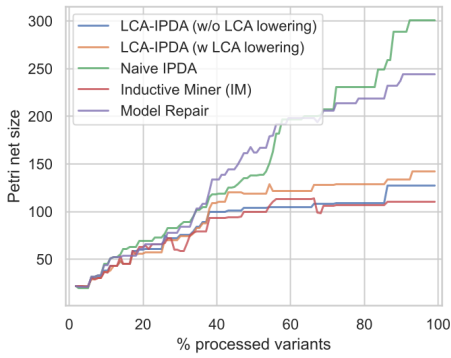
Figure 5.13: Results regarding the *size of the WF-nets*, i.e., the number of transitions and places, for the *road traffic fine management event log* [56] using different initial process models



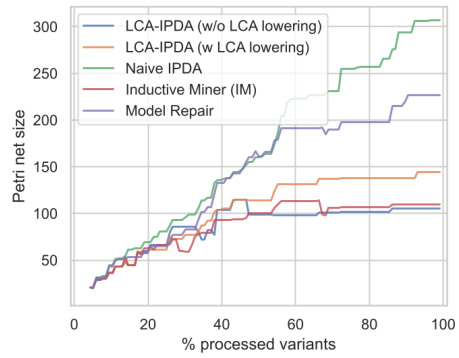
(a) Initial model covers first variant



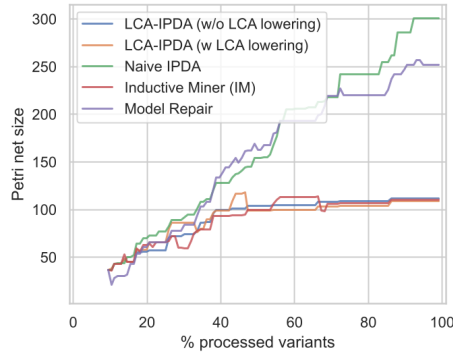
(b) Initial model covers top 1% variants



(c) Initial model covers top 2% variants



(d) Initial model covers top 5% variants



(e) Initial model covers top 10% variants

Figure 5.14: Results regarding the size of the *WF-nets*, i.e., the number of transitions and places, for the *receipt event log* [110] using different initial process models

### 5.4.3. Discussion & Threats to Validity

In the experiments, we executed each approach for a given event log and a given initial process model once, in particular due to the high computational complexity of the experiments. While we fixed the order in which trace variants are incrementally added, the model repair approaches and the LCA-IPDA with and without LCA lowering use internally alignments. As there can be several optimum alignments for a particular trace model combination, there is a certain degree of randomness as to which alignment is used. The resulting process models might be affected by the specific optimal alignment found.

As discussed already in Section 5.4.2, the significance of the F-measure values is limited. As all logs exhibit a power law distribution [214], the precision values are the main driver for changes in the F-measure values because fitness values are close to 1 for most models. However, precision values have limitations as discussed in [202].

## 5.5. Illustrative Example

This section contains an illustrative example to demonstrate the advantages of the LCA-IPDA over the Inductive Miner. In particular, the example shows how duplicate labels can lead to more precise models. Recall that duplicate labels are a central argument why the LCA-IPDA approach produces often better results than the Inductive Miner, as a representative of conventional process discovery algorithms. We use the road traffic fine management event log [56] for this example. Figure 5.15 provides an overview of a subset of contained activities in this log. Furthermore, Figure 5.15 depicts the previously added traces  $A$  and the trace to be added next  $\sigma_{next}$ .

	CF	Create Fine		RRAP	Receive Result Appeal from Prefecture
	SF	Send Fine		NRAO	Notify Result Appeal to Offender
	IFN	Insert Fine Notification		SAP	Send Appeal to Prefecture
	IDAP	Insert Date Appeal to Prefecture		P	Payment
	AP	Add Penalty		AJ	Appeal to Judge

(a) Overview activity abbreviations

$$A = \left\{ \langle CF, SF, IFN, AP, P \rangle, \langle CF, SF, IFN, AP, P, P \rangle, \langle CF, SF, IFN, IDAP, AP, P \rangle \right\}$$

$$\sigma_{next} = \langle CR, P, SF, IFN, AP, P \rangle$$

(b) Previously added traces  $A$  and a trace to be added next  $\sigma_{next}$

Figure 5.15: Contextual information for the illustrative example using the road traffic fine management log [56]

Figure 5.16 shows the results. Process tree  $\Lambda_1$  depicted in Figure 5.16a is considered the initial model with  $A \subseteq \mathbb{L}(\Lambda_1)$  and  $\sigma_{next} \notin \mathbb{L}(\Lambda_1)$ . When invoking the LCA-IPDA for  $\Lambda_1$ ,  $A$ , and  $\sigma_{next}$ , tree  $\Lambda_2$  as shown in Figure 5.16b is returned. Comparing  $\Lambda_1$  and  $\Lambda_2$ , we

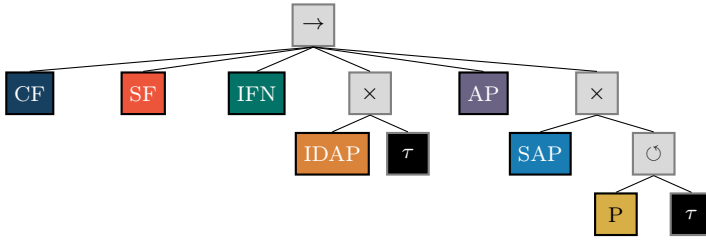
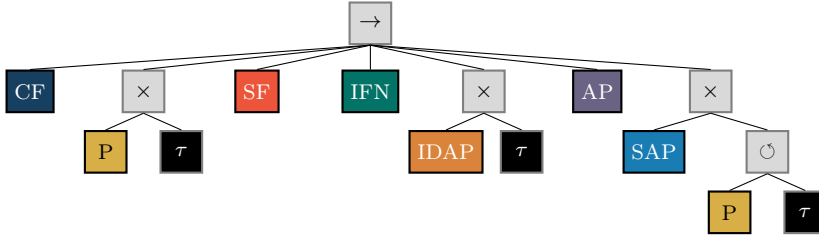
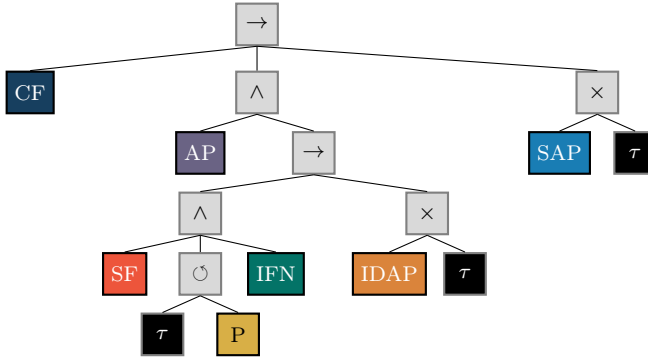
(a) Initial process tree  $\Lambda_1$ (b) Process tree  $\Lambda_2$  returned by the LCA-IPDA(c) Process tree  $\Lambda_3$  returned by the Inductive Miner; tree  $\Lambda_3$  has lower precision than  $\Lambda_2$  returned by the LCA-IPDA regarding the traces  $A$  and trace  $\sigma_{next}$ 

Figure 5.16: Discovered process models using the LCA-IPDA and the conventional process discovery algorithm Inductive Miner using the traces as specified in Figure 5.15

observe that  $\Lambda_2$  allows after executing activity CF the optional execution of activity P. Furthermore, note that tree  $\Lambda_2$  contains duplicate labels, i.e., activity P occurs twice. In contrast, when invoking the Inductive Miner for traces  $A \cup \{\sigma_{next}\}$ , we obtain  $\Lambda_3$  depicted in Figure 6.1c. Note that  $\Lambda_3$  allows for much more behavior than specified in  $A \cup \{\sigma_{next}\}$  because many activities can be executed in parallel. In short,  $\Lambda_2$  is clearly more precise than  $\Lambda_3$  with respect to the traces  $A \cup \{\sigma_{next}\}$ .

## 5.6. Trace Ordering Effects

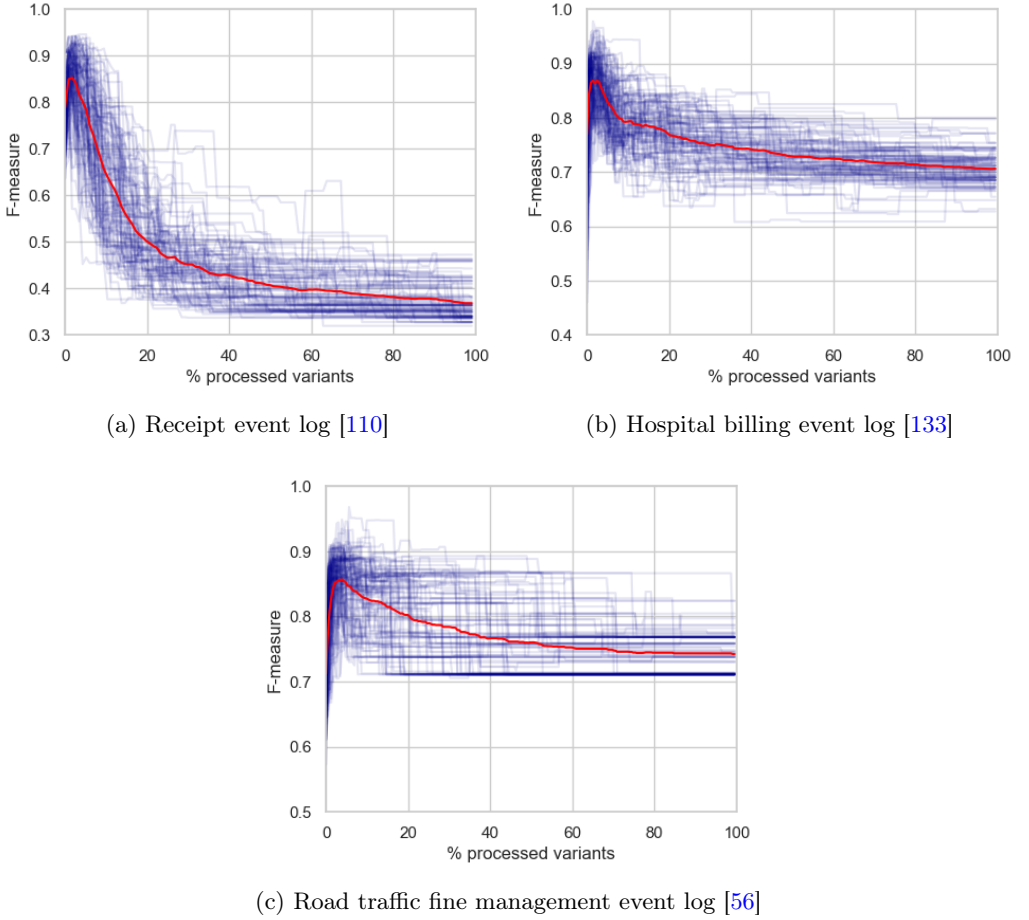


Figure 5.17: Trace ordering effects for different event logs; starting from the same initial model, each blue line represents one specific trace ordering and the red line represents the average of the blue lines

In the conducted experiments, cf. Section 5.4, we fixed the order in which trace variants are incrementally added, i.e., we sorted the trace variants based on their frequency. However, when applying the proposed incremental process discovery framework, outlined in Figure 5.1 (page 110), the resulting model might differ depending on the order in which traces are added incrementally. We refer to this phenomenon as *trace ordering effects*, i.e., starting from the same initial process model and incrementally adding the same traces in different orders results in other final process models. Note that trace ordering effects do not originate from the IPD framework illustrated in Figure 5.1 (page 110) itself, but



rather from the specific instantiations of this framework. As an example, the LCA-IPDA (cf. Section 5.3) is affected by trace ordering effects. Thus, the resulting process model may differ depending on the order in which traces are incrementally added. Figure 5.17 illustrates various trace orderings and their effect on the F-measure for real-life event logs. Each blue line represents an individual trace ordering. Starting from the same initial model, eventually, all traces from the event log have been incrementally added with LCA-IPDA. Overall, we observe that the span between the lowest and largest F-measure is large. Further, Thus, ordering traces to be added next significantly impacts the F-measure of the resulting process model. In short, trace ordering effects are a relevant practical phenomenon when considering incremental process discovery.

The remainder of this section is structured as follows. Section 5.6.1 proposes a framework that allows to calculate trace ordering recommendations. Next, Section 5.6.2 provides examples of concrete instantiations of this framework, i.e., specific strategies that determine a order over traces to be added next. In Section 5.6.3, we present experimental results of evaluating the framework and its specific instantiations, presented in Section 5.6.2, on real-life event logs.

### 5.6.1. Framework for Recommending Trace Orderings

This section proposes a framework for *Next Trace Ordering Strategies (NTOSs)*. The framework determines a trace ordering from a set of traces to be added next; thus, it determines the trace to be added next  $\sigma_{next}$  from a set of trace candidates. The Next Trace Ordering Strategy (NTOS) framework can be easily incorporated into the previously presented incremental process discovery framework. Consider Figure 5.18 depicting an NTOS embedded into the IPD framework. We highlight in red new respectively altered elements compared to the original incremental process discovery framework shown in Figure 5.1 (page 110). In detail, instead of an individual (user-)selected trace to be added next  $\sigma_{next}$ , potentially multiple (user-)selected trace candidates to be added next exist, i.e.,  $C_0 \subseteq \bar{L}^s \cup X$ . Further, an NTOS is placed between the (user-)selected trace candidates and the IPDA. The NTOS is responsible for selecting an individual trace  $\sigma_{next}$  from the trace candidates because the IPDA, as specified in Definition 5.1 (page 111), assumes a single trace to be added next. After adding the trace  $\sigma_{next}$ , which is selected by the NTOS, trace  $\sigma_{next}$  is removed from the set of trace candidates  $C_0$ . Except for the described modifications, the incremental process discovery framework remains as previously introduced in Figure 5.1.

An NTOS is generally composed of multiple, sequentially-aligned strategy components. Figure 5.19 illustrates the structure of an NTOS. The depicted NTOS comprises  $n$  strategy components, referred to as  $sc_1, \dots, sc_n$ . The first strategy component obtains the complete set of (user-)selected trace candidates to be added next, i.e.,  $C_0$ . Further, strategy component  $sc_1$  as well as all subsequent strategy components  $sc_2, \dots, sc_n$  have access to the set of previously added traces  $A$ , the process tree  $\Lambda$ , and the trace pool  $L \uplus L'$ . Access to the trace pool allows strategy components to take frequency information about individual traces into account. Note that not every concrete instantiation of a strategy component uses all this provided information to determine a rating respectively ordering among trace candidates. The output of a strategy component is a total order over the trace candidates that were provided as input. This total order represents an ordering

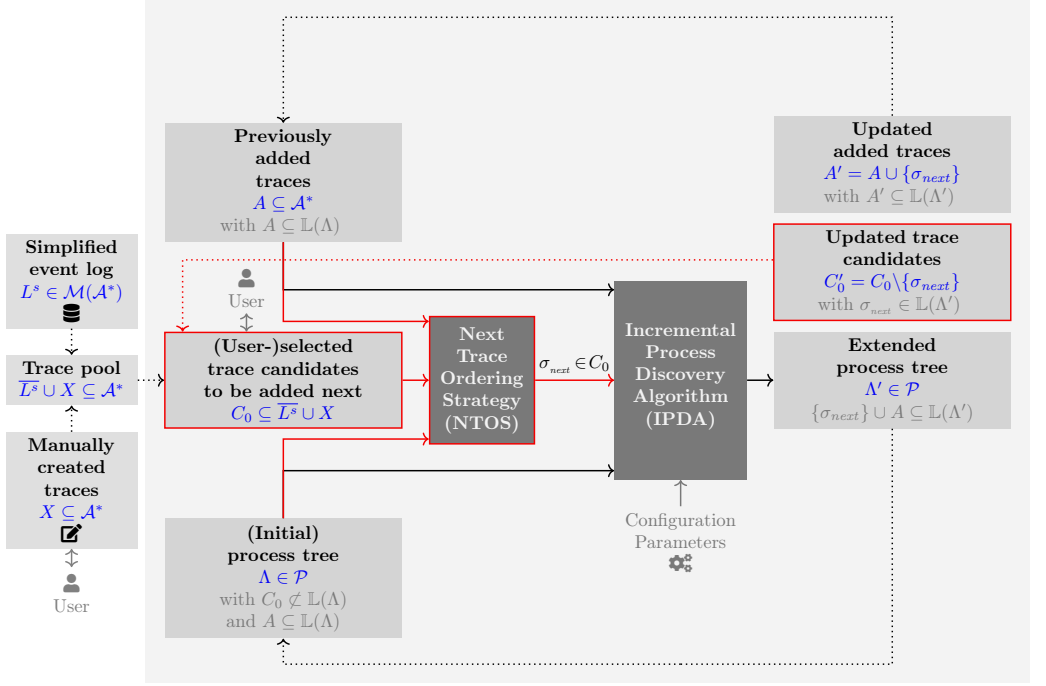


Figure 5.18: Embedding an NTOS into the incremental process discovery framework; compared to Figure 5.1 (page 110), multiple trace candidates to be added next have been selected, and a NTOS recommends from these candidates one specific trace to be added next, i.e.,  $\sigma_{next}$

from best to worst suitable trace to be added next; thus, each trace among the trace candidates is rated to be added next. Below, we formally define a strategy component.

**Definition 5.3** (Strategy Component)

A strategy component is a function  $sc : \mathbb{P}(\mathcal{A}^*) \times \mathcal{P} \times \mathbb{P}(\mathcal{A}^*) \times \mathcal{M}(\mathcal{A}^*) \rightarrow \mathcal{O}_{\leq}(\mathcal{A}^*)$  that maps

1. previously added traces  $A \subseteq \mathcal{A}^*$ ,
2. (initial) process tree  $\Lambda \in \mathcal{P}$ ,
3. (user-) selected trace candidates to be added next  $C_0 \subseteq \mathcal{A}^*$ , and
4. trace pool  $\overline{L^s} \cup X \subseteq \mathcal{A}^*$

to a total order over the (user-) selected trace candidates to be added next, i.e.,  $(C_0, \leq) \in \mathcal{O}_{\leq}(\mathcal{A}^*)$ . We denote the universe of strategy components by  $\mathcal{SC}$ .

Reconsider Figure 5.19. After each strategy component, we apply a *filter* function that removes the worst-suited trace candidates. Whenever we apply the *filter* function, we must provide a filter rate, denoted by  $r$ , determining the ratio of trace candidates to be

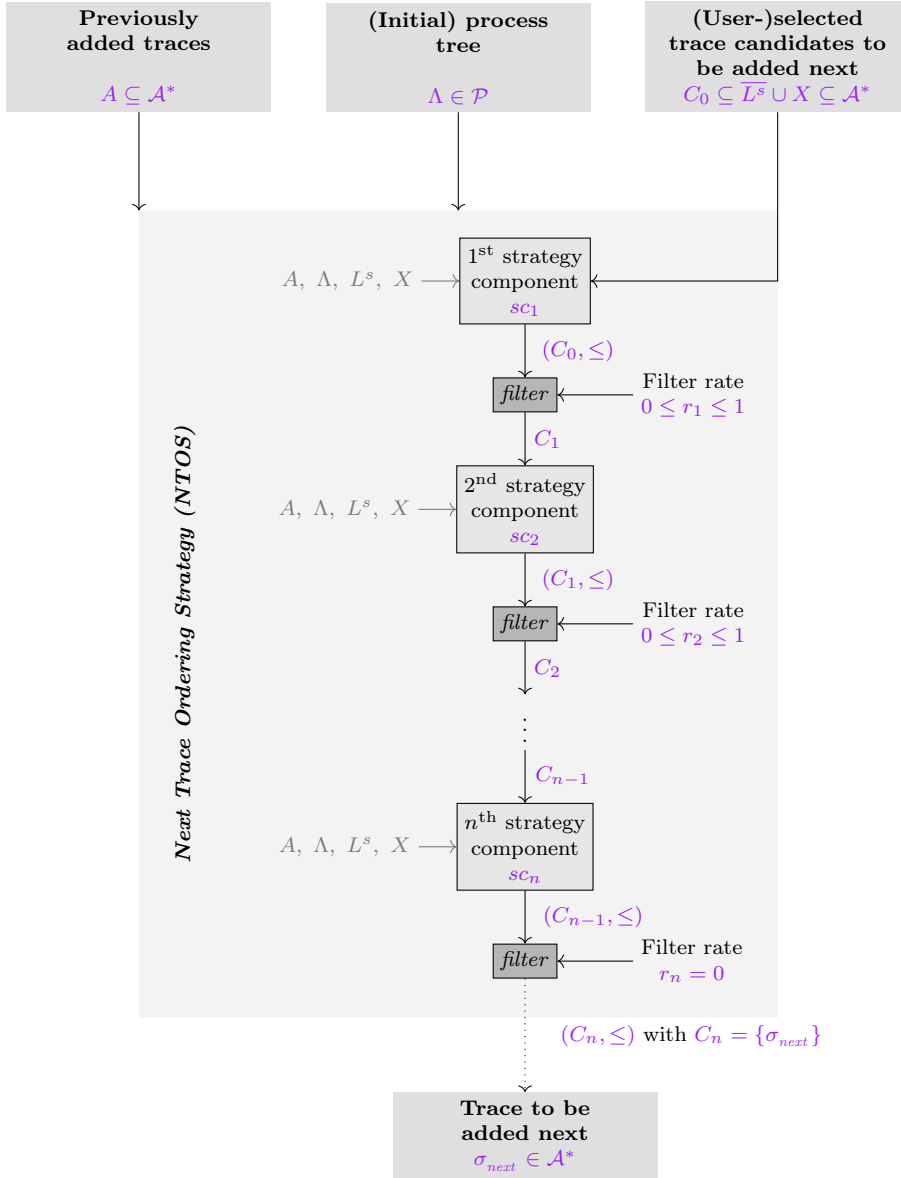


Figure 5.19: Schematic structure of a NTOS consisting of multiple, sequentially-aligned strategy components; each strategy component orders the trace candidates from best to worst fitting, and subsequent filtering functions filter the worst trace candidates before the following strategy component is invoked

filtered. For instance, if  $r = 0.7$ , *filter* removes 30% of the worst trace candidates. If  $r = 1$ , no trace candidate will be eliminated, while  $r = 0$  will result in only one trace candidate remaining. Thus, the following subset relations hold among the trace candidate sets  $C_0, \dots, C_n$  within an NTOS.

$$C_n \subseteq C_{n-1} \subseteq \dots \subseteq C_1 \subseteq C_0$$

Accordingly, each strategy component paired with the subsequent application of *filter* can be considered a knock-out step that reduces the number of trace candidates to be added next. Below, we formally define the function *filter*.

**Definition 5.4** (Filter function *filter*)

The function  $\text{filter} : \mathcal{O}_{\leq}(\mathcal{A}^*) \times [0, 1] \rightarrow \mathbb{P}(\mathcal{A}^*)$  maps a total ordered set  $(C, \leq) \in \mathcal{O}_{\leq}(\mathcal{A}^*)$  and a filter rate  $0 \leq r \leq 1$  to a set of trace candidates  $C' \subseteq \mathcal{A}^*$  such that:

1.  $C' \subseteq C$ ,
2.  $|C'| = \max \{1, \lceil r * C \rceil\}$ , and
3.  $\forall c' \in C' \forall c \in C \setminus C' (c' \leq c)$ .

Finally, we define NTOSs consisting of sequentially aligned strategy components (cf. Definition 5.3) and applications of the *filter* function (cf. Definition 5.4). Reconsider Figure 5.19 for an illustration of an NTOS structure.

**Definition 5.5** (Next Trace Ordering Strategy (NTOS))

A NTOS is a non-empty sequence of  $n \in \mathbb{N}$  pairs, each consisting of a strategy component and a filter rate. Thus,  $\rho = \langle (sc_1, r_1), \dots, (sc_n, r_n) \rangle \in (\mathcal{SC} \times [0, 1])^*$  with  $r_n = 0$  is a NTOS.<sup>a</sup>

<sup>a</sup>The last filter rate  $r_n$  is required to be zero to ensure only one trace candidate eventually remains.

We made a deliberate choice to create an NTOS using a sequence of strategy components paired with subsequent filtering.<sup>11</sup> Each strategy component orders the trace candidates from best to worst, and subsequently we filter out the worst candidates. This approach was taken to ensure that recommendations can be computed quickly in an interactive process discovery setting. By using multiple strategy components, the framework can consider different aspects when evaluating which trace candidate to add next. The idea behind this framework is to begin with fast evaluations in the initial strategy components. More complex evaluations are then performed in the later strategy components, which receive fewer trace candidates as the earlier strategy components have already filtered out some of the options.

### 5.6.2. Sample Instantiations of Strategy Components

This section provides examples of specific strategy component instantiations according to Definition 5.3. We present six strategy components in total as listed in Table 5.1.

<sup>11</sup>Note that also non-sequential arrangements of strategy-components are conceivable. For instance, each strategy component evaluates all trace candidates, and eventually, one overall ordering is determined; thus, no intermediate filtering between strategy components is applied.

Table 5.1.: Overview of the six strategy component instantiations

Abbreviation	Name	Specific/General
C	Alignment Costs	general
M	Missing Activities	general
L	Levenshtein Distance	general
D	Duplicates	general
H	LCA Height	specific
B	Brute Force	general

Note that these six strategy component instantiations are only examples and many other strategy components are conceivable.

### Alignment Costs

Alignments, as introduced in Section 3.5, are a state-of-the-art conformance checking technique that quantify the mismatches between a provided trace and a process model. Further, alignments provide diagnostic information such as missing activities and unexpected process behavior. Alignments are typically assigned costs. The *standard cost function* assigns costs of one to visible model moves and log moves; other moves are assigned costs zero [45]. When computing an optimal alignment for each trace candidate, the associated costs are used to order/rank the trace candidates from low to high costs. Trace candidates with high costs have many deviations and are thus not close to the current process model. The intention behind this strategy component is to favor trace candidates that are already close to the specified behavior by the current process model, i.e., trace candidates having low alignment costs. Note that alignment computation suffers from the *state space explosion problem* and has, therefore, an exponential time complexity in worst case [45].

### Missing Activities

When incrementally discovering a process model, it's common to find that the first models obtained don't encompass all the process activities recorded in the event data. Since not every trace in real-life event logs contains all possible executable process activities of a process, missing activities within early-stage process models are possible. The 'missing activities' strategy makes use of this observation. It ranks trace candidates based on the number of activity labels in the process tree  $\Lambda$ . Trace candidates solely containing process activities already in the current process tree  $\Lambda$  are assigned cost zero. In contrast, costs for other trace candidates correspond to the number of unique activity labels not yet part of the process tree  $\Lambda$ . The rationale behind this strategy component is similar to the 'alignment costs' component, i.e., favoring trace candidates that are close to the current process model over traces containing new activities that are not yet present in the tree  $\Lambda$ .

## Levenshtein Distance

The Levenshtein distance, originally defined over binary sequences [124], also known as *edit distance* allows us to measure the similarity between two sequences. Thus, we can utilize the Levenshtein distance to measure the similarity of the two traces. In short, the Levenshtein distance counts the number of edit operations to convert one sequence into another. Three potential edit operations exist insertion, deletion, and substitution. Below, we formally define the Levenshtein distance.

**Definition 5.6** (Levenshtein distance)

Let  $X$  be an arbitrary set and  $\sigma_1, \sigma_2 \in X^*$  be sequences. We recursively define the function  $lev : X^* \times X^* \rightarrow \mathbb{N}_0$  as follows.

$$lev(\sigma_1, \sigma_2) = \begin{cases} |\sigma_1| & \text{if } |\sigma_2| = 0 \\ |\sigma_2| & \text{if } |\sigma_1| = 0 \\ lev(\langle \sigma_1(2), \dots, \sigma_1(|\sigma_1|) \rangle, \langle \sigma_2(2), \dots, \sigma_2(|\sigma_2|) \rangle) & \text{if } \sigma_1(1) = \sigma_2(1) \\ 1 + \min \left\{ \begin{array}{l} lev(\langle \sigma_1(2), \dots, \sigma_1(|\sigma_1|) \rangle, \sigma_2), \\ lev(\sigma_1, \langle \sigma_2(2), \dots, \sigma_2(|\sigma_2|) \rangle), \\ lev(\langle \sigma_1(2), \dots, \sigma_1(|\sigma_1|) \rangle, \langle \sigma_2(2), \dots, \sigma_2(|\sigma_2|) \rangle) \end{array} \right\} & \text{otherwise} \end{cases}$$

We use the Levenshtein distance to determine the trace candidate that shares most behavior with all other trace candidates. We also use frequency information about the trace candidates to weigh their distances. Since the proposed strategy components are mainly designed to improve the discovered process models' F-measure and frequency information are vital in F-measure calculation, this strategy component considers frequency information. Table 5.2 depicts an example of three trace candidates having different frequencies in the trace pool. For each trace, we compute the weighted Levenshtein distance, i.e., the Levenshtein distance of the current trace candidate to all other trace candidates, each multiplied by the occurrence of the other trace candidate. For instance, first trace  $\langle a, b \rangle$  is compared to the second trace  $\langle a, b, b \rangle$  and the third trace  $\langle a, c \rangle$ . In this example, the first trace candidate would be ranked first. The rationale behind this strategy component is to favor trace candidates with a lot of behavior in common with all other trace candidates that will be incrementally added later.

## Duplicates

This strategy is tailored to the LCA-IPDA introduced in Section 5.3. Further, we assume that function *discovery*, which is used within Algorithm 5.2 line 9 (page 118) for rediscovering subtrees, is instantiated with a conventional process discovery algorithm that returns process trees having no *duplicate labels*, i.e., no two leaf vertices having the same activity label.<sup>12</sup>

In general, duplicate labels can increase the precision of a process model and are, therefore, often desirable [128, 240]. Recall that the LCA-IPDA determines subtrees that must

<sup>12</sup>Many process discovery algorithms cannot discover process models with duplicate labels, for instance, the inductive miner algorithms [120].

Table 5.2.: Example of the weighted Levenshtein distance strategy component for three trace candidates (adapted from [179, Table 2])

Trace candidate	Frequency in trace pool*	Weighted Levenshtein distance						Rank
$\langle a, b \rangle$	100	50	*	$lev(\langle a, b \rangle, \langle a, b, b \rangle)$	+	20	*	1
		$lev(\langle a, b \rangle, \langle a, c \rangle) = 70$						
$\langle a, b, b \rangle$	50	100	*	$lev(\langle a, b, b \rangle, \langle a, b \rangle)$	+	20	*	2
		$lev(\langle a, b, b \rangle, \langle a, c \rangle) = 140$						
$\langle a, c \rangle$	20	100	*	$lev(\langle a, c \rangle, \langle a, b \rangle)$	+	50	*	3
		$lev(\langle a, c \rangle, \langle a, b, b \rangle) = 200$						

\* Note that the trace pool as defined in Definition 5.3 contains no frequency information about the contained traces. However, frequency information about the traces can be obtained from the originally provided event log  $L^2$ , cf. Figure 5.18. In addition, the trace pool can also contain manually defined traces for which the user must also enter frequency information if the weighted Levenshtein distance strategy component is to be used.

be altered. Altering a subtree involves rediscovering the subtree from a corresponding sublog, cf. Algorithm 5.2 (page 118). If the determined subtree contains duplicate labels and under the assumption that *discovery* is instantiated with a discovery algorithm unable to discover a process tree with duplicate labels, the rediscovered subtree no longer contains duplicate labels. Thus, the rediscovery removes the potentially desirable duplicate labels in the determined subtree that have been learned so far. Therefore, this strategy component, called *Duplicates*, favors trace candidates whose first LCA does not contain leaf nodes with duplicate labels. The trace candidates are ranked in ascending order based on the number of duplicate leaf vertices. Thus, the strategy component favors trace candidates that when being incrementally added to the process model do not affect a subtree containing duplicate labels.

Finally, we will briefly elucidate one shortcoming of this strategy. When incrementally adding a trace to a process model, a subtree is determined that causes the first (block of) deviation(s) and is replaced by a new one. However, until the trace to be added fits the process model, multiple subtrees may be altered by the LCA-IPDA, cf. Algorithm 5.2 line 5 (page 118). However, strategy component *Duplicates* considers the first subtree only when determining the order of trace candidates. Considering only the first one is done because executing the LCA-IPDA is necessary to know the potential next subtree that is being altered. However, there are potential cases where the first subtree that must be altered does not contain duplicate labels; however, the second subtree, which needs to be altered after altering the first one, does contain duplicate labels. Since the strategy component only considers the first one, the trace candidate would be rated good, although when adding this candidate, duplicate labels would disappear from the overall process model.

## LCA Height

Next to *Duplicates*, *LCA Height* is the second strategy component that is tailored for the LCA-IPDA. The key idea of this strategy is to avoid changing large parts of the already learned process model upon adding a new trace. Thus, the strategy prefers trace candidates that lead to only minor changes in the process model. Therefore, the strategy computes for each trace candidate the height of the first subtree that will be changed by LCA-IPDA.<sup>13</sup> The height of an LCA is defined by the path length from the determined LCA, i.e., the root vertex of the determined subtree to the root vertex of the entire tree. Trace candidates are then descending ordered based on the first LCA's height.

## Brute-Force

The *Brute-Force* strategy separately applies the LCA-IPDA to all trace candidates in  $C_i$  and the process tree  $\Lambda$ .<sup>14</sup> As a result,  $|C_i|$  different process trees are obtained. A quality metric, i.e., the F-measure representing the harmonic mean of fitness (recall) and precision, is calculated on the given trace pool  $L \uplus L'$  for each obtained process tree. The trace candidate that yields a process tree with the highest F-measure is ranked first. Thus, the *Brute-Force* strategy can be considered as a greedy strategy, that locally optimizes for the highest F-measure. Note that this strategy is computationally very expensive since we calculate the F-measure based on alignments for various process trees in each incremental execution.

### 5.6.3. Evaluation

This section presents an evaluation of the proposed NTOS, cf. Section 5.6.1, using the introduced strategy component instantiations, cf. Section 5.6.2. The overall goal of the evaluation is to demonstrate that by applying trace ordering strategies, on average better process models can be discovered compared to random trace orderings. Note that the parameter space of potential experiments is vast. Reconsider the introduced NTOS framework depicted in Figure 5.19 (page 143). First, the number of strategy components can be varied. Further, we can freely choose the instantiation of each strategy component. Note that strategy component instantiations can occur multiple times; for instance, we can utilize the identical strategy component instantiation at the beginning and the end of an NTOS. Moreover, we can freely select a filter rate for each strategy component, except for the last strategy component (cf. Figure 5.19). In short, the introduced NTOS framework depicted in Figure 5.19 (page 143) has a vast parameter space.

The remainder of this section is divided into three parts. First, we introduce the experimental setup. Subsequently, we present the results. Eventually, we discuss the results and potential threats to validity.

<sup>13</sup>Similar to *Duplicates*, note that *LCA Height* can only determine the first subtree that is altered by LCA-IPDA. Therefore, there is a risk that *LCA Height* rates the first subtree as good and thus the corresponding trace candidate, but further subtrees must be changed by LCA-IPDA, which the strategy would rate as bad.

<sup>14</sup>Note that the  $i$  refers to the position of the *Brute-Force* strategy component within an NTOS.



## Experimental Setup

To ensure independence from any particular user selecting trace candidates to be added next from the trace pool (cf.  $C_0$  in Figure 5.18 on page 142), we assume that all traces from a given event log  $L$  are selected as trace candidates, and all candidates are eventually added to the model incrementally. Thus, in the beginning, the set of trace candidates represents the entire event log, i.e.,  $C_0 = \bar{L}$ . After each incremental discovery step, the last added trace is removed from the set of trace candidates  $C_0$  as depicted in Figure 5.18 (page 142).

To keep the overall parameter space for potential experiments manageable, we fixed the length of evaluated NTOSs to six, i.e.,  $n = 6$  in Figure 5.19 (page 143). Thus, each evaluated NTOS consists of six strategy components. Recall the six specific strategy components introduced in Section 5.6.2 and their corresponding abbreviations shown in Table 5.1 (page 145). We created all potential orderings by shuffling the order of the strategies: C, M, L, D, and H. Finally, the brute force (B) strategy component is added to each NTOS as the last strategy component. Note that the brute force (B) strategy component is computationally expensive; therefore, we always add this strategy component at the end, where the fewest trace candidates need to be ordered. The above-described approach leads to  $5! = 120$  different NTOSs. To avoid further expansion of the parameter space, we use one filter rate for each strategy component within a NTOS; thus,  $r_1 = \dots, r_5$  and  $r_6 = 0$  (as specified in Definition 5.5). In short, we test a large part of all sequential combinations of the proposed strategy components in this evaluation.

We denote a particular NTOS as a sequence of abbreviations of the contained strategy components. For instance, the strategy L-H-C-M-D-B F-Rate 10 refers to the NTOS where first the Levenshtein distance component is applied and finally the brute force component. All components within this specific strategy use a filter rate of  $r_1 = \dots, r_5 = 0.1 \hat{=} 10\%$ . We applied the different NTOSs on real-life event logs using the LCA-IPDA, cf. Section 5.3.2. Furthermore, we use the same event logs that we already used in Section 5.4. Furthermore, we measured the F-measure of each incrementally discovered process tree using the entire given event log.

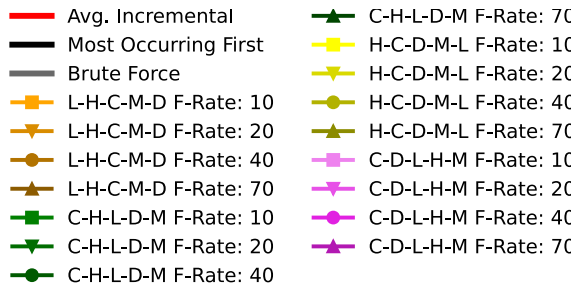


Figure 5.20: Legend for the subsequently presented plots, cf. Figures 5.21 to 5.23

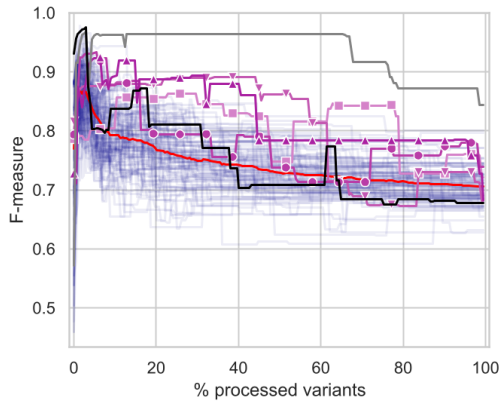
## Results

In Figures 5.21 to 5.23, we depict the results of 16 dynamic trace ordering strategies, a static strategy, i.e., most occurring trace variant first (black line), the brute force component as a stand-alone strategy (gray line), random trace orderings (blue lines), and the average of the random trace orderings (red line). Note that we only show a selection of the strategies evaluated; we evaluated many more combinations of strategy components and show here the best performing ones. Figure 5.20 depicts the legend listing all orderings evaluated strategies.

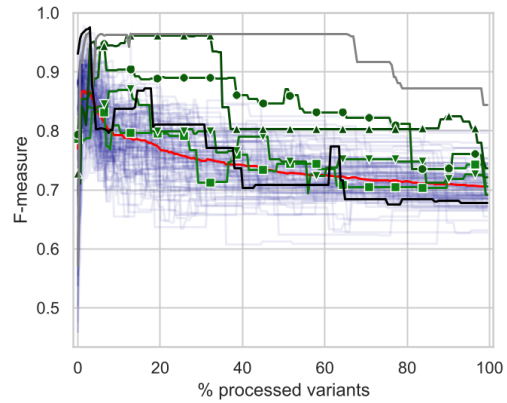
For each event log we plot the F-measure that has been calculated based on the entire event log for all the incremental discovered process models.

We observe that for all four event logs, the trace candidate order has a significant impact on the F-measure, cf. the large area covered by the blue lines in Figures 5.21 to 5.23. The solid red line represents the average of the blue lines. Thus, the red line can be seen as a baseline as it represents the quality of the models if a random trace order is applied. We see that *most strategies are clearly above the red line*. Thus, applying a strategy is often better than randomly selecting trace candidates. Note that with incremental process discovery, the goal is often *not* to include all traces from the event log, as event logs often have data quality issues. We observe that the brute force approach as a stand-alone strategy (gray line) often performs better than the other strategies, although the brute force approach can be considered as a *greedy* algorithm, i.e., it is only locally optimizing the F-measure. For the hospital billing event log in particular, the brute force strategy outperforms all other strategies in the majority of sections in terms of the number of variants processed.

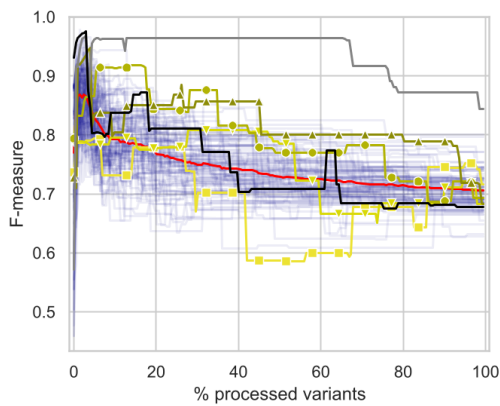
In short, the proposed LCA-IPDA is affected by trace ordering effects. The order of traces incrementally added can significantly impact the quality, i.e., the F-measure, of the discovered process models. Moreover, trace ordering strategies can help obtain better process models when compared to random trace orders.



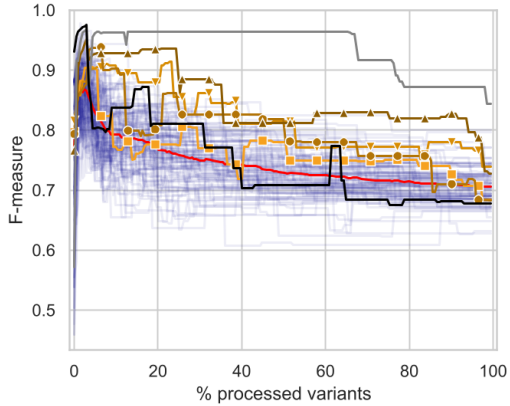
(a) Strategy C-D-L-H-M-B with different f-rates



(b) Strategy C-H-L-D-M-B with different f-rates

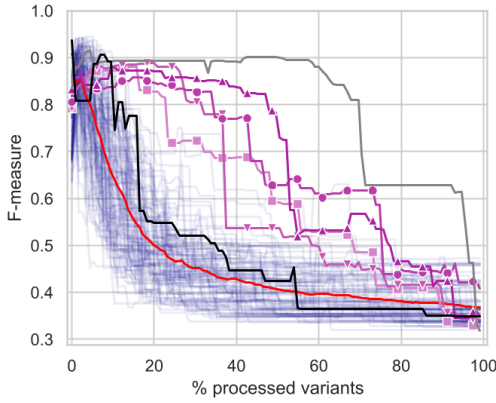


(c) Strategy H-C-D-M-L-B with different f-rates

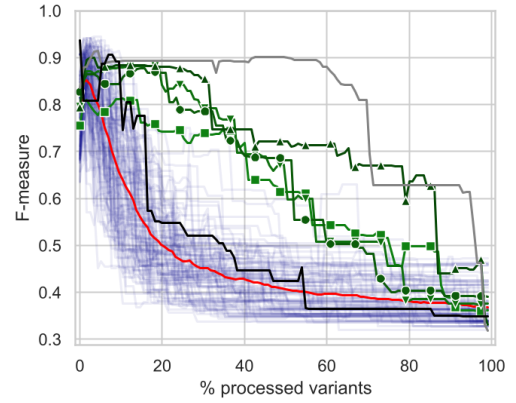


(d) Strategy L-H-C-M-D-B with different f-rates

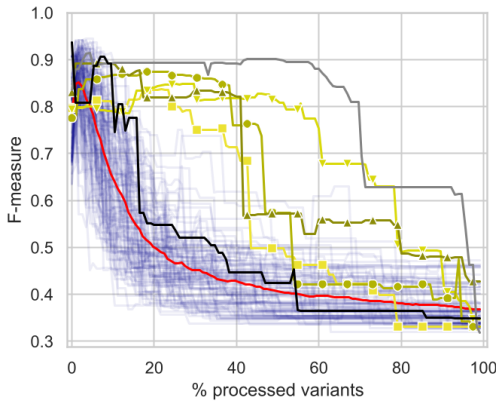
Figure 5.21: F-measure values of the incrementally discovered process trees using different NTOSs using the *hospital billing event log* [133]



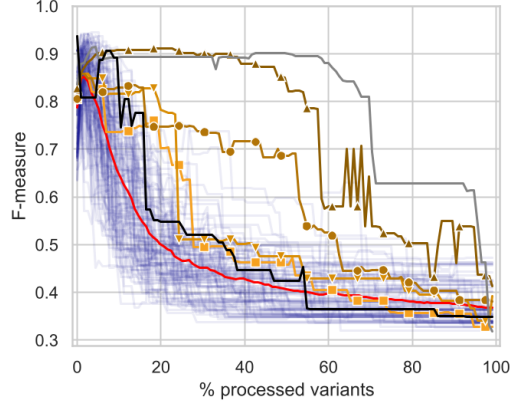
(a) Strategy C-D-L-H-M-B with different f-rates



(b) Strategy C-H-L-D-M-B with different f-rates

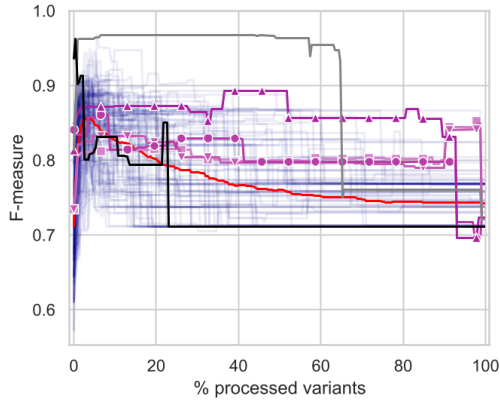


(c) Strategy H-C-D-M-L-B with different f-rates

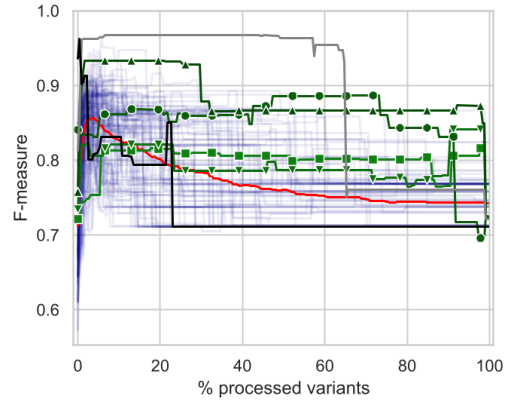


(d) Strategy L-H-C-M-D-B with different f-rates

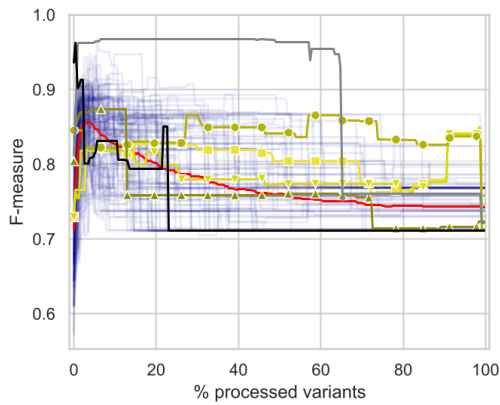
Figure 5.22: F-measure values of the incrementally discovered process trees using different NTOSs using the *receipt event log* [110]



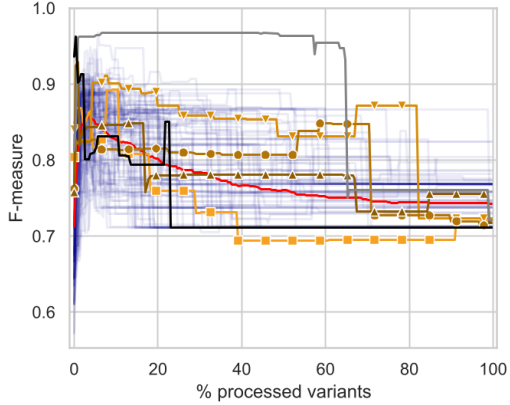
(a) Strategy C-D-L-H-M-B with different f-rates



(b) Strategy C-H-L-D-M-B with different f-rates



(c) Strategy H-C-D-M-L-B with different f-rates



(d) Strategy L-H-C-M-D-B with different f-rates

Figure 5.23: F-measure values of the incrementally discovered process trees using different NTOSs using the *road traffic fine management event log* [56]

## 5.7. Conclusion

We introduced a fundamental incremental process discovery framework that allows users to gradually discover process models from event data (cf. Section 5.1). Central to this framework is an IPDA, which requires a process tree, a (user-)selected trace to be added next from a trace pool, and potentially previously added traces. The IPDA alters the provided process tree such that the altered tree supports the previously added traces and the newly selected trace. This described altering of a tree is counted as an increment. Between increments, i.e., upon selecting the next trace to be added, the (intermediate) process tree can be reviewed and also edited manually if necessary before further traces are added incrementally.

We proposed two specific instantiations of IPDAs: the naive IPDA (cf. Section 5.2) and the LCA-IPDA (cf. Section 5.3). The naive IPDA should be considered as the most straightforward algorithm and is used for comparison purposes only. The central idea of the naive IPDA is to resolve each deviation in an alignment between the process tree and the trace to be added next individually by applying resolving rules (cf. Figure 5.2 on page 113). As discussed, the naive IPDA cannot add parallel or loop operators to a model and is, therefore, severely limited in terms of the process trees that can be discovered. In comparison, the LCA-IPDA determines subtrees in the provided process tree that cause deviations between the trace to be added next and the provided process tree. By employing a conventional process discovery algorithm, the core idea of the LCA-IPDA is to rediscover determined subtrees from so-called sub logs that specify trace fragments the determined subtree should support such that previously added traces and the trace to be added next are supported. For example, compared to the widely used Inductive Miner algorithm family [120], the LCA-IPDA can discover process trees with duplicate labels. Generally, process models with duplicate labels can be more precise than those without [128, 240]. Thus, the ability to discover models with duplicate labels of the proposed LCA-IPDA is considered an advantage.

Subsequently, we investigated trace ordering effects within incremental process discovery. Trace ordering effects apply if different orderings in which (user-)selected traces are incrementally added to the initially same process model result in different process models. Further, we modified the presented incremental process discovery framework to allow for multiple (user-)selected traces to be added next, i.e., trace candidates to be added next (cf. Figure 5.18). In this regard, we defined NTOSs, which recommend from a set of trace candidates to be added next a single trace that is eventually incrementally added by an IPDA.

## Chapter 6.

# Supporting Trace Fragments in Incremental Process Discovery

This chapter is largely based on the following published work.

- D. Schuster, N. Föcking, S. J. van Zelst, and W. M. P. van der Aalst. *Incremental discovery of process models using trace fragments*. In C. Di Francescomarino, A. Burattin, C. Janiesch, and S. Sadiq, editors, *Business Process Management, volume 14159 of Lecture Notes in Computer Science*, pages 55–73. Springer, 2023. doi:10.1007/978-3-031-41620-0\_4 [185]

Most process discovery algorithms [15, 60, 235], including incremental process discovery, as proposed in Chapter 5, consider process executions, i.e., *traces*, recorded in the event data to be *complete*. Thus, traces are assumed to span the process from start to completion. In contrast, *incomplete* traces that do not cover a complete process execution, referred to as *trace fragments*, are usually considered noise and therefore filtered during event data preparation [24, 31, 35] because most algorithms do not support them respectively would falsely treat them as complete traces resulting in inaccurate or even wrong results.<sup>1</sup> In addition, trace fragments are usually not marked as such in the event logs, which makes it difficult to support or filter them.

State-of-the-art process discovery lacks support for trace fragments. However, trace fragments may provide valuable information similar to complete traces. Filtering them, therefore, may result in the loss of valuable data from event logs. As a result, there often exist cases for which not all relevant events have been extracted. For this reason, [30] proposes an approach to extend incomplete traces to complete traces. In short, trace fragments are a regular phenomenon; most process discovery approaches do not support trace fragments or consider trace fragments as complete and, therefore, are often filtered in event data preprocessing phases.

We provide an illustrative example in Figure 6.1 to motivate the need for trace fragment support. Figure 6.1b provides an overview of the various activities, which have been abbreviated and color-coded for better readability. Figure 6.1b shows an initial

<sup>1</sup>Note that trace fragments respectively incomplete traces are also sometimes referred to as partial traces [35].

process model that has been discovered from the two complete traces  $\langle CF, SF, IFN, ISDAP, AP, RRAP, NRAO, P \rangle$  and  $\langle CF, SF, IFN, IDAP, AP, SAP, P \rangle$  using the Inductive Miner. The model precisely describes the two traces. Next, consider the trace postfix  $\langle IFN, P, AP, P \rangle$  that should be (incrementally) added to the initial model shown in Figure 6.1b. Note that since it is a postfix alignment, other activities could happen before. When using the proposed LCA-IPDA (cf. Chapter 5), which does not support trace fragments, i.e., all trace fragments are treated as complete ones, we obtain the model shown in Figure 6.1c. Note that the Inductive Miner returns the same model when using the two initial traces and the trace postfix. Recall that the Inductive Miner also does not support trace fragments. Consider the model shown in Figure 6.1c. Activities CP, SF, IDAP, RRAP, NRAO, and SAP become optional since they are not contained in the provided trace postfix. However, neither the trace postfix nor the two initially used traces require these activities to be optional. Thus, the model shown in Figure 6.1c is imprecise. In contrast, the model depicted in Figure 6.1d is more precise than the one shown in Figure 6.1c. For example, activities CF, SF, and IFN are not optional, i.e., they must be executed at the beginning according to the model. The fact that these activities must be executed at the beginning is consistent with the two initially-used traces and with the trace postfix. For the trace postfix, these activities can be ignored as a trace postfix does not contain any information on activities that are executed at the start. Overall, the model shown in Figure 6.1d, which is discovered with the technique proposed in this chapter, is preferred over the model shown in Figure 6.1c because the latter is more imprecise concerning the two initially used traces and the trace postfix. Moreover, the model in Figure 6.1d contains duplicate labels, i.e., there exists two leave nodes that are labeled IFN. For instance, the Inductive Miner is not capable of discovering models having duplicate labels. In short, the example shows the need to support trace fragments if users want to consider process behavior from trace fragments in process discovery.

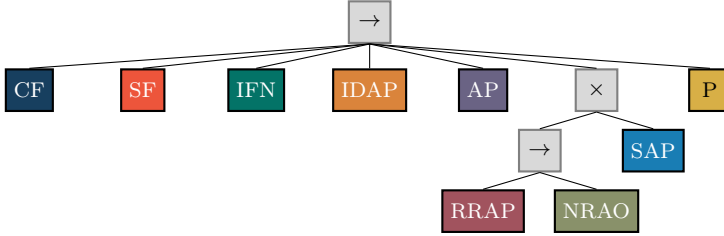
The central research question addressed in this chapter is: *How can trace fragments, i.e., trace prefixes/infixes/postfixes, be incrementally added to a process model?* We answer this research question by extending the incremental process discovery framework introduced in Section 5.1 to support trace fragments. Further, we provide a specific instantiation of the extended framework. We propose a novel *Trace-Fragment-Supporting Incremental Process Discovery Algorithm (TFS-IPDA)* that allows gradually discovering models from complete traces (as introduced in Chapter 5) and trace fragments. The proposed Trace-Fragment-Supporting Incremental Process Discovery Algorithm (TFS-IPDA) builds on the proposed LCA-IPDA.

The remainder of this chapter is organized as follows. Section 6.1 introduces the extended incremental process discovery framework that incorporates trace fragments besides complete traces. Finally, we present a concrete LCA-IPDA that instantiates the extended framework introduced in Section 6.1 and employs infix and postfix alignments presented in Chapter 4.

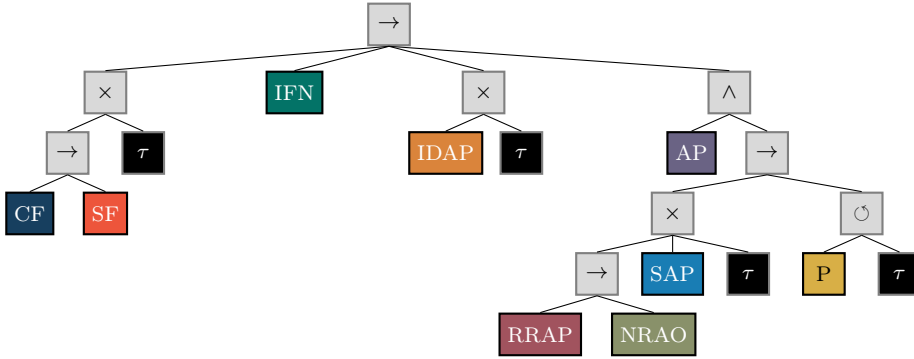


CF	Create Fine	RRAP	Receive Result Appeal from Prefecture
SF	Send Fine	NRAO	Notify Result Appeal to Offender
IFN	Insert Fine Notification	SAP	Send Appeal to Prefecture
IDAP	Insert Date Appeal to Prefecture	P	Payment
AP	Add Penalty		

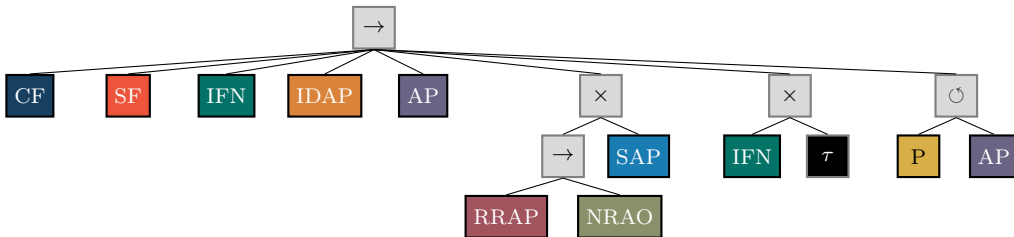
(a) Overview activity abbreviations



(b) Initial process model discovered using the Inductive Miner using the traces  $\langle CF, SF, IFN, IDAP, AP, RRAP, NRAO, P \rangle$  and  $\langle CF, SF, IFN, IDAP, AP, SAP, P \rangle$



(c) Adding trace postfix  $\langle IFN, P, AP, P \rangle$  using the LCA-IPDA (cf. Chapter 5), which considers the trace postfix to be complete



(d) Adding trace postfix  $\langle IFN, P, AP, P \rangle$  using the IPDA proposed in this chapter, which considers the trace postfix as a postfix

Figure 6.1: Example from the road traffic fine management log [56] showing the impact if a trace postfix is falsely considered as a complete trace within IPD

## 6.1. Extended IPD Framework

This section presents the extended IPD framework that supports trace fragments, i.e., trace prefixes, infixes, and suffixes, besides complete traces. Further, we briefly elaborate on the potential origins of trace fragments and why trace fragments are worth to consider. To this end, we first introduce notation conventions used throughout this chapter. We use the symbol

$$\square \in \{cmplt, pre, inf, pos\}$$

to indicate the type of an artifact, for instance, a trace or an alignment. We refer to complete as *cmplt*, prefix as *pre*, infix as *inf*, and postfix as *pos*. Recall Definition 3.30 (page 69), specifying the language of a process tree, i.e., complete traces supported by a process tree. Below, we define a given process tree's prefix, infix, postfix, and complete language.

**Definition 6.1** (Process Tree Prefix/Infix/Postfix/Complete Language)

Let  $\Lambda \in \mathcal{P}$ . We define its

- complete language as  $\mathbb{L}_{cmplt}(\Lambda) = \mathbb{L}(\Lambda)$  (cf. Definition 3.30 on page 69)<sup>a</sup>,
- prefix language as  $\mathbb{L}_{pre}(\Lambda) = \{\sigma_1 \mid \sigma_1, \sigma_2 \in \mathcal{A}^* \wedge \sigma_1 \circ \sigma_2 \in \mathbb{L}(\Lambda)\}$
- infix language as  $\mathbb{L}_{inf}(\Lambda) = \{\sigma_2 \mid \sigma_1, \sigma_2, \sigma_3 \in \mathcal{A}^* \wedge \sigma_1 \circ \sigma_2 \circ \sigma_3 \in \mathbb{L}(\Lambda)\}$ , and
- postfix language as  $\mathbb{L}_{pos}(\Lambda) = \{\sigma_2 \mid \sigma_1, \sigma_2 \in \mathcal{A}^* \wedge \sigma_1 \circ \sigma_2 \in \mathbb{L}(\Lambda)\}$ .

<sup>a</sup>For completeness reasons and better distinguishability, we introduce a second symbol in this chapter for the complete language of a process tree, i.e.,  $\mathbb{L}_{cmplt}(\Lambda) = \mathbb{L}(\Lambda)$ .

Figure 6.2 illustrates the extended IPD framework. We use red outlines in the extended framework depicted in Figure 6.2 to highlight essential modifications compared to the IPD framework depicted in Figure 5.1 (page 110). Fundamentally, the extended framework remains as presented in Figure 5.1. The essential modification appears in trace to be added next  $\sigma_{next}$  that can be either a complete trace, a trace prefix, an infix, or postfix. The corresponding interpretation  $\square$  of  $\sigma_{next}$  indicates the given type, i.e.,  $\square \in \{cmplt, pre, inf, pos\}$ . Further, the set of previously added traces  $A$  (cf. Figure 5.1) is now divided into four sets: previously added complete traces  $A_{cmplt}$ , trace prefixes  $A_{pre}$ , trace infixes  $A_{inf}$ , and trace postfixes  $A_{pos}$ . Like in the initial framework (cf. Figure 5.1), the user selected trace respectively trace fragment, the (initial) process tree, and the previously added traces and trace fragments are fed into a TFS-IPDA, which is central to the framework. The TFS-IPDA returns an extended process tree  $\Lambda'$  that supports  $\sigma_{next}$ , the previously added traces, and trace fragments. Finally,  $\sigma_{next}$  is added to the corresponding set of previously added traces or trace fragments depending on its interpretation  $\square$ . Definition 6.2 on page 160 formally dspecifies TFS-IPDAs.

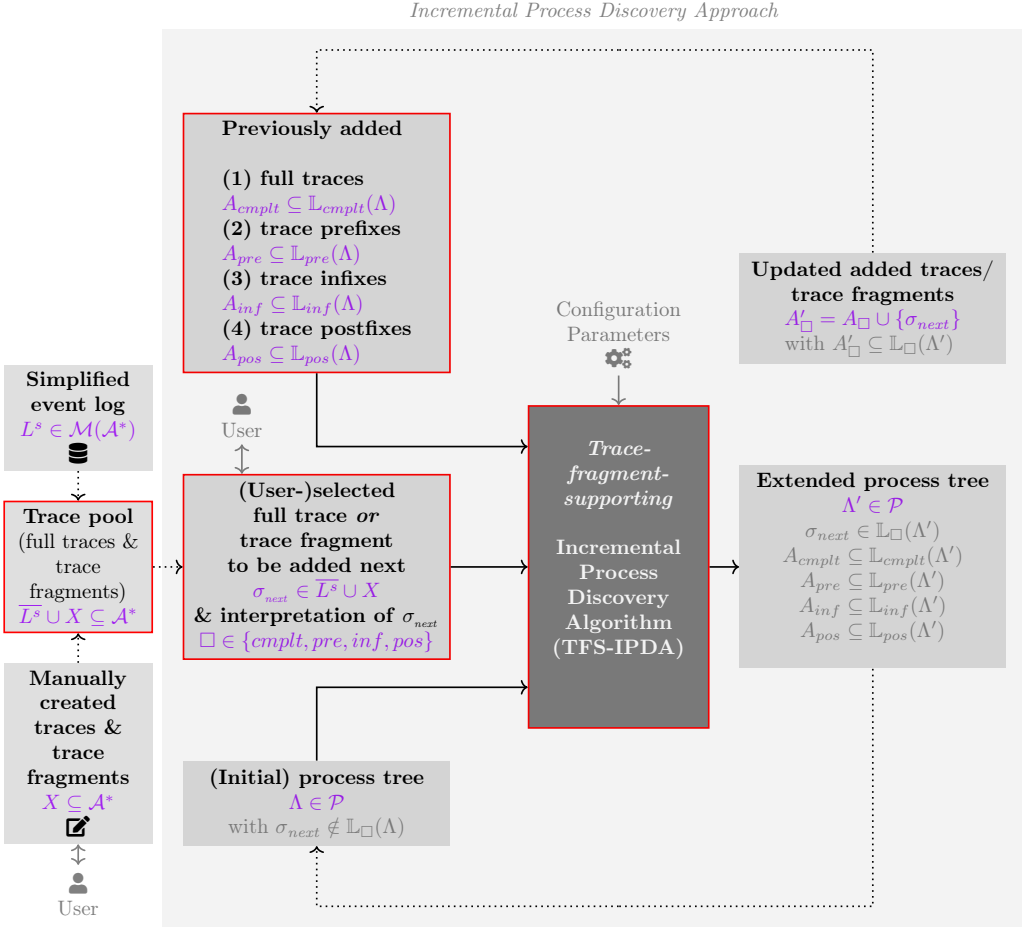


Figure 6.2: Input-output perspective of the proposed incremental process discovery framework extended to support trace fragments

**Definition 6.2** (Trace-Fragment-Supporting Incremental Process Discovery Algorithm (TFS-IPDA))

The function

$$tfsIpda : \mathbb{P}(\mathcal{A}^*) \times \mathbb{P}(\mathcal{A}^*) \times \mathbb{P}(\mathcal{A}^*) \times \mathbb{P}(\mathcal{A}^*) \times \mathcal{A}^* \times \mathcal{P} \rightarrow \mathcal{P}$$

is an TFS-IPDA if for any

- previously added
  - complete traces  $A_{cmplt} \subseteq \mathcal{A}^*$  with  $A_{cmplt} \subseteq \mathbb{L}_{cmplt}(\Lambda)$ ,
  - trace prefixes  $A_{pre} \subseteq \mathcal{A}^*$  with  $A_{pre} \subseteq \mathbb{L}_{pre}(\Lambda)$ ,
  - trace infixes  $A_{inf} \subseteq \mathcal{A}^*$  with  $A_{inf} \subseteq \mathbb{L}_{inf}(\Lambda)$ ,
  - trace postfixes  $A_{pos} \subseteq \mathcal{A}^*$  with  $A_{pos} \subseteq \mathbb{L}_{pos}(\Lambda)$ ,
- complete trace or trace fragment  $\sigma_{next} \in \mathcal{A}^*$  to be added next with corresponding interpretation  $\square \in \{cmplt, pre, inf, pos\}$ , and
- (initial) tree  $\Lambda \in \mathcal{P}$

it holds that

- $\sigma_{next} \in \mathbb{L}_{\square}(tfsIpda(A_{cmplt}, A_{pre}, A_{inf}, A_{pos}, \sigma_{next}, \Lambda))$ ,
- $A_{cmplt} \subseteq \mathbb{L}_{cmplt}(tfsIpda(A_{cmplt}, A_{pre}, A_{inf}, A_{pos}, \sigma_{next}, \Lambda))$ ,
- $A_{pre} \subseteq \mathbb{L}_{pre}(tfsIpda(A_{cmplt}, A_{pre}, A_{inf}, A_{pos}, \sigma_{next}, \Lambda))$ ,
- $A_{inf} \subseteq \mathbb{L}_{inf}(tfsIpda(A_{cmplt}, A_{pre}, A_{inf}, A_{pos}, \sigma_{next}, \Lambda))$ , and
- $A_{pos} \subseteq \mathbb{L}_{pos}(tfsIpda(A_{cmplt}, A_{pre}, A_{inf}, A_{pos}, \sigma_{next}, \Lambda))$ .

If  $A_{cmplt} \not\subseteq \mathbb{L}_{cmplt}(\Lambda) \vee A_{pre} \not\subseteq \mathbb{L}_{pre}(\Lambda) \vee A_{inf} \not\subseteq \mathbb{L}_{inf}(\Lambda) \vee A_{pos} \not\subseteq \mathbb{L}_{pos}(\Lambda)$ , function  $tfsIpda$  is undefined.

As stated at the beginning of this chapter, trace fragments are a frequent phenomenon in real-life event data. In the extended framework depicted in Figure 6.2, for the sake of simplicity, we represent a trace pool that contains both complete traces and trace fragments. As in the initial framework (cf. Figure 5.1), an event log and manually-created traces/trace fragments may fill the trace pool. However, the information if a trace is a complete trace or a trace fragment is in most event logs not present. Thus, users must utilize domain knowledge from the process under study to detect trace fragments from event logs. Alternatively, users may apply fully automated techniques to detect and label trace fragments; for instance, a classifier that can detect trace fragments is proposed in [24].

The above approaches for obtaining trace fragments presume their existence within the event log and merely focus on their detection. Although complete traces are available, users may only require specific trace fragments from complete ones in some instances. Thus, users do not incrementally add the complete trace but rather a fragment of a

complete one. For example, imagine a more extensive process consisting of various stages that may even cross organizational boundaries. Starting from a rough initial process model that covers the entire process, users can gradually discover the different process stages consecutively to maintain an overview and to more easily incorporate the expertise of the process participants of individual process stages into the process model to be discovered. In such a scenario, users of incremental process discovery might extract fragments from complete traces and add them to the pool of traces and trace fragments. Finally, users might apply frequent pattern mining approaches to obtain trace fragments from complete traces. For instance, in [249], an algorithm is proposed to discover frequent subsequences from sequences that can be easily adapted to traces as considered in process mining.

In summary, trace fragments may originate from various sources, respectively approaches. Trace fragments may be *automatically or manually identified* from event logs, which usually do not label traces as complete, prefix, infix, or postfix. Further, incremental process discovery users can *manually create* trace fragments when a particular process behavior that should be incorporated into the process model but is not recorded in the event data. Finally, *frequent pattern mining* approaches can be applied to discover frequent trace fragments from complete traces.

## 6.2. Trace-Fragment-Supporting IPDA

This section introduces a specific TFS-IPDA that can be embedded in the extended framework illustrated in Figure 6.2. The proposed TFS-IPDA builds upon the LCA-IPDA, introduced in Section 5.3. In the remainder of this section, Section 6.2.1 presents a running example. Subsequently, Section 6.2.2 presents the algorithm formally.

### 6.2.1. Running Example

This section presents a running example of the proposed TFS-IPDA. Figure 6.3 depicts an input process tree  $\Lambda$ . As in the LCA-IPDA (cf. Algorithm 5.2), the tree is initially extended by artificial start and end activities, cf. the red highlighted elements in Figure 6.3. In the following, we refer to the extended process tree as  $\Lambda$ .

Further, assume as input trace (fragment)  $\sigma_{next}$  to be added next, its interpretation  $\square \in \{cmplt, pre, inf, pos\}$ , previously added complete traces  $A_{cmplt}$ , and previously added trace fragments (i.e.,  $A_{pre}, A_{inf}, A_{pos}$ ). Below, we list these inputs. Like in Section 5.3, we extend the traces and trace fragments with the artificial start and completion activities correspondingly; for instance, we only extend trace postfixes by the end activity  $\blacksquare$  but not with the start activity. We highlight the extensions below using red font color.

- $\square = postfix$
- $\sigma_{next} = \langle a, a, f, \blacksquare \rangle$  with  $\sigma_{next} \notin \mathbb{L}_{pos}(\Lambda)$
- $A_{cmplt} = \{ \sigma_1 = \langle \blacktriangleright, a, b, a, b, f, e, a, \blacksquare \rangle \}$  with  $\sigma_1 \in \mathbb{L}_{cmplt}(\Lambda)$
- $A_{pre} = \{ \sigma_2 = \langle \blacktriangleright, d, c, e, a \rangle \}$  with  $\sigma_2 \in \mathbb{L}_{pre}(\Lambda)$

- $A_{inf} = \{\sigma_3 = \langle c, f, a \rangle, \sigma_4 = \langle b, d, c, e, a \rangle\}$  with  $\sigma_3, \sigma_4 \in \mathbb{L}_{inf}(\Lambda)$
- $A_{pos} = \{\sigma_5 = \langle f, a, \blacksquare \rangle\}$  with  $\sigma_5 \in \mathbb{L}_{pos}(\Lambda)$

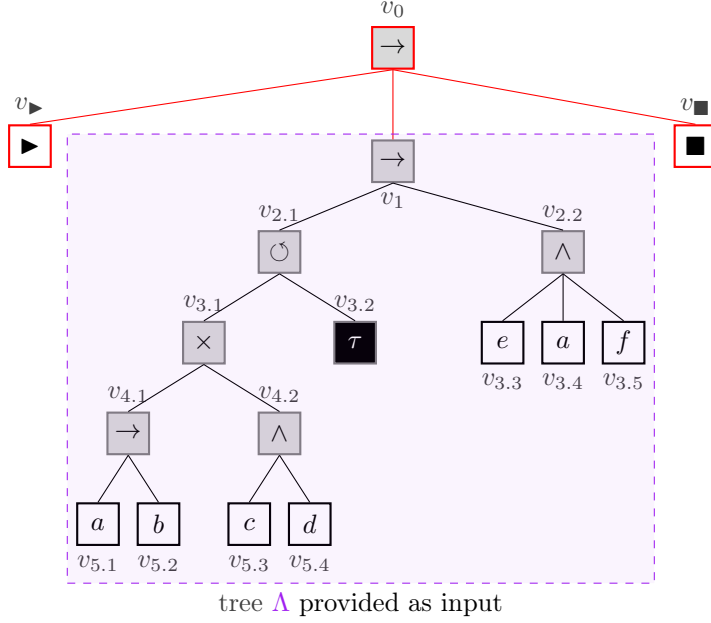


Figure 6.3: Input process tree  $\Lambda$ , which is extended by artificial start  $\blacktriangleright$  and end  $\blacksquare$  activities; extensions are highlighted in red

Next, an optimal postfix alignment is computed to reveal the deviations between trace postfix  $\sigma_{next}$  and tree  $\Lambda$ . Figure 6.4 depicts an optimal postfix alignment  $\gamma_{pos}$  indicating a deviation in its second move, i.e., a log move on the second  $a$  activity.

	1.	2.	3.	4.	5.	6.	7.
$\gamma_{pos} =$	$a$	$a$	$f$	$\gg$	$\gg$	$\blacksquare$	$\gg$
	$(v_{3.4}, a)$	$\gg$	$(v_{3.5}, f)$	$(v_{2.2}, close)$	$(v_1, close)$	$(v_{\blacksquare}, \blacksquare)$	$(v_0, close)$

Figure 6.4: Optimal postfix alignment  $\gamma_{pos} \in \Gamma_{pos}^{opt}(\Lambda, \sigma_{next})$  with one deviation-indicating alignment move at index 2, i.e., a log move on activity  $a$

Utilizing the information in  $\gamma_{pos}$  (cf. Figure 6.4), we compute the corresponding subtree of  $\Lambda$  causing the deviation. In this case, we compute the LCA from the tree vertices corresponding to the two synchronous moves at position 1 and 3. Thus,  $lca_{\Lambda}(v_{3.4}, v_{3.5}) = v_{2.2}$ . Vertex  $v_{2.2}$  represents the root vertex of the subtree, i.e.,  $\Delta_{\Lambda}(v_{2.2})$ , causing the first

(and only) deviation indicated in  $\gamma_{pos}$ , cf. Figure 6.4. Figure 6.5 visualizes the subtree. As we can observe, subtree  $\Lambda_{LCA}$  does not support the execution of activity  $a$  more than once; however,  $\sigma_{next}$  contains two  $a$  activities that, according to  $\gamma_{pos}$ , should be executed between executing vertex  $v_{3.4}$  and  $v_{3.5}$ .

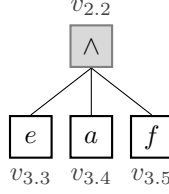


Figure 6.5: Problematic subtree  $\Lambda_{LCA} = \Delta_{\Lambda}(v_{2.2})$  causing the deviation indicated in  $\gamma_{pos}$  (cf. Figure 6.4) between  $\sigma_{next}$  and  $\Lambda$

Similar to the LCA-IPDA (cf. Algorithm 5.2), we compute a corresponding log for subtree  $\Lambda_{LCA}$  containing traces that  $\Lambda_{LCA}$  must support. Below, we depict the corresponding sublog  $L_{LCA}$  for  $\Lambda_{LCA}$ .

$$L_{LCA} = \left[ \begin{array}{ll} \langle e, a, a, f \rangle, & \text{derived from } \sigma_{next} \\ \langle f, e, a \rangle, & \text{derived from } \sigma_1 \\ \langle e, a, f \rangle^2, & \text{derived from } \sigma_2, \sigma_4 \\ \langle f, a, e \rangle, & \text{derived from } \sigma_3 \\ \langle e, f, a \rangle & \text{derived from } \sigma_5 \end{array} \right]$$

Consider the trace  $\langle e, a, a, f \rangle$  shown above. As indicated, this trace is derived from  $\sigma_{next} = \langle a, a, f \rangle$ . Simply adding  $\sigma_{next} = \langle a, a, f \rangle$  to the sublog  $L_{LCA}$  results in an unnecessary imprecise subtree upon rediscovering  $\Lambda_{LCA}$  using function *discovery* (cf. Definition 3.32). Function *discovery* would return a tree that contains activity  $e$  as an optional activity. However, none of the traces and trace fragments provided as inputs require  $e$  to be optional, cf. the input traces and trace fragments shown earlier. To this end, we extend  $\sigma_{next} = \langle a, a, f \rangle$  by an  $e$  activity in the beginning, i.e., we add  $\langle e \rangle \circ \langle a, a, f \rangle$  to the sublog  $L_{LCA}$ . Similarly, when replaying the trace prefix  $\sigma_2 = \langle \blacktriangleright, d, c, e, a \rangle$ , only the last two activities, i.e.  $\langle e, a \rangle$ , are replayed in subtree  $\Lambda_{LCA}$ . Thus, we also extend for this trace prefix the corresponding trace that is added to  $L_{LCA}$ ; we add  $\langle e, a, f \rangle$  to  $L_{LCA}$ .

Next, we invoke *discovery* on the constructed sublog  $L_{LCA}$ . The obtained subtree  $discovery(L_{LCA}) \in \mathcal{P}$  replaces the subtree  $\Delta_{\Lambda}(v_{2.2})$  in  $\Lambda$ . Figure 6.6 illustrates the result, i.e., tree  $\Lambda$  with replaced subtree  $\Lambda_{LCA}$ .<sup>2</sup> Since  $\gamma_{pos}$  (cf. Figure 6.4) indicates only one deviation, which we just resolved in  $\Lambda$ , we know that  $\sigma_{next} \in \mathbb{L}_{pos}(\Lambda)$ . Further, altered process tree  $\Lambda$  (cf. Figure 6.6) still supports all previously added traces and trace fragments.

<sup>2</sup>Depending on the concrete instantiation of the function *discovery*, other process trees than the one depicted in Figure 6.6 are conceivable.

Finally, we remove the added vertices  $v_0, v_{\blacktriangleright}$ , and  $v_{\blacksquare}$  from the tree depicted in Figure 6.6. Likewise, we remove the artificial start and end activities from previously added traces and trace fragments, i.e.,  $A_{\blacksquare}, A_{pre}, A_{inf}, A_{pos}$ . In the subsequent section, we formally introduce the proposed TFS-IPDA in detail.

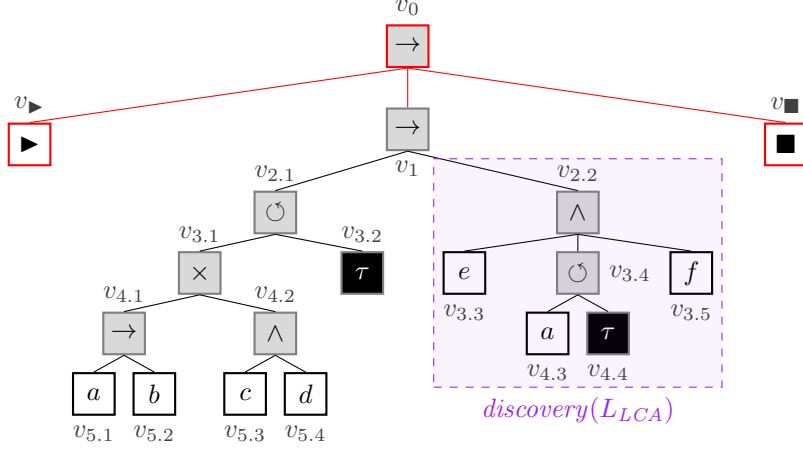


Figure 6.6: Process tree  $\Lambda$  after replacing  $\Delta_{\Lambda}(v_{2.2})$  by  $discovery(L_{LCA})$

### 6.2.2. Algorithm

Algorithm 6.1 presents the proposed TFS-IPDA. The essential structure of Algorithm 6.1 is very similar to the LCA-IPDA algorithm presented in Algorithm 5.2 (page 118). For this reason, we highlight significant changes in Algorithm 6.1 compared to Algorithm 5.2. In the following, we describe the changes in detail.

First, the *input preprocessing phase* is executed. Identical to Algorithm 5.2, the tree  $\Lambda$  is extended by artificial start and end activities (line 1). In line 2, the only change compared to Algorithm 5.2 is the consideration of the interpretation  $\square$  of  $\sigma_{next}$  to ensure adding it in the corresponding set of traces/trace fragments. As before, we extend complete traces by a start  $\blacktriangleright$  and end activity  $\blacksquare$  (line 3). Accordingly, we add the start activity  $\blacktriangleright$  to each trace prefix (line 4). Likewise, we add the end activity  $\blacksquare$  to each trace postfix (line 5). Note that we do not modify trace infixes since other activities can happen before and after; thus, adding start or end activities is not feasible.



**Algorithm 6.1:** TFS-IPDA

---

**Input:**  $A_{cmplt} \subseteq \mathcal{A}^*$ ,  $A_{pre} \subseteq \mathcal{A}^*$ ,  $A_{inf} \subseteq \mathcal{A}^*$ ,  $A_{pos} \subseteq \mathcal{A}^*$ ,  $\sigma_{next} \in \mathcal{A}^*$ ,  
 $\square \in \{cmplt, pre, inf, pos\}$ , // interpretation of trace (fragment)  $\sigma$   
 $\Lambda \in \mathcal{T}$   
**Output:**  $\Lambda \in \mathcal{T}$

**begin**

```

1  /* input preprocessing phase */
2   $\Lambda \leftarrow \text{extend } \Lambda \text{ by artificial } \blacktriangleright \text{ and } \blacksquare \text{ activities}$  // cf. Figure 5.6 (page 119)
3   $A_{\square} \leftarrow A_{\square} \cup \{\sigma_{next}\}$  // adding  $\sigma_{next}$  to corresponding  $A_{\square}$ 
4   $A_{cmplt} \leftarrow \{\langle \blacktriangleright \rangle \circ \sigma \circ \langle \blacksquare \rangle \mid \sigma \in A_{cmplt}\}$  // extend complete traces by start & end activities
5   $A_{pre} \leftarrow \{\langle \blacktriangleright \rangle \circ \sigma \mid \sigma \in A_{pre}\}$  // extend trace prefixes by a start activity
6   $A_{pos} \leftarrow \{\sigma \circ \langle \blacksquare \rangle \mid \sigma \in A_{pos}\}$  // extend trace postfixes by an end activity
7  /* main phase */
8  let  $\gamma \in \Gamma_{\square}^{opt}(\Lambda, \sigma_{next})$  // calculate an optimal  $\square$  alignment  $\gamma$ 
9  while deviation( $\gamma$ ) do //  $\sigma_{next} \notin \mathbb{L}(\Lambda)$ 
10      $i \leftarrow \text{firstDeviationMvIndex}(\gamma)$  // first deviation-indicating move index
11      $\Lambda_{LCA} \leftarrow \text{subtree}(\Lambda, \gamma, i)$  // Definition 5.2 (page 122)
12     if  $\Lambda_{LCA} \neq \text{undefined}$  then // subtree  $\Lambda_{LCA}$  could be determined
13          $L_{LCA} \leftarrow \text{computeSublog}^{TFS}(\Lambda, \Lambda_{LCA}, A)$  // Algorithm 5.3 (page 124)
14          $\Lambda \leftarrow \text{replace } \Lambda_{LCA} \sqsubseteq \Lambda \text{ by } \text{discovery}(L_{LCA})$ 
15     else // no subtree  $\Lambda_{LCA}$  could be determined
16          $\Lambda \leftarrow \text{extend } \Lambda \text{ according to Figure 6.7}$ 
17     let  $\gamma \in \Gamma_{\square}^{opt}(\Lambda, \sigma_{next})$  // calculate an optimal alignment  $\gamma$ 
18  /* output postprocessing phase */
19   $\Lambda \leftarrow \text{remove artificial } \blacktriangleright \text{ and } \blacksquare \text{ activities from } \Lambda$  // added in line 1
20   $\Lambda \leftarrow \text{apply process tree reduction rules to } \Lambda$  // cf. [120, Chapter 5]
21  return  $\Lambda$ 

```

---

In short, the preprocessing phase of the input consists only of slightly more lines since different trace fragment types are handled individually compared to Algorithm 5.2, which only considers complete traces. Otherwise, the input preprocessing phase are very alike when comparing Algorithm 5.2 and Algorithm 6.1.

The *main phase* of the algorithm starts at line 6 until line 15. The first difference compared to Algorithm 5.2 is that we calculate an optimal  $\square$  alignment in lines 6 and 15, i.e., an alignment according to the interpretation of  $\sigma_{next} \in \{cmplt, pre, inf, pos\}$ . Using the optimal  $\square$  alignment, we determine the subtree  $\Lambda_{LCA}$  in line 9. Recall that *subtree* (cf. Definition 5.2)) always returns a subtree  $\Lambda_{LCA}$  when invoked by Algorithm 5.2. However,

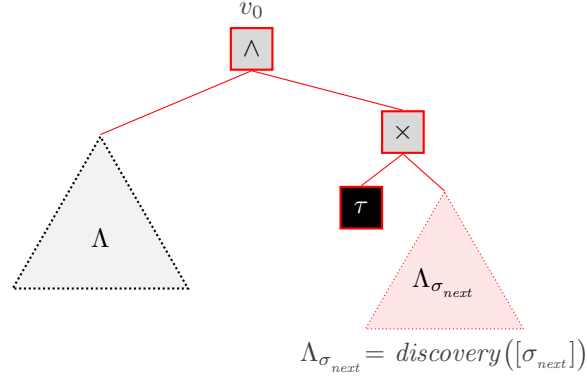


Figure 6.7: Extending tree  $\Lambda$  above its root node, cf. Algorithm 6.1 line 14; red highlighted elements are added to  $\Lambda$

when dealing with trace infixes in Algorithm 6.1, *subtree* might be undefined for specific inputs. Thus, Algorithm 6.1 makes a case distinction depending on whether  $\Lambda_{LCA}$  could be determined, cf. lines 10 to 14.

If  $\Lambda_{LCA}$  is determined (line 10), we compute a sublog  $L_{LCA}$  for the subtree  $\Lambda_{LCA}$  (line 11). Subsequently, we replace subtree  $\Lambda_{LCA}$  by a rediscovered one from  $L_{LCA}$  (Line 12), analogously to Algorithm 5.2. In the other case, i.e.,  $\Lambda_{LCA}$  could not be determined (line 13), we modify tree  $\Lambda$  as illustrated in Figure 6.7. Note that this case only applies if  $\sigma_{next}$  is a trace infix, and  $\Lambda$  does not contain any of its activities, cf. Definition 5.2.<sup>3</sup> In this case, we have no reference point in the tree  $\Lambda$  where the non-fitting infix  $\sigma_{next}$  should take place in tree  $\Lambda$ . Therefore, we extend tree  $\Lambda$  at its root vertex by a new vertex  $v_0$  labeled with the parallel operator (cf. Figure 6.7). Further, we add below the new root vertex  $v_0$  a tree that allows optionally execution of the trace infix  $\sigma_{next}$ . To this end, we utilize the function *discovery* that returns a tree  $\Lambda_{\sigma_{next}}$  with  $\sigma_{next} \in \mathbb{L}_{cmplt}(\Lambda_{\sigma_{next}})$ . Thus, applying this extension rule to  $\Lambda$ , we can guarantee that the extended process tree supports the trace infix  $\sigma_{next}$ .

The remaining parts of Algorithm 6.1 are similar to Algorithm 5.2. Thus, after resolving the first block of deviations as indicated in  $\gamma$ , again an optimal  $\square$  alignment is computed (line 15) and, in case of further deviations, the loop (cf. line 7–line 15) is executed again. Finally, after resolving all deviations between  $\sigma_{next}$  and  $\Lambda$ , the output postprocessing phase is executed, identical to Algorithm 5.2.

<sup>3</sup>If  $\sigma_{next}$  is a trace infix,  $\sigma_{next}$  is not extended by the artificial start and end activities compared to complete traces, prefix, and postfix traces, cf. lines 2 to 5. Thus, if no activity from  $\sigma_{next}$  is contained in process tree  $\Lambda$ , the corresponding optimal infix alignment contains only log moves. Therefore, function *subtree* cannot find a subtree since the optimal infix alignment contains no reference point in the tree.

### Sublog Calculation

This section covers the sublog calculation for the determined subtree  $\Lambda_{LCA}$ , cf. Algorithm 6.1 line 11. Recall the running example with  $\Lambda_{LCA} = \Delta_{\Lambda}(v_{2.2})$ , cf. Figure 6.5 (page 163). The alignment  $\gamma_{pos}$  (cf. Figure 6.4) contains a log move on activity  $a$ ; thus, subtree  $\Lambda_{LCA} = \Delta_{\Lambda}(v_{2.2})$  should support the replay of two  $a$  activities. Note that we show  $\gamma_{pos}$  from Figure 6.4 also in the lower right of Figure 6.8. In the LCA-IPDA approach, introduced in Chapter 5, we iterate over the alignment, look for openings/closings of the determined subtree and extract traces. Applying this approach to  $\gamma_{pos}$  does not immediately work because the alignment  $\gamma_{pos}$  does not contain an opening of  $\Lambda_{LCA}$ 's root vertex  $v_{2.2}$ , cf. Figure 6.8. In detail,  $\gamma_{pos}$ 's first move is a synchronous move on vertex  $v_{3.4}$  labeled  $a$ , which belongs to  $\Lambda_{LCA}$ . Thus,  $\Lambda_{LCA}$ 's root vertex  $v_{2.2}$  must have been opened before, which is not contained in  $\gamma_{pos}$ . Assume that the missing opening of  $\Lambda_{LCA}$ 's root vertex is negligible, we simply assume it was opened before. Applying the trace extraction to  $\gamma_{pos}$  as for the LCA-IPDA, cf. *extractSubTraces* in Algorithm 5.4, results in the trace  $\langle a, a, f \rangle$ . Thus, upon rediscovering  $\Lambda_{LCA}$ , activity  $e$  will be optional because the trace  $\langle a, a, f \rangle$  extracted from  $\gamma_{pos}$  does not contain an  $e$  activity. However, when looking at all previously added traces and trace fragments, they do not require activity  $e$  to be optional, cf. Section 6.2.1. Likewise, the trace postfix to be added next  $\sigma_{next} = \langle a, a, f \rangle$  does not require activity  $e$  to be optional in subtree  $\Lambda_{LCA}$ . Thus, when adding trace  $\langle a, a, f \rangle$  to the sublog  $L_{LCA}$ , which is used to rediscover subtree  $\Lambda_{LCA}$ , the resulting process tree would be unnecessarily imprecise because activity  $e$  would be optional due to trace  $\langle a, a, f \rangle$  contained in the sublog  $L_{LCA}$  used for rediscovery.

To avoid imprecise subtrees upon rediscovery, we must adapt the technique to extract traces from an alignment. To this end, we extend alignment  $\gamma_{pos}$  into a complete alignment. Consider Figure 6.8 showing a complete alignment  $\gamma \in \Gamma(\Lambda, \sigma_{next})$  that is composed of a prefix alignment  $\gamma_{pre} \in \Gamma_{pre}(\Lambda, \langle \rangle)$  aligning the empty sequence and the optimal postfix alignment  $\gamma_{pos} \in \Gamma_{pos}^{opt}(\Lambda, \sigma_{next})$ . Note that  $\gamma_{pre}$  is not an optimal alignment. Further, the complete alignment  $\gamma_{\blacksquare}$  is not necessarily optimal. Considering the complete alignment  $\gamma$  depicted in Figure 6.8. In general, when iterating over the extended alignment, i.e., a complete alignment, it is now guaranteed that we find to any opening/closing of an inner vertex a corresponding closing/opening. We observe the opening of  $\Delta_{LCA}$ 's root vertex in alignment move 12, which is part of the prefix alignment  $\gamma_{pre}$ , and its closing in alignment move 17. Between the opening and closing of  $\Lambda_{LCA}$ 's root vertex four alignment moves are contained that we process from left to right. Move 13 represents a visible model move on activity  $e$ . Usually, we would ignore visible model moves. However, this visible model move is *not* part of the optimal postfix alignment  $\gamma_{pos}$ ; thus, we add activity  $e$  to the trace. From move 14 until 16 we proceed as usual; we add two times an  $a$  and once a  $f$  activity to the trace. In alignment move 17,  $\Lambda_{LCA}$ 's root vertex is closed and never opened again. Thus, we finally return the extracted trace  $\langle e, a, a, f \rangle$  and add this one to the sublog  $L_{LCA}$ . Recall the complete sublog  $L_{LCA}$  depicted in Section 6.2.1.

Subsequently, we present an adapted function *computeSublog*<sup>TFS</sup> that builds upon the idea of extending prefix/infix/postfix alignments as exemplified above. The adapted function *computeSublog*<sup>TFS</sup> is invoked in Algorithm 6.1 line 11. Algorithm 6.2 specifies *computeSublog*<sup>TFS</sup>. In essence, Algorithm 6.2 iterates over all traces and trace fragments,

---

**Algorithm 6.2:**  $\text{computeSublog}^{TFS}$  (called in Algorithm 6.1 line 11)
 

---

```

input :  $\Lambda \in \mathcal{P}$ , // entire process tree
 $\Lambda_{LCA} \subseteq \Lambda \in \mathcal{P}$ , // subtree causing deviation(s)
 $A_{cmplt}$ , // previously added complete traces and  $\sigma_{next}$  if it is complete trace
 $A_{pre}$ , // previously added trace prefixes and  $\sigma_{next}$  if it is trace prefix
 $A_{inf}$ , // previously added trace infixes and  $\sigma_{next}$  if it is trace infix
 $A_{pos} \subseteq \mathcal{A}^*$  // previously added trace postfixes and  $\sigma_{next}$  if it is trace postfix
output:  $L_{LCA} \subseteq \mathcal{M}(\mathcal{A}^*)$  // sub-log for  $T_{LCA}$ 
begin
1   $L_{LCA} \leftarrow \emptyset$  // initialize sublog for  $\Lambda_{LCA}$ 
2  forall  $\sigma \in A_{cmplt}$  do
3    let  $\gamma \in \Gamma^{opt}(\Lambda, \sigma)$  // calculate an optimal complete alignment
4     $L_{LCA} \leftarrow L_{LCA} \uplus \text{extractSubTraces}^{TFS}(\Lambda_{LCA}, \gamma, \{1, \dots, |\gamma|\})$ 
      // Algorithm 6.3
5  forall  $\sigma \in A_{pre}$  do
6     $\gamma \leftarrow \gamma_{pre} \circ \gamma_{pos}$  such that
      1.  $\gamma_{pre} \in \Gamma_{pre}^{opt}(\Lambda, \sigma)$ 
      2.  $\gamma_{pos} \in \Gamma_{pos}(\Lambda, \langle \rangle)$ 
      3.  $\gamma_{pre} \circ \gamma_{pos} \in \Gamma(\Lambda, \sigma)$ 
7     $I \leftarrow \{1, \dots, i\}$  such that  $\langle \gamma(1), \dots, \gamma(i) \rangle = \gamma_{pre}$ 
8     $L_{LCA} \leftarrow L_{LCA} \uplus \text{extractSubTraces}^{TFS}(T_{LCA}, \gamma, I)$  // Algorithm 6.3
9  forall  $\sigma \in A_{inf}$  do
10    $\gamma \leftarrow \gamma_{pre} \circ \gamma_{inf} \circ \gamma_{pos}$  such that
      1.  $\gamma_{pre} \in \Gamma_{pre}(\Lambda, \langle \rangle)$ 
      2.  $\gamma_{inf} \in \Gamma_{inf}^{opt}(\Lambda, \sigma)$ 
      3.  $\gamma_{pos} \in \Gamma_{pos}^{opt}(\Lambda, \langle \rangle)$ 
      4.  $\gamma_{pre} \circ \gamma_{inf} \circ \gamma_{pos} \in \Gamma(\Lambda, \sigma)$ 
11    $I \leftarrow \{i, \dots, i + |\gamma_{inf}|\}$  such that  $\langle \gamma(i), \dots, \gamma(i + |\gamma_{inf}|) \rangle = \gamma_{inf}$ 
12    $L_{LCA} \leftarrow L_{LCA} \uplus \text{extractSubTraces}^{TFS}(T_{LCA}, \gamma, I)$  // Algorithm 6.3
13  forall  $\sigma \in A_{pos}$  do
14    $\gamma \leftarrow \gamma_{pre} \circ \gamma_{pos}$  such that
      1.  $\gamma_{pre} \in \Gamma_{pre}(\Lambda, \langle \rangle)$ 
      2.  $\gamma_{pos} \in \Gamma_{pos}^{opt}(\Lambda, \sigma)$ 
      3.  $\gamma_{pre} \circ \gamma_{pos} \in \Gamma(\Lambda, \sigma)$ 
15    $I \leftarrow \{i, \dots, |\gamma|\}$  such that  $\langle \gamma(i), \dots, \gamma(|\gamma|) \rangle = \gamma_{pos}$ 
16    $L_{LCA} \leftarrow L_{LCA} \uplus \text{extractSubTraces}^{TFS}(T_{LCA}, \gamma, I)$  // Algorithm 6.3
17  return  $L_{LCA}$ 
  
```

---

complete alignment $\gamma \in \Gamma(\Lambda, \sigma_{next})$												
prefix alignment $\gamma_{pre} \in \Gamma_{pre}(\Lambda, \langle \rangle)$						optimal postfix alignment (cf. Figure 6.4) $\gamma_{pos} \in \Gamma_{pos}^{opt}(\Lambda, \sigma_{next})$						
1	2	...	11	12	13	14	15	16	17	18	19	20
$\gg$	$\gg$	...	$\gg$	$\gg$	$\gg$	$a$	$a$	$f$	$\gg$	$\gg$	■	$\gg$
$(v_0,$ $open$	$(v_1,$ $\blacktriangleright$	...	$(v_{2.1},$ $close$	$(v_{2.2},$ $open$	$(v_{3.3},$ $e$	$(v_{3.4},$ $a$	$\gg$	$(v_{3.5},$ $f$	$(v_{2.2},$ $close$	$(v_1,$ $close$	$(v_{19},$ $\blacksquare$	$(v_0,$ $close$
$\Lambda_{LCA}$ opens						$\Lambda_{LCA}$ closes						

Figure 6.8: Extending the optimal postfix alignment  $\gamma_{pos}$  from the running example shown in Figure 6.4 (page 162) into a complete alignment  $\gamma$

computes alignments, and extracts traces that the subtree  $\Lambda_{LCA}$  must support. Note that for the extraction a slightly modified version of *extractSubTraces* is used; we refer to the modified version as *extractSubTraces*<sup>TFS</sup>, which will be introduced afterwards. Four sections can be distinguished in *computeSublog*<sup>TFS</sup>, cf. Algorithm 6.2.

1. For *complete* traces, an optimal alignment is computed (line 3) and the modified extraction function *extractSubTraces*<sup>TFS</sup> is called (line 4), returning the traces for the sublog  $L_{LCA}$ . Note that *extractSubTraces*<sup>TFS</sup> takes one more argument, i.e., a set indicating the alignment moves in the complete alignment  $\gamma$  that belong to an optimal complete/prefix/infix/postfix alignment. Since we compute an optimal complete alignment, we provide all indices as input, i.e.,  $\{1, \dots, |\gamma|\}$ .
2. For trace *prefixes*, an optimal prefix alignment  $\gamma_{pre}$  is computed that is extended into a complete alignment with a postfix alignment  $\gamma_{pos}$  (line 6). Next, the indices  $I = \{1, \dots, |\gamma_{pre}|\}$  constituting the optimal prefix alignment  $\gamma_{pre}$  within the complete alignment  $\gamma$  are collected (line 7). Finally, *extractSubTraces*<sup>TFS</sup> is invoked (line 8).
3. For trace *infixes*, an optimal infix alignment  $\gamma_{inf}$  is computed that is extended into a complete alignment with a prefix alignment  $\gamma_{pre}$  before  $\gamma_{inf}$  and a postfix alignment  $\gamma_{pos}$  after  $\gamma_{inf}$  (line 10). Next, the indices constituting the optimal infix alignment  $\gamma_{inf}$  within the complete alignment  $\gamma$  are collected (line 11), i.e.,  $I = \{|\gamma_{pre}| + 1, \dots, |\gamma_{pre}| + |\gamma_{inf}|\}$ . Finally, *extractSubTraces*<sup>TFS</sup> is invoked (line 12).
4. For trace *postfixes*, an optimal postfix alignment  $\gamma_{pos}$  is computed that is extended into a complete alignment  $\gamma$  with a prefix alignment  $\gamma_{pre}$  before  $\gamma_{pos}$  (line 14).

For example, recall Figure 6.8. Next, the indices  $I = \{|\gamma_{pre}| + 1, \dots, |\gamma|\}$  constituting the optimal postfix alignment  $\gamma_{pos}$  within the complete alignment  $\gamma$  are collected (line 15). In the example depicted in Figure 6.8,  $I = \{14, \dots, 20\}$ . Finally,  $extractSubTraces^{TFS}$  is invoked (line 16).

In short,  $computeSublog^{TFS}$  simply calculates an optimal complete alignment for complete traces and extracts traces from the complete alignment for the sublog  $L_{LCA}$ . Note that if only complete traces are considered,  $computeSublog^{TFS}$  (cf. Algorithm 5.3) returns the same sublog as  $computeSublog$  (cf. Algorithm 6.2). For trace fragments, a corresponding optimal prefix/infix/postfix alignment is calculated and extended into a complete alignment, which is not necessarily optimal.

Computing first optimal prefix/infix/postfix alignments and subsequently extending them into complete alignments is necessary to: 1) obtain complete running sequences of the tree and 2) to ensure that if parts of the trace fragment fit the process tree, these parts of the trace fragment are synchronized with the process tree. Simply calculating an optimal complete alignment for a given trace fragment is not feasible, although requirement 1) would be satisfied, since it might be optimal to not synchronize any activity from a given trace fragment with a process tree in certain cases; thus, the optimal complete alignment contains no synchronous move. However, when computing a corresponding optimal prefix/infix/postfix alignment, synchronous moves would appear in certain cases. For instance, consider the process tree depicted below, specifying that between start ► and completion ■ either nothing ( $\tau$ ) or the sequence containing  $a, b, c, \dots, x, y, z$  occurs.

$$\rightarrow \left( \blacktriangleright, \times(\tau, \rightarrow(a, b, c, \dots, x, y, z)), \blacksquare \right)$$

Consider the postfix  $\langle y, z, \blacksquare \rangle$ . A corresponding optimal postfix alignment aligns the activities  $y, z$ , and  $\blacksquare$  with the process tree above; thus, an optimal postfix alignment would not contain any deviation-indicating alignment move, i.e., no log and visible model moves. In contrast, any optimal complete alignment for  $\langle y, z, \blacksquare \rangle$ , contains log moves on activities  $y$  and  $z$  because it is cheaper (according to the standard cost function) to not synchronize activities  $y$  and  $z$  as synchronizing these activities implies visible model moves on activities  $a, b, c, \dots, x$ . Thus, the model part of an optimal complete alignment would execute the  $\tau$  instead of the sequence  $\rightarrow(a, b, c, \dots, x, y, z)$ . Consequently, log moves on activities  $y$  and  $z$  are present. Only one synchronous move on the artificial activity  $\blacksquare$  is present in an optimal complete alignment.

Subsequently, we present  $extractSubTraces^{TFS}$  in Algorithm 6.3, which is invoked in Algorithm 6.2 lines 4, 8, 12 and 16. Again, we highlight changes compared to  $extractSubTraces$ , cf. Algorithm 5.4 (page 125). Overall, note that the differences between  $extractSubTraces$  and  $extractSubTraces^{TFS}$  are minor. Two changes apply to the inputs. First, the provided alignment  $\gamma$  is not necessarily optimal, compared to Algorithm 5.4. Second,  $extractSubTraces^{TFS}$  requires next to the subtree  $\Lambda_{LCA}$  and the alignment  $\gamma$  a third input, i.e., a set  $I \subseteq \{1, \dots, |\gamma|\}$ . The elements of  $I$  represent the indices of the alignment moves in  $\gamma$  that belong to an optimal prefix/infix/postfix alignment in case  $\gamma$  is an extended alignment. For example, reconsider the extended alignment shown in Figure 6.8. For this extended alignment,  $I = \{14, \dots, 20\}$ .

Besides the changes in the inputs, the other change affects the condition specifying how a trace  $\sigma'$  is constructed, which is eventually added to  $L_{LCA}$ ; consider line 18 in

**Algorithm 6.3:**  $extractSubTraces^{TFS}$  (called in Algorithm 6.2)

---

**input** :  $\Lambda_{LCA} = (V_{LCA}, E_{LCA}, \Sigma_{LCA}, \lambda_{LCA}, r_{LCA}, <_{LCA}) \sqsubseteq \Lambda \in \mathcal{P}$ ,  
 $\gamma \in \Gamma(\Lambda, \sigma)$ , // (opt.) alignment  $\gamma$  for some  $\sigma \in \mathcal{A}^*$  and a tree  $\Lambda$   
 $I \subseteq \{1, \dots, |\gamma|\}$  // alignment move indices belonging to an optimal  
complete/prefix/infix/postfix alignment for  $\sigma$  and  $\Lambda$  (cf. Algorithm 5.3)  
**output:**  $L_{LCA} \subseteq \mathcal{M}(\mathcal{A}^*)$  // sublog for  $\Lambda_{LCA}$

**begin**

```

1   $L_{LCA} \leftarrow []$  // initialize sub-log for  $\Lambda_{LCA}$ 
2  forall  $1 \leq i \leq |\gamma|$  do // iterate over alignment moves
3       $\sigma' \leftarrow \langle \rangle$  // initialize trace eventually added to  $L_{LCA}$ 
4      if  $V_{LCA} = \{r_{LCA}\}$  then // Case 1:  $\Lambda_{LCA}$  contains only one vertex  $r_{LCA}$ 
5          while  $modelVertex(\gamma(i)) \neq r_{LCA}$  do
6              if  $logMv(\gamma(i))$  then
7                   $\sigma' \leftarrow \sigma' \circ \langle traceLabel(\gamma(i)) \rangle$  // add log moves
8                   $i \leftarrow i+1$ 
9              if  $modelVertex(\gamma(i)) = r_{LCA}$  then
10                  $\sigma' \leftarrow \sigma' \circ \langle modelLabel(\gamma(i)) \rangle$  //  $modelLabel(\gamma(i)) = \lambda_{LCA}(r_{LCA})$ 
11                  $i \leftarrow i+1$ 
12                 if  $\forall i \leq j \leq |\gamma| \left( \neg syncMv(\gamma(j)) \wedge \neg invModelMv(\gamma(j)) \right)$  then
13                      $\sigma' \leftarrow \sigma' \circ \langle traceLabel(\gamma(j)), \dots, traceLabel(\gamma(|\gamma|)) \rangle_{\downarrow \mathcal{A}}$ 
14                  $L_{LCA} \leftarrow L_{LCA} \uplus [\sigma']$  // add trace  $\sigma'$  to the sublog of  $\Lambda_{LCA}$ 
15             else // Case 2:  $\Lambda_{LCA}$  contains more than one vertex
16                 if  $modelVertex(\gamma(i)) = r_{LCA} \wedge modelLabel(\gamma(i)) = open$  then
17                     while  $modelVertex(\gamma(i)) \neq r_{LCA} \vee modelLabel(\gamma(i)) \neq close$  do
18                         if  $modelVertex(\gamma(i)) \in V_{LCA} \wedge$ 
19                              $\left( syncMv(\gamma(i)) \vee \left( visModelMv(\gamma(i)) \wedge i \notin I \right) \right)$  then
20                              $\sigma' \leftarrow \sigma' \circ \langle modelLabel(\gamma(i)) \rangle$ 
21                             else if  $traceLabel(\gamma(i)) \in \mathcal{A}$  then //  $traceLabel(\gamma(i)) \neq \gg$ 
22                                  $\sigma' \leftarrow \sigma' \circ \langle traceLabel(\gamma(i)) \rangle$ 
23                              $i \leftarrow i+1$ 
24                          $L_{LCA} \leftarrow L_{LCA} \uplus [\sigma']$  // add trace  $\sigma'$  to the sublog of  $\Lambda_{LCA}$ 
25             return  $L_{LCA}$ 

```

---

Algorithm 6.3. We add the label of a model vertex to  $\sigma'$  if it is part of the subtree  $\Lambda_{LCA}$  and either a synchronous move or a visible model move that is outside of the extended optimal prefix/infix/postfix alignment, i.e.,  $i \notin I$ . In comparison, *extractSubTraces* (cf. Algorithm 5.4) only adds the label of a model vertex to the trace  $\sigma'$  if its a sync move on a model vertex that corresponds to the subtree  $\Lambda_{LCA}$ . The extended condition in line 18 (Algorithm 6.3) Apart from the modified condition and the changes in the input, *extractSubTraces*<sup>TFS</sup> (Algorithm 5.4) equals *extractSubTraces* (Algorithm 6.3).

Reconsider the extended alignment  $\gamma$  depicted in Figure 6.8 (page 169) and the subtree  $\Lambda_{LCA}$  shown in Figure 6.5 (page 163). When calling *extractSubTraces*<sup>TFS</sup> with  $\gamma$ ,  $\Lambda_{LCA}$ , and  $I = \{14, \dots, 20\}$ , we iterate over  $\gamma$  until the 12. move, representing the opening of  $\Lambda_{LCA}$ 's root vertex. Next, alignment move 13 is processed, which is a visible model move on vertex  $v_{3,3}$  labeled  $e$ . Since alignment move 13 is not part of the optimal postfix alignment, i.e.,  $13 \notin I$ , we add  $e$  to  $\sigma'$ ; thus,  $\sigma' = \langle e \rangle$  after processing alignment move 13. All subsequent moves of  $\gamma$  are part of the optimal postfix alignment and thus, are processed identical to *extractSubTraces* (cf. Algorithm 5.4). Finally, the sequence  $\sigma' = \langle e, a, a, f \rangle$  is returned.

## 6.3. Evaluation

This section presents an evaluation of the proposed TFS-IPDA. The goal of the evaluation is to demonstrate that distinguishing trace fragments and complete traces may lead to better process models. Therefore, we compare the LCA-IPDA (cf. Section 5.3) with the TFS-IPDA as proposed in this chapter.

### 6.3.1. Experimental Setup

We compare TFS-IPDA with LCA-IPDA. We use publicly available real-life event logs: BPI Ch. 2020–Request for Payment [232], Road Traffic Fine Management [56], and Receipt Phase of an Environmental Permit Application Process [110]. Note that all event logs used do not distinguish complete traces and trace fragments; all traces are assumed to be complete. To obtain trace fragments, we applied the following steps.

1. We determine the total time span covered by the event log, i.e. the time from the earliest to the latest event. Next, we remove all cases having events located in the first or last 20% of the time span. By removing these cases, we aim to filter incomplete traces. Note that real-life event logs often have data quality issues and also contain trace fragments. We consider all remaining cases as complete.
2. We iterate over the remaining complete traces to change some of them into trace fragments, i.e., we artificially create trace fragments. With probability  $\frac{1}{2}$  we alter a complete trace into a trace fragment. With a unified likelihood of  $\frac{1}{3}$ , we apply one of the following changes. Let  $x = \max\{1, \lfloor 20\% \text{ average trace length} \rfloor\}$ .<sup>4</sup>
  - a) We remove the first  $x$  activities from the complete trace. As a result, we obtain a trace postfix.

<sup>4</sup>Note that we denote the rounding of a number  $y \in \mathbb{R}$  to the nearest integer by  $\lfloor y \rfloor \in \mathbb{N}$ .



- b) We remove the last  $x$  activities from the complete trace. As a result, we obtain a trace prefix.
- c) We remove the first  $x$  and last  $x$  activities from the complete trace. As a result, we obtain a trace infix.

In case the above approach yields empty trace fragments, we ignore them. We calculate fitness, precision, and the f-measure using the event log obtained after applying the first step described above. We discover a process tree from the 1% most frequent variants. This process tree serves as an initial model.

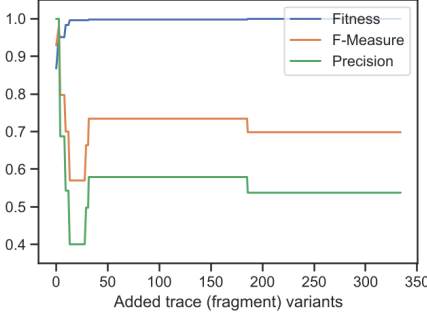
### 6.3.2. Results

Figure 6.9 reports the results for the three different event logs; per row we show the results of one event log. The plots on the left show the results for LCA-IPDA, while the plots on the right show the results for TFS-IPDA. Recall that LCA-IPDA considers all provided traces as complete traces. Overall, we observe that TFS-IPDA outperforms LCA-IPDA in many scenarios. Comparing the values for fitness, precision, and f-measure, most times TFS-IPDA scores higher than LCA-IPDA. Although these results were expected to a certain degree, the results clearly demonstrate how much of a difference it makes to distinguish between complete traces and trace fragments, i.e. LCA-IPDA compared to TFS-IPDA.

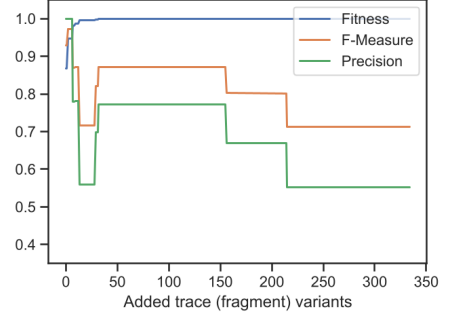
As mentioned in earlier chapters, the goal of incremental process discovery is generally only to include some behavior. Incorporating all behavior is often not desired since almost all real-life event logs are affected by data quality issues. Thus, only comparing the values upon adding the last trace (fragment) variant in Figure 6.9 is not decisive. We observe that TFS-IPDA scores slightly lower in later stages for the event logs BPI Ch. 2020 and the Receipt Phase of an Environmental Permit Application Process. For instance, consider the slightly higher f-measure in Figure 6.9c compared to Figure 6.9d after adding more than 100 trace (fragment) variants. Since many event logs exhibit a Pareto distribution [214], i.e., a few trace variants cover large parts of the overall observed process behavior, we find for all event logs that we quickly obtain process models supporting more than 90% of the observed behavior. Thus, after adding a few of the most frequent trace (fragment) variants, we reach high fitness values, cf. Figure 6.9. Due to this phenomenon, we see that the f-measure, i.e., the harmonic mean of fitness and precision, is mainly driven by precision. Thus, with some offset, the curves for f-measure are almost identical to those for precision the higher the fitness value is.

### 6.3.3. Discussion & Threats to Validity

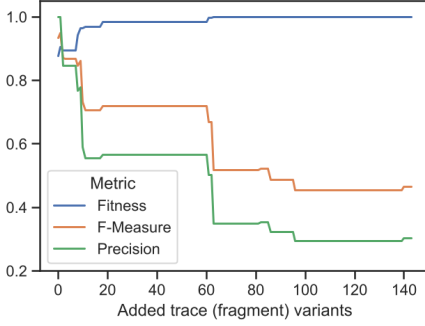
In the above presented results, we compute fitness, precision, and f-measure based on the event log that we obtain after applying the first step as described in Section 6.3.1. We do this because, to the best of our knowledge, there are currently no methods for computing fitness and precision based on alignments for trace fragments. However, we provide a different event log, the one obtained after applying the second step (cf. Section 6.3.1), to the two discovery approaches. Thus, we eventually evaluate the process models obtained



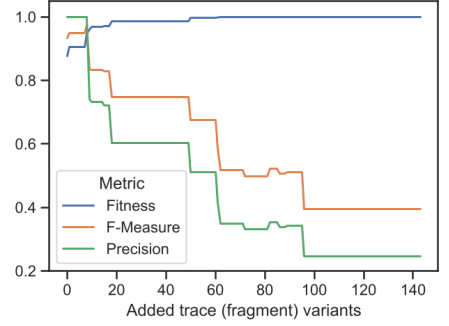
(a) Road Traffic Fine Management log—LCA-IPDA



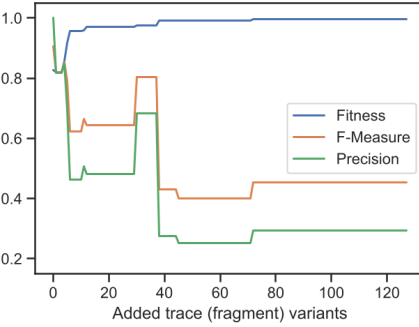
(b) Road Traffic Fine Management log—TFS-IPDA



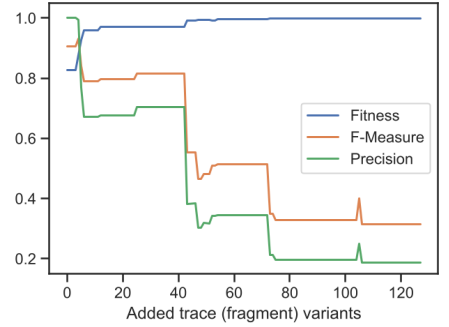
(c) BPI Ch. 2020-Request for Payment log—LCA-IPDA



(d) BPI Ch. 2020-Request for Payment log—TFS-IPDA



(e) Receipt Phase of an Environmental Permit Application Process—LCA-IPDA



(f) Receipt Phase of an Environmental Permit Application Process log—TFS-IPDA

Figure 6.9: Comparing LCA-IPDA (left column), which considers all provided traces as complete, with TFS-IPDA (right column), which distinguishes between complete traces and trace fragments (adapted from [185, Figure 8])

only with an event log consisting solely complete traces. However, since we are evaluating both algorithms this way, this is not of concern since both are treated the same.

Further, we make an initial effort to ensure the event log contains only complete traces, cf. the first step in Section 6.3.1. However, the approach taken does not guarantee that only complete traces remain. For instance, there might exist cases spanning the entire time range of the event log but represent actually trace fragments, i.e., before or after events occurred for these cases that are not captured in the log. Without utilizing domain knowledge about the underlying process, no automated technique can detect truly complete traces. In short, there might be actual trace fragments in the event log, which both techniques consider complete, although they are trace fragments. However, since both algorithms are equally affected,

## 6.4. Conclusion

This chapter extended the previously introduced incremental process discovery framework by trace fragments, cf. Figure 6.2. Trace fragments are a natural phenomenon in event logs and are often considered a data quality issue [35]. Therefore, many process mining techniques solely focus on complete traces. The extended IPD framework allows to utilize trace fragments along with complete traces. Core to this extended framework is a trace-fragment-supporting IPDA. We instantiated the framework by extending the previously proposed LCA-IPDA into a TFS-IPDA, cf. Section 6.2. We showed how large parts of the LCA-IPDA could be easily adapted to support trace fragments. Further, the proposed TFS-IPDA demonstrates how alignments for trace fragments—recall that we proposed infix and postfix alignments in Chapter 4—can be employed within IPD.

From the user's point of view, trace-fragment-supporting IPD offers new opportunities. For instance, it allows process experts to focus on specific process stages during the discovery by considering only trace fragments covering a specific process stage. Thus, trace-fragment-supporting IPD enhances the overall incremental discovery idea. While IPD generally facilitates the gradual incorporation of complete traces, trace-fragment-supporting IPD facilitates the gradual discovery of individual process stages. Especially in complex processes, where various domain experts from different process stages are required to comprehensively cover the entire process, the focus on trace fragments covering different phases can be advantageous for process discovery. The gradual discovery of process stages from corresponding trace fragments means that not all experts are always needed simultaneously. For instance, at the beginning, an initial model can describe the rough arrangement of the entire process, whose individual process stages are then discovered step by step in the process discovery phase.



---

## Chapter 7.

# Freezing Process Model Parts in Incremental Process Discovery

---

This chapter is largely based on the following published work.

- D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. *Freezing sub-models during incremental process discovery*. In A. Ghose, J. Horkoff, V. E. Silva Souza, J. Parsons, and J. Evermann, editors, *Conceptual Modeling, volume 13011 of Lecture Notes in Computer Science*, pages 14–24. Springer, 2021. doi:10.1007/978-3-030-89022-3\_2 [176]

So far, we focused within incremental process discovery on iteratively user-selected process behavior, i.e., complete traces (cf. Chapter 5) and trace fragments (cf. Chapter 6). Reconsider the simplistic IPD overview depicted in Figure 1.6 (page 14), which generally considers domain knowledge as a further input besides user-selected process behavior and the (initial) process model. Note that the incremental selection of the process behavior, i.e., complete traces or trace fragments, has so far been the essential form of interaction between algorithm and user.

This chapter extends the incremental process discovery framework introduced in Chapter 5, cf. Figure 5.1 (page 110), by allowing users to freeze parts within the process model provided as input. Freezing parts of the input process model restricts the IPDA to not alter these frozen parts. We refer to IPDAs supporting process model freezing as *freezing-enabled*. Thus, a freezing-enabled IPDA allows users to steer and restrict the process discovery phase because the options regarding potential process model modifications of the freezing-enabled IPDA are limited by the user.

Freezing subtrees within IPD allows users to influence the returned model. By freezing subtrees, the options of an IPDA to modify the provided process tree are restricted. Figure 7.1 provides an exemplary comparison between classic, non-freezing-enabled IPD and freezing-enabled IPD. Note that the examples depicted in Figure 7.1 are general and that the actual change in the process trees depends on the specific instantiation of the IPDAs in use. Figure 7.1a lists the previously added traces  $A$  and the trace to be added next  $\sigma_{next}$ .

Figure 7.1b shows the process tree  $\Lambda_1$  that is provided to a classic, non-freezing-enabled IPDA along with  $A$  and  $\sigma_{next}$ . The output tree  $\Lambda_2$  can be seen in Figure 7.1c. Comparing

$\Lambda_1$  and  $\Lambda_2$ , we observe that the choice operator was changed into a loop operator, cf.  $v_{3.2}$  in  $\Lambda_1$  (Figure 7.1b) and  $\Lambda_2$  (Figure 7.1c). Thus, tree  $\Lambda_2$  allows to execute activity  $d$  infinite times. Further, the order of the vertex labeled  $d$  has been changed with the  $\tau$  labeled vertex, cf. vertices  $v_{4.1}$  and  $v_{4.2}$  in Figures 7.1b and 7.1c. Thus, tree  $\Lambda_2$  supports  $\sigma_{next}$  and the previously added traces contained in  $A$ .

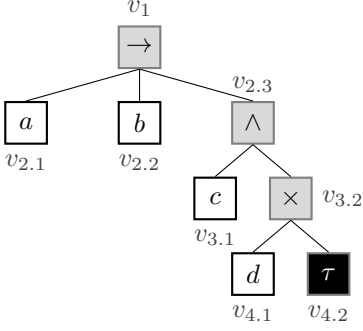
Figure 7.1d shows the process tree  $\Lambda_1$ , identical to Figure 7.1b, with one frozen subtree rooted at vertex  $v_{2.3}$  that is provided to a freezing-enabled IPDA along with  $A$  and the trace to be added next  $\sigma_{next}$ . The output tree  $\Lambda'_2$  of a freezing-enabled IPDA is shown in Figure 7.1e. As highlighted, the frozen subtree  $\Lambda_{i_1} \hat{=}_\Sigma \wedge (c, \times (\tau, d))$  is contained in  $\Lambda'_2$ . Since the freezing-enabled IPDA is not allowed to modify the frozen subtree, a new subtree was added before the frozen one in  $\Lambda'_2$ , cf. the subtree rooted at vertex  $v_{2.3}$ . This subtree allows the execution of activity  $d$  optionally. Thus, tree  $\Lambda'_2$  supports  $\sigma_{next}$  and previously added traces contained in  $A$ . Further, tree  $\Lambda'_2$  supports the execution of activity  $d$  at most twice.

Comparing the two obtained trees  $\Lambda_2$  and  $\Lambda'_2$ , we clearly see that freezing subtrees can influence the resulting process tree. Tree  $\Lambda'_2$  is more precise than  $\Lambda_2$  with respect to the traces  $A \cup \{\sigma_{next}\}$ . Tree  $\Lambda_2$  supports, for instance, traces like  $\langle a, b, d, c, d \rangle$ ,  $\langle a, b, c, d, d \rangle$ , and  $\langle a, b, d, d, d, c \rangle$  that are all *not* part of the added traces, i.e.,  $A \cup \{\sigma_{next}\}$ , cf. Figure 7.1a. Further,  $\Lambda'_2$  does not contain a loop operator. On the contrary,  $\Lambda'_2$  is more complex regarding the number of elements, i.e., vertices and edges, than tree  $\Lambda_2$ . Further note that process tree  $\Lambda'_2$  contains duplicate labels, cf.  $v_{3.2}$  and  $v_{4.2}$  (Figure 7.1e). In short, this simple example shows that the option to freeze subtrees within IPD enables users to steer and influence the process discovery phase positively.

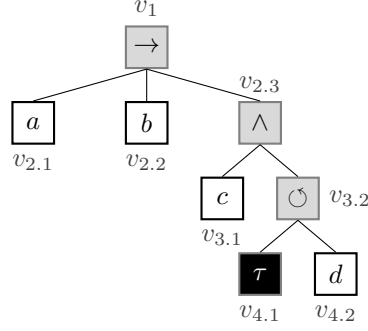
The remainder of this chapter is organized as follows. Section 7.1 introduces the extended framework that embeds a freezing-enabled IPDA. Subsequently, Section 7.2 presents a baseline freezing-enabled IPDA, instantiating the extended framework. Further, Section 7.3 presents the freezing-enabled LCA-IPDA, which is the essential contribution of this chapter. Section 7.4 presents an evaluation of the proposed freezing-enabled IPDAs. Subsequently, Section 7.5 discusses an illustrative example to showcase the advantage of freezing submodels during IPD. Finally, Section 7.6 concludes this chapter.

$$A = \{\langle a, b, c \rangle, \langle a, b, c, d \rangle, \langle a, b, d, c \rangle\} \quad \sigma_{next} = \langle a, b, d, d, c \rangle$$

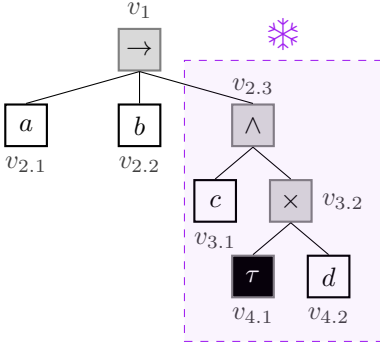
(a) Input: previously added traces  $A$  and trace to be added next  $\sigma_{next}$



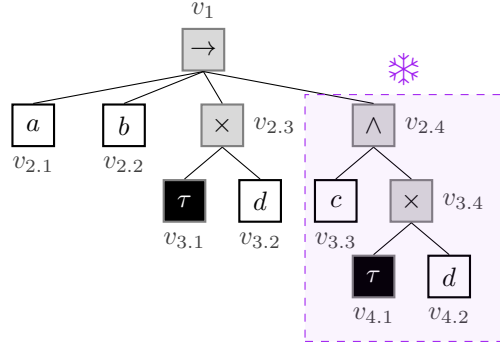
(b) Input process tree  $\Lambda_1$  to a non-freezing-enabled IPDA



(c) Output process tree  $\Lambda_2$  of a non-freezing-enabled IPDA



(d) Input process tree  $\Lambda_1$  to a freezing-enabled IPDA; highlighted subtree is considered frozen



(e) Output process tree  $\Lambda'_2$  of a freezing-enabled IPDA; highlighted subtree is considered frozen

Figure 7.1: Exemplary comparison of classic, non-freezing-enabled IPD (as introduced in Chapter 5), shown in Figures 7.1b and 7.1c, to freezing-enabled IPD (as proposed in this chapter), shown in Figures 7.1d and 7.1e

### 7.1. Extended IPD Framework

This section formally introduces the extended framework, i.e., the freezing-enabled incremental process discovery framework. In the interest of clarity, we focus only on complete traces in this chapter. However, it should be noted that the presented framework and the concrete instantiations can easily be adapted to support trace fragments. Therefore, in the following, we will only occasionally give brief insights into what would need to be adjusted accordingly in the case of trace fragments. In the remainder of this section, we will introduce and formally define freezing-enabled IPD.

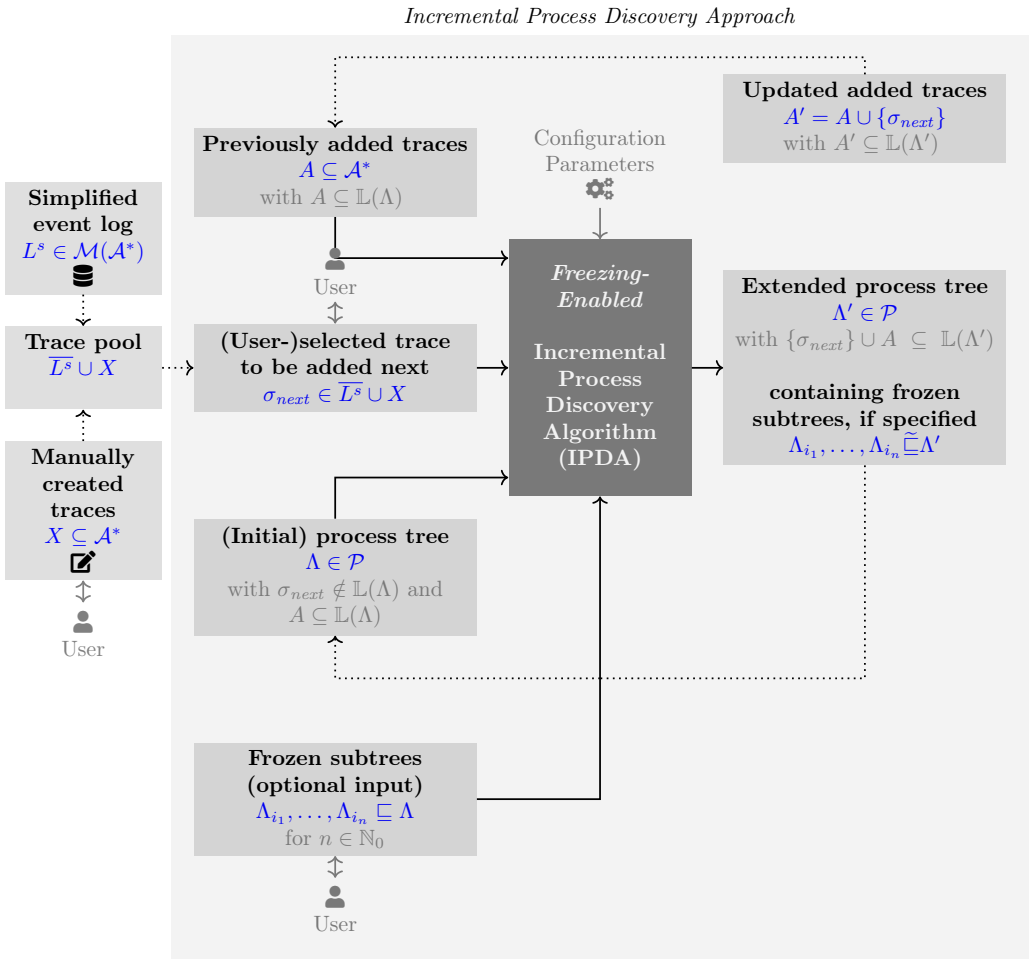


Figure 7.2: Input-output perspective of the proposed incremental process discovery framework extended to support submodel freezing

Figure 7.2 depicts the extended IPD framework that allows users to freeze submodels.



Compared to the framework shown in Figure 5.1 (page 110), the extended framework shown in Figure 7.2 allows users to optionally specify subtrees  $\Lambda_{i_1}, \dots, \Lambda_{i_n} \subseteq \Lambda$  of the provided input process tree that should be frozen. Further, the IPDA is replaced by a freezing-enabled IPDA. Finally, the output process tree  $\Lambda'$  is guaranteed to contain the frozen subtrees, i.e.,  $\Lambda_{i_1}, \dots, \Lambda_{i_n} \subseteq \Lambda'$  besides the guarantee that the previously added traces and the trace to be added next are supported, i.e.,  $\{\sigma_{next}\} \cup A \subseteq \mathbb{L}(\Lambda')$ . Subsequently, we define a freezing-enabled IPDA.

**Definition 7.1** (Freezing-Enabled Incremental Process Discovery Algorithm)

The function

$$feIpda : \mathcal{P} \times \mathbb{P}(\mathcal{P}) \times \mathcal{A}^* \times \mathbb{P}(\mathcal{A}^*) \rightarrow \mathcal{P}$$

is a freezing-enabled IPDA if for any

- (initial) tree  $\Lambda \in \mathcal{P}$ ,
- frozen subtrees  $\{\Lambda_{i_1}, \dots, \Lambda_{i_n}\} \in \mathbb{P}(\mathcal{P})$  with
  - $n \in \mathbb{N}_0$
  - $\Lambda_{i_1}, \dots, \Lambda_{i_n} \subseteq \Lambda$
  - $\forall 1 \leq j < k \leq n \left( \Lambda_{i_j} \not\subseteq \Lambda_{i_k} \wedge \Lambda_{i_k} \not\subseteq \Lambda_{i_j} \right)$
- trace  $\sigma_{next} \in \mathcal{A}^*$ , and
- previously added traces  $A \in \mathbb{P}(\mathcal{A}^*)$  with  $A \subseteq \mathbb{L}(\Lambda)$

it holds that

- $A \cup \{\sigma_{next}\} \subseteq \mathbb{L}(feIpda(\Lambda, \{\Lambda_{i_1}, \dots, \Lambda_{i_n}\}, \sigma_{next}, A))$  and
- $\Lambda_{i_1}, \dots, \Lambda_{i_n} \subseteq feIpda(\Lambda, \{\Lambda_{i_1}, \dots, \Lambda_{i_n}\}, \sigma_{next}, A)$

If  $A \not\subseteq \mathbb{L}(\Lambda)$ ,  $\exists j, k \in \{1, \dots, n\} (j \neq k \wedge \Lambda_{i_j} \subseteq \Lambda_{i_k})$ , or  $\Lambda_{i_1}, \dots, \Lambda_{i_n} \not\subseteq \Lambda$ , function  $feIpda$  is undefined.

## 7.2. Naive Freezing-Enabled IPDA

This section presents a naive freezing-enabled IPDA. The core idea of this naive approach is to simply apply a non-freezing-enabled IPDA, for instance, the LCA-IPDA proposed in Section 5.3.2. If the resulting process tree misses frozen subtrees, these missing frozen subtrees are reinserted into the output process tree. During reinsertion, no further semantic analysis of the output process tree is performed. Further, the reinserted frozen subtrees are inserted in such a way that they merely extend the existing language of the output process tree, which is returned by the non-freezing-enabled IPDA.

Algorithm 7.1 presents a naive freezing-enabled IPDA according to Definition 7.1. First, a non-freezing-enabled IPDA is applied (line 1). Next, we check in the returned process tree  $\Lambda'$  if all frozen subtrees specified in  $T_{frozen}$  are contained. Any frozen subtree that is not contained in  $\Lambda'$  is added to the set  $T_{missing}$  (line 2). In case  $T_{missing}$  is empty (line 3), all frozen subtrees are contained in  $\Lambda'$ ; further, since function  $ipda$  guarantees that the returned tree  $\Lambda'$  supports  $\sigma_{next}$  and previously added traces  $A$ , we return  $\Lambda'$  (line 4). Otherwise, i.e., some frozen subtrees are not contained in  $\Lambda'$  (line 5), we add

**Algorithm 7.1:** Naive freezing-enabled IPDA

---

**Input:**  $A \subseteq \mathcal{A}^*$ , // previously added traces  
 $\sigma_{next} \in \mathcal{A}^*$ , // trace to be added next  
 $\Lambda \in \mathcal{T}$ , // (initial) process tree  
 $T_{frozen} = \{\Lambda_{i_1}, \dots, \Lambda_{i_n}\}$  with  $\Lambda_{i_1}, \dots, \Lambda_{i_n} \subseteq \Lambda$  (for  $n \geq 0$ ) // optional (i.e.,  $n \geq 0$ ) frozen subtrees of  $\Lambda$   
**Output:**  $\Lambda' \in \mathcal{T}$  // with (1)  $A \cup \{\sigma_{next}\} \subseteq \mathbb{L}(\Lambda')$  and (2)  $\Lambda_1, \dots, \Lambda_n \subseteq \Lambda'$

**begin**

```

1   $\Lambda' \leftarrow ipda(\Lambda, \sigma_{next}, A)$  // apply a non-freezing-enabled IPDA, cf. Definition 5.1
2   $T_{missing} \leftarrow \{\Lambda_{frozen} \mid \Lambda_{frozen} \in T_{frozen} \wedge \Lambda_{frozen} \not\subseteq \Lambda'\}$ 
3  if  $T_{missing} = \emptyset$  then
4    return  $\Lambda'$  //  $\Lambda'$  contains all frozen subtrees  $T_{frozen}$ 
5  else
6    return  $\wedge \left( \Lambda', \times(\Lambda_{i_1}, \tau), \dots, \times(\Lambda_{i_j}, \tau) \right)$  for  $\{\Lambda_{i_1}, \dots, \Lambda_{i_j}\} = T_{missing}$ 
    // extend  $\Lambda'$  such that all frozen subtrees that are missing in  $\Lambda'$  are optionally executable in parallel to  $\Lambda'$ 

```

---

all missing frozen subtrees  $T_{missing}$  as optional subtrees in parallel to  $\Lambda'$  and return (page 182). Adding the missing frozen subtrees as optional subtrees in parallel to  $\Lambda'$  only extends the language of the returned tree; thus, all traces supported by  $\Lambda'$  are also supported by the tree returned in line 6. Figure 7.3 visualizes the returned subtree that contains the missing frozen subtree as optional subtrees in parallel to  $\Lambda'$ .

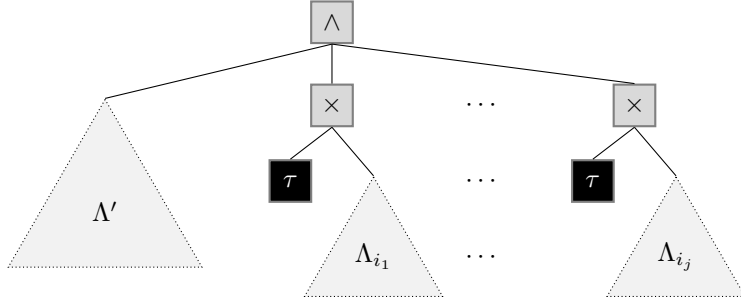


Figure 7.3: Adding missing frozen subtrees  $\Lambda_{i_1}, \dots, \Lambda_{i_j}$  as optional subtrees in parallel to  $\Lambda'$ , cf. Algorithm 7.1 line 6

Subsequently, we present an example of the above presented naive freezing-enabled IPDA specified in Algorithm 7.1. Reconsider the input process tree  $\Lambda_1$  with the frozen subtree  $\Delta_{\Lambda_1}(v_{2.3})$ , depicted in Figure 7.1d (page 179). Further, consider the previously added traces  $A$  and the trace to be added next  $\sigma_{next}$  as specified in Figure 7.1a (page 179).

When applying Algorithm 7.1 to these inputs, we could obtain in line 1 the tree  $\Lambda_2$  depicted in Figure 7.1c (page 179).<sup>1</sup> Note that  $\Lambda_2$  does not contain the frozen subtree. Thus, after executing line 2, the frozen subtree highlighted in Figure 7.1d, i.e.,  $\Delta_{\Lambda_1}(v_{2.3})$ , is contained in  $T_{missing}$ . Since  $T_{missing}$  is not empty, we extend the tree  $\Lambda_2$ , which we obtained by applying *ipda* in line 1, as illustrated in Figure 7.3. Figure 7.4 illustrates the resulting tree that is returned by Algorithm 7.1 (line 6).

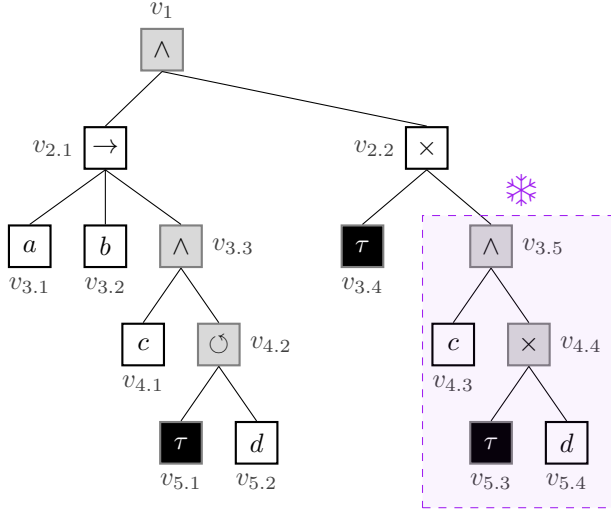


Figure 7.4: Example output tree returned by Algorithm 7.1 for the input tree  $\Lambda_1$  with frozen subtree  $\Delta_{\Lambda_1}(v_{2.3})$ , trace to be added next  $\sigma_{next}$  (cf. Figure 7.1a), and previously added traces  $A$  (cf. Figure 7.1a); the highlighted subtree corresponds to the frozen subtree

The example tree shown in Figure 7.4 illustrates that the naive freezing-enabled IPDA has significant limitations. Comparing the tree resulting from the naive approach, shown in Figure 7.4, with the tree  $\Lambda_2$ , shown in Figure 7.1c, indicates that the naive approach returns a less precise and more complex process tree—complexity in this case refers to the size of the process tree. Again, both compared trees depend on certain algorithmic instantiations, so both trees can only be seen as one example out of many. Tree  $\Lambda_2$  depends on the used freezing-enabled IPDA used. Similarly, the tree shown in Figure 7.4 depends on the employed instantiation of the *ipda* function (cf. Algorithm 7.1 line 1), i.e., a non-freezing-enabled IPDA.

In conclusion, the proposed naive freezing-enabled IPDA (Algorithm 7.1) demonstrates that given a non-freezing-enabled IPDA, one can easily create a freezing-enabled IPDA that conforms to Definition 7.1; recall that the function call of *ipda* is central in Algorithm 7.1. However, the example process tree that is returned by the naive freezing-enabled IPDA—the exact process tree depends on the concrete instantiation of function *ipda* in Algorithm 7.1 line 1—shows that the naive approach has clear limitations, i.e., the

<sup>1</sup>As mentioned earlier, the output depends on the specific instantiation of the *ipda* function.

process tree may become large and imprecise compared to potential other process trees. Therefore, in the following, we present the freezing-enabled LCA-IPDA that devotes more significant effort to obtaining frozen subtrees rather than simply looking downstream to see if frozen trees have been lost, as the proposed naive approach does.

### 7.3. Freezing-Enabled LCA-IPDA

This section introduces the freezing-enabled LCA-IPDA compared to the naive approach shown before. Also, the advanced approach embeds a given non-freezing-enabled IPDA similar to the naive approach presented in Section 7.2. However, instead of simply applying the non-freezing-enabled IPDA to the given process tree and reinserting missing frozen subtrees after the incremental process discovery phase into the resulting process tree, the advanced approach pursues another strategy. In essence, the advanced approach modifies the inputs provided to the non-freezing-enabled IPDA as well as the output process tree.

#### 7.3.1. Overview

This section provides a detailed overview of the proposed freezing-enabled LCA-IPDA. Consider Figure 7.5 providing an overview of the proposed approach and its various components. Central to the freezing-enabled LCA-IPDA is a *non-freezing-enabled* IPDA. Further, the freezing-enabled LCA-IPDA comprises four components that modify the inputs provided to the non-freezing-enabled IPDA and the output process tree returned by the non-freezing-enabled IPDA. These four components are highlighted as black numbered boxes in Figure 7.5. The four components, the modified inputs (i.e.,  $\tilde{A}$ ,  $\tilde{\sigma}_{next}$ , and  $\tilde{\Lambda}$ ), the non-freezing-enabled IPDA, and the output process tree  $\tilde{\Lambda}'$  returned by the IPDA constitute the proposed freezing-enabled LCA-IPDA.

Subsequently, we outline the four components that are central to the freezing-enabled LCA-IPDA, cf. Figure 7.5.

- Component (1) replaces the frozen subtrees  $\Lambda_{i_1}, \dots, \Lambda_{i_n} \subseteq \Lambda$  in the input process tree  $\Lambda$  by a single leaf vertex with a unique label each. We refer to the modified input process tree as  $\tilde{\Lambda}$ . Section 7.3.2 describes component (1) in detail.
- Component (2) replaces any full occurrences of the frozen subtrees in  $\sigma_{next}$  with the unique label already used in component (1). We refer to the modified trace to be added next as  $\tilde{\sigma}_{next}$ . Note that  $\tilde{\sigma}_{next} \in \mathbb{L}(\tilde{\Lambda})$  iff  $\sigma_{next} \in \mathbb{L}(\Lambda)$ . Section 7.3.3 describes component (2) in detail.
- Component (3) is similar to component (2) and is applied to the traces that have already been added, i.e.,  $A$ . Executions of frozen subtrees in the previously added traces are detected and replaced by the corresponding unique label. We refer to the modified traces as  $\tilde{A}$ . Note that  $A \subseteq \mathbb{L}(\Lambda)$  and  $\tilde{A} \subseteq \mathbb{L}(\tilde{\Lambda})$ .
- Component (4) reinserts the frozen subtrees replaced by component (1) back into  $\tilde{\Lambda}'$ . We refer to the resulting process tree as  $\Lambda'$  with  $\{\sigma_{next}\} \cup A \in \mathbb{L}(\Lambda')$ . Section 7.3.5 describes component (4) in detail.

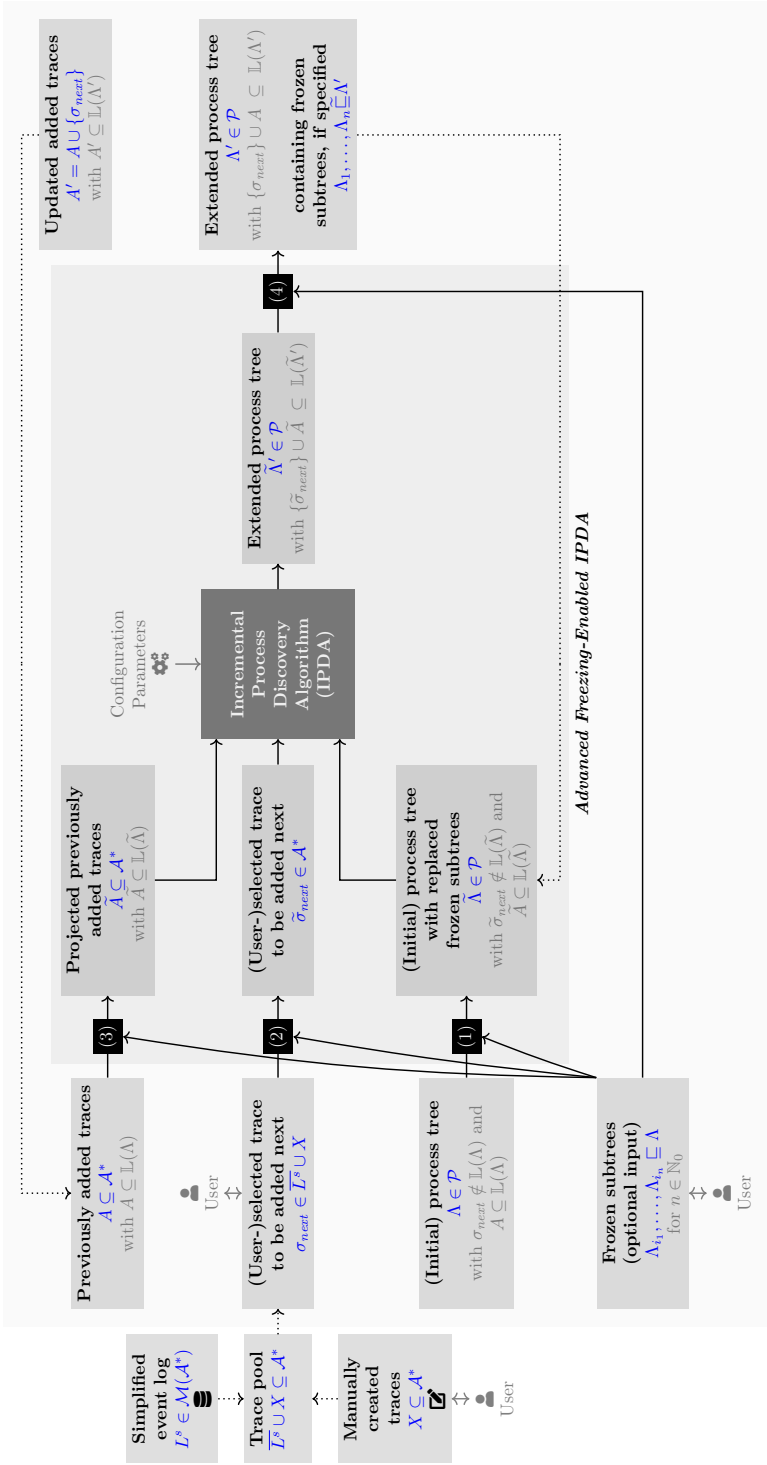


Figure 7.5: Overview of the advanced freezing-enabled IPDA, which consists of four central components modifying the input- s/outputs of a non-freezing-enabled IPDA; (1) replaces the frozen subtrees in  $\Lambda$ , (2) projects the trace to be added next  $\sigma_{next}$ , (3) projects the previously added traces, and (4) reinserts the frozen subtrees in  $\tilde{\Lambda}'$

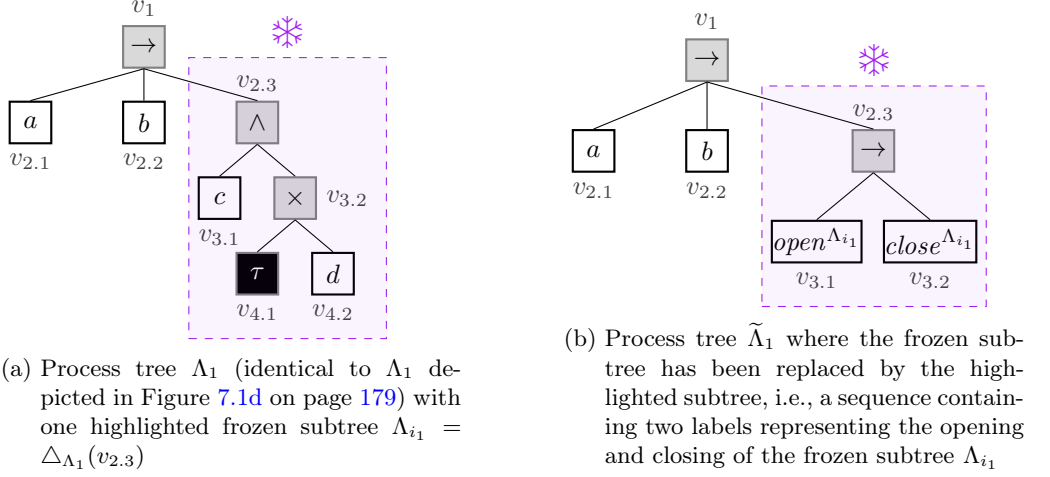


Figure 7.6: Example of applying component (1) to the process tree  $\Lambda_1$  with highlighted frozen subtree

In brief, the freezing-enabled LCA-IPDA employs a non-freezing-enabled IPDA. Therefore, the advanced approach modifies the inputs and output of the non-freezing-enabled IPDA. In total, four components are comprised within the advanced approach. The following sections provide a detailed introduction to these four components.

### 7.3.2. Component (1)—Replacing Frozen Subtrees

This section introduces component (1), which modifies the provided (initial) process tree  $\Lambda$ , cf. Figure 7.5. In brief, component (1) replaces each frozen subtree  $\Lambda_{i_1}, \dots, \Lambda_{i_n} \sqsubseteq \Lambda$  in tree  $\Lambda$  by two unique labels, one representing the opening and one the closing of the frozen subtree. Note that the replacement labels are not present in  $\Lambda$  nor the trace pool  $L \cup L'$ .

For example, consider  $\Lambda_1$  with one frozen subtree  $\Lambda_{i_1} = \triangle_{\Lambda_1}(v_{2.3})$  depicted in Figure 7.6a. Component (1) replaces frozen subtree  $\Lambda_{i_1}$  by the subtree  $\rightarrow (open^{\Lambda_{i_1}}, close^{\Lambda_{i_1}})$ . Figure 7.6b shows the resulting process tree  $\tilde{\Lambda}_1$ . Note that we assume that the replacement labels  $open^{\Lambda_{i_1}}$  and  $close^{\Lambda_{i_1}}$  are not included in any trace to be added in the future. The two labels  $open^{\Lambda_{i_1}}$  and  $close^{\Lambda_{i_1}}$  allow us to track when a frozen subtree is opened and when it is closed again.

In the general case, we iteratively replace each frozen subtree  $\Lambda_{i_j} \in \{\Lambda_{i_1}, \dots, \Lambda_{i_n}\}$  for  $n \in \mathbb{N}_0$  by a corresponding subtree  $\rightarrow (open^{\Lambda_{i_j}}, close^{\Lambda_{i_j}})$ . Recall that we require that all frozen subtrees  $\Lambda_{i_1}, \dots, \Lambda_{i_n}$  do not overlap, i.e., they do not contain each other, cf. Definition 7.1 on page 181. Hence, the order in which we replace the frozen subtrees is irrelevant, as all frozen subtrees can be replaced accordingly by component (1).

### 7.3.3. Component (2)—Projecting Trace to be Added Next

This section introduces component (2), which modifies the trace to be added next  $\sigma_{next}$ , cf. Figure 7.5. Component (2) modifies the input trace  $\sigma_{next}$  such that occurrences of the frozen subtrees are replaced by the corresponding labels, as already used by component (1). In the following, component (2) is introduced using a running example. The subsequent explanation details the functionality of component (2) in the general case.

Recall tree  $\Lambda_1$  with one frozen subtree  $\Lambda_{i_1} = \triangle_{\Lambda_1}(v_{2.3})$ , cf. Figure 7.6a (page 186). Component (1) altered  $\Lambda_1$  into  $\tilde{A}$ , cf. Figure 7.6b. Further, recall  $\sigma_{next} = \langle a, b, d, d, d, c \rangle$  (cf. Figure 7.1a). The goal of component (2) is to detect potential full executions of the frozen subtree and replace these full executions accordingly in  $\sigma_{next}$ . To this end, we compute an optimal alignment between  $\sigma_{next}$  and an *abstraction tree* that allows the replay of the frozen subtree arbitrarily many times. Consider Figure 7.7 showing the corresponding abstraction tree. The loop operator (cf.  $v_1$ ) allows to replay the frozen subtree, which is highlighted in Figure 7.7, any number of times.

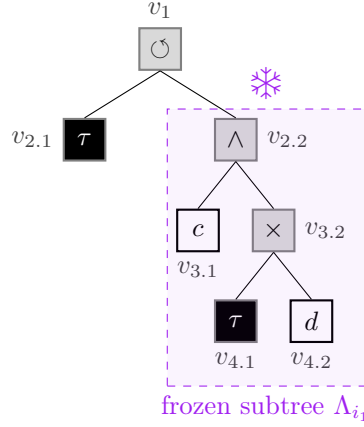


Figure 7.7: Abstraction tree  $\Lambda_1^{abstraction}$  for the frozen subtree  $\Lambda_{i_1} \hat{=}_{\Sigma} \wedge (c, \times (\tau, d))$  of tree  $\Lambda_1$  (cf. Figure 7.6a)

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
$a$	$b$	$\gg$	$\gg$	$\gg$	$\gg$	$d$	$\gg$	$d$	$c$	$\gg$	$\gg$
$\gg$	$\gg$	$(v_1, open)$	$(v_{2.1}, \tau)$	$(v_{2.2}, open)$	$(v_{3.2}, open)$	$(v_{4.2}, d)$	$(v_{3.2}, close)$	$\gg$	$(v_{3.1}, c)$	$(v_{2.2}, close)$	$(v_1, close)$

Figure 7.8: Optimal alignment for trace to be added next  $\sigma_{next} = \langle a, b, d, d, d, c \rangle$  and abstraction tree  $\Lambda_1^{abstraction}$  depicted in Figure 7.7

Figure 7.8 illustrates an optimal alignment for the abstraction tree shown in Figure 7.7 and trace to be added next  $\sigma_{next}$ . The alignment contains a complete execution of the frozen subtree, cf. alignment moves 5–8, 10, and 11. In alignment move 5 the frozen

$$\tilde{\sigma}_{next} = \langle \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1. & 2. & 3. & 4. & 5. & 6. & 7. & 8. & 9. & 10. & 11. & 12. \\ \hline a, & b, & & & open^{\Lambda_{i_1}}, & & & & d, & & close^{\Lambda_{i_1}} & \\ \hline \end{array} \rangle$$

Figure 7.9: Projected trace to be added next  $\tilde{\sigma}_{next} = \langle a, b, open^{\Lambda_{i_1}}, d, close^{\Lambda_{i_1}} \rangle$  derived from the optimal alignment depicted in Figure 7.8

subtree is opened and in move 11 it is closed again. In between there exists no visible model move on a vertex that belongs to the frozen subtree. Thus, we replace all labels in  $\sigma_{next}$  that are replayed in a full execution of the frozen subtree, i.e., the first  $d$  and the  $c$  in  $\sigma_{next}$ . Figure 7.9 shows how we derive the output trace  $\tilde{\sigma}_{next}$  from the computed alignment shown in Figure 7.8. The first two moves represent log moves; thus, we adopt the activity labels to  $\tilde{\sigma}_{next}$ . Recall that the alignment shown in Figure 7.8 contains a full execution of the frozen subtree  $\Lambda_{i_1}$ , which is rooted at vertex  $v_{2.2}$  (in the abstraction tree, cf. Figure 7.7). The frozen subtree is opened in the 5<sup>th</sup> alignment move; thus, we add the corresponding replacement label  $open^{\Lambda_{i_1}}$  to  $\tilde{\sigma}_{next}$ . Next, we ignore alignment move 7 since it is part of a full execution of the frozen subtree  $\Lambda_{i_1}$ . However, the 9<sup>th</sup> alignment move is a log move on  $d$ ; thus, we add the  $d$  to  $\tilde{\sigma}_{next}$ . The 10<sup>th</sup> alignment move is again a synchronous move within the frozen subtree and is therefore ignored. Finally, the 11<sup>th</sup> alignment move closes the full execution of the frozen subtree. Below, we summarize the input trace  $\sigma_{next}$  and the output trace  $\tilde{\sigma}_{next}$  of component (2).

$$\begin{aligned} \sigma_{next} &= \langle a, b, d, d, c \rangle \\ \tilde{\sigma}_{next} &= \langle a, b, open^{\Lambda_{i_1}}, d, close^{\Lambda_{i_1}} \rangle \end{aligned}$$

In the general case, i.e.,  $n \in \mathbb{N}_0$  frozen subtrees  $\Lambda_{i_1}, \dots, \Lambda_{i_n}$  are provided, we generate one abstraction tree containing all frozen subtrees. Figure 7.10 depicts the abstraction tree for the general case. This abstraction tree allows the replay of any frozen subtree at any time arbitrarily often. As in the example presented above, the trace to be added next  $\sigma_{next}$  is aligned with the abstraction tree. If a frozen subtree is fully replayed in an alignment, we remove the replayed activity labels and add instead the opening and closing of the frozen subtree, cf. the example shown in Figures 7.8 and 7.9. We refer to a frozen subtree as fully replayed within an alignment if, between an opening and closing of that frozen subtree, no visible model moves on leaf vertices belonging to that frozen subtree exist. Note that there can be multiple openings and closings of the same frozen subtree within an alignment; thus, both full and partial executions of the same frozen subtree can occur in an alignment.

In the following, we formally present the extraction of the projected trace to be added next  $\tilde{\sigma}_{next}$ . To this end, we specify the function *belongsToFullExecution* in Definition 7.2. Given an alignment  $\gamma \in \Gamma^{opt}(\Lambda^{abstraction}, \sigma_{next})$  for the trace to be added next  $\sigma_{next}$  and the abstraction tree  $\Lambda^{abstraction}$ , an alignment move index  $i \in \{1, \dots, |\gamma|\}$ , and the set of frozen subtrees  $T_{frozen}$ , the function returns true if the alignment move  $\gamma(i)$  is part of a frozen subtree, i.e., either a model move or synchronous move on a leaf vertex belonging to the frozen subtree, and the current execution of the frozen subtree is a full one, i.e., no visible model moves occur. In other words, the current alignment move is part of a deviation-free execution of a frozen subtree.



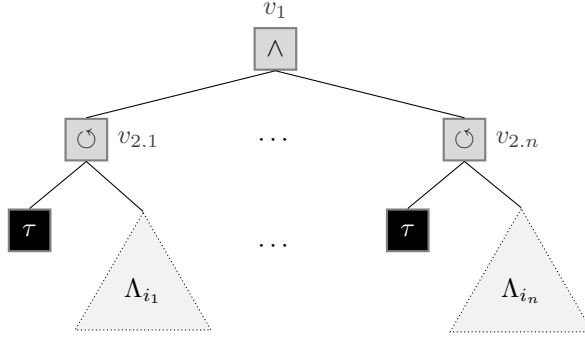


Figure 7.10: General structure of the abstraction tree  $\Lambda^{abstraction}$  for  $n \in \mathbb{N}_0$  frozen subtrees  $\Lambda_{i_1}, \dots, \Lambda_{i_n} \subseteq \Lambda$ ; each frozen subtree in  $\Lambda^{abstraction}$  can be replayed any number of times and in parallel with other frozen subtrees

**Definition 7.2** (Function *belongsToFullExecution*)

The function *belongsToFullExecution* :  $\Gamma^{opt} \times \mathbb{N} \times \mathbb{P}(\mathcal{P}) \rightarrow \mathbb{B}$  takes

- an optimal alignment  $\gamma \in \Gamma^{opt}(\Lambda^{abstraction}, \sigma_{next})$ ,
- an alignment move index  $i \in \{1, \dots, |\gamma|\} \subseteq \mathbb{N}$ , and
- a set of  $n$  frozen subtrees  $T_{frozen} = \left\{ \Lambda_{i_1} = (V_{i_1}, E_{i_1}, \Sigma_{i_1}, \lambda_{i_1}, r_{i_1}, <_{i_1}), \dots, \Lambda_{i_n} = (V_{i_n}, E_{i_n}, \Sigma_{i_n}, \lambda_{i_n}, r_{i_n}, <_{i_n}) \right\} \in \mathbb{P}(\mathcal{P})$  of tree  $\Lambda$

and returns *true* if the alignment move  $\gamma(i)$  is part of a full execution of a frozen subtree, i.e., no visible model move on a leaf vertex of the frozen subtree exists between opening and closing. Below, we formally specify the function.

$$belongsToFullExecution(\gamma, i, T_{frozen}) = true \quad \text{iff}$$

$$\begin{aligned} \exists 1 \leq m \leq n \bigg( & modelVertex(\gamma(i)) \in V_{i_m} \wedge \\ & \exists 1 \leq j \leq i \leq k \leq |\gamma| \bigg( modelVertex(\gamma(j)) = r_{i_m} \wedge \\ & modelLabel(\gamma(j)) = open \wedge modelVertex(\gamma(k)) = r_{i_m} \wedge \\ & modelLabel(\gamma(k)) = close \wedge \forall l \in \{j, \dots, k\} \bigg( \\ & visModelMv(\gamma(l)) \Rightarrow modelVertex(\gamma(l)) \notin V_{i_m} \bigg) \bigg) \end{aligned}$$

**Algorithm 7.2:**  $project^{\sigma_{next}}$ 


---

**Input:**  $\gamma \in \Gamma^{opt}(\Lambda^{abstraction}, \sigma_{next})$ , // optimal alignment for the trace to be added  
 $next \ \sigma_{next} \in \mathcal{A}^*$  and abstraction process tree  $\Lambda^{abstraction} \in \mathcal{P}$   
 $T_{frozen} = \{ \Lambda_{i_1} = (V_{i_1}, E_{i_1}, \Sigma_{i_1}, \lambda_{i_1}, r_{i_1}, <_{i_1}), \dots, \Lambda_{i_n} = (V_{i_n}, E_{i_n}, \Sigma_{i_n}, \lambda_{i_n}, r_{i_n}, <_{i_n}) \}$   
with  $\Lambda_{i_1}, \dots, \Lambda_{i_n} \subseteq \Lambda$  //  $n \geq 0$  frozen subtrees of  $\Lambda$

**Output:**  $\tilde{\sigma}_{next} \in \mathcal{A}^*$  // projected trace to be added next

**begin**

```

1   $\tilde{\sigma} \leftarrow \langle \rangle$ 
2  for  $1 \leq j \leq |\gamma|$  do // iterate over alignment moves
3    if  $syncMv(\gamma(j)) \wedge (modelVertex(\gamma(j)) \notin V_{i_1} \cup \dots \cup V_{i_n} \vee$ 
       $\neg belongsToFullExecution(\gamma, i, T_{frozen}))$  then // add label from sync. move
      if the sync. move does not involve a vertex from a frozen subtree or is
      part of a not fully executed frozen subtree
4       $\tilde{\sigma} \leftarrow \tilde{\sigma} \circ \langle modelLabel(\gamma(j)) \rangle$ 
5    if  $invModelMv(\gamma(j)) \wedge \exists 1 \leq k \leq n (modelVertex(\gamma(j)) = r_{i_k}) \wedge$ 
       $modelLabel(\gamma(j)) = open \wedge belongsToFullExecution(\gamma, i, T_{frozen})$  then
      // frozen subtree  $\Lambda_{i_k}$  is opened and is fully executed in subsequent moves
       $\Rightarrow$  add corresponding opening label to  $\tilde{\sigma}$ 
6       $\tilde{\sigma} \leftarrow \tilde{\sigma} \circ \langle open^{\Lambda_{i_k}} \rangle$ 
7    if  $invModelMv(\gamma(j)) \wedge \exists 1 \leq k \leq n (modelVertex(\gamma(j)) = r_{i_k}) \wedge$ 
       $modelLabel(\gamma(j)) = close$  then // frozen subtree  $\Lambda_{i_k}$  is closed and was
      fully executed  $\Rightarrow$  add corresponding closing label to  $\tilde{\sigma}$ 
8       $\tilde{\sigma} \leftarrow \tilde{\sigma} \circ \langle close^{\Lambda_{i_k}} \rangle$ 
9    if  $logMv(\gamma(i))$  then
10      $\tilde{\sigma} \leftarrow \tilde{\sigma} \circ \langle traceLabel(\gamma(i)) \rangle$ 
11 return  $\tilde{\sigma}$ 

```

---

Algorithm 7.2 specifies the projection function  $project^{\sigma_{next}}$  that extracts projected trace  $\tilde{\sigma}_{next}$  from a provided optimal alignment  $\gamma \in \Gamma^{opt}(\Lambda^{abstraction}, \sigma_{next})$  and the set of frozen subtrees  $T_{frozen}$ . The function iterates over the provided alignment  $\gamma$  and iteratively constructs the projected trace  $\tilde{\sigma}$ . Four case distinctions are made by Algorithm 7.2, cf. lines 3, 5, 7 and 9.

- If alignment move  $\gamma(i)$  is a synchronous move (line 3) whose executed leaf vertex is not part of a frozen subtree *or* the executed leaf vertex is part of a not fully executed frozen subtree (cf. Definition 7.2), we add the activity label to  $\tilde{\sigma}_{next}$  (line 4).
- If alignment move  $\gamma(i)$  is an invisible model move representing the opening of a frozen subtree *and* the frozen subtree is fully executed (cf. Definition 7.2), we add the corresponding opening label to  $\tilde{\sigma}_{next}$  (line 6).

- If alignment move  $\gamma(i)$  is an invisible model move representing the closing of a frozen subtree *and* the frozen subtree was fully executed (cf. Definition 7.2), we add the corresponding closing label to  $\tilde{\sigma}_{next}$  (line 8).
- If alignment move  $\gamma(i)$  is a log move (line 9), we add the corresponding activity label to  $\tilde{\sigma}_{next}$  (line 10).

Alignment movements other than those specified above or which do not satisfy the above conditions are ignored by Algorithm 7.2. Finally, it should be noted, that since optimal alignments are generally not unique, the trace  $\tilde{\sigma}_{next}$  obtained by component (2) for a given set of frozen subtrees  $\Lambda_{i_1}, \dots, \Lambda_{i_n}$  and trace  $\sigma_{next}$  may not be unique. However, component (2) always guarantees that the returned trace  $\tilde{\sigma}_{next} \in \mathbb{L}(\tilde{\Lambda})$  iff  $\sigma_{next} \in \mathbb{L}(\Lambda)$ .

#### 7.3.4. Component (3)—Projecting Previously Added Traces

This section introduces component (3), which modifies the previously added traces  $A$ , cf. Figure 7.5 (page 185). Other than in component (2), which modifies the trace to be added next  $\sigma_{next}$ , we know that traces contained in  $A$  fit the process tree  $\Lambda$ , i.e.,  $A \subseteq \mathbb{L}(\Lambda)$ . However, after modifying tree  $\Lambda$  into  $\tilde{\Lambda}$  (cf. component (1) introduced in Section 7.3.2), traces contained in  $A$  do not fit  $\tilde{\Lambda}$  because  $\tilde{\Lambda}$  contains replacement labels for the replaced frozen subtrees that are not contained in traces from  $A$ .<sup>2</sup>

Similar to component (2), component (3) computes optimal alignments for the traces contained in  $A$  and  $\Lambda$ . Since all traces in  $A$  are supported by  $\Lambda$ , the optimal alignments computed contain no deviations; hence, these optimal alignments are free of log and visible model moves. Thus, when a frozen subtree's root vertex is opened, we know this frozen subtree is fully executed; there is no need to check this as in component (2). Below, we summarize the central steps taken by component (3).

7

1. For each trace  $\sigma \in A$ , an optimal alignment, i.e.,  $\gamma \in \Gamma^{opt}(\Lambda, \sigma_j)$ , is computed.
2. The projected trace  $\tilde{\sigma}$  is extracted from  $\gamma$  by applying the projection function  $project^A$  introduced in Algorithm 7.3.
3. Projected previously added trace  $\tilde{\sigma}$  is added to the set of projected previously added traces, i.e.,  $\tilde{A}$ .

Algorithm 7.3 specifies the function  $project^A$  that extracts the projected trace from a given alignment, cf. the 2<sup>nd</sup> step of component (3). As input, function  $project^A$  is provided an optimal input for a previously added trace  $\sigma \in A$  and tree  $\Lambda$ ; further, the set of frozen subtrees  $T_{frozen}$  is provided. After initializing the projected trace  $\tilde{\sigma}$  (line 1),  $project^A$  iterates over the alignment moves of  $\gamma$  (line 2). Three case distinctions are made per alignment move, cf. lines 3, 5 and 7.

- If a synchronous move is reached containing a leaf vertex that is not part of a frozen subtree (line 3), we add the corresponding label of that leaf vertex to  $\tilde{\sigma}$  (line 4).

<sup>2</sup>Only in the case if no frozen subtrees (i.e,  $n = 0$ ) are provided, component (1) does not alter the provided tree; thus,  $\Lambda = \tilde{\Lambda}$ . In this particular case, traces contained in  $A$  also fit  $\tilde{\Lambda}$ .

- If an invisible model move is reached representing the opening of a frozen subtree's root (line 5), we add the corresponding unique opening label of that frozen subtree to  $\tilde{\sigma}$  (line 6).
- If an invisible model move is reached representing the closing of a frozen subtree's root (line 7), we add the corresponding unique closing label of that frozen subtree to  $\tilde{\sigma}$  (line 8).

In summary, Algorithm 7.3 specifying  $project^A$  is similarly structured as Algorithm 7.2 specifying  $project^{\sigma_{next}}$ . Since all previously add traces  $A$  are supported by process tree  $\Lambda$ , frozen subtrees are always fully executed if they occur. As a result, Algorithm 7.3 can be seen as a simplified version of Algorithm 7.2.

---

**Algorithm 7.3:**  $project^A$ 


---

**Input:**  $\gamma \in \Gamma^{opt}(\Lambda, \sigma)$ , // optimal alignment for a previously added trace  $\sigma \in A$  and process tree  $\Lambda \in \mathcal{P}$

$T_{frozen} = \{ \Lambda_{i_1} = (V_{i_1}, E_{i_1}, \Sigma_{i_1}, \lambda_{i_1}, r_{i_1}, <_{i_1}), \dots, \Lambda_{i_n} = (V_{i_n}, E_{i_n}, \Sigma_{i_n}, \lambda_{i_n}, r_{i_n}, <_{i_n}) \}$   
 with  $\Lambda_{i_1}, \dots, \Lambda_{i_n} \sqsubseteq \Lambda$  //  $n \geq 0$  frozen subtrees of  $\Lambda$

**Output:**  $\tilde{\sigma} \in \mathcal{A}^*$  // projected previously added trace

**begin**

```

1   $\tilde{\sigma} \leftarrow \langle \rangle$ 
2  for  $1 \leq j \leq |\gamma|$  do // iterate over alignment moves
3    if  $syncMv(\gamma(j)) \wedge modelVertex(\gamma(j)) \notin V_{i_1} \cup \dots \cup V_{i_n}$  then // add
      label from sync. move if the sync. move does not involve a vertex from a
      frozen subtree
4       $\tilde{\sigma} \leftarrow \tilde{\sigma} \circ \langle modelLabel(\gamma(j)) \rangle$ 
5    if  $invModelMv(\gamma(j)) \wedge \exists 1 \leq k \leq n (modelVertex(\gamma(j)) = r_{i_k}) \wedge$ 
       $modelLabel(\gamma(j)) = open$  then // a frozen subtree is opened, add
      corresponding opening label to  $\tilde{\sigma}$ 
6       $\tilde{\sigma} \leftarrow \tilde{\sigma} \circ \langle open^{\Lambda_{i_k}} \rangle$ 
7    if  $invModelMv(\gamma(j)) \wedge \exists 1 \leq k \leq n (modelVertex(\gamma(j)) = r_{i_k}) \wedge$ 
       $modelLabel(\gamma(j)) = close$  then // a frozen subtree is closed, add
      corresponding closing label to  $\tilde{\sigma}$ 
8       $\tilde{\sigma} \leftarrow \tilde{\sigma} \circ \langle close^{\Lambda_{i_k}} \rangle$ 
9  return  $\tilde{\sigma}$ 

```

---

For example, when applying component (3) to the running example, i.e., previously added traces  $A = \{ \langle a, b, c \rangle, \langle a, b, c, d \rangle, \langle a, b, d, c \rangle \}$  and tree  $\Lambda_1$  with frozen subtree  $\Lambda_{i_1} = \triangle_{\Lambda_1}(v_{2.3})$  (cf. Figure 7.6a on page 186), we obtain the following projected previously added traces.

- $\sigma_1 = \langle a, b, c \rangle$  is projected to  $\tilde{\sigma}_1 = \langle a, b, open^{\Lambda_{i_1}}, close^{\Lambda_{i_1}} \rangle$
- $\sigma_2 = \langle a, b, c, d \rangle$  is projected to  $\tilde{\sigma}_2 = \langle a, b, open^{\Lambda_{i_1}}, close^{\Lambda_{i_1}} \rangle$

- $\sigma_3 = \langle a, b, d, c \rangle$  is projected to  $\tilde{\sigma}_3 = \langle a, b, open^{\Lambda_{i_1}}, close^{\Lambda_{i_1}} \rangle$

As a result,  $\tilde{A} = \{ \langle a, b, open^{\Lambda_{i_1}}, close^{\Lambda_{i_1}} \rangle \}$ . Since when replaying any trace from  $A$  onto  $\Lambda_1$ , the frozen subtree  $\Lambda_{i_1}$  is always executed last, i.e., no other leaf vertex is executed afterwards. Further, while executing the frozen subtree  $\Lambda_{i_1}$ , no other subtree that is not part of  $\Lambda_{i_1}$  is executed in parallel. Thus, all traces from  $A$  are projected to the same trace, see  $\tilde{A}$  above.

### 7.3.5. Component (4)—Reinserting Frozen Subtrees

Reconsider the overview of the freezing-enabled LCA-IPDA shown in Figure 7.5 (page 185). Components (1), (2), and (3) modify the inputs fed into a non-freezing-enabled IPDA. This section introduces component (4), which modifies the output process tree  $\tilde{\Lambda}'$ , which is returned by the non-freezing-enabled IPDA, cf. Figure 7.5. Recall that  $\tilde{\Lambda}$ , which is fed into the non-freezing-enabled IPDA, does not contain any frozen subtree; all frozen subtrees have been replaced by component (1) (cf. Section 7.3.2). Thus, the resulting tree  $\tilde{\Lambda}'$  returned by the IPDA does not contain the frozen subtrees.<sup>3</sup> Component (4) reinserts the frozen subtrees into  $\tilde{\Lambda}'$  such that the overall freezing-enabled LCA-IPDA adheres to Definition 7.1 (page 181). Due to applying a non-freezing-enabled IPDA (cf. Figure 7.5), replacement labels used to replace frozen subtrees might occur more than once in tree returned by the IPDA. Therefore, component (4) must find appropriate positions within  $\tilde{\Lambda}'$  to reinsert the frozen subtrees.

#### Example

For example, reconsider the running example with  $\tilde{\Lambda}_1$  (cf. Figure 7.6b on page 186),  $\tilde{A} = \{ \langle a, b, open^{\Lambda_{i_1}}, close^{\Lambda_{i_1}} \rangle \}$  (cf. Section 7.3.4), and  $\tilde{\sigma}_{next} = \langle a, b, open^{\Lambda_{i_1}}, d, close^{\Lambda_{i_1}} \rangle$  (cf. Section 7.3.3). Figure 7.11 depicts a potential process tree  $\tilde{\Lambda}'_1$  returned by the deployed IPDA inside the freezing-enabled LCA-IPDA. Note that the following traces are in the language, according to the definition of an IPDA (cf. Definition 5.1).

$$\{ \tilde{\sigma}_{next} \} \cup \tilde{A} \subseteq \mathbb{L}(\tilde{\Lambda}'_1)$$

Recall that component (1) replaced the frozen subtree  $\Lambda_{i_1}$  in  $\Lambda_1$  with  $\rightarrow (open^{\Lambda_{i_1}}, close^{\Lambda_{i_1}})$ . However, the subtree  $\rightarrow (open^{\Lambda_{i_1}}, close^{\Lambda_{i_1}})$  is not contained in  $\tilde{\Lambda}'_1$ , i.e., between the leaf vertex  $v_{3.1}$  labeled  $open^{\Lambda_{i_1}}$  and  $v_{3.3}$  labeled  $close^{\Lambda_{i_1}}$  a new subtree rooted at vertex  $v_{3.2}$  labeled  $\times$  has been added by the IPDA, cf. Figure 7.11. Therefore, component (4) must calculate an appropriate position for reinserting frozen subtree  $\Lambda_{i_1}$ . To this end, component (4) calculates an LCA from all vertices that are labeled with the corresponding replacement labels  $open^{\Lambda_{i_1}}$  and  $close^{\Lambda_{i_1}}$ . In the example tree  $\tilde{\Lambda}'_1$ , vertex  $v_{3.1}$  labeled  $open^{\Lambda_{i_1}}$  and  $v_{3.3}$  labeled  $close^{\Lambda_{i_1}}$  are used for determining the LCA.

$$lca_{\tilde{\Lambda}'_1}(v_{3.1}, v_{3.3}) = v_{2.3}$$

<sup>3</sup>However, in rare cases, tree  $\tilde{\Lambda}'$  returned by the IPDA (cf. Figure 7.5) may contain some of the frozen subtrees, as the IPDA may have created the frozen subtrees through the changes applied to the tree. Nevertheless, tree  $\tilde{\Lambda}'$  still contains the replacement labels that must be replaced by the corresponding frozen subtrees.

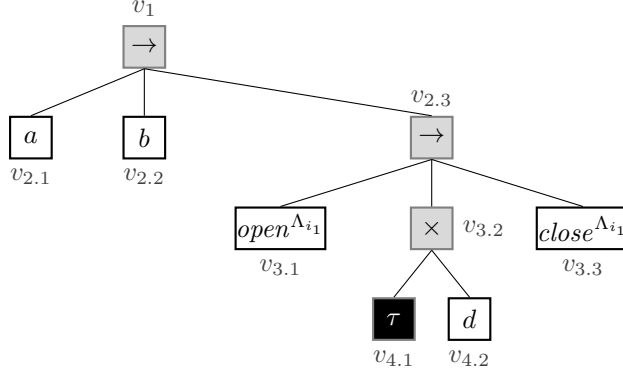


Figure 7.11: Potential process tree  $\tilde{\Lambda}'_1$  returned by the IPDA embedded in the advanced freezing-enabled IPDA (cf. Figure 7.5) for the running example

Next, we do a semantic analysis of the determined subtree  $\Delta_{\tilde{\Lambda}'_1}(v_{2.3})$  in which we determine how often vertices labeled  $open^{\Lambda_{i_1}}$  and vertices labeled  $close^{\Lambda_{i_1}}$  can be replayed. In the determined subtree  $\Delta_{\tilde{\Lambda}'_1}(v_{2.3})$  (cf. Figure 7.11), exactly once a vertex labeled  $open^{\Lambda_{i_1}}$  must be executed (i.e.,  $v_{3.1}$ ) as well as exactly once a vertex labeled  $close^{\Lambda_{i_1}}$  (i.e.,  $v_{3.3}$ ). Further, no other vertices exist labeled with the corresponding replacement labels, and neither can the vertices  $v_{3.1}$  and  $v_{3.3}$  be skipped. Thus, we reinsert the frozen subtree  $\Lambda_{i_1}$  in parallel to the subtree  $\Delta_{\tilde{\Lambda}'_1}(v_{2.3})$ . Figure 7.13 visualizes the intermediate process tree obtained. Next, vertices representing the closing and opening of the reinserted frozen subtree can be removed; vertices  $v_{4.1}$  and  $v_{4.3}$  can be removed as indicated in Figure 7.13. Consequently, also vertex  $v_{3.1}$  can be removed since it only contains a single child vertex, i.e.,  $v_{4.2}$ , after removing  $v_{4.1}$  and  $v_{4.3}$ . Figure 7.13 depicts the finally returned tree  $\Lambda'$  with:  $\Lambda_{i_1} \subseteq \Lambda'$  and  $\{\sigma_{next}\} \cup A \subseteq \mathbb{L}(\Lambda')$ .

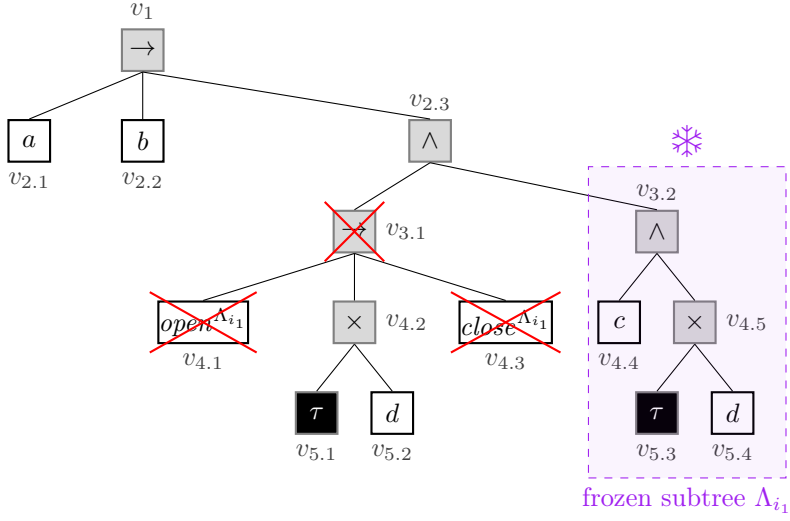


Figure 7.12: Intermediate process tree after reinserting frozen subtree  $\Lambda_{i_1}$  in parallel to determined the subtree  $\Delta_{\tilde{\Lambda}_1}(v_{2.3})$  (cf. Figure 7.11); red crossed vertices and corresponding edges are removed in a postprocessing step

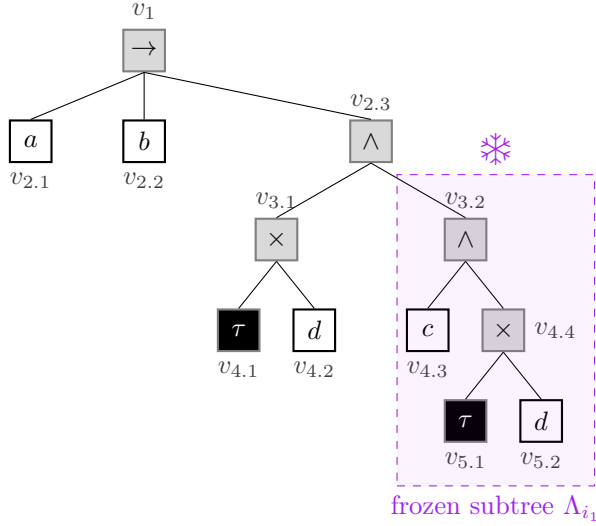


Figure 7.13: Process tree  $\Lambda'$  returned by component (4); crossed out elements shown in the intermediate process tree (cf. Figure 7.12) have been removed

## General Case

Following, we present the general case of component (4). To this end, we first introduce the so-called Semantic Tree Analysis (STA). Afterwards, we present the algorithm specifying component (4).

**Semantic Tree Analysis (STA)** The STA is an integral function used within component (4). Given a process tree  $\Lambda \in \mathcal{P}$  and a label  $l \in \mathcal{A}$ , the STA returns how often vertices labeled  $l$  within  $\Lambda$  can/must be replayed. The STA makes a simplifying assumption and only considers three fundamental cases: can an activity not occur at all, can an activity occur once, and can an activity occur more than once, i.e. at least twice, in a trace contained in the language of the process tree. These three fundamental cases can be further combined. Five possible outcomes may result from the STA for a given tree  $\Lambda$  and label  $l$ .

- $\{0\}$  corresponds to *zero*. Potential vertices labeled with the provided label  $l$  can never be executed. Hence, no vertex exists with the specified label  $l$ .<sup>4</sup>
- $\{1\}$  corresponds to *once*. Exactly once, a leaf vertex labeled  $l$  must be executed in any running sequence.
- $\{0, 1\}$  corresponds to *at most once*. At most, one leaf vertex labeled  $l$  may be executed in any running sequence. Thus, the tree has running sequences that do not contain any leaf vertex labeled  $l$  or exactly one leaf vertex labeled  $l$ .
- $\{0, \infty\}$  corresponds to *zero to many*. Vertices labeled  $l$  may be executed arbitrarily often, i.e., cases  $\{0\}$  and  $\{1\}$  apply and there exists at least one running sequence that contains two leaf vertices labeled  $l$ . Note that these two leaf vertices labeled  $l$  can be identical, for example, consider a leaf vertex enclosed in a loop construct.
- $\{1, \infty\}$  corresponds to *once to many*. At least one vertex labeled with  $l$  must be executed, i.e., case  $\{0\}$  does not apply, case  $\{1\}$  applies, and there exists at least one running sequence that contains two leaf vertices labeled  $l$ .

Subsequently, we introduce the helper function *count* (Definition 7.3). Eventually, we formally define the STA (Definition 7.4) using Definition 7.3.

### Definition 7.3 (*count*)

Let  $\sigma \in \mathcal{A}^*$  be a trace and  $l \in \mathcal{A}$  be an activity label.

Function  $\text{count} : \mathcal{A}^* \times \mathcal{A} \rightarrow \mathbb{N}_0$  returns the number of occurrences of  $l$  in  $\sigma$ .

$$\text{count}(\sigma, l) = \left| \{i \mid 1 \leq i \leq |\sigma| \wedge \sigma(i) = l\} \right|$$

<sup>4</sup>Recall that process trees represent sound WF-nets. Thus, any leaf vertex is executable in a process tree, i.e., no *dead* vertices exist in a process tree.



**Definition 7.4** (Semantic Tree Analysis (STA))

Let  $\Lambda \in \mathcal{P}$  be a process tree and  $l \in \mathcal{A}$  be an activity label.

Semantic tree analysis (STA) is a function

$$sta : \mathcal{P} \times \mathcal{A} \rightarrow \left\{ \{0\}, \{1\}, \{0, 1\}, \{0, \infty\}, \{1, \infty\} \right\}$$

that maps  $\Lambda$  and  $l$  to one of the five above-specified sets that indicate how often leaf vertices labeled  $l$  can/must be executed within the given tree  $\Lambda$ .

$$sta(\Lambda, l) = \begin{cases} \{0\} & \text{if } \forall \sigma \in \mathbb{L}(\Lambda) \ (count(\sigma, l) = 0) \\ \{1\} & \text{if } \forall \sigma \in \mathbb{L}(\Lambda) \ (count(\sigma, l) = 1) \\ \{0, 1\} & \text{if } \exists \sigma, \sigma' \in \mathbb{L}(\Lambda) \ (count(\sigma, l) = 0 \wedge count(\sigma', l) = 1) \wedge \\ & \forall \sigma \in \mathbb{L}(\Lambda) \ (count(\sigma, l) \leq 1) \\ \{1, \infty\} & \text{if } \min_{\sigma \in \mathbb{L}(\Lambda)} count(\sigma, l) = 1 \wedge \exists \sigma \in \mathbb{L}(\Lambda) \ (count(\sigma, l) > 1) \\ \{0, \infty\} & \text{if } \min_{\sigma \in \mathbb{L}(\Lambda)} count(\sigma, l) = 0 \wedge \exists \sigma \in \mathbb{L}(\Lambda) \ (count(\sigma, l) > 1) \end{cases}$$

The STA will be used within component (4) to determine how often vertices that are labeled with replacement labels of frozen subtrees can/must be executed. For example, reconsider process tree  $\tilde{\Lambda}_1$  depicted in Figure 7.11 (page 194). Applying STA on this subtree for the two replacement labels of frozen subtree  $\Lambda_{i_1}$  results in:

- $sta(\tilde{\Lambda}_1, open^{\Lambda_{i_1}}) = \{1\}$
- $sta(\tilde{\Lambda}_1, close^{\Lambda_{i_1}}) = \{1\}$

Thus, all traces supported by  $\tilde{\Lambda}_1$  contain exactly once  $open^{\Lambda_{i_1}}$  and  $close^{\Lambda_{i_1}}$  each. Since the frozen subtree  $\Lambda_{i_1}$  is opened once and closed once in any trace, we derive that the frozen subtree must always be executed once.

In the general case, however, we might obtain different results for the opening and closing replacement labels of a frozen subtree when applying  $sta$ . For example, the STA might return that the label representing the opening of a frozen subtree can be potentially executed multiple times (i.e., case  $\{1, \infty\}$  or  $\{0, \infty\}$ ), however, STA finds that the corresponding closing label must be executed exactly once. In this case, although we can potentially open the frozen subtree multiple times, we can and must close it exactly once. Hence, we can derive that the frozen subtree must always be executed once since we can close it only once. In this case, the multiple options for opening the frozen subtree, i.e., executing a leaf vertex labeled with the corresponding replacement label indicating the opening, are to be understood as different states in the model in which the frozen subtree can be opened, but ultimately it is only opened once. To derive such conclusions, we therefore introduce the operator

$$\sqcup : \left\{ \{0\}, \{1\}, \{0, 1\}, \{0, \infty\}, \{1, \infty\} \right\} \times \left\{ \{0\}, \{1\}, \{0, 1\}, \{0, \infty\}, \{1, \infty\} \right\} \rightarrow \left\{ \{0\}, \{1\}, \{0, 1\}, \{0, \infty\}, \{1, \infty\} \right\}$$

that maps the outcome of two *sta* results to one. Note that the operator  $\sqcup$  is specifically defined for the usage of combining the results of an opening and closing replacement label of an individual frozen subtree. Thus, when a frozen subtree is opened, it is also closed again, i.e., a leaf vertex is executed, labeled with the corresponding replacement label indicating its opening, followed by the execution of a leaf vertex labeled with the corresponding replacement label indicating its closing. Table 7.1 (page 198) specifies the  $\sqcup$  operator, which takes into account the fact that every tree that is opened must also be closed again.

Table 7.1.: Definition of the  $\sqcup$  operator for two results obtained from function *sta* (cf. Definition 7.4), i.e., defining  $sta(\Lambda_1, l_1) \sqcup sta(\Lambda_2, l_2)$  for arbitrary trees  $\Lambda_1, \Lambda_2 \in \mathcal{P}$  and labels  $l_1, l_2 \in \mathcal{A}$

$\sqcup$	$\{0\}$	$\{1\}$	$\{0, 1\}$	$\{0, \infty\}$	$\{1, \infty\}$
$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$
$\{1\}$	$\{0\}$	$\{1\}$	$\{1\}$	$\{1\}$	$\{1\}$
$\{0, 1\}$	$\{0\}$	$\{1\}$	$\{0, 1\}$	$\{0, 1\}$	$\{1\}$
$\{0, \infty\}$	$\{0\}$	$\{1\}$	$\{0, 1\}$	$\{0, \infty\}$	$\{1, \infty\}$
$\{1, \infty\}$	$\{0\}$	$\{1\}$	$\{1\}$	$\{1, \infty\}$	$\{1, \infty\}$

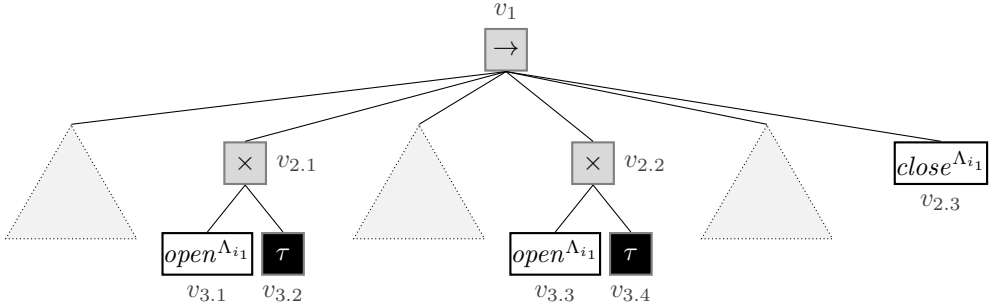


Figure 7.14: Example tree  $\tilde{\Lambda}'_2$  that contains twice a replacement label for opening but only once for closing a frozen subtree  $\Lambda_{i_1}$

For example, consider the exemplary tree  $\tilde{\Lambda}'_2$  depicted in Figure 7.14 that contains replacement labels for a frozen subtree  $\Lambda_{i_1}$ . Applying STA for this tree and the opening/closing labels (i.e.,  $open^{\Lambda_{i_1}}$  and  $close^{\Lambda_{i_1}}$ ) yields the following information.

- $sta(\tilde{\Lambda}'_2, open^{\Lambda_{i_1}}) = \{0, \infty\}$
- $sta(\tilde{\Lambda}'_2, close^{\Lambda_{i_1}}) = \{1\}$

Since the vertex  $v_{2.3}$  must always be executed, i.e., the frozen subtree  $\Lambda_{i_1}$  is always closed, and we can optionally execute both vertices (i.e.,  $v_{3.1}$  and  $v_{3.3}$ ) representing the opening

of  $\Lambda_{i_1}$ , we can conclude that the frozen subtree  $\Lambda_{i_1}$  must be always executed once.

$$\{0, \infty\} \sqcup \{1\} = \{1\} \quad (\text{according to Table 7.1})$$

**Algorithmic Description of Component (4)** Consider Algorithm 7.4 specifying all relevant steps. As input, component (4) (Algorithm 7.4) is provided the tree  $\tilde{\Lambda}'$ , frozen subtrees  $T_{\text{frozen}}$ , the trace to be added next  $\sigma_{\text{next}}$ , and the previously added traces  $A$ . Algorithm 7.4 gradually reinserts each frozen subtree into  $\tilde{\Lambda}'$ . As specified in line 1, variable  $j$  represents the index of the frozen subtree, which is about to be reinserted into  $\tilde{\Lambda}'$  next. First, the partly projected trace to be added next  $\tilde{\sigma}_{\text{next}}^{\Lambda_{i_1}, \dots, \Lambda_{i_j}}$  is calculated (line 2) by applying Algorithm 7.2 using the so far processed frozen subtrees  $\Lambda_{i_1}, \dots, \Lambda_{i_j}$ . Similarly, the partly projected previously added traces  $\tilde{A}^{\Lambda_{i_1}, \dots, \Lambda_{i_j}}$  are computed (line 3). Next, the LCA is computed from all vertices that are labeled with the replacement labels of the currently considered frozen subtree  $\Lambda_{i_j}$ .<sup>5</sup> Assuming that the LCA vertex  $v_{LCA}$  is defined, we compute the subtree  $\Lambda_c$  rooted at  $v_{LCA}$  (line 6). The subtree  $\Lambda_c$  represents a candidate that we are using for reinserting the frozen subtree. For example, recall the running example discussed at the beginning of this section, cf.  $v_{2.3}$  in Figure 7.11.

Having determined the reinsertion subtree candidate  $\Lambda_c$ , next, we enter a loop (line 8) that ends upon the partly projected trace  $\tilde{\sigma}_{\text{next}}^{\Lambda_{i_1}, \dots, \Lambda_{i_j}}$  and previously added traces  $\tilde{A}^{\Lambda_{i_1}, \dots, \Lambda_{i_j}}$  fit tree  $\tilde{\Lambda}'$ . First, we do a STA of subtree candidate  $\Lambda_c$  (line 9). As a result, we obtain the information how often the frozen subtree  $\Lambda_{i_j}$  must/can be replayed in  $\Lambda_c$ . According to the value  $S$  (line 9), we apply the corresponding case illustrated in Figure 7.15 (Figure 7.15). The shown cases specify reinsertion rules for the frozen subtree  $\Lambda_{i_j}$  giving the reinsertion candidate  $\Lambda_c$ . Figure 7.15a depicts the initial case, i.e., the determined tree candidate  $\Lambda_c$  with root vertex  $r_c$  and a potential parent vertex of  $r_c$ . Subsequently, we briefly describe the five reinsertion rules.

- **Case  $\{0\}$ —Figure 7.15b**

In this case, the entire tree  $\tilde{\Lambda}'$  does not allow the execution of the frozen subtree. Thus, neither partly projected previously added traces  $\tilde{A}^{\Lambda_{i_1}, \dots, \Lambda_{i_j}}$  nor  $\tilde{\sigma}_{\text{next}}^{\Lambda_{i_1}, \dots, \Lambda_{i_j}}$  require the frozen subtree  $\Lambda_{i_j}$ . However, since Definition 7.1 requires that any frozen subtree is also part of the final tree, we must add frozen subtree  $\Lambda_{i_j}$  to  $\tilde{\Lambda}'$ . We add a new root node labeled  $\times$  and the frozen subtree  $\Lambda_{i_j}$  as an option next to the tree  $\tilde{\Lambda}'$ . In this way, the language of  $\tilde{\Lambda}'$  is not changed, and the overall tree only allows for additional behavior specified in frozen subtree  $\Lambda_{i_j}$ .

- **Case  $\{1\}$ —Figure 7.15c**

In this case, the frozen subtree  $\Lambda_{i_j}$  must always be executed exactly once. Thus,

<sup>5</sup>Note that although component (1) (cf. Section 7.3.2) adds only one vertex labeled *open* <sup>$\Lambda_{i_j}$</sup>  and one vertex labeled *close* <sup>$\Lambda_{i_j}$</sup>  to the tree when replacing the frozen subtrees, the invoked non-freezing-enabled IPDA (cf. Figure 7.5 page 185) can return a tree  $\tilde{\Lambda}'$  that contains more than these two vertices added by component (1) that are labeled with the replacement labels of frozen subtree  $\Lambda_{i_j}$ . Hence, the LCA calculation in Algorithm 7.4 line 4 uses a set of vertices as input.

**Algorithm 7.4:** Component (4)

---

**Input:**  $\tilde{\Lambda}' = (\tilde{V}', \tilde{E}', \tilde{\Sigma}', \tilde{\lambda}', \tilde{r}', \tilde{z}') \in \mathcal{P}$ , // tree returned by the  
non-freezing-enabled IPDA, cf. Figure 7.5

$T_{\text{frozen}} = \{\Lambda_{i_1}, \dots, \Lambda_{i_n}\}$  with  $\Lambda_{i_1}, \dots, \Lambda_{i_n} \sqsubseteq \Lambda$ , //  $n \geq 0$  frozen subtrees of  $\Lambda$

$\sigma_{\text{next}} \in \mathcal{A}^*$ , // trace to be added next

$A \subseteq \mathcal{A}^*$  // previously added traces

**Output:**  $\Lambda' \in \mathcal{P}$  //  $\Lambda'$  with  $\{\sigma_{\text{next}}\} \cup A \subseteq \mathbb{L}(\Lambda')$  and  $\Lambda_{i_1}, \dots, \Lambda_{i_n}$

**begin**

1   **for**  $1 \leq j \leq n$  **do** // iterate over frozen subtrees  $\Lambda_{i_1}, \dots, \Lambda_{i_n} \sqsubseteq \Lambda'$

2     let  $\tilde{\sigma}_{\text{next}}^{\Lambda_{i_1}, \dots, \Lambda_{i_j}}$  bet the partly projected trace after replacing full executions  
   of frozen subtrees  $\Lambda_{i_1}, \dots, \Lambda_{i_j}$  in  $\sigma_{\text{next}}$  // apply Algorithm 7.2 for a part  
   of frozen subtrees, i.e.,  $\Lambda_{i_1}, \dots, \Lambda_{i_j}$

3     let  $\tilde{A}^{\Lambda_{i_1}, \dots, \Lambda_{i_j}}$  bet the set of partly projected previously added traces after  
   replacing full executions of frozen subtrees  $\Lambda_{i_1}, \dots, \Lambda_{i_j}$  in all traces  $\sigma \in A$   
   // apply Algorithm 7.3 for a part of frozen subtrees, i.e.,  $\Lambda_{i_1}, \dots, \Lambda_{i_j}$

4      $v_{LCA} \leftarrow lca \left( \left\{ v \in \tilde{V}' \mid \tilde{\lambda}'(v) \in \{open^{\Lambda_{i_j}}, close^{\Lambda_{i_j}}\} \right\} \right)$  // calculate the  
   LCA from all vertices labeled with the replacement labels of  $\Lambda_{i_j}$

5     **if**  $v_{LCA}$  is undefined **then**

6        $\tilde{\Lambda}' \leftarrow$  apply case  $\{0\}$  as illustrated in Figure 7.15b // reinsert frozen  
      subtree  $\Lambda_{i_j}$  into  $\tilde{\Lambda}'$

7      $\Lambda_c \leftarrow \Delta_{\tilde{\Lambda}'}(v_{LCA})$  //  $\Lambda_c \sqsubseteq \tilde{\Lambda}'$  is the first insert candidate at which we try  
      to reinsert the frozen subtree  $\Lambda_{i_j}$

8     **while**  $\{\tilde{\sigma}_{\text{next}}^{\Lambda_{i_1}, \dots, \Lambda_{i_j}}\} \cup \tilde{A}^{\Lambda_{i_1}, \dots, \Lambda_{i_j}} \not\subseteq \mathbb{L}(\tilde{\Lambda}')$  **do**

9        $S \leftarrow sta(\Lambda_c, open^{\Lambda_{i_j}}) \sqcup sta(\Lambda_c, close^{\Lambda_{i_j}})$  // determine how often frozen  
      subtree  $\Lambda_{i_j}$  can be replayed in  $\Lambda_c$  by utilizing semantic tree analysis,  
      cf. Definition 7.4 and Table 7.1 (pages 197 and 198)

10        $\tilde{\Lambda}' \leftarrow$  apply case  $S$  as illustrated in Figure 7.15 // reinsert frozen  
      subtree  $\Lambda_{i_j}$  into  $\tilde{\Lambda}'$

11        $\tilde{\Lambda}' \leftarrow$  remove vertices labeled  $open^{\Lambda_{i_j}}$  or  $close^{\Lambda_{i_j}}$  and connected edges

12       **if**  $\{\tilde{\sigma}_{\text{next}}^{\Lambda_{i_1}, \dots, \Lambda_{i_j}}\} \cup \tilde{A}^{\Lambda_{i_1}, \dots, \Lambda_{i_j}} \not\subseteq \mathbb{L}(\tilde{\Lambda}')$  **then**

13           $\tilde{\Lambda}' \leftarrow$  undo changes applied in lines 10 and 11

14           $\Lambda_c \leftarrow \Delta_{\tilde{\Lambda}'}(parent(r_c))$  // try next higher subtree as insertion  
          candidate;  $r_c$  denotes the root vertex of  $\Lambda_c$

15    $\Lambda' \leftarrow$  apply reduction rules to  $\tilde{\Lambda}'$

16   **return**  $\Lambda'$

---

we add the frozen subtree in parallel to the insertion candidate  $\Lambda_c$  and connect the new vertex labeled  $\wedge$  to the old parent of  $r_c$ .

- **Case  $\{0, 1\}$ —Figure 7.15d**

In this case, the frozen subtree  $\Lambda_{i_j}$  can be executed at most once. Thus, we add the frozen subtree parallel to the insertion candidate  $\Lambda_c$  as an optional subtree and connect the new vertex labeled  $\wedge$  to the old parent of  $r_c$ .

- **Case  $\{0, \infty\}$ —Figure 7.15e**

In this case, the frozen subtree  $\Lambda_{i_j}$  must be executable any number of times. Thus, we add the frozen subtree in parallel to the insertion candidate  $\Lambda_c$  within a loop construct and connect the new vertex labeled  $\wedge$  to the old parent of  $r_c$ . Note that the first subtree of a loop vertex must always be executed while the second subtree can be optionally executed, cf. Definition 3.29 (page 68). Thus, the loop construct shown allows the frozen subtree  $\Lambda_{i_j}$  to be replayed any number of times, including skipping.

- **Case  $\{1, \infty\}$ —Figure 7.15f**

In this case, the frozen subtree must be executable any number of times; however, it may not be skipped. The reinsertion rule resembles the one from case  $\{0, \infty\}$  with one difference, i.e., the order below the added loop vertex between the frozen subtree  $\Lambda_{i_j}$  and the leaf vertex labeled  $\tau$  is changed. Thus, frozen subtree  $\Lambda_{i_j}$  must be executed at least once.

After applying the corresponding reinsertion rule from Figure 7.15 in Algorithm 7.4 line 10, we check if the partly projected traces and trace fit process tree, which contains the frozen subtree  $\Lambda_{i_j}$ .

Finally, we briefly elaborate on the termination of component (4), i.e., Algorithm 7.4. In detail, the while loop (lines 8 to 14) continues until the partly projected previously added traces  $\tilde{A}^{\Lambda_{i_1}, \dots, \Lambda_{i_j}}$  and the partly projected trace  $\tilde{\sigma}_{next}^{\Lambda_{i_1}, \dots, \Lambda_{i_j}}$  are supported by  $\tilde{\Lambda}'$ . Eventually, the while loop (lines 8 to 14) might reach the root of  $\tilde{\Lambda}'$ , i.e.,  $\Lambda_c = \tilde{\Lambda}'$ . At this point at the latest the while loop terminates because in this last iteration the frozen subtree  $\Lambda_{i_j}$  is placed in parallel to the entire process tree  $\tilde{\Lambda}'$ , cf. reinsertion rules for cases  $\{1\}$ ,  $\{0, 1\}$ ,  $\{0, \infty\}$ , and  $\{1, \infty\}$  shown in Figures 7.15c to 7.15f.<sup>6</sup> Inserting the frozen subtree  $\Lambda_{i_j}$  in parallel to the entire process tree  $\tilde{\Lambda}'$  always works since it can be executed in parallel any time while executing  $\tilde{\Lambda}'$ . However, reinserting the frozen subtree  $\Lambda_{i_j}$  parallel to the entire tree  $\tilde{\Lambda}'$  might lead to an imprecise overall tree. Thus, Algorithm 7.4 always tries to reinsert the frozen subtree  $\Lambda_{i_j}$  as low as possible in the tree hierarchy of  $\tilde{\Lambda}'$  by starting at the subtree rooted at  $v_{LCA}$  (line 4).

<sup>6</sup>Note that case  $\{0\}$  was already covered before the while loop in line 6.

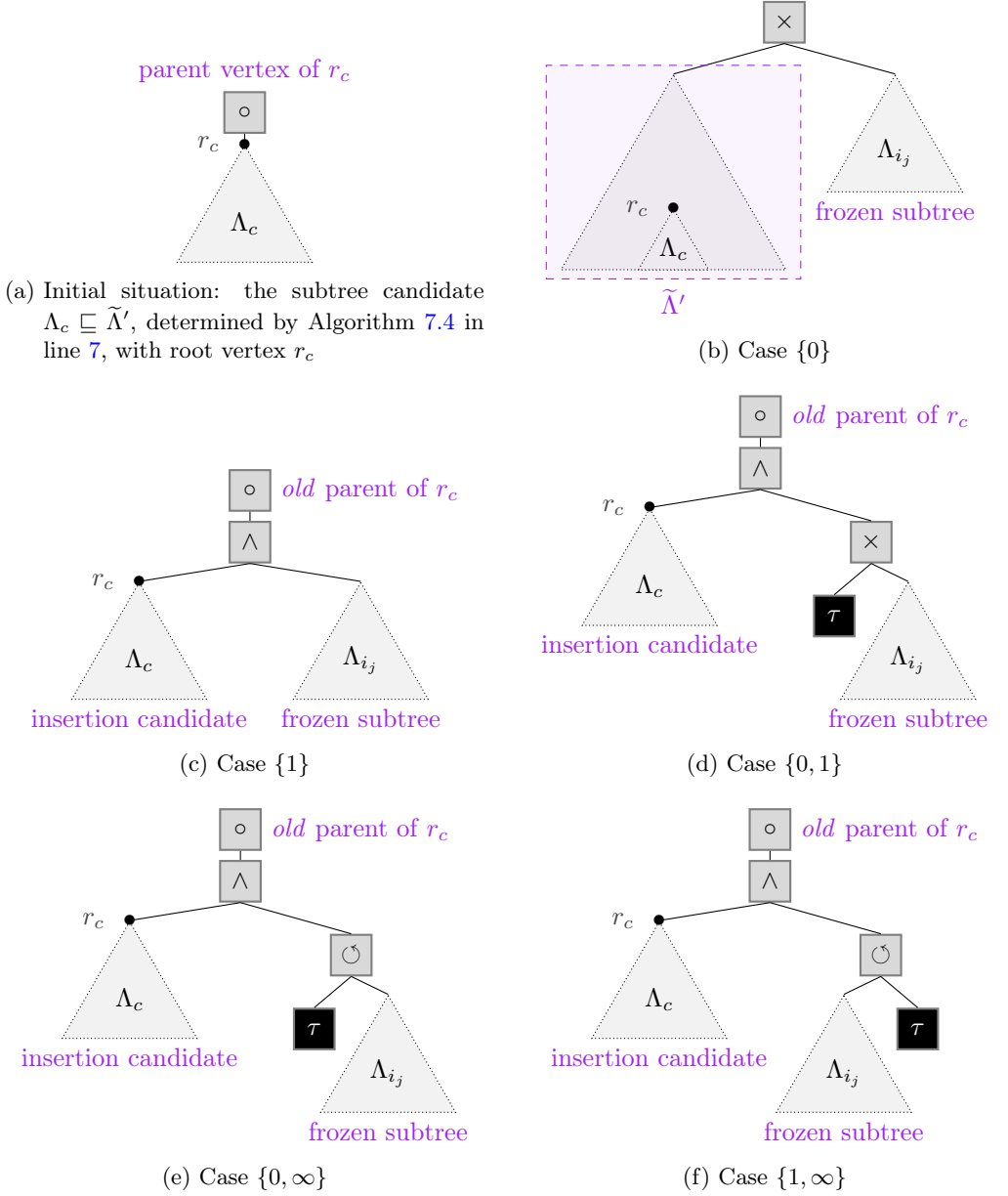


Figure 7.15: Overview of the four different cases that can occur as a result of the STA (cf. Definition 7.4) in Algorithm 7.4 line 9; note that in all figures the depicted parent vertex of  $r_c$ , with label  $\circ \in \otimes$ , does not exist if  $\Lambda_c = \tilde{\Lambda}'$

## 7.4. Evaluation

This section presents an experimental evaluation of the proposed freezing approach. Focus of the evaluation is to demonstrate that the proposed freezing extension may lead to better process models regarding the F-measure. Below, Section 7.4.1 presents the experimental setup. Subsequently, Section 7.4.2 presents the results while Section 7.4.3 discusses the results and lists threats to validity.

### 7.4.1. Experimental Setup

We compare the LCA-IPDA presented in Section 5.3.2 with the freezing-enabled LCA-IPDA presented in Section 7.3. We use publicly available event logs. To counteract ordering effects, we sort the trace variants per event log according to frequency, starting with the most frequent variant. We use the IM to discover an initial model from the most frequent trace variant. To ensure equal conditions at the beginning for both approaches. Subsequently, we iteratively add each trace variant from the log using the LCA-IPDA and the freezing enabled LCA-IPDA. For the latter one, we additionally make use of the freezing option. To decide on the subtree(s) that should be frozen, we apply a brute-force approach in each iteration, as described below. For clarification, we denote

- the tree obtained after adding the  $i$ -th trace variant when applying the LCA-IPDA as  $\Lambda_i^{noFreezing}$ , and
- the tree obtained after adding the  $i$ -th trace variant when applying the freezing-enabled LCA-IPDA as  $\Lambda_i^{freezing}$ .

Below, we describe the steps executed in iteration  $i + 1$  for both approaches.

1. First, we apply the LCA-IPDA using tree  $\Lambda_i^{noFreezing}$ , the previously added trace variants, and the trace variant to be added in this iteration. For the resulting tree  $\Lambda_{i+1}^{noFreezing}$ , we calculate the F-measure using the entire event log.
2. Next, we apply the freezing-enabled LCA-IPDA using tree  $\Lambda_i^{freezing}$  for *each* subtree, which we freeze, of  $\Lambda_i^{freezing}$  that is rooted at an inner vertex. Thus, if  $\Lambda_i^{freezing}$  contains  $n$  inner vertices, we execute the freezing-enabled LCA-IPDA  $n$  times. Each time we freeze one individual subtree that is rooted at an inner vertex. As a result, we obtain  $n$  trees, denoted as  $\Lambda_{i+1}^{freezing,1}, \dots, \Lambda_{i+1}^{freezing,n}$ . Additionally, we also compute the tree with no subtree frozen, denoted as  $\Lambda_{i+1}^{freezing}$ .<sup>7</sup> For each resulting tree, we compute the F-measure using the entire event log.
3. We compare the F-measure values of the process trees  $\Lambda_{i+1}^{freezing,1}, \dots, \Lambda_{i+1}^{freezing,n}$  and  $\Lambda_{i+1}^{freezing,n+1}$  with the F-measure of tree  $\Lambda_{i+1}^{noFreezing}$ .
  - If no tree from  $\Lambda_{i+1}^{freezing}, \Lambda_{i+1}^{freezing,1}, \dots, \Lambda_{i+1}^{freezing,n}$  has a higher F-measure value than  $\Lambda_{i+1}^{noFreezing}$ , we continue the next iteration of the freezing-enabled LCA-IPDA with the tree yielding the highest F-measure.

<sup>7</sup>Note that if no tree is frozen, the freezing-enabled LCA-IPDA is identical to the LCA-IPDA.

- If one of the trees  $\Lambda_{i+1}^{freezing,1}, \dots, tree_{i+1}^{freezing,n}$  has a higher F-measure than tree  $\Lambda_{i+1}^{noFreezing}$ , we know that freezing a subtree is advantageous. Next, we collect all trees from  $\Lambda_{i+1}^{freezing,1}, \dots, tree_{i+1}^{freezing,n}$  that have a higher F-measure than  $\Lambda_{i+1}^{noFreezing}$ . Further, we collect the corresponding subtrees that were frozen. Using the determined set of frozen subtrees, which were frozen in tree  $\Lambda_i^{freezing}$  and lead to a higher F-measure, we apply again the freezing-enabled LCA-IPDA for *every combination* of subtrees, i.e., the power set of subtrees. With this step we want to test whether the freezing of several subtrees, which individually already deliver better results compared to  $\Lambda_{i+1}^{noFreezing}$ , can further improve the result. Assume  $m$  distinct combinations exists that each include at least two subtrees; eventually, we yield  $n + m + 1$  trees  $\Lambda_{i+1}^{freezing}, \Lambda_{i+1}^{freezing,1}, \dots, \Lambda_{i+1}^{freezing,n}, \Lambda_{i+1}^{freezing,n+1}, \dots, \Lambda_{i+1}^{freezing,n+m}$ . Finally, we continue with the tree that leads the highest F-measure, which serves as an input in iteration  $i + 2$ .

Since the described brute force approach is highly computationally intensive, we terminate the experiments after one week of computing time for each event log.

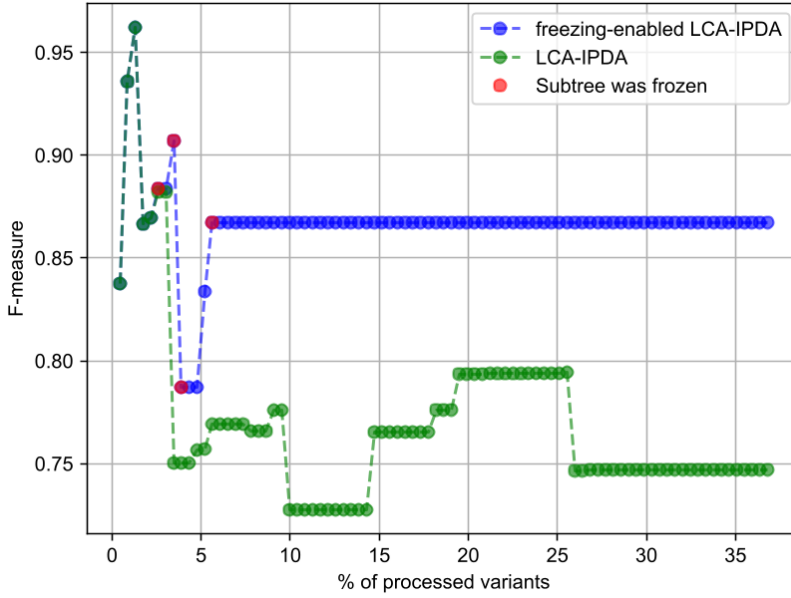
### 7.4.2. Results

Figures 7.16 and 7.17 show the obtained results for four different event logs: BPI Ch. 2020 travel permits event log [232], BPI Ch. 2012 event log [228], road traffic fine management log [56], and the hospital billing event log [133]. Each plot contains two lines; the green line represents the LCA-IPDA (cf. Section 5.3.2), while the blue line represents the freezing-enabled LCA-IPDA (cf. Section 7.3). Each dot represents a data point, i.e., an iteration within IPDA. Recall that each experiment was conducted for one week. Thus, the x-axes of the depicted plots do not end at 100% of processed variants. Note that some dots corresponding to the blue line, representing the freezing-enabled LCA-IPDA, are colored red. A red dot indicates that this tree was obtained by the freezing-capable LCA-IPDA with one or more frozen subtrees in the input tree.

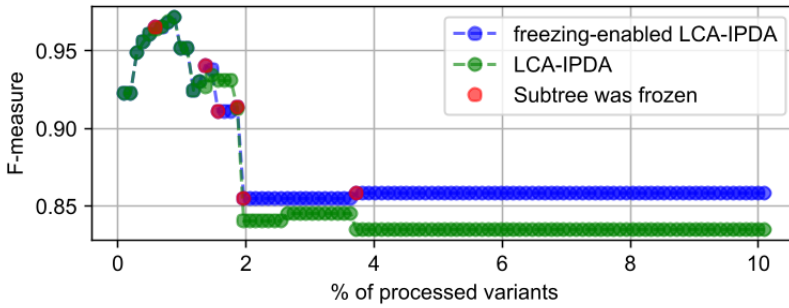
Overall, we observe that the blue line is most of the time above the green one; thus, freezing-enabled LCA-IPDA is outperforming LCA-IPDA most of the time. Further, we observe that freezing is often applied. Note that if, for one approach, the following data point is at the same y-value as the previous one, the input tree equals the output tree since the trace variant added in this iteration already fits the tree. Hence, no freezing is applied since the algorithm must not alter the input tree. For example, this pattern can be seen in Figure 7.16b in the area from 4% to 10% of processed variants for both approaches. Considering these patterns and their interpretation, it can be stated that freezing is frequently used. Further, we can see that when freezing is applied, the F-measure value is significantly higher compared to LCA-IPDA, which does not apply to freeze. In conclusion, the conducted experiments showcase that freezing subtrees within IPD can lead to process models with higher F-measures.

Figures 7.18 and 7.19 compare the number of vertices of the incrementally discovered process trees for the two approaches. For three event logs (cf. Figures 7.18b, 7.19a and 7.19b), we observe that the freezing-enabled LCA-IPDA discovers larger process





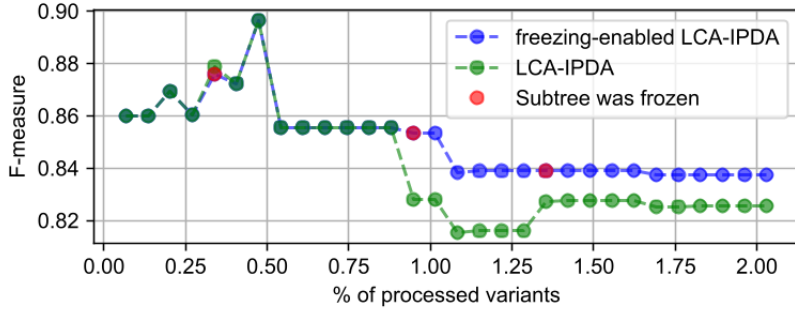
(a) Road traffic fine management event log [56]



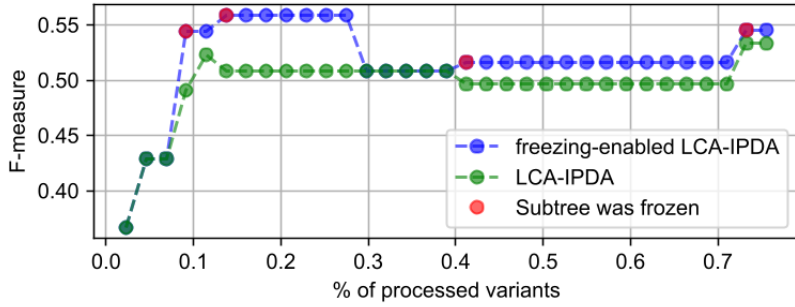
(b) Hospital billing event log [133]

Figure 7.16: Comparing the  $F$ -measure of the incrementally discovered process models using the LCA-IPDA and the freezing-enabled LCA-IPDA; both approaches start from the same initial model and add the trace variants in identical order (part 1/2)

models compared to the ones discovered by the LCA-IPDA. The tree size often changes, especially in the iterations where freezing is applied. While this difference is minor in Figures 7.19a and 7.19b, the differences between the models are significant for the event log depicted in Figure 7.18b. When inspecting individual models, we observe that



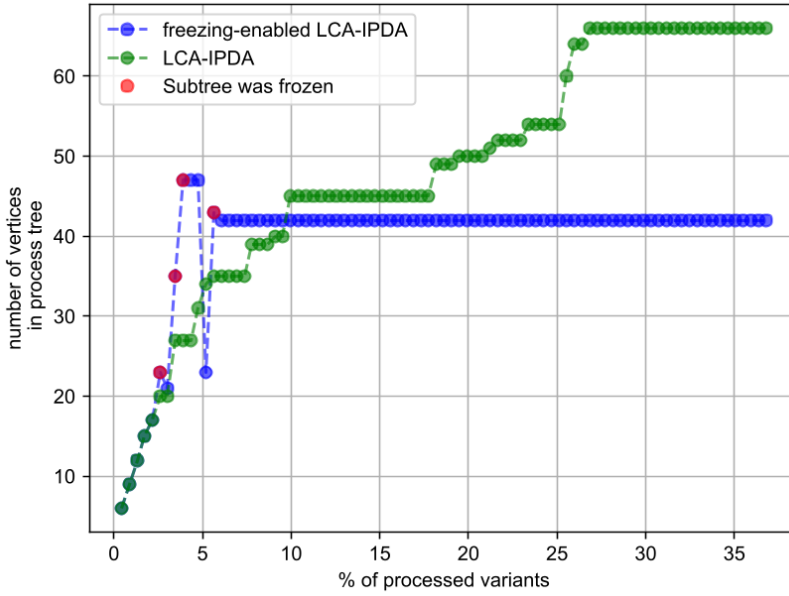
(a) BPI Ch. 2020 travel permits event log [232]



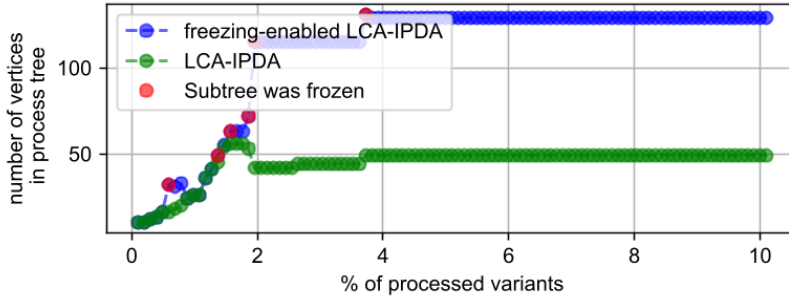
(b) BPI Ch. 2012 event log [228]

Figure 7.17: Comparing the *F-measure* of the incrementally discovered process models using the LCA-IPDA and the freezing-enabled LCA-IPDA; both approaches start from the same initial model and add the trace variants in identical order (part 2/2)

the models discovered by the freezing-enabled LCA-IPDA contain more duplicate labels than the models discovered by the LCA-IPDA. This increased number of duplicate labels explains the corresponding increase in the *F-measure*, cf. Figures 7.16b, 7.17a and 7.17b). Interestingly, for the road traffic fine management log, cf. Figure 7.18a, we observe that the freezing-enabled LCA-IPDA discovers smaller process models than the LCA-IPDA in later stages. At the same time, these smaller models also have higher *F-measure* values, cf. Figures 7.16a and 7.18a. In short, freezing submodels during IPD can lead to better process models regarding the *F-measure* but also impacts the tree size; recall that in three out of four reported logs, the tree size is larger when applying freezing compared to when not applying freezing.

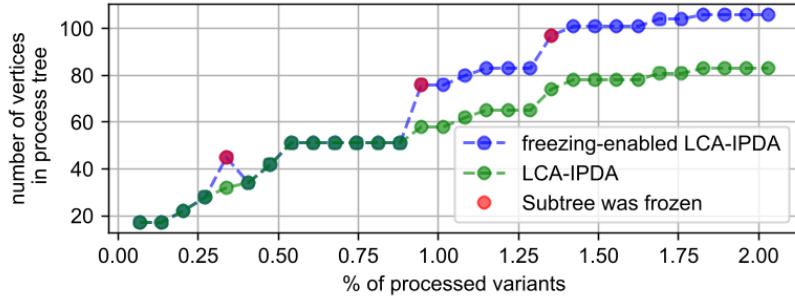


(a) Road traffic fine management event log [56]

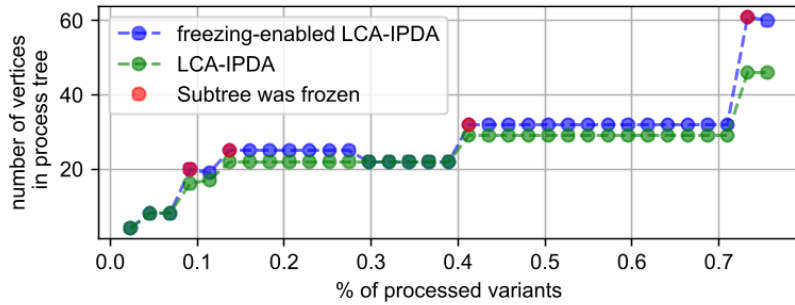


(b) Hospital billing event log [133]

Figure 7.18: Comparing the *number of vertices* of the incrementally discovered process models using the LCA-IPDA and the freezing-enabled LCA-IPDA; both approaches start from the same initial model and add the trace variants in identical order (part 1/2)



(a) BPI Ch. 2020 travel permits event log [232]



(b) BPI Ch. 2012 event log [228]

Figure 7.19: Comparing the *number of vertices* of the incrementally discovered process models using the LCA-IPDA and the freezing-enabled LCA-IPDA; both approaches start from the same initial model and add the trace variants in identical order (part 2/2)

### 7.4.3. Discussion & Threats to Validity

A shortcoming of the above-described experiments is the high computational effort for the applied brute force approach to determine whether and, if so, which subtrees should be frozen. For example, for the logs shown in Figure 7.17, only very few variants are processed. The significance of the results is therefore limited.

Nevertheless, the results generally indicate that freezing subtrees can improve the resulting process trees concerning the F-measure. However, the applied brute force approach is not feasible for real-life applications as it is too computationally expensive. Thus, the chosen experimental setup can be considered an ideal situation in which all options are considered when deciding, i.e., if and if so, which subtrees should be frozen. Finally, freezing might not always be applied to obtain higher F-measures. Users might also use freezing to prevent the LCA-IPDA from altering certain parts in the model.

## 7.5. Illustrative Example

This section, presents a concrete example to showcase the effect of freezing in the context of IPD besides the quantitative evaluation presented in the previous section. For the example, we use the road traffic fine management event log [56] that we used already in the example of the previous chapter, cf. Figure 6.1 on page 157.

	CF	Create Fine		RRAP	Receive Result Appeal from Prefecture
	SF	Send Fine		NRAO	Notify Result Appeal to Offender
	IFN	Insert Fine Notification		SAP	Send Appeal to Prefecture
	IDAP	Insert Date Appeal to Prefecture		P	Payment
	AP	Add Penalty		AJ	Appeal to Judge

(a) Overview activity abbreviations

$$\begin{aligned}
 A = \{ & \langle CF, SF, IFN, IDAP, SAP, AP, RRAP, NRAO, AJ \rangle, \\
 & \langle CF, SF, IFN, IDAP, SAP, RRAP, AP, NRAO, AJ \rangle, \\
 & \langle CF, SF, IFN, IDAP, SAP, AP, RRAP, NRAO, AJ, P \rangle \} \\
 \sigma_{next} = & \langle CF, SF, IFN, AP, P, P \rangle
 \end{aligned}$$

(b) Previously added traces  $A$  and trace to be added next  $\sigma_{next}$

Figure 7.20: Contextual information for the illustrative example using the road traffic fine management log [56]

Figure 7.21 presents various information about the example. Figure 7.20a lists the activity labels, their corresponding color coding, and their abbreviations. Figure 7.20b shows the previously added traces  $A$  and trace  $\sigma_{next}$  that is about to be incrementally added. The initial process tree that we use for IPD is depicted in Figure 7.21a. Note

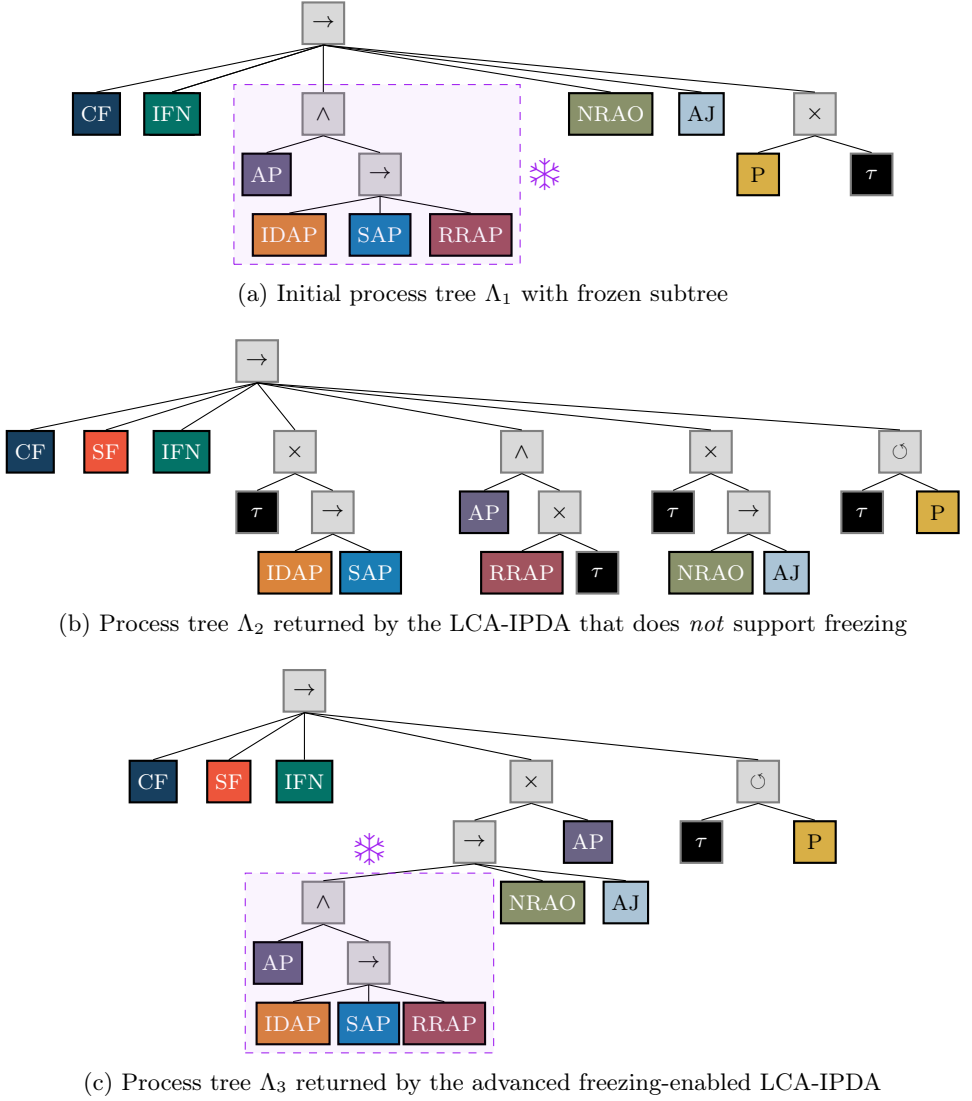


Figure 7.21: Discovered process models using a non-freezing-enabled (i.e., LCA-IPDA, cf. Section 5.3.2) and freezing-enabled IPDA (i.e., freezing-enabled LCA-IPDA, cf. Section 7.3) using the traces as specified in Figure 7.20

that this tree supports all traces specified in  $A$ . However, the tree does not support  $\sigma_{next}$  because the tree does not allow to execute activity P twice, and activities IFN, IDAP, SAP, and RRAP must always be executed, although these are not presented in  $\sigma_{next}$ .

Figure 7.21b shows tree  $\Lambda_2$  obtained when invoking the LCA-IPDA (cf. Section 5.3.2).

Note that the LCA-IPDA is not freezing-enabled. Therefore, the LCA-IPDA ignores that the highlighted subtree in  $\Lambda_1$  is to be considered as frozen (cf. Figure 7.21a). We observe that  $\Lambda_2$  does not contain the frozen subtree from  $\Lambda_1$ . Thus, if a user wanted to preserve this subtree during IPD, the user must undo the last iteration or manually edit  $\Lambda_2$ .

In contrast, Figure 7.21c shows process tree  $\Lambda_3$  obtained when invoking the freezing-enabled LCA-IPDA (cf. Section 7.3). As guaranteed by the freezing-enabled LCA-IPDA, the frozen subtree  $\wedge(AP, \rightarrow(IDAP, SAP, RRAP))$  from  $\Lambda_1$  is preserved in  $\Lambda_3$ . Moreover, tree  $\Lambda_3$  is more precise than  $\Lambda_2$  and might, therefore, be favored. Tree  $\Lambda_2$  allows to skip activities IDAP, SAP, RRAP, NRAO, AJ, and P simultaneously. In  $\Lambda_2$ , the choice between executing AP or the subtree containing activities AP, IDAP, SAP, RRAP, NRAO, and AJ must be made. Thus,  $\Lambda_2$  allows for much more behavior, which is not part of  $A \cup \{\sigma_{next}\}$ , than  $\Lambda_3$ . In summary, this example illustrates how freezing submodels can improve precision. Besides improving precision, freezing allows users to steer the incremental discovery by restricting the potential solution space.

## 7.6. Conclusion

This chapter extended the IPD framework introduced in Section 5.1, cf. Figure 5.1 (page 110). Freezing subtrees within IPD offers unique opportunities for users to steer the process discovery. While the IPD framework and the extensions introduced in Sections 5.1 and 6.2 focus on the process behavior, i.e., the event data, as the central input through which users control or influence the process discovery, the freezing extension offers a different dimension of influence possibilities by users. In short, the freezing extension allows freezing of subtrees in the input process model that are not altered by the freezing-enabled IPDA when adding a new trace  $\sigma_{next}$  to the input model. Thus, freezing allows users to influence the process model returned by the freezing-enabled IPDA.

The proposed freezing extension offers several directions for future work. For example, a recommendation system could suggest to users which parts of the tree discovered so far are worth preserving and should therefore be frozen. Further, other definitions of freezing-enabled IPDAs are conceivable. At the moment (cf. Definition 7.1 on page 181), trace  $\sigma_{next}$  is guaranteed to fit the eventually returned process tree. If a frozen subtree causes deviations between the current process tree and  $\sigma_{next}$ , the tree parts outside the frozen subtree are modified to support the trace. In such a case, however, one could alternatively involve users and provide them the feedback that a frozen subtree causes a deviation between the selected trace  $\sigma_{next}$  and the entire tree. Next, it would be up to the users to decide whether:

1. the tree should remain frozen and the problem should be solved outside the frozen tree (as presented in this chapter),
2. the tree should be defrosted to allow its manipulation,
3. only deviations between trace  $\sigma_{next}$  and the tree that do not affect the frozen subtree are resolved.

The third option, however, implies that  $\sigma_{next}$  is only partly supported by the resulting tree. In short, freezing subtrees within IPD opens various opportunities for future research.





## Part III.

# Facilitating Interaction with Event Data



---

# Chapter 8.

## Defining & Visualizing Variants

---

This chapter is largely based on the following published work.

- D. Schuster, F. Zerbato, S. J. van Zelst, and W. M. P. van der Aalst. *Defining and visualizing process execution variants from partially ordered event data*. Information Sciences, 657:119958, 2024. doi:10.1016/j.ins.2023.119958 [188]
- D. Schuster, L. Schade, S. J. van Zelst, and W. M. P. van der Aalst. *Visualizing trace variants from partially ordered event data*. In J. Munoz-Gama and X. Lu, editors, Process Mining Workshops, volume 433 of Lecture Notes in Business Information Processing, pages 34–46. Springer, 2022. doi:10.1007/978-3-030-98581-3\_3 [183]

Event logs generally include many process executions, i.e., individual cases, each containing several events describing the execution of activities. To handle these large volumes of event data, *process execution variants* (hereinafter referred to simply as *variants*) are key abstractions widely used within process mining techniques and practices. Recall that variants are also an essential abstraction in (incremental) process discovery. Variants describe process executions whose activities share identical order relations. Thus, a one-to-many relationship between variants and individual process executions exists, i.e., one variant may summarize various cases.

Analyzing variants allows process analysts to comprehend differences among process executions based on their control flow or performance [203]. Further, analysts can filter event data based on variants satisfying specific conditions [236], for instance, control flow constraints over activities. During exploratory event data analysis, *variant visualizations* aid in understanding the occurrence and order of frequent activity patterns and the level of process structuredness [252]. Such insights help analysts to make informed decisions about the analyses performed on a specific event log [207]. In short, variants and corresponding visualizations are essential for successfully applying process mining.

Variants are primarily defined for traces as formally introduced in Definition 3.20 on page 61), i.e., sequentially-aligned respectively *totally-ordered* activities that are consid-

ered to be atomic [211].<sup>1</sup> Two traces that correspond to the same simplified trace (cf. Definition 3.21 on page 61), i.e., their sequences of activity labels is identical, correspond to the same variant. Thus, variants for totally-ordered, time-point-based events are sequences of activity labels. Figure 8.1 shows in the top right a widely used variant visualization consisting of totally-ordered, time-point-based activities represented as a sequence of colored chevrons, which can be found in diverse process mining tools [26, 234].

In literature, the terms *parallel* and *concurrent* are distinguished. “A system is said to be concurrent if it can support two or more actions in progress at the same time. A system is said to be parallel if it can support two or more actions executing simultaneously.” [37,

216

p. 3] The key difference is the term *in progress*. Multiple actions can be in progress at the same time but this does not imply they are also executed simultaneously. Note that the definition of *parallel* is consistent with the interpretation taken in this thesis, i.e., two activities are being considered parallel if their execution overlaps in time. Further, this thesis does not distinguish if an activity is *in progress* or actually *executed*.

Recall the event log shown in Table 3.1 (page 59); events 8247 and 8248 overlap in their execution if considering the timestamp and the duration.<sup>2</sup> Formally, two events  $e_1, e_2 \in \mathcal{E}$  overlap if the following condition is met.

$$\{x \mid e_1^t \leq x \leq e_1^t + e_1^d\} \cap \{x \mid e_2^t \leq x \leq e_2^t + e_2^d\} \neq \emptyset$$

Recall that  $e^t, e^d \in \mathbb{R}_{\geq 0}$  (cf. Definition 3.17 on page 60) for any event  $e \in \mathcal{E}$ . Moreover, recall that if the duration of an event is greater than zero, i.e.,  $e^d > 0$ , we consider the timestamp  $e^t$  as the activity's execution start time point. We consider two events parallel if they overlap in time, as described above. Since each event describes the execution of an activity, we will also refer to parallel events as parallel activities in the following.

Parallel activities are a common phenomenon in event logs and, hence, in real-life processes. However, variants based on totally-ordered activities lack the expressiveness to capture parallel activities adequately. For instance, different activities within a process execution having identical timestamps must be sequentialized<sup>3</sup>. When process mining tools and algorithms enforce sequentialization, it can lead to inaccurate conclusions by analysts who may not be aware of the imposed order. Such imposed order relations can result in flawed analysis and decision-making.

In [219], variants consisting of partially ordered, time-point-based activities were proposed; Figure 8.1 shows the corresponding chevron-based visualization of two example variants. These variants can capture parallel activities; however, they consider activities time-point-based, i.e., atomic. Consequently, these variants lack expressiveness regarding activities' execution durations, i.e., these variants ignore duration information as, for example, present in the event log excerpt depicted in Table 3.1 (page 59). Additionally, the authors introduce time granularity modification in the context of event data and variants [219].

This chapter considers partially ordered event data with *heterogeneous temporal information* about the executed activities. The term heterogeneous temporal information refers to activities that are either atomic, representing a time point [108], or spanning a time interval [109]. Recall the event log shown in Table 3.1 (page 59). For instance, the first event with ID 8245 has a timestamp but no duration information; thus, event 8245 represents a time point. In contrast, the event with ID 8246 has a timestamp and duration information; thus, it represents a time interval from 17.06.21 08:32:23 until 18.06.21 12:01:11 (corresponds to the stated duration of 1d, 3h, 28m, and 48s). We focus on both point-based and interval-based activity executions since the level of detail in recording activities can vary; heterogeneous temporal information is a regular phenomenon in event

<sup>2</sup>If an event contains duration information, we interpret the timestamp as the start of the activity's execution. Thus, such events  $e$  having a duration greater zero describe the time interval from  $e^t$  to  $e^t + e^d$ .

<sup>3</sup>Activities' timestamps are usually used as a first-order criterion. If this is insufficient because two events within a case have identical timestamps, a second-order criterion is needed. Recall the definition of traces Definition 3.20 (page 61) that uses the event ID as a second-order criterion.

data capturing actual processes [57, 228, 229]. We propose two definitions and visualizations for variants that comprise partially ordered time-point and interval-based activities. We refer to the proposed variants as high-level and low-level variants. Both definitions address different abstraction levels and thus complement each other.

In addition to defining high-level and low-level variants, we address changes in time granularity and their effects on these variants in this chapter. Defining an appropriate time granularity is critical to gaining valuable insights when analyzing event data from real-world processes. Most event data contain timestamps that are expressed in seconds or even milliseconds. While analyzing some processes requires this temporal precision, using these precise timestamps can be detrimental when analyzing other processes. For example, for specific processes, it may not matter whether one activity occurred a few seconds before another or vice versa; instead, process analysts are interested in whether these activities were performed on the same day and, therefore, want to consider these activities as parallel because they set the temporal *bottom granularity* as days.

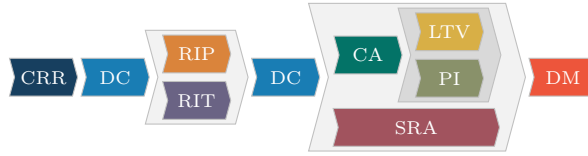
The subsequent sections are organized as follows. Section 8.1 introduces to the two proposed variants, i.e., high- and low-level variants, and highlights the connection between them. Subsequently, Section 8.2 formally introduces high-level variants, while Section 8.3 introduces low-level variants. Section 8.4 elaborates on the computation of the proposed variants. In Section 8.5, we discuss the impact of time granularity changes to the proposed variants, i.e., how do the proposed variants change when we switching to a coarser time granularity. Section 8.6 presents an evaluation consisting of automated experiments that focus on computational aspects and a user study focusing on usefulness and ease of use of the proposed variants. Finally, Section 8.7 concludes this chapter.

## 8.1. Overview

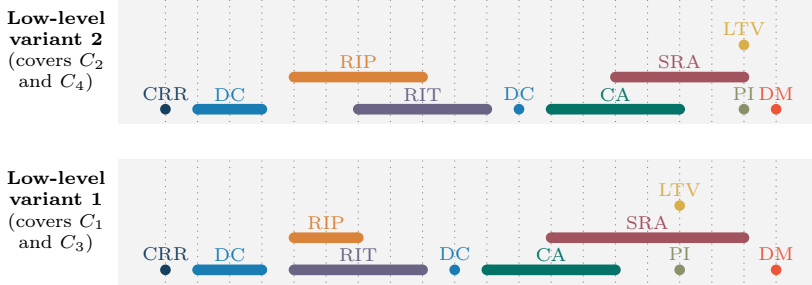
This section provides an overview on high-level and low-level variants. Consider Figure 8.2 showing an comprehensive overview of cases, low-level, and high-level variants. Figure 8.2c depicts four cases  $C_1, \dots, C_4$ . Each case comprises events representing a time point as well as event representing a time interval. Note that the cases shown in Figure 8.2c have a lower granularity of days, meaning that each time point or time interval is mapped to full days. Consider  $C_1$  shown in Figure 8.2c. Case  $C_1$  corresponds to the events with case ID 134 in the event log excerpt shown in Table 3.1 (page 59).<sup>4</sup>

Figure 8.2b shows two low-level variants. It is important to note that the x-axis provides no time information. Therefore, the length of the elements cannot be used to infer the duration of activities. Additionally, recall that the variants proposed in this chapter focus on the ordering relationships among activities. The first low-level variant covers  $C_2$  and  $C_4$ . Note that low-level variants are generally composed of circles and horizontally aligned bars. Each element represents an activity; circles represent time-point-based activities, while bars represent time-interval-based activities. For example, low-level variant 1 indicates that activity CRR is executed first, followed by activity DC. Next, activities RIP and RIT start simultaneously; however, activity RIP's execution completes while RIT is still being executed. Next, activity CA is executed, followed

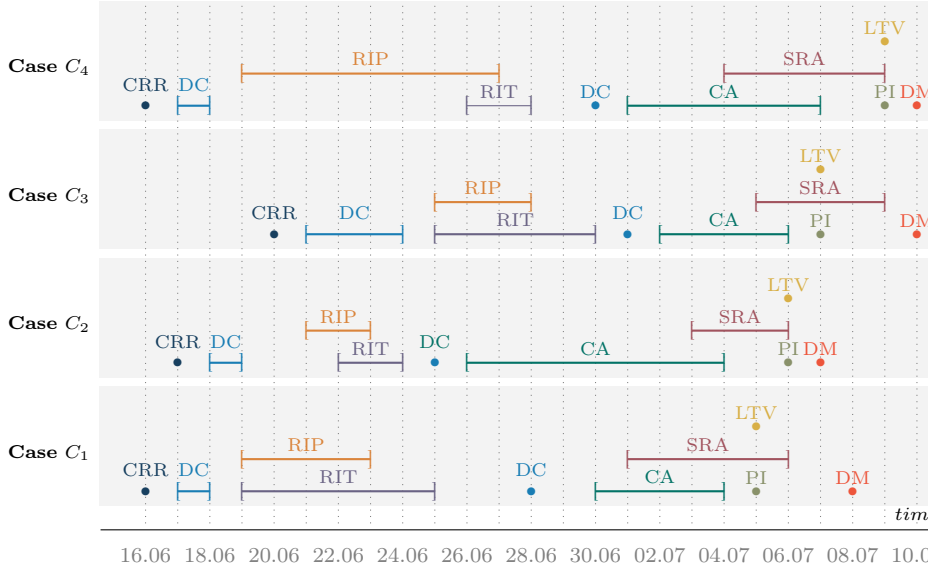
<sup>4</sup>From the timestamps of the events with case ID 134, only the year, month and day information is taken into account in Figure 8.2c.



(a) Variant level: *High-level variant* that comprises the behavior described by the two low-level variants (Figure 8.2b) and thus the four cases  $C_1, \dots, C_4$  (Figure 8.2c)



(b) Variant level: *Low-level variants* whose described process behavior comprises the cases in Figure 8.2c; Section 8.3 formally introduces low-level variants



(c) Case level: Visualization of four cases; each case consists of activities either representing a time point, represented by  $\bullet$ , or time interval, represented by  $[-]$

Figure 8.2: Example of four cases and corresponding high- and low-level variants from a mortgage application process (partly adapted from [188, Figure 3])

by activity CA. While activity CA is still being executed, the execution of activity SRA starts. Next, the execution of activity CA is completed. While SRA is still being executed, activities LTV and PI are executed in parallel. Afterwards, the execution of SRA is completed, and eventually, activity DM is executed. In summary, low-level variants indicate the different relations among activities that represent time-points as well as time-intervals. Further, low-level variants distinguish different relations between time-intervals that are overlapping, similar to *Allen's interval algebra* [11], which generally defines potential relations for intervals.

Figure 8.2a shows a high-level variant. As with low-level variants, it is impossible to infer the duration of the activities from high-level variants, i.e., the x-axis is not to be understood as a time axis. High-level variants address a higher level of abstraction than low-level variants. High-level variants are visualized employing chevrons, cf. Figure 8.2a. When the executions of activities overlap, the chevrons representing those activities are aligned vertically. Activities whose executions do not overlap are aligned horizontally. The visualized high-level variant covers all for cases  $C_1, \dots, C_4$  (cf. Figure 8.2c). Thus, the single high-level variant also covers the two low-level variants. The illustrated high-level variant indicates that all cases start with activity CRR followed by DC. Subsequently, activities RIP and RIT are executed in parallel, followed by activity DC. Next, activity SRA is executed. In parallel to SRA, activity CA is first executed, followed by activities LTV and PI, both executed in parallel to SRA. Eventually, activity DM is executed. Compared to the two low-level variants, the high-level variant does not indicate, for example, the exact relation between activities RIP and RIT; both activities are shown in parallel, cf. Figure 8.2a. In contrast, from the two low-level variants we observe, for example, that the execution of activity RIT in all four cases  $C_1, \dots, C_4$  exceeds the execution of activity RIP. In short, high-level variants provide a greater abstraction in terms of order relationships between activities compared to low-level variants.

Generally, a one-to-many relation between high-level and low-level variants exists, as exemplified in Figure 8.1. Further, a one-to-many relation between variants and cases exists. Both proposed variants, i.e., low-level and high-level ones, generalize existing variant definitions, i.e., variants for totally-ordered time-point-based activities as well as for partially-ordered time-point-based activities as shown in Figure 8.1. If an event log contains only events that represent time points, and for each case, all associated events can be sequentially aligned based on their timestamps alone, then both high-level and low-level variants for that event log correspond to the variants for totally-ordered time-point-based activities shown in Figure 8.1.<sup>5</sup> Similarly, if an event log includes only cases consisting of time-point-based activities that may be parallel, i.e., partially ordered. For such logs, the visualization of variants assuming partially ordered time-point-based activities (cf. Figure 8.1) equals the visualization of corresponding high-level variants. Further, the visualization of low-level variants contains the identical information as the visualization of variants for partially ordered time-point-based activities (cf. Figure 8.1). In conclusion,

<sup>5</sup>The visualization of high-level variants for such an event log is identical to variants assuming totally ordered time point-based activities. The visualization of low-level variants for such an event log would of course not be identical to those for totally-ordered time-point-based activities, because low-level variants use circles and bars instead of chevrons, but the information contained in the low-level variant visualization is identical to the chevron-based visualization of variants for totally-ordered time-point-based activities.



the proposed high-level and low-level variants and corresponding visualizations generalize existing variant definitions and visualizations, which focus on time-point-based activities.

## 8.2. High-Level Variants

Following, Section 8.2.1 defines the *high-level case view* that is a labeled partially ordered set over the elements contained in a case. The high-level case view is utilized in Section 8.2.2 that introduces the calculation and visualization of high-level variants.

### 8.2.1. High-Level Case View

As exemplified in Figure 8.2, high-level variants consider two activities in parallel if their execution somehow overlaps in time, i.e., the exact overlap is irrelevant, for example, compared to low-level variants. If the executions of two activities do not overlap in time, these activities are considered sequentially ordered.

For example, recall case  $C_1$ . Figure 8.3a illustrates the contained activities on a time-axis. Figure 8.3b illustrates the *high-level case view*, i.e., a partially ordered set over the events from  $C_1$ . Whenever two events overlap in time, these events are unrelated to each other in the high-level case view. For instance, activities SRA, LTV, and PI are all unrelated to each other, cf. Figure 8.3b. Below we define the high-level case view.

**Definition 8.1** (High-level case view)

Let  $C \subseteq \mathcal{E}$  be a case. The *high-level case view* of  $C$  is a labeled, strict partially ordered set over the events in  $C$ , i.e.,  $(C, \prec^{HL}, \Sigma, \lambda)$  with

- $e_1 \prec^{HL} e_2$  iff  $e_1^t + e_1^d < e_2^t$  for arbitrary  $e_1, e_2 \in C$
- $\Sigma = \{e^a \mid e \in C\} \subseteq \mathcal{A}$
- $\lambda(e) = e^a$  for  $e \in C$

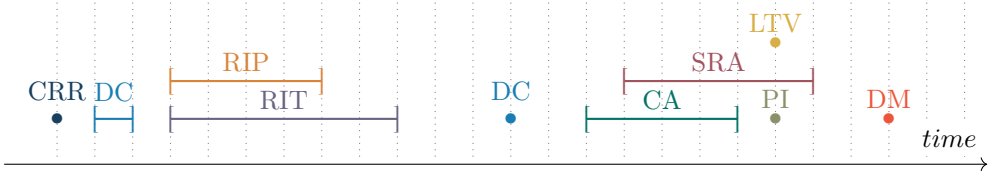
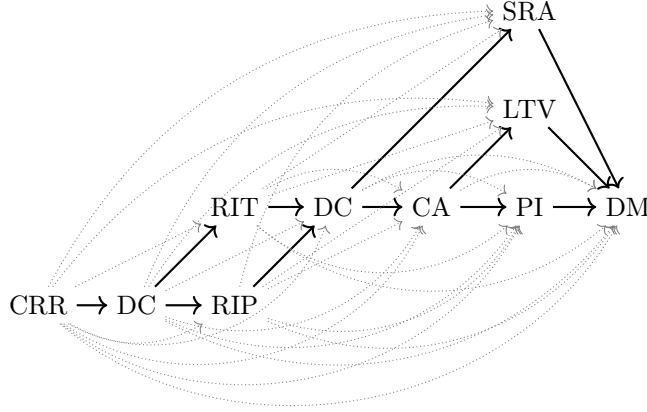
Assume an event log  $L$  containing multiple cases. Two cases are covered by the same high-level variant if their high-level case views, i.e., labeled ordered sets, are *isomorphic*, cf. Definition 3.7 on page 51.

**Definition 8.2** (High-level variant)

Let  $C_1, \dots, C_n \in \mathcal{C}$  be arbitrary cases. Let  $(C_1, \prec_1^{HL}, \Sigma_1, \lambda_1), \dots, (C_n, \prec_n^{HL}, \Sigma_n, \lambda_n)$  be correspondingly high-level case views (cf. Definition 8.1). We say that cases  $C_1, \dots, C_n$  belong to one high-level variant iff

$$(C_1, \prec_1^{HL}, \Sigma_1, \lambda_1) \cong \dots \cong (C_n, \prec_n^{HL}, \Sigma_n, \lambda_n).$$

Recall the four cases  $C_1, \dots, C_4$  shown in Figure 8.2c. The high-level case views of these four cases are all isomorphic to each other. Thus, these cases belong to the same *high-level variant*, cf. Definition 8.2.

(a) Case  $C_1$  as shown in Figure 8.2c(b) Graph representation of  $(C_1, \prec^{HL}, \Sigma, \lambda)$ ; each node represents an event and is labeled according to the activity label, solid arcs represent the transitive reduction, and solid together with dotted arcs represent the transitive closureFigure 8.3: Case  $C_1$  and its corresponding high-level case view (partly adapted from [188, Figure 4])

### 8.2.2. Calculation & Visualization of High-Level Variants

In this section, a visualization for high-level variants is proposed and a corresponding layout algorithm is presented. The input to the visualization algorithm is a high-level case view, i.e., a labeled ordered set according to Definition 8.1. The visualization approach returns a chevron-based visualization as shown in Figure 8.2a (page 219). We selected chevrons to represent activities because of their widespread use in process mining. Though alternative geometric shapes to chevrons are possible.

Subsequently, we define sequential and parallel partitions for labeled ordered sets. These partitions are necessary for calculating the variants. For a sequential partition holds that elements of two different subsets are related to each other.

**Definition 8.3** (Sequential partition of a labeled ordered set)

Let  $(X, \triangleleft, \Sigma, \lambda)$  be an arbitrary labeled ordered set.<sup>a</sup> Further, let  $X_1, \dots, X_n$  be a partition of set  $X$ .<sup>b</sup> We refer to the ordered sets  $(X_1, \triangleleft_1, \Sigma_1, \lambda_1), \dots, (X_n, \triangleleft_n, \Sigma_n, \lambda_n)$  as a sequential partition if the following conditions are satisfied.

- $n > 1$
- $\forall 1 \leq i < j \leq n \ \forall x \in X_i \ \forall x' \in X_j \ (x \triangleleft x')$
- $\forall 1 \leq i \leq n \ \forall x, x' \in X_i \ (x \triangleleft_i x' \Leftrightarrow x \triangleleft x')$

<sup>a</sup>Recall that the symbol  $\triangleleft$  represents an arbitrary order.

<sup>b</sup>Thus, (1)  $X_1, \dots, X_n \subseteq X$ , (2)  $X_1 \cup \dots \cup X_n = X$ , and (3)  $\forall 1 \leq i < j \leq n \ (X_i \not\subseteq X_j \wedge X_j \not\subseteq X_i)$ .

Analogous to sequential partitions for labeled ordered sets, we define parallel partitions. For a parallel partition, it holds that elements of two different subsets are unrelated.

**Definition 8.4** (Parallel partition of a labeled ordered set)

Let  $(X, \triangleleft, \Sigma, \lambda)$  be an arbitrary labeled ordered set. Further, let  $X_1, \dots, X_n$  be a partition of set  $X$ . We refer to the ordered sets  $(X_1, \triangleleft_1, \Sigma_1, \lambda_1), \dots, (X_n, \triangleleft_n, \Sigma_n, \lambda_n)$  as a parallel partition if the following conditions are satisfied.

- $n > 1$
- $\forall 1 \leq i < j \leq n \ \forall x \in X_i \ \forall x' \in X_j \ (\neg (x \triangleleft x') \wedge \neg (x' \triangleleft x))$
- $\forall 1 \leq i \leq n \ \forall x, x' \in X_i \ (x \triangleleft_i x' \Leftrightarrow x \triangleleft x')$

Next, we define the notion of *maximality* for sequential and parallel partitions of labeled ordered sets. Given a sequential or parallel partition  $(X_1, \triangleleft_1, \Sigma_1, \lambda_1), \dots, (X_n, \triangleleft_n, \Sigma_n, \lambda_n)$  of a labeled ordered set  $(X, \triangleleft, \Sigma, \lambda)$ . The sequential/parallel partition  $(X_1, \triangleleft_1, \Sigma_1, \lambda_1), \dots, (X_n, \triangleleft_n, \Sigma_n, \lambda_n)$  is called *maximal* iff there exists no  $m > n$  such that  $(X_1, \triangleleft_1, \Sigma_1, \lambda_1), \dots, (X_m, \triangleleft_m, \Sigma_m, \lambda_m)$  is a sequential/parallel partition, too.

Subsequently, we present formal properties of parallel and sequential partitions. Finally, we derive that the proposed recursive partitioning of labeled ordered sets is deterministic, i.e., there are no order effects.

**Lemma 8.2.1** (Sequential and parallel partitions cannot coexist). *Let  $(X, \triangleleft, \Sigma, \lambda)$  be an arbitrary labeled strictly partially ordered set a sequential and a parallel partition cannot coexist, i.e., at most one of the two exists.*

*Proof.* We proof Lemma 8.2.1 by contradiction. Let  $(X, \triangleleft, \Sigma, \lambda)$  be an arbitrary labeled strictly partially ordered set with a sequential partition  $(X_1, \triangleleft_1, \Sigma_1, \lambda_1), \dots, (X_n, \triangleleft_n, \Sigma_n, \lambda_n)$  with  $n > 1$ . Assume there also exists a parallel partition  $(X'_1, \triangleleft'_1, \Sigma'_1, \lambda'_1), \dots, (X'_m, \triangleleft'_m, \Sigma'_m, \lambda'_m)$  with  $m > 1$ . For  $1 \leq j \leq m$  and  $1 \leq i \leq n$ , assume that for an arbitrary  $x_1 \in X$  it holds that  $x_1 \in X'_j$  and  $x_1 \in X_i$ . Since a sequential partition exists,

we know that:

$$\forall x_2 \in X_{i+1} \cup \dots \cup X_n (x_1 \prec x_2)$$

and

$$\forall x_3 \in X_1 \cup \dots \cup X_{i-1} (x_3 \prec x_1).$$

Since a parallel partition exists, all  $x_2, x_3$  as above-specified are contained in  $X'_j$ . Hence,

$$\{x\} \cup X_1 \cup \dots \cup X_{i-1} \cup X_{i+1} \cup \dots \cup X_n \subseteq X'_j.$$

Since

$$\forall x_3 \in X_1 \cup \dots \cup X_{i-1} \forall x_2 \in X_{i+1} \cup \dots \cup X_n (x_3 \prec x_2 \wedge x_3 \prec x \wedge x \prec x_2)$$

it follows from Definition 8.4 that

$$X'_j = X_1 \cup \dots \cup X_n = X.$$

Hence,

$$\forall k \in \{1, \dots, m\} \setminus \{j\} (X'_k = \emptyset)$$

because  $X'_1, \dots, X'_m$  is a partition of  $X$ . This contradicts the assumption that there exists a parallel partition with  $m > 1$ . The other direction of this proof is symmetrical.  $\square$

**Lemma 8.2.2** (Uniqueness of maximal sequential partitions). *Let  $(X, \prec, \Sigma, \lambda)$  be an arbitrary labeled strictly partially ordered set. If a maximal sequential partition  $(X_1, \prec_1, \Sigma_1, \lambda_1), \dots, (X_n, \prec_n, \Sigma_n, \lambda_n)$  with  $n > 1$  exists, this sequential partition is unique.*

*Proof.* We proof Lemma 8.2.2 by contradiction. Let  $(X, \prec, \Sigma, \lambda)$  be an arbitrary labeled strictly partially ordered set. Assume this set has two maximal sequential partitions  $(X_1, \prec_1, \Sigma_1, \lambda_1), \dots, (X_n, \prec_n, \Sigma_n, \lambda_n)$  and  $(X'_1, \prec'_1, \Sigma'_1, \lambda'_1), \dots, (X'_n, \prec'_n, \Sigma'_n, \lambda'_n)'$  with  $n > 1$  that are not identical. Hence,

$$\exists 1 \leq i \leq n \forall 1 \leq j \leq n (X_i \neq X'_j).$$

In particular,  $X_i \neq X'_i$ . Hence,

$$\exists x \in X_i \cup X'_i \left( \left( x \in X_i \wedge x \notin X'_i \right) \vee \left( x \notin X_i \wedge x \in X'_i \right) \right).$$

Assume  $x \in X_i \wedge x \notin X'_i$  holds. Note that the other case is symmetric. Hence,

$$x \in X'_1 \cup \dots \cup X'_{i-1} \cup X'_{i+1} \cup \dots \cup X'_n = X \setminus X'_i.$$

In the following we make a case distinction.

1. Assume  $x \in X'_1 \cup \dots \cup X'_{i-1}$ . According to Definition 8.3  $\forall x' \in X_i (x \prec x')$ .
2. Assume  $x \in X'_{i+1} \cup \dots \cup X'_n$ . According to Definition 8.3  $\forall x' \in X_i (x' \prec x)$ .

Since  $x \in X_i$  it follows in both above-described cases that  $x \prec x$ . This contradicts our assumption that  $(X, \prec, \Sigma, \lambda)$  is an arbitrary labeled strictly partially ordered set because irreflexibility is not satisfied.  $\square$

Finally, we show that maximal parallel partitions are unique.

**Lemma 8.2.3** (Uniqueness of maximal parallel partitions). *Let  $(X, \prec, \Sigma, \lambda)$  be an arbitrary labeled strictly partially ordered set. If a maximal parallel partition  $(X_1, \prec_1, \Sigma_1, \lambda_1), \dots, (X_n, \prec_n, \Sigma_n, \lambda_n)$  with  $n > 1$  exists, this parallel partition is unique.*

*Proof.* Note that when representing  $(X, \prec, \Sigma, \lambda)$  as a graph, a maximal parallel partition corresponds to its connected components. The set of connected components of a graph is unique by definition.  $\square$

From Lemmas 8.2.2 and 8.2.3 we know that maximal partitions, both sequential and parallel, are unique. Moreover, Lemma 8.2.1 states that at most either a sequential or parallel partition exists. Recall that we exclude trivial partitions that contain only one set by requesting  $n > 1$ , cf. Definitions 8.3 and 8.4. In the following, we always assume maximum partitions when referring to sequential or parallel partitions.

The layout approach for high-level variants recursively partitions a given high-level case view by applying sequential and parallel partitions (cf. Definitions 8.3 and 8.4). The given high-level case view is recursively partitioned until no partition can be found anymore. Each partition of size  $n$  results in a set of  $n$  chevrons. Further, the partition type indicates the positioning of chevrons. Parallel partitions result in vertically aligned chevrons, while sequential partitions result in horizontally aligned chevrons. Through the recursive partitioning, a hierarchical structure emerges, i.e., a tree structure.

Consider Figure 8.4 showing the recursive layout approach for the high-level variant depicted in Figure 8.3a covering the four cases  $C_1, \dots, C_4$ . First, a sequential partitioning of size  $n = 6$  is found, cf. top left in Figure 8.4. Thus, we draw six chevrons horizontally aligned and add the activity labels to the chevrons, cf. intermediate high-level variant visualization (I) in Figure 8.4. If a chevron contains only one activity label, we color that chevron to provide better distinguishability between activities. Next, we recursively try to find a parallel partitioning in all sets containing more than one element.<sup>6</sup> We find a parallel partition of size two in the set containing activities RIT and RIP; moreover, we find a parallel partition of size two in the set containing activities SRA, CA, LTV, and PI. Both parallel partitions result in two vertically aligned chevrons each, cf. visualization (II) in Figure 8.4. At this recursion level, only one subset with cardinality greater one remains, i.e., the subset containing activities CA, LTV and PI. This subset can be sequentially partitioned into two subsets. Intermediate visualization (III) shows the corresponding two sequentially aligned chevrons. Finally, a parallel partition is found in the last recursion level in the subset containing activities LTV and PI. The final high-level variant visualization is depicted in the lower right, cf. variant visualization IV in Figure 8.4. Figure 8.5 depicts the corresponding tree structure based on the recursive partitioning by applying sequential, visualized in Figure 8.5 by symbol  $\Rightarrow$ , and parallel partitions, visualized by symbol  $\Downarrow$ .

In short, a given high-level case view is recursively partitioned by alternately applying sequential and parallel partitioning. The recursion stops as soon as no more partitioning can be applied. Each partitioning of size  $n$  leads to  $n$  chevrons, which are horizontally aligned if it is a sequential partitioning or vertically if it is a parallel partitioning.

<sup>6</sup>Note that after finding a maximal sequential partition, we cannot find a further sequential partition in one of the subsets because this would imply that the original sequential partition is not maximal.

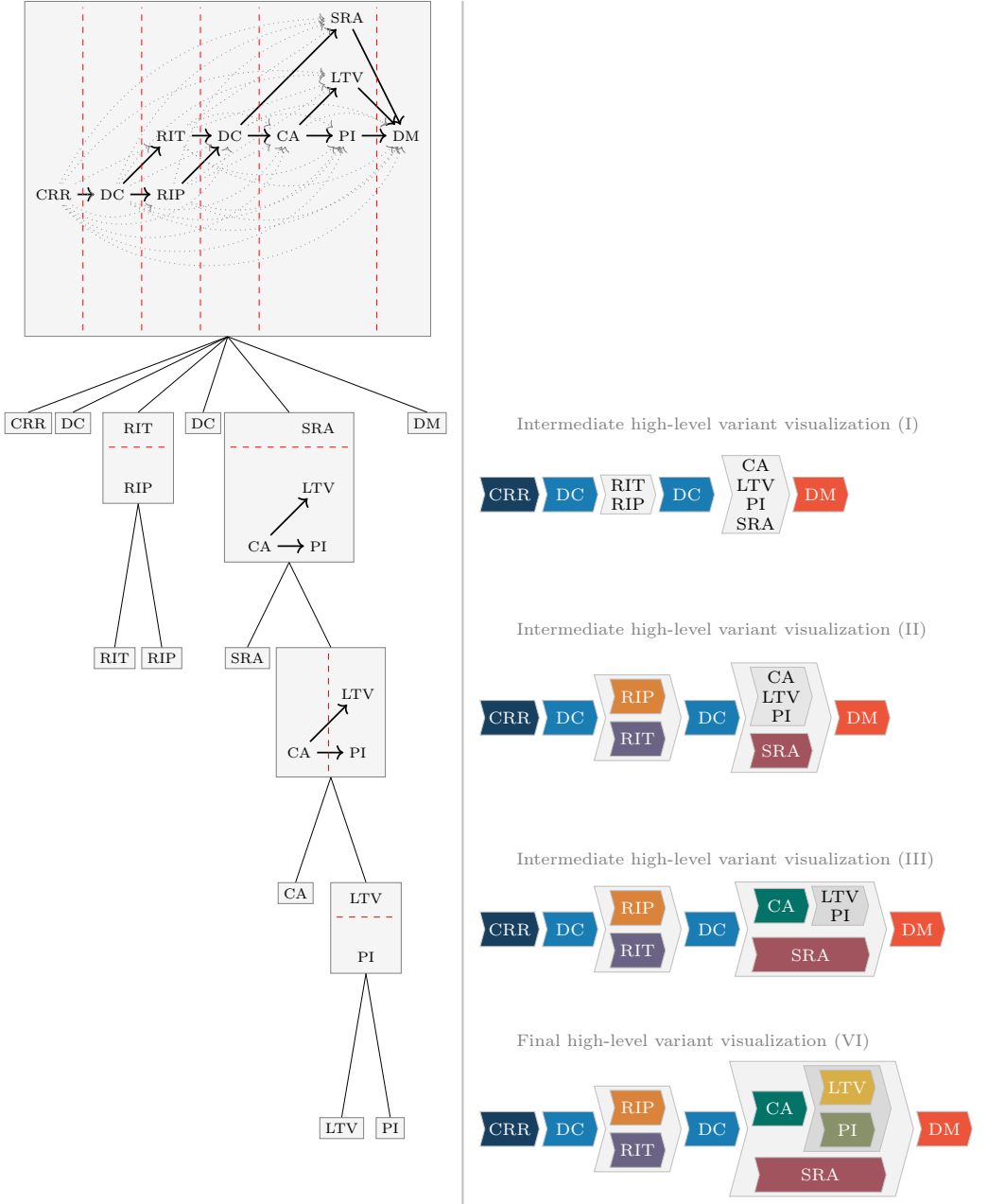


Figure 8.4: Recursive partitioning of  $(C_1, \prec^{HL}, \Sigma, \lambda)$ , horizontal red dashed lines symbolize parallel partitions and vertical ones sequential partitions (on the left side); the right side shows the corresponding (intermediate) variant visualization after each recursion level (partly adapted from [188, Figure 5])

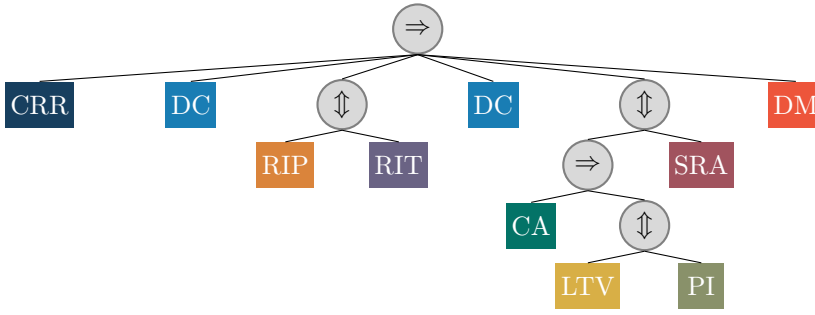


Figure 8.5: Final tree structure of the high-level variant calculated in Figure 8.4; symbol  $\Rightarrow$  represents a sequential partition and  $\Updownarrow$  a parallel partition

### 8.2.3. Limitations of the High-level Variant Visualization

In the shown example, cf. Figure 8.4, parallel and sequential partitions are recursively applied until all activities end up in singletons. However, in the general case, this does not hold, i.e., subsets with more than one activity may exist for which no further partition (cf. Definitions 8.3 and 8.4) can be found. For example, consider the case  $C_5$  depicted in Figure 8.6a. Note that case  $C_5$  is not part of the running example, cf. Figure 8.1. The corresponding high-level case view of  $C_5$  is shown in Figure 8.6b. Note that  $C_5$  is similar to  $C_1$  (cf. Figure 8.3 on page 222); therefore, their high-level case views are similar. In summary, the second DC activity represents a time interval overlapping the execution of activity CA, compared to  $C_1$  (Figure 8.3 on page 222).

Figure 8.7 illustrates the recursive calculation of the high-level variant visualization. In the first recursion level, a parallel partition for the subset containing activities RIT and RIP is found; however, we cannot find a partition for the subset containing activities CA, DC, PI, LTV, and SRA. Thus, the recursion stops for this subset. After applying the parallel partition in the second recursion level, only singletons and the subset for which we cannot find any partition remain. Thus, the layout algorithm stops. The high-level variant visualization shown in the lower right is eventually returned. The gray chevron with the activity labels DC, CA, SRA, LTV, and PI indicates that these activities happen in a not further specified order. Figure 8.8 shows the corresponding tree representation of the constructed high-level variant visualization. Note that we use the symbol  $\parallel$  to indicate that activities below can occur in any order, i.e., no order is specified. Further, note that the order of subtrees below nodes labeled  $\Updownarrow$  is irrelevant because parallel partitions do not impose any sequential order; thus, the order of subtrees does not matter. Similarly, the order of the activities below a node labeled  $\parallel$  does not matter because  $\parallel$  indicates that these activities can be executed in any order respectively in an unspecified order.

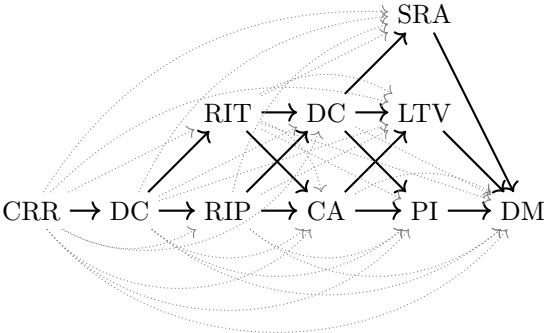
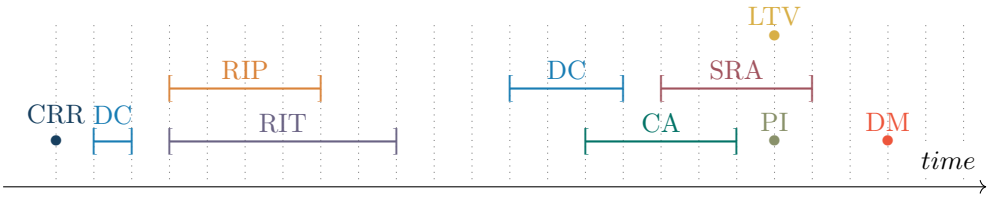


Figure 8.6: Case  $C_5$  and its corresponding high-level case view



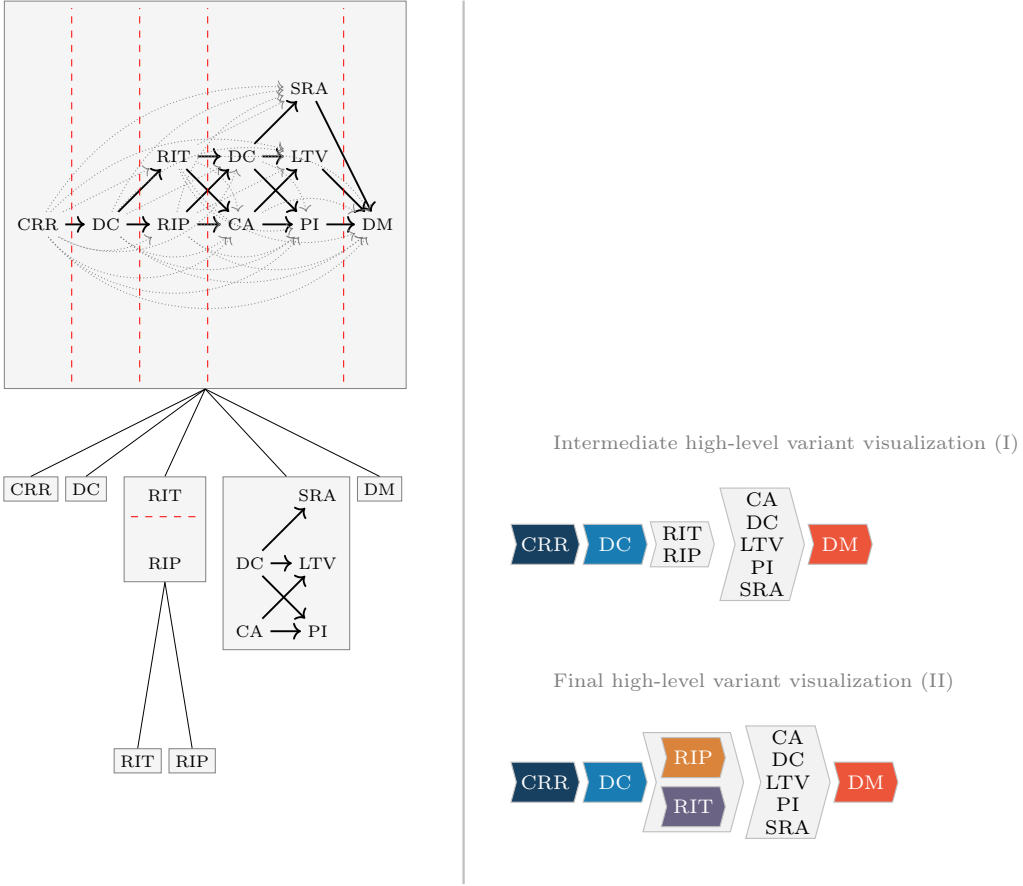


Figure 8.7: The left side shows the recursive partitioning of  $(C_5, \prec^{HL}, \Sigma, \lambda)$  where horizontal red dashed lines symbolize parallel partitions and vertical ones symbolize sequential partitions, while the right side shows after each recursion level the corresponding (intermediate) variant visualization

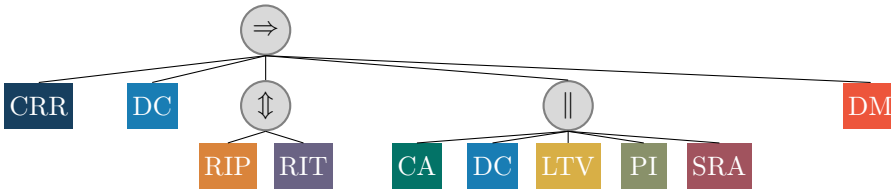


Figure 8.8: Final tree structure of the high-level variant calculated in Figure 8.7; symbol  $\parallel$  indicates that activities can happen in any order (i.e., no partition could be applied to the corresponding activities)

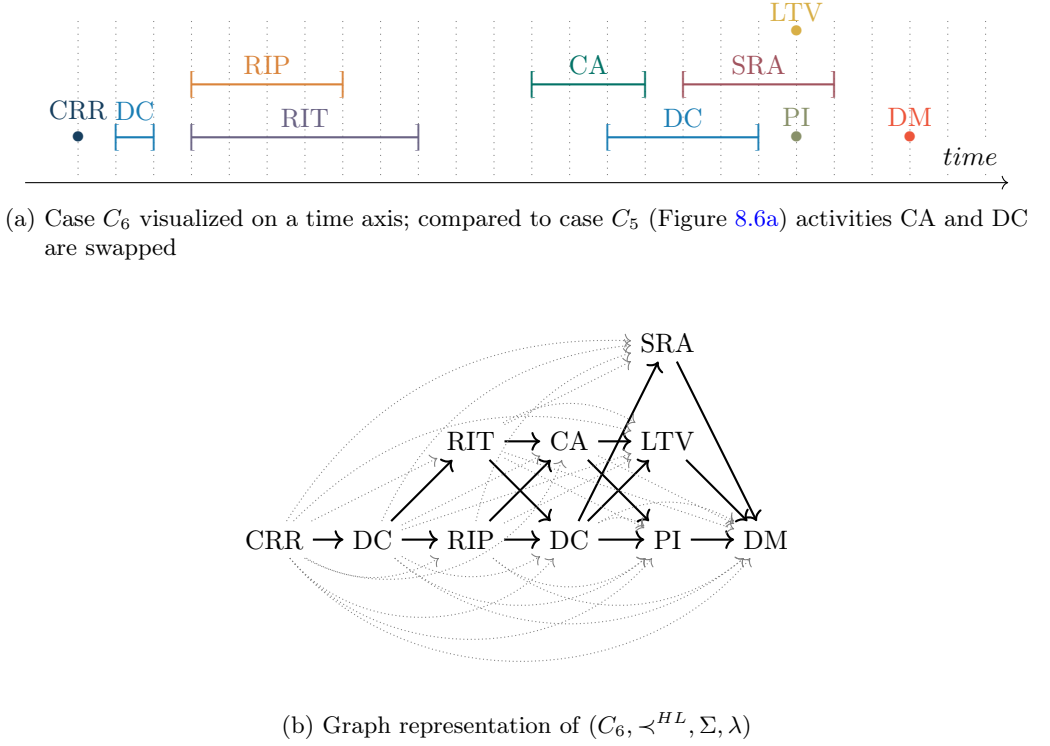


Figure 8.9: Case  $C_6$  and its corresponding high-level case view; compared to  $C_5$  (cf. Figure 8.9 on page 230), only CA and the second occurrence of DC are swapped

Consider case  $C_6$  depicted in Figure 8.9a (page 230) and its corresponding high-level case view shown in Figure 8.9b (page 230). Compared to case  $C_5$  (cf. Figure 8.6a), activities CA and DC are swapped. When applying the layout algorithm to the high-level case view of  $C_6$ , we yield the same variant visualization as for case  $C_5$ , cf. the variant visualization in the bottom right of Figure 8.7. Thus, although the high-level case views of case  $C_5$  and  $C_6$  differ, cf. Figures 8.6b and 8.9b, the corresponding visualization does not. However, we deliberately designed the high-level variant visualization in the manner presented to provide the most comprehensive and compact visualization possible that does not require any graph-like structures.

Given that two cases can have non-isomorphic high-level case views, for example, cases  $C_5$  and  $C_6$ , and at the same time having an identical high-level variant visualization implies that the definition of high-level variants can have two appearances.

1. Recall Definition 8.2 (page 221) specifying that two cases belong to the same high-level variant if their high-level case views are isomorphic. Applying this definition, a high-level variant visualization may actually cover multiple variants according to Definition 8.2 in certain scenarios.

2. Alternatively, we can refer to high-level variants as all cases that have the same visualization. Since we can represent each high-level variant visualization in a tree structure, consider Figure 8.8, we can specify that cases whose tree structure of the high-level variant visualization is isomorphic belong to the same variant. Note that the order of subtrees below inner nodes labeled  $\parallel$  or  $\updownarrow$  is irrelevant.<sup>7</sup> Only for subtrees of a node labeled  $\Rightarrow$ , which indicates a sequential partitioning, the order of the subtrees matters.

In the remainder of this chapter, we consider the second definition of high-level variants, i.e., two cases are covered by the same high-level variant if their visualization equals respectively the corresponding tree structures of their visualizations are isomorphic, recall that the order of subtrees below nodes labeled  $\parallel$  or  $\updownarrow$  does not matter.

## 8.3. Low-Level Variants

Following, Section 8.3.1 defines the *low-level case view* that is a labeled partially ordered set over the elements contained in a case. The low-level case view is utilized in Section 8.3.2 that introduces the calculation and visualization of low-level variants.

### 8.3.1. Low-Level Case View

As exemplified in Figure 8.1 (page 216), low-level variants provide a more detailed view on ordering relations among activities compared to high-level variants. Low-level variants aim to categorize the overlap of activity executions rather than simplistically considering any form of overlap of two activity executions as parallel.

For example, recall case  $C_1$ . Figure 8.10a illustrates the contained activities on a time-axis. Figure 8.10b illustrates the low-level case view of  $C_1$ , that is, a partially ordered set over the events that are split into two elements if their duration is greater than zero. For example, the first activity in time, i.e.,  $CRR$ , contains no duration information respectively the duration value is zero; thus, the activity is represented as a point in Figure 8.10a. Thus, the low-level case view contains only the element  $(CRR, \blacksquare)$ . In contrast, the second event in time, i.e., the execution of  $DC$ , contains duration information. Thus, we split this event into two elements, i.e.,  $(DC, \blacktriangleright)$  and  $(DC, \blacksquare)$ , and put these elements in relation to all other elements. For instance, Figure 8.10b shows that after the completion of the first  $DC$ , activities  $RIP$  and  $RIT$  are started simultaneously, i.e.,  $(RIP, \blacktriangleright)$  and  $(RIT, \blacktriangleright)$  are unrelated to each other.

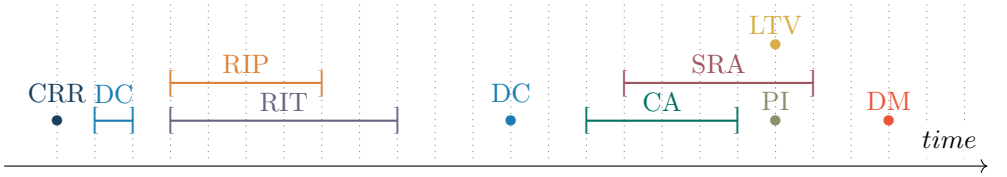
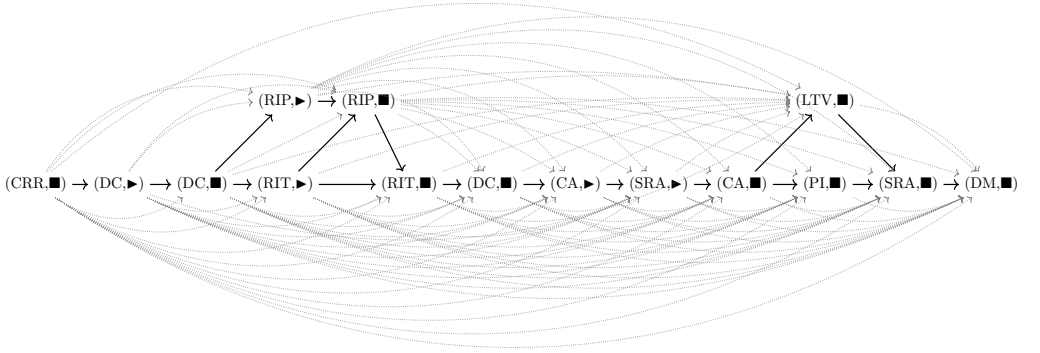
In general, for events  $e$  with duration information, we create two elements that represent the start  $(e, \blacktriangleright)$  and completion  $(e, \blacksquare)$ . For events  $e$  having no duration information, we create only one element representing its completion  $(e, \blacksquare)$ . Below, we formally define the low-level case view.

<sup>7</sup>Recall that the symbol  $\updownarrow$  indicates a parallel partition; thus, the order of subtrees does not matter. Similarly, this applies to the symbol  $\parallel$  that represents that no partition could be applied, and thus, the activities below can occur in any order. To fix the issue of potential different subtree orderings, one can enforce a lexicographical order for subtrees below nodes labeled  $\updownarrow$  or  $\parallel$ .

**Definition 8.5** (Low-level case view)

Let  $C \subseteq \mathcal{E}$  be a case. We define  $\tilde{C} = \{(e, \blacktriangleright) \mid e \in C \wedge e^d > 0\} \cup \{(e, \blacksquare) \mid e \in C\}$ . The low-level case view of case  $C$  is a labeled, strict partially ordered set over the set  $\tilde{C}$ , i.e.,  $(\tilde{C}, \prec^{LL}, \Sigma, \lambda)$  such that for arbitrary  $(e_1, x_1), (e_2, x_2) \in \tilde{C}$  it holds that  $(e_1, x_1) \prec^{LL} (e_2, x_2)$  iff

- $e_1^t < e_2^t$  if  $x_1 = \blacktriangleright \wedge x_2 = \blacktriangleright$
- $e_1^t < e_2^t + e_2^d$  if  $x_1 = \blacktriangleright \wedge x_2 = \blacksquare$
- $e_1^t + e_1^d < e_2^t$  if  $x_1 = \blacksquare \wedge x_2 = \blacktriangleright$
- $e_1^t + e_1^d < e_2^t + e_2^d$  if  $x_1 = \blacksquare \wedge x_2 = \blacksquare$

(a) Case  $C_1$  as shown in Figure 8.2c

(b) Graph representation of  $(C_1, \prec^{LL}, \Sigma, \lambda)$ ; each node represents either a full event, the start, or the end of an event and is labeled according to the activity label, solid arcs represent the transitive reduction, and solid together with dotted arcs represent the transitive closure

Figure 8.10: Case  $C_1$  and its corresponding low-level case view

Analogously to high-level variants, a low-level variant covers all cases whose low-level case views are isomorphic. Below, Definition 8.6 formalizes low-level variants.

**Definition 8.6** (Low-level variant)

Let  $C_1, \dots, C_n \in \mathcal{C}$  be arbitrary cases. Let  $(\tilde{C}_1, \prec_1^{HL}, \Sigma_1, \lambda_1), \dots, (\tilde{C}_n, \prec_n^{HL}, \Sigma_n, \lambda_n)$  be correspondingly low-level case views (cf. Definition 8.5). We say that cases  $C_1, \dots, C_n$  belong to one low-level variant iff  $(\tilde{C}_1, \prec_1^{HL}, \Sigma_1, \lambda_1) \cong \dots \cong (\tilde{C}_n, \prec_n^{HL}, \Sigma_n, \lambda_n)$ .

### 8.3.2. Calculation & Visualization of Low-Level Variants

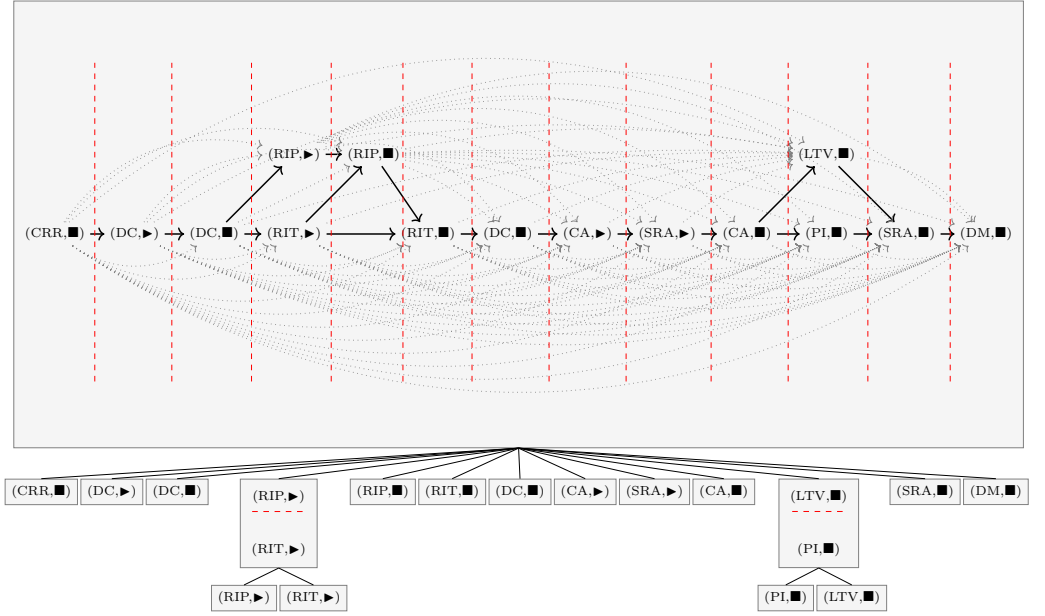
This section introduces a visualization for low-level variants and presents a corresponding layout algorithm. The input to the visualization approach is a low-level case view, i.e., a labeled ordered set according to Definition 8.5. The visualization approach returns a circle and bar-based visualizations as exemplified in Figure 8.2b (page 219).

As for visualizing high-level variants, we utilize sequential and parallel partitioning as specified in Definitions 8.3 and 8.4 (page 223). Consider Figure 8.11a showing the low-level case view  $(C_1, \prec^{LL}, \Sigma, \lambda)$  of  $C_1$ . As indicated, we find a sequential partition of size 13. As a result, two subsets remain that contain more than one element. In both subsets, we find a parallel partition of size two. After the second recursion level we stop since all subsets contain only one element. The obtained tree structure allows to place dots for each element, cf. Figure 8.11b. Sequential partitions indicate the horizontal alignment of these points, while parallel partitions indicate their vertical alignment.

Next, elements that correspond to the same activity are connected, cf. Figure 8.11c. Note that some elements remain alone, i.e., elements representing the execution of an activity with no duration information. For instance, element  $(\text{CRR}, \blacksquare)$  is not connected with another element as the underlying event contains no duration information and thus no element  $(\text{CRR}, \blacktriangleright)$  exists. After connecting elements, cf. Figure 8.11c, we change the labels of circles and bars to the activity name. As a final step, the gap width in which no activity is performed is reduced, cf. Figure 8.11d. For instance, compare the gap between CRR and DC in Figures 8.11c and 8.11d.

The resulting low-level variant visualization shown in Figure 8.11d contains much more information than the high-level variant covering case  $C_1$ , cf. Figure 8.2a. For instance, low-level variants distinguish between time-point-based activities, visualized as circles, and time-interval-based activities, visualized as bars. Further, the low-level variant shows, for example, that activities RIP and RIT start simultaneously; however, RIP ends while RIT is still being executed. As for high-level variants, the bars' length does not allow for concluding timing duration information. For example, although the bar representing CA is longer than the bar representing SRA, the conclusion cannot be made that the execution of activity CA is shorter than that of SRA. In short, low-level variants focus purely on ordering relationships among activities as high-level variants. However, low-level variants can visualize more ordering relations between activities being executed in parallel than high-level variants.

Finally, note that the proposed visualization of low-level variants is not affected by any scenario in which neither a sequential nor parallel partition can be applied during recursive partitioning. Since all elements in the low-level case view represent time points,



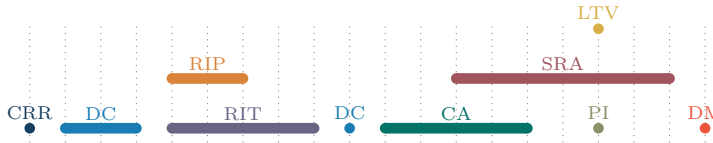
(a) Recursive partitioning of low-level case view  $(C_1, \prec^{LL}, \Sigma, \lambda)$  where horizontal red dashed lines symbolize parallel partitions and vertical ones sequential partitions



(b) 1<sup>st</sup> step: Placing start and end nodes according to the above-calculated partitioning



(c) 2<sup>nd</sup> step: Connecting nodes representing the start and completion of activities



(d) 3<sup>rd</sup> visualization step: Reduction of the gap width in which no activity is performed

Figure 8.11: Low-level variant visualization for  $C_1$  (partly adapted from [188, Figure 6])

there can always either a parallel or sequential partition be applied. In comparison, recall that elements of the high-level case view may represent time intervals or time points. Thus, the recursion might only stop for high-level variants before not all elements end up in singletons.

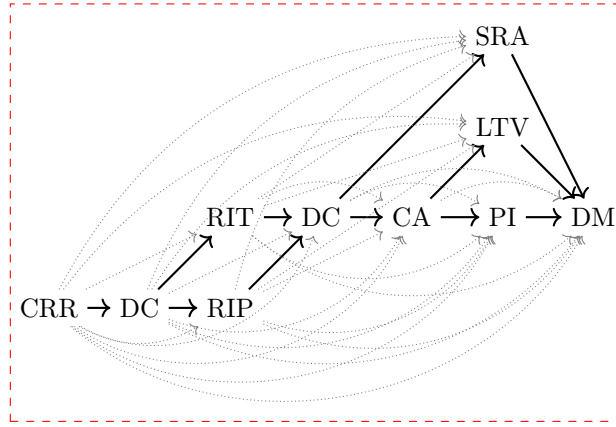
## 8.4. Computing High- & Low-Level Variants

This section briefly outlines the implementation of the high-level and low-level variant computation. Recall that the core of both variant types is the recursive partitioning as exemplified in Figures 8.4 and 8.11a, and the grouping of cases to variants. In the implementation of high- and low-level variants, we reduce the problem of computing maximal sequential and parallel partitions (cf. Definitions 8.3 and 8.4) to the well-known problem of computing components of a graph [103].

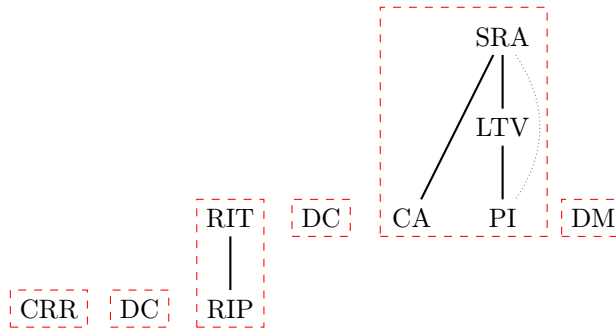
As introduced in Sections 8.2.2 and 8.3.2, we first create the high-/low-level case view (cf. Definitions 8.1 and 8.5), i.e., a partially ordered set. In the remainder of this section, we refer to the graph representing the high-level/low-level case view simply as *case view graph*. Next to the case view graph, we create the corresponding *unrelated graph*. The unrelated graph contains the same nodes as the corresponding case view graph but only connects any two nodes that are *unrelated* in the corresponding case view. For instance, Figure 8.12a depicts the high-level case view graph, and Figure 8.12b depicts the corresponding unrelated graph. Note that any two elements that are not connected in the case view graph are connected in the corresponding unrelated graph; for example, activities RIT and RIP are unrelated in the high-level case view and thus connected in the corresponding unrelated graph, cf. Figure 8.12b.

Subsequently, we look for components in the case view graph and in the unrelated graph. If the case view graph contains more than one component, we found a parallel partition (cf. Definition 8.4) in the corresponding case view. Analogously, if we find more than one component in the unrelated graph, we found a sequential partition (cf. Definition 8.3) in the case view. In the example depicted in Figure 8.12, the case view graph contains a single component but the unrelated graph contains more than one component; thus, we found a sequence partition. Afterwards, we recursively continue on the partitions found as exemplified in Figure 8.4. Since either a parallel or a sequential partition can be found in general, only one of the two graphs contains more than one component. In short, computing maximal parallel and sequential partitions is reduced to finding components in a graph, which can be solved in linear time [103].

Recall Definitions 8.1 and 8.5 specifying high-/low-level case views. Note that these definitions are defined over individual cases. Thus, we specified in Definitions 8.2 and 8.6 that cases having isomorphic high-/low-level case views are covered by the same high-/low-level variant. Thus, the problem could be reduced to a graph isomorphism problem. However, when computing the visualization of variants, we obtain a tree structure. We can utilize this tree structure and apply tree isomorphism instead of graph isomorphism to detect if two cases belong to the same high-/low-level variant. Note that checking whether two trees are isomorphic can be done in polynomial time [42].



(a) High-level *case view graph* (identical to the example depicted in Figure 8.3) containing one component highlighted by a red dashed box



(b) *Unrelated graph* that corresponds to the high-level case view graph shown above in Figure 8.12a and contains six components highlighted by red dashed boxes

Figure 8.12: Example of a high-level case view graph and corresponding unrelated graph (partly adapted from [188, Figure 11])

## 8.5. Time Granularity Modifier

Gaining valuable insights from event data is essential for making informed decisions. The right level of temporal abstraction of the event data being analyzed is critical to reach such insights. Setting the right level of temporal abstraction may reveal patterns and trends in the event data, which might remain hidden if the temporal abstraction is not set appropriately. Note that most information systems use discrete time domains with seconds or even milliseconds as the smallest time unit. According to [10], this smallest time unit is referred to as the *bottom granularity*. Analyzing event data at the millisecond level may lead to incorrect conclusions depending on analysis objectives.



Recall the example event log describing a mortgage application process, cf. Table 3.1. Since the individual process executions span several days, the analysis at the level of milliseconds appears too fine-grained. Sometimes, whether a particular activity was initiated a few minutes before or after another activity is irrelevant. It may not provide any relevant information or could even be misleading. For example, the exact timing of whether activity "Request information from requester" was executed a few minutes before "Request information from a third party" is irrelevant. Instead, knowing that both activities were executed on the same day and, therefore, parallel from a control flow perspective is adequate. Thus, analysts are interested in variants indicating such parallelism.

As exemplified above, a low bottom granularity may not add value to the process analysis and may lead to wrong conclusions. Furthermore, the timestamps recorded for the events of an event log might also be inaccurate, i.e., they do not accurately reflect reality. For example, reconsider the mortgage application process. Assume a process participant makes several decisions over a period of time and records these decisions all at once in an information system. Furthermore, assume that the information system cannot track when the decision-making process begins. As a result, the timestamps for the activity "decision made" do not accurately reflect the actual time the decisions were made; instead, they indicate the entry of the decision into the system. Additionally, since there is no record of the start time for the "decision made" activity, there is no information available on its duration.

The above examples demonstrate the importance of adjusting temporal granularity during process mining analyses. Also in [219], introducing variants for partially-ordered, time-point-based activities, the authors recognized the importance of adjusting time granularity. We define the adjustment of the time granularity as a function called Time Granularity Modifier (TGM) that modifies timestamps.<sup>8</sup>

**Definition 8.7** (Time granularity modifier (TGM))

The function  $f^{TGM} : \mathbb{R}^+ \rightarrow \mathbb{R}^+$  is called a TGM iff

$$\forall t_1, t_2 \in \mathbb{R}_{\geq 0} \left( t_1 < t_2 \Rightarrow f^{TGM}(t_1) \leq f^{TGM}(t_2) \wedge \right. \\ \left. t_1 = t_2 \Rightarrow f^{TGM}(t_1) = f^{TGM}(t_2) \right).$$

Figure 8.13 shows different time granularity modifiers applied to case  $C_1$ . For instance, when applying the time granularity modifier *days*, the timestamps' hours, minutes, and seconds information are ignored. Thus, the bottom granularity is hours. As a result, the two start timestamps of the activity RIT and RIP become identical. In comparison, RIP starts before RIT when looking at  $C_{1,H}$  (Figure 8.13). The third granularity modifier sets the bottom granularity to calendar weeks. For example, as a result, the first execution of activity DC falls into a single calendar week. Thus, activity DC's start and completion timestamps are equal, so DC is considered atomic. In comparison, the first execution of activity DC represented an interval using the before-introduced time granularity modifier. The same applies to activity CA. In conclusion, time granularity modifiers change the

<sup>8</sup>Recall that we define timestamps as real numbers, cf. Definition 3.18.

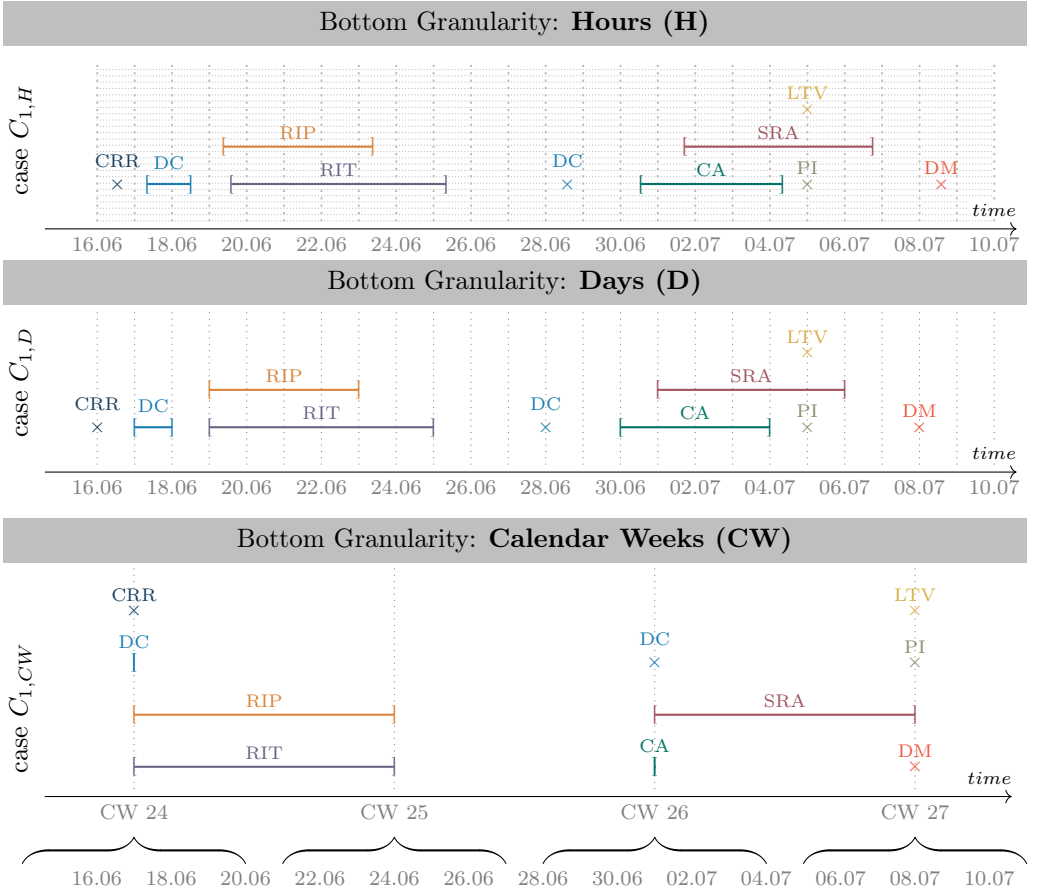


Figure 8.13: Different time granularity modifiers applied to case  $C_1$  (figure partly adapted from [188, Figure 8])

timestamps of the different activities of a case and hence might change the order relations among them. These modified order relations ultimately affect the variants describing these cases.

Depending on the chosen bottom granularity, we obtain different variants for the same case. We exemplify the effect of different TGMs on the variant describing case  $C_1$ . Figure 8.14 depicts cases  $C_{1,H}$  and  $C_{1,D}$ , which are represented by the same high-level variant. When switching the bottom granularity to calendar weeks, i.e.,  $C_{1,CW}$ , the corresponding high-level variant changes, cf. Figure 8.14. Visually speaking, we observe that with coarser bottom granularity, more activities tend to become parallel. As a result, the variants increase vertically and decrease horizontally in size.

Figure 8.15 (page 240) depicts the low-level variants for  $C_1$  when applying different TGMs. Each bottom granularity results in a different low-level variant for  $C_1$ . Between

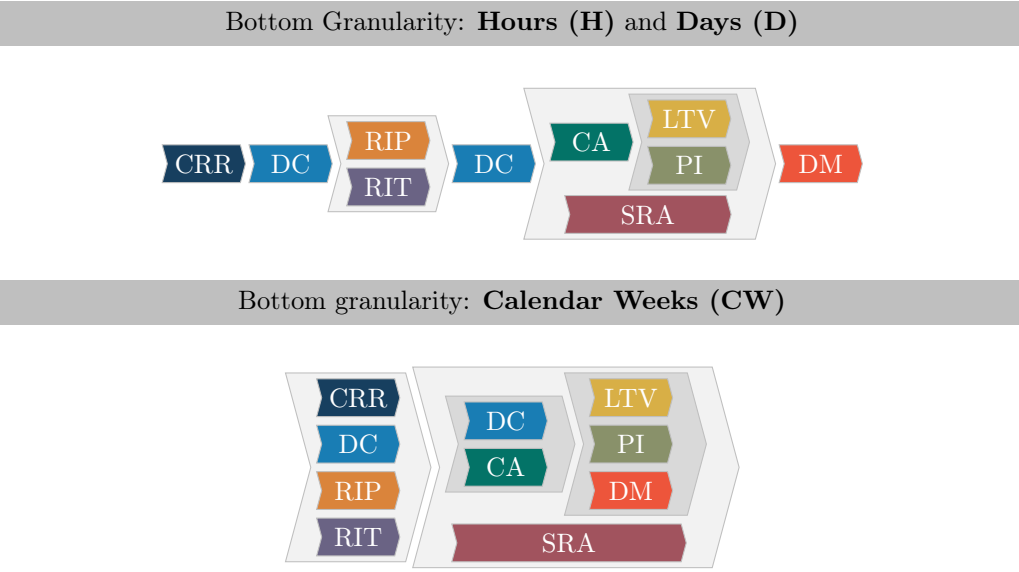


Figure 8.14: High-level variants for case  $C_1$  per TGM; the first high-level variant covers  $C_{1,H}$  and  $C_{1,D}$  while the second variant covers  $C_{1,CW}$  (partly adapted from [188, Figure 9])

the low-level variant covering  $C_{1,H}$  and  $C_{1,D}$ , the only difference is that the start of activities RIP and RIT is in parallel for  $C_{1,D}$ , whereas RIP starts before RIT for  $C_{1,H}$ . For instance, the low-level variant for the bottom granularity days indicates that activities RIP and RIT start on the same day; however, RIT ends at least one day later than RIP.<sup>9</sup> We observe that many activities happen in parallel for the bottom granularity of calendar weeks. Overall, we observe similar behavior of the low-level variants compared to high-level variants when moving to coarser bottom granularities, i.e., activities tend to be executed more in parallel for coarser bottom granularities. However, these are not unexpected observations.

<sup>9</sup>Recall that variants are intended to visualize order relations among activities. Thus, we cannot derive any further temporal information regarding how longer RIT is executed compared to RIP than one unit of the bottom granularity, i.e., one day in the given example.

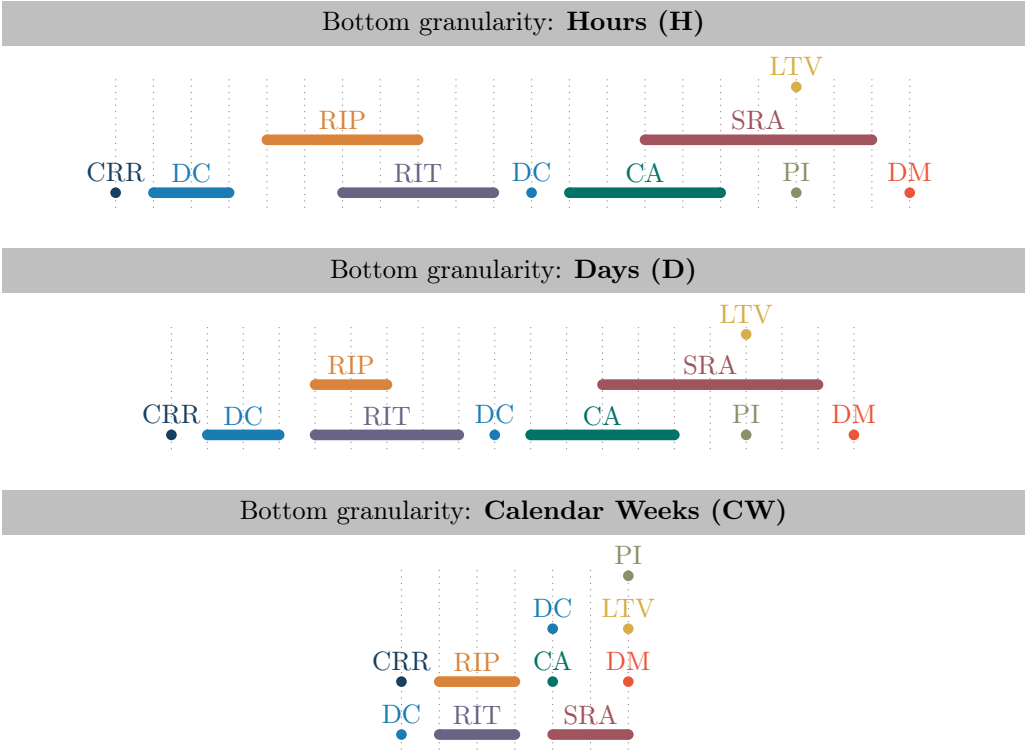


Figure 8.15: Low-level variants for case  $C_1$  per TGM; the first low-level variant covers  $C_{1,H}$ , the second variant  $C_{1,D}$ , and the third one  $C_{1,CW}$  (partly adapted from [188, Figure 10])

## 8.6. Evaluation

This section presents an evaluation of the proposed high- and -low-level variants. The evaluation is split into two parts. Section 8.6.1 presents automated experiments that primarily focus on demonstrating the applicability regarding computational effort to real-life event logs. Further, the automated experiments focus on how variants change upon applying different TGMs, as exemplified in Section 8.5. Section 8.6.2 presents a user study that evaluates how high-level variants support process analysts in real-life analysis tasks compared to existing variant visualizations. Main focus of the user study is the evaluation of the usefulness and effectiveness of the proposed high-level variants.

### 8.6.1. Automated Experiments

In this section, we present the automated experiments conducted. The main objective of these experiments is to demonstrate that the proposed variants can be computed on real-life event logs within a reasonable time, thus showing their practical applicability. Additionally, the automated experiments provide insights into the number of variants and their dimensions regarding height and width for different time granularity modifiers.

#### Experimental Setup

Table 8.1 overviews the real-life event logs used. All logs exhibit partially ordered event data<sup>10</sup>, i.e., there exist cases containing events with identical timestamps or events representing time intervals overlap with intervals specified by other events. The BPI Ch. 18 contains only time-point-based activities (cf. Table 8.1), while the remaining logs contain activities with heterogeneous temporal information. For these event logs, we compute high-level and low-level variants as presented in Sections 8.2 to 8.4.

Table 8.1.: Overview of the event logs used for the automated experiments

Event log	#Cases	Timestamps*	Bottom granularity
Hospital Billing [133]	100,000	start & completion	seconds
BPI Challenge 2012 (BPI Ch. 12) [228]	13,087	start & completion	milliseconds
BPI Challenge 2017 (BPI Ch. 17) [229]	31,509	start & completion	milliseconds
BPI Challenge 2018 (BPI Ch. 18) [233]	150,370	completion	milliseconds

\* Having a timestamp for the start & completion of an activity's execution is identical to having duration information, as exemplified in the event log depicted in Table 3.1 (page 59) and defined in Definition 3.18 (page 60).

Table 8.2.: Total number of variants for different logs and time granularities; the number in parentheses below indicates the proportion of variants that contain parallel process behavior, i.e., at least one parallel partition is found in the corresponding high-/low-level case view (adapted from [188, Table 3])

Event log	Variant type	Time granularity					
		ms	s	m	h	d	mo.
Hospital Billing	high-level	-	667 (93.9%)	1,071 (97.4%)	1,048 (98.1%)	1,182 (98.4%)	1,389 (99.0%)
	low-level	-	685 (93.9%)	1,308 (97.1%)	1,331 (97.9%)	1,516 (98.2%)	1,773 (98.9%)
BPI Ch. 12	high-level	3,830 (98.8%)	3,766 (99.6%)	4,594 (100%)	5,220 (100%)	5,241 (100%)	4,080 (100%)
	low-level	3,855 (98.9%)	3,947 (99.7%)	5,737 (100%)	5,702 (100%)	5,257 (100%)	4,080 (100%)
BPI Ch. 17	high-level	5,937 (88.9%)	8,484 (100%)	9,665 (100%)	8,516 (100%)	9,454 (100%)	6,551 (100%)
	low-level	5,946 (88.9%)	9,137 (100%)	10,088 (100%)	8,995 (100%)	9,752 (100%)	6,551 (100%)
BPI Ch. 18	high-level	30,122 (100%)	33,407 (100%)	35,396 (100%)	29,313 (100%)	29,476 (100%)	28,294 (100%)
	low-level	30,122 (100%)	33,407 (100%)	35,396 (100%)	29,313 (100%)	29,476 (100%)	28,294 (100%)

## Results

Table 8.2 shows the number of high-level and low-level variants for different event logs and time granularities. Note that the high-level and low-level variants are identical for BPI Ch. 18, which only consists of time-point-based activities. This observation is expected because high-level variants are identical to low-level variants regarding the information value they contain if an event log contains solely time-point-based activities. For the other logs, there are often more low-level variants than high-level variants—as discussed in Section 8.1, a one-to-many relation exists between high-level and low-level variants. The numbers in parentheses (cf. Table 8.2) indicate the share of variants that indicate parallel activities. We observe that the coarser the time granularity, the more variants indicate parallel process behavior. Overall, we observe that many variants indicate parallel process

<sup>10</sup>We used the event abstraction technique from [131] on the Hospital Billing event log to obtain activities with start and completion timestamps, i.e., activities with duration information according to Definition 3.17.

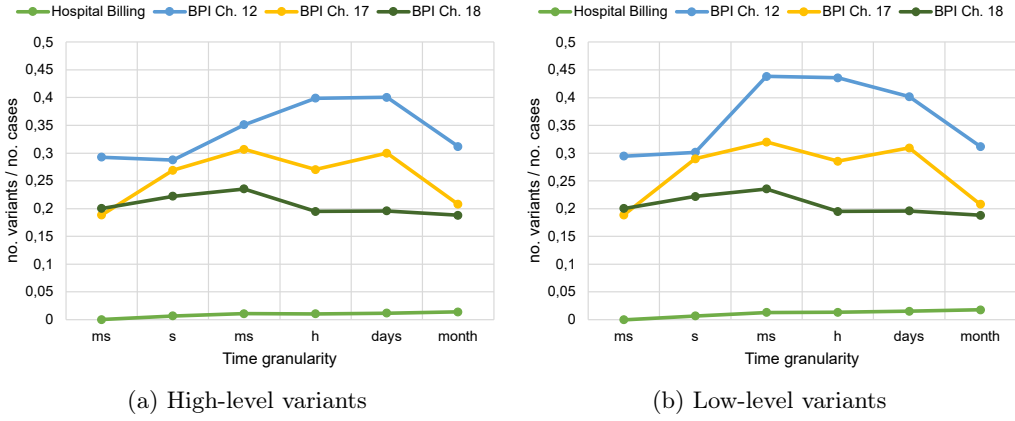


Figure 8.16: Ratio between high-level/low-level variants and cases across different bottom granularities for different event logs

behavior, i.e., all values are clearly above 90% for all logs and time granularities.

Figure 8.16 plots the ration between variants and cases, i.e., the number of variants divided by the number of cases. Figure 8.16a is showing the results for high-level variants, while Figure 8.16b is showing the results for low-level variants. The underlying data is taken from Table 8.2. We observe that, for example, the hospital billing event log contains a large number of variants that are infrequent, i.e., each variant describes only a few cases. As a result, the ratio across all time granularities is near zero. In contrast, the other three event logs contain variants that summarize larger quantities of the cases contained in these logs. Moreover, we observe that changing the bottom granularity impacts the depicted ratio. In short, a change of the bottom granularity does not necessarily lead to a reduction in the number of variants obtained.

Table 8.3 displays the time in seconds taken to calculate high-level and low-level variants from the provided event logs per time granularity modifier. The time taken to calculate high-level variants is significantly longer than that taken for low-level variants across all logs. The longer calculation time for high-level variants in comparison to low-level variants originates from the more involved recursion. When calculating low-level variants, at most two recursion levels are executed since all elements are atomic, cf. Section 8.3. One of the following scenarios applies.

- Every element is parallel. Hence, a parallel partition is found and the recursive partitioning stops.
- All elements are sequentially ordered. Hence, a sequential partition is found and the recursive partitioning stops.
- Some elements are sequentially ordered; thus, a sequential partition is found. Recursively, parallel partitions are found for all non-singleton subsets.

Table 8.3.: Calculation time (seconds) of high-/low-level variants for different logs and time granularities (adapted from [188, Table 4])

Event log	Variant type	Time granularity					
		ms	s	m	h	days	mo.
Hospital Billing	high-level	-	25	26	26	25	22
	low-level	-	18	18	17	13	10
BPI Ch. 12	high-level	26	28	29	32	35	41
	low-level	4	4	5	5	5	6
BPI Ch. 17	high-level	38	50	61	59	65	63
	low-level	10	12	13	13	14	14
BPI Ch. 18	high-level	673	771	900	825	900	1,180
	low-level	61	80	84	77	77	85

In contrast, events in high-level variants may represent time points or time intervals. As a result, the high-level case view can have more complex patterns than the low-level case view. These more complex patterns lead to more recursion levels when calculating the partitions, for example, consider Figure 8.4 (page 226).

When time granularity is coarser, computation time usually increases, as shown in Table 8.3. The time it takes to compute variants depends on two factors. First, the more variants there are, the longer it takes to compute them all. Second, the coarser the time granularity, the longer it takes to compute a single variant. The second point is a result of the implementation. Upon applying a time granularity modifier, all timestamps in an event log are modified when the corresponding events are used for any calculation. This means that the entire event log does not have to be recreated with changed time stamps, but instead, time stamps are adjusted on the fly as required. For example, if the bottom granularity is set to hours, the values for milliseconds, seconds, and minutes in the timestamps of the currently considered events are set to zero. The values reported in Table 8.3 include these on-the-fly timestamp adjustments. The coarser the time granularity, the more values must be set to zero, resulting in increased calculation times.

Table 8.4 reports the average dimensions, i.e., width and height, of the variants per event log and time granularity. As expected, both high-level and low-level variants increase in height and decrease in width when changing to coarser time granularities. Furthermore, Table 8.4 shows that high-level variants' average height is greater or equal than low-level variants' height.

Finally, we present the share of high-level variants that contain non-singleton partitions as discussed in Section 8.2.3. As expected, we do not observe this phenomenon for BPI Ch. 18 since this log contains only atomic activities; thus, patterns, as exemplified in Section 8.2.3, cannot occur. For the other logs, we find that, in general, only a small number of variants is affected by the phenomenon that when sequence and parallel partitions are applied recursively, non-singletons remain.



Table 8.4.: Average width and height (rounded to integers), i.e., the number of chevrons/bars respectively dots, of high-level and low-level variants for different logs and time granularities (adapted from [188, Table 5])

Dimension	Event log	Variant type	Time granularity					
			ms	s	m	h	d	mo.
width	Hospital Billing	high-level	-	6	6	5	5	3
		low-level	-	10	8	7	6	4
	BPI Ch. 12	high-level	20	18	13	10	6	2
		low-level	36	33	22	10	6	2
	BPI Ch. 17	high-level	19	13	9	6	5	2
		low-level	23	15	10	7	5	2
	BPI Ch. 18	high-level	58	52	36	29	25	9
		low-level	58	52	36	29	25	9
height	Hospital Billing	high-level	-	3	3	4	4	5
		low-level	-	2	3	3	4	5
	BPI Ch. 12	high-level	3	4	6	8	10	19
		low-level	3	4	5	7	10	19
	BPI Ch. 17	high-level	3	5	5	8	8	15
		low-level	2	3	4	7	8	15
	BPI Ch. 18	high-level	3	4	6	8	10	21
		low-level	3	4	6	8	10	21

Table 8.5.: Share of high-level variants that contain non-singleton partitions, cf. Section 8.2.3 (adapted from [188, Table 6])

Event log	Time granularity					
	ms	s	m	h	d	mo.
Hospital Billing	-	13%	4%	3%	2%	3%
BPI Ch. 12	0%	0%	14%	1%	0%	0%
BPI Ch. 17	7%	17%	14%	13%	10%	0%
BPI Ch. 18	0%	0%	0%	0%	0%	0%

8.6.2. User Study

This section presents a user study, assessing how high-level variants support users in event data analysis tasks compared to existing variant visualizations, cf. Figure 8.1. The user study compares the *usefulness* and *ease of use* [53] of the proposed high-level variant visualization with existing visualizations. Moreover, we collected open feedback from study participants on high-level variants’ positive and negative aspects. The remainder of this section is organized as follows. Section 8.6.2 describes the design of the study, and

Section 8.6.2 presents the findings. Finally, Section 8.6.2 provides a discussion.

## Study Design

The conducted user study compares three variant visualizations: (A) variant visualization for totally ordered time-point-based activities, (B) variant visualization for partially ordered time-point-based activities [219], and (C) variant visualization for partially ordered time-point- and time-interval-based activities, i.e., high-level variants.

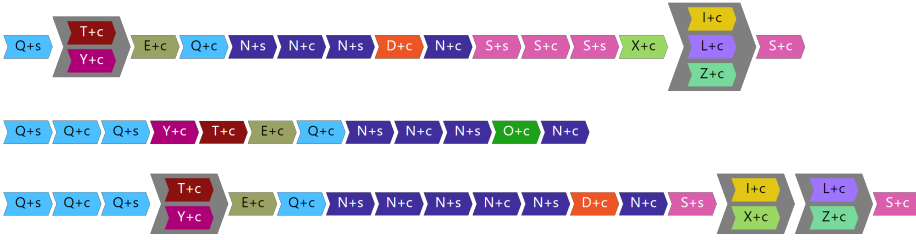
Figure 8.17 shows the three variant visualizations considered in this study; Figures 8.17a to 8.17c visualize all the same event log, i.e., a small sample from a real-life event log to ensure realistic activity patterns. We intentionally kept the event log used small because we found during piloting the study design that participants find even a small number of variants challenging, especially in the variant visualization (A), cf. Figure 8.17. Note that we randomized the activity labels per visualization. Thereby, it is difficult to detect for study participants that all three visualizations show the same process behavior. Since variant visualizations (A) and (B) assume activities to be atomic, we split the start and potential completion of an activity into two events. For example, the activity label “E+s” represents the start of an activity E and the label “E+c” its completion.

In the following, we present the procedure of the user study, which was carried out in the form of an online questionnaire.

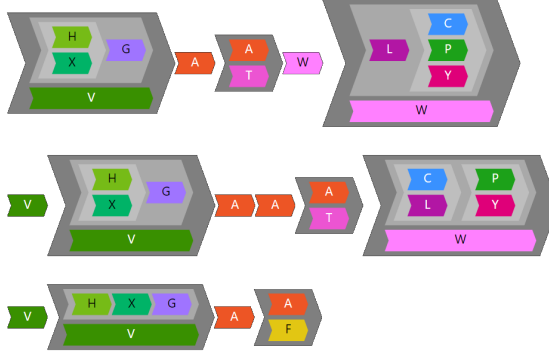
1. The *demographic questions* ask for participant information as well as an assessment of process mining experience.
2. A brief *video tutorial* introduces the three considered visualizations (A), (B), and (C), cf. Figure 8.17 to ensure participants have all necessary information to solve the subsequent tasks.
3. A *comprehension & familiarization task* checks whether the participants understand the three visualizations. Further, this first task allows participants to familiarize with the kinds of questions asked subsequently. Moreover, we use the results of this task to filter participants who show a poor understanding of the three visualizations. The task was not recognizable to the participants as a comprehension & familiarization task, but was simply presented as a regular task.
4. *Task (1) “Extracting Patterns from Variant Visualizations”* contains control-flow-related questions about the visualized activities.  
*Overall, this task comprises five data analysis questions per visualization, i.e., 15 questions in total*
5. *Task (2) “Identifying Variants based on Patterns”* is about identifying variants that contain activity patterns described in natural language.  
*Overall, this task comprises five data analysis questions per visualization, i.e., 15 questions in total*
6. *Perceived usefulness* and *ease of use* questions about the three visualizations.
7. *Open questions* collect qualitative feedback about variant visualization (C), i.e., high-level variants (cf. Section 8.2).



(a) Visualization (A) assuming time-point-based activities; chevrons represent atomic events where ‘+s’ represents the start and ‘+c’ the completion of an activity



(b) Visualization (B) assuming partially ordered time-point-based activities; chevrons represent atomic events where ‘+s’ represents the start and ‘+c’ the completion of an activity



(c) Visualization (C) assuming partially ordered time-point-based and time-interval-based activities, i.e., the proposed high-level variants

Figure 8.17: Variant visualizations describing the same event log; activity labels are randomly changed per visualization (partly adapted from [188, Figure 14])

Within a task, we use the same event log for all visualizations, i.e., subsets of real-life event logs. We used letters to label activities in the log to mitigate the influence of potential domain knowledge that study participants might have. We also applied label randomization such that the questions per visualization differ, cf. Figure 8.17, and randomized the answer options across different questions. In this way, we aimed to reduce the learning effect and ensure fairness between the three visualizations since we always asked for the same patterns. After each task, we also asked participants to indicate the perceived task difficulty per visualization to estimate how much effort the task required. Regarding usefulness and ease of use, we designed the questionnaire following the validated question items presented in [53]. Finally, we asked participants to provide open feedback on visualization (C), and list positive and negative aspects.

The questions posed in tasks (1) and (2) represent realistic process mining analysis tasks. Task (1) represents an event data exploration task. Study participants are shown variant visualizations and a list of activity patterns described in natural language. Participants had to consider the provided variant visualization to identify true statements, i.e., statements describing patterns that occurred in one of the visualized variants. Task (2) represents an event data filtering task. Study participants are shown variant visualizations and activity patterns described in natural language. Participants had to select variants containing the provided activity patterns.

Each task consists of five questions; we ask the same five questions per variant visualization, resulting in 15 questions per task. We use the same event log per task, i.e., tasks (1) and (2). However, as described above, we randomly change activity labels per variant visualization, cf. Figure 8.17. Thereby, it was tough for participants to detect that we used the same event data for each variant visualization. Although we derived the event data from a real-life event log, we intentionally chose individual letters for activity names to mitigate the influence of potential domain knowledge some study participants might have. We randomized the answer options to ensure fairness between the three visualizations and reduce the learning effect, as we always asked for the same patterns. Additionally, participants were asked to indicate the perceived task difficulty per visualization to estimate how much effort the task required.

At the end of the questionnaire, i.e., after tasks (1) and (2), we asked questions concerning usefulness and ease of use for all three visualizations. We used the validated question items that were presented in [53]. Finally, participants could provide open feedback, including positive and negative aspects, regarding visualization (C).

We conducted the user study as an online questionnaire during the summer of 2022. We invited computer and data science students from the Business Process Intelligence and Advanced Process Mining courses offered at RWTH Aachen University. Thus, invited students had a fundamental understanding of process mining. Moreover, we invited process mining researchers and industry professionals. A total of 58 participants completed the questionnaire. Seven participants indicating low comprehension of the three variant visualizations were filtered based on the comprehension and familiarization task results. Of the 51 remaining participants, 32 are bachelor/master students, 15 are Ph.D. students, two are postdoc/professors, and one is working in the industry. All participants had expertise in process mining according to their self-rating, with an average reported expertise of 4.1 out of 6. None of the participants reported having no expertise.

Findings

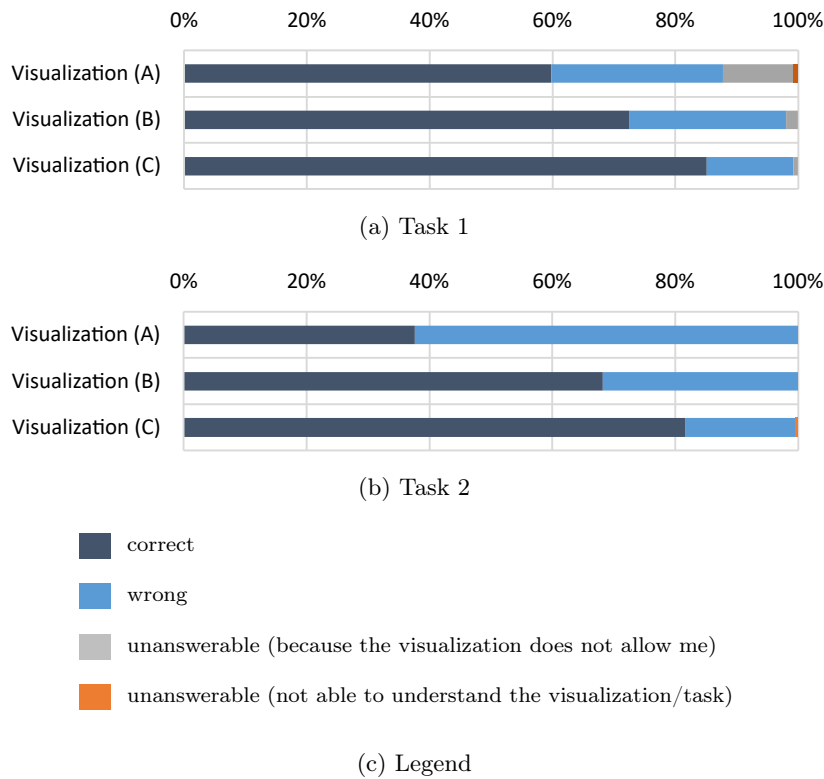


Figure 8.18: Average distribution of accurate responses based on the questions per visualization for tasks (1) and (2); participants tend to answer most questions correctly when they use variant visualization (C), i.e., high-level variants (partly adapted from [188, Figure 15])

Figure 8.18 shows the distribution of correctly answered questions per task and visualization. Note that each question in tasks (1) and (2) contained correct and incorrect answer options. Overall, participants made fewer errors when using visualization (C) than with (B) and (A). We also note that, on average, across all five questions, about 10% of the study participants stated that visualization (A) did not allow them to answer the questions posed, especially for questions related to activities performed in parallel.

Figure 8.19 shows task difficulty for each visualization as perceived by the study participants. Recall that we ask the same questions per variant visualization; however, study participants were unaware since we randomized the activity labels. When the study participants were using variant visualization (C), they perceived the task difficulty as lowest compared to (B) and (A). Figure 8.20 shows the participants' confidence in providing the correct answer. Like the perceived task difficulty (cf. Figure 8.19), participants have the

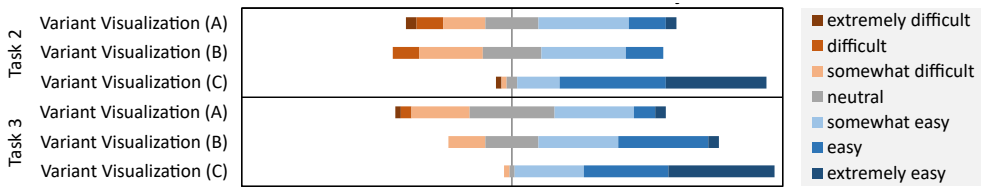


Figure 8.19: Perceived task difficulty per visualization and task (adapted from [188, Figure 16])

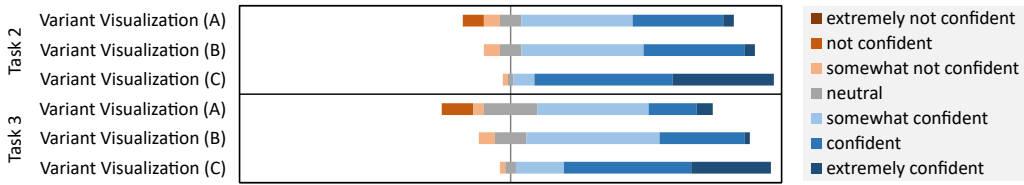


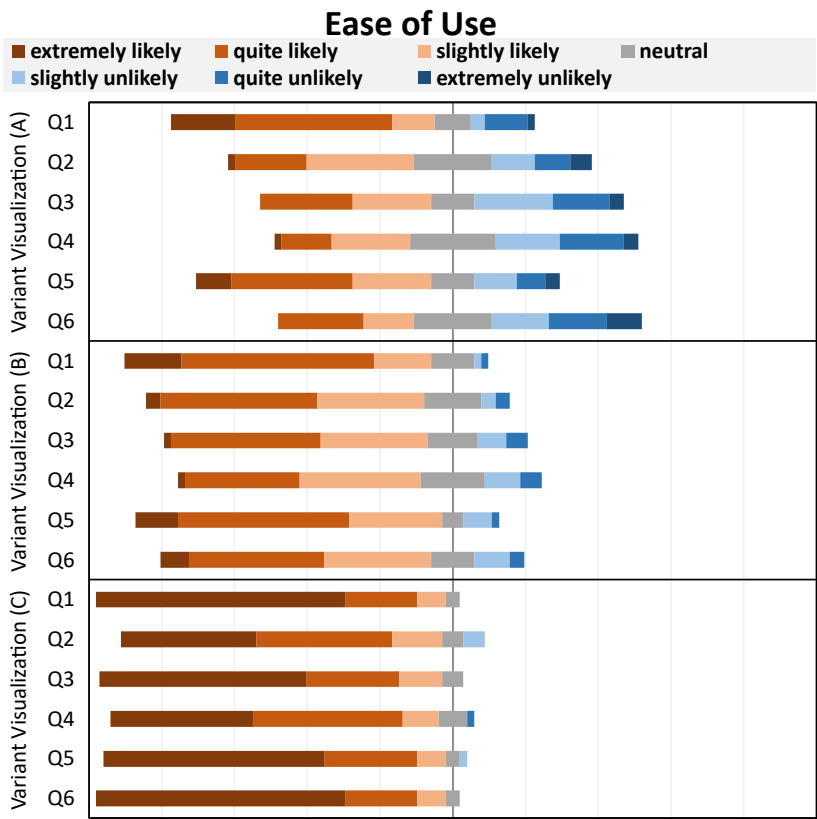
Figure 8.20: Perceived confidence of having given the correct answer per task and visualization

highest confidence when using variant visualization (C). In particular, the proportion of participants who state that they are “extremely confident” is significantly higher.

Figures 8.21 and 8.22 show the *usefulness* and *ease of use* of the visualizations [53]. The results indicate that variant visualization (C) received the highest scores for both usefulness and ease of use, followed by (B) and finally (A). In particular, visualization (C) scored significantly better in usefulness and ease of use than the other visualizations, as evidenced by the percentage of respondents who rated it as “extremely likely.”

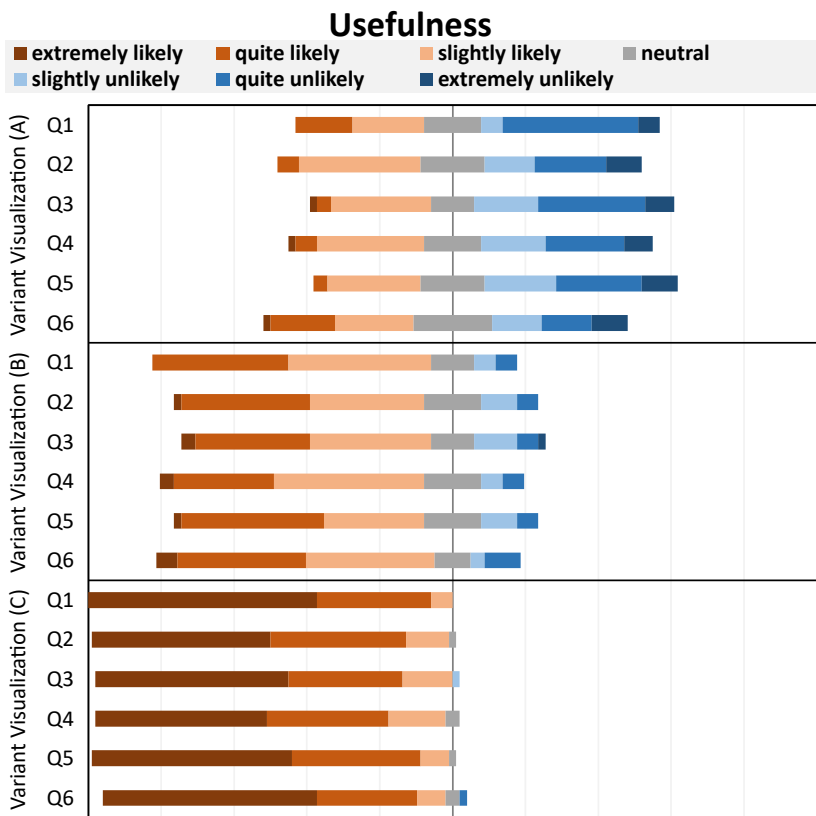
Finally, we provide the results from the open feedback. Below, we summarize the most frequently mentioned positive aspects related to (C) from 49 responses. The study participants indicate that visualization (C) is easy to interpret and presents variants compactly. Furthermore, participants reported that visualization (C) enables them to identify parallelism within variants easily and made them noticeably faster at completing the posed tasks than the other visualizations. We have received 47 responses regarding negative aspects. Participants noted that high-level variants might be too abstract in some scenarios where information on the exact overlap of activities is required. Moreover, they mentioned that visualization (C) does not allow for distinguishing between atomic or interval-based activities. We find the two negative aspects mentioned interesting, as the proposed low-level variants (cf. Section 8.3) address them; recall that low-level variants were not part of this user study. Finally, participants reported concerns about the vertical size of the variants, which can become considerably large in case many activities are executed in parallel. These concerns are justified, as we have seen, for example, in the automated experiments, that variants can grow significantly vertically if the bottom granularity increases, cf. Table 8.4. Based on the conducted user study, the proposed variant visualization (C) is more useful and easier to use for process analysts in event

data analysis tasks when compared to existing variant visualizations, especially in the presence of parallel activities. Moreover, visualization (C) leads to users drawing fewer incorrect conclusions.



- Q1 Learning to operate the visualization would be easy for me.
- Q2 I would find it easy to get the visualization do what I want it to do.
- Q3 My interaction with the visualization would be clear and understandable.
- Q4 I would find the visualization to be flexible to interact with.
- Q5 It would be easy for me to become skillful at using the visualization.
- Q6 I would find the visualization easy to use.

Figure 8.21: Perceived ease of use of the three variant visualizations for solving the posed tasks (partly adapted from [188, Figure 17])



- Q1 Learning to operate the visualization would be easy for me.
- Q2 I would find it easy to get the visualization do what I want it to do.
- Q3 My interaction with the visualization would be clear and understandable.
- Q4 I would find the visualization to be flexible to interact with.
- Q5 It would be easy for me to become skillful at using the visualization.
- Q6 I would find the visualization easy to use.

Figure 8.22: Perceived usefulness of the three variant visualizations for solving the posed tasks (partly adapted from [188, Figure 17])

Discussion & Threats to validity

We asked participants to complete the questionnaire in one go. The study was unsupervised; thus, we could not supervise the participants to assess if they worked on the questionnaire without interruption. As a result, some participants may not have completed the entire questionnaire in one go and may not have remembered all the details



when answering the questions on usefulness and ease of use located at the end of the questionnaire. However, study participants took a similar amount of time to complete the questionnaire as the participants in the pilot, indicating limited risk.

We acknowledge that our sample primarily consists of bachelor's, master's, and Ph.D. students, which may limit the generalizability of our findings. Although research suggests students can replace experts in user studies [89, 197], generalizations from the questionnaire population to process mining experts are limited. In addition, due to the small number of study participants, the proposed study does not claim statistical significance. Thus, further studies are needed to generalize our findings to a broader group of subjects. Still, since the core contribution of this chapter is the definition and visualization of novel types of variants, we consider the limitations concerning generalization less critical.

The study aimed to compare high-level variant visualizations with existing ones. However, it did not assess other contributions mentioned in the chapter, such as the low-level variants and the temporal granularity modifiers. We deliberately chose to exclude low-level variants from this user study based on the results of our pilot. While piloting the study, we discovered that answering the questions required significant time and concentration, taking about one to two hours to complete the questionnaire. To properly test low-level variants, participants should have answered questions requiring at least a similar amount of time. Given that the study was designed as a voluntary online survey, there was a high risk that participants would manifest tiredness after some time or might drop out or not participate. Thus, conducting a separate study evaluating low-level variants is more advisable to mitigate these risks.

As elaborated above, this user study did not consider the time granularity modifiers for reasons similar to those of the low-level variants. Moreover, to properly test the usefulness of time granularity modifiers, we require participants who understand the process captured in the log and can decide which modifier to apply based on this knowledge. In this scenario, there are more suitable instruments than an unsupervised questionnaire. Instead, an observational study with expert analysts might provide more insight into how analysts use various time granularity modifiers during analysis and what variants they consider valuable for achieving specific objectives.

## 8.7. Conclusion

This chapter proposed two complementary variant definitions and corresponding visualizations, i.e., high-level and low-level variants. The proposed variant definitions assume partially ordered event data with heterogeneous temporal information, cf. Definition 3.17. As the names indicate, high-level variants allow to explore process behavior from an event log at a higher level of abstraction compared to low-level variants. Recall that a one-to-many relationship between high-level and low-level variants exists, i.e., one high-level variant may comprise multiple low-level variants. Moreover, we discussed the importance of finding the right level of temporal abstraction. We introduced temporal granularity modifier and showcased their impact on the proposed variants. The conducted automated experiments show that the proposed variants and corresponding visualizations are computable in a reasonable amount of time on real-life event logs. The conducted user study revealed high scores for usefulness and ease of use for the proposed high-level variants

compared to existing variant visualizations when partially ordered event data is analyzed.

In the context of incremental process discovery, variants are central. Especially for the practical realization of IPD, variant visualizations are essential, as they allow users to comprehend, assess, and eventually select process behavior from an event log. However, although we mainly motivate the proposed variants in the context of IPD, they are of general relevance for various other process mining tasks. Particularly in view of the fact that partially ordered event data still receives far less attention than totally ordered event data within process mining research, these variant visualizations represent a significant contribution to this field [123].

---

# Chapter 9.

## Query Language for Variants

---

This chapter is largely based on the following published work.

- *D. Schuster, M. Martini, S. J. van Zelst, and W. M. P. van der Aalst. Control-flow-based querying of process executions from partially ordered event data. In J. Troya, B. Medjahed, M. Piattini, L. Yao, P. Fernández, and A. Ruiz-Cortés, editors, Service-Oriented Computing, volume 13740 of Lecture Notes in Computer Science, pages 19–35. Springer, 2022. doi:10.1007/978-3-031-20984-0\_2 [181]*

In the previous chapter, we presented variants for partially ordered event data. Although variants are a fundamental abstraction in process mining to handle large amounts of event data, the number of variants for a given log may be too large to explore all variants manually using visualizations. Therefore, there is a clear motivation for query language support for the proposed variants, i.e., in general, partially ordered event data. While various query languages allow querying of event data [155, 156, 157, 243], specifically supporting partially ordered event data is missing.

Querying variants is of crucial importance for various process mining applications and tasks. In the context of this thesis, the incremental selection of process behavior during incremental process discovery is central. Supporting users in this selection task with suitable tools, such as a query language, is essential for implementing and adapting incremental process discovery. Also, apart from incremental process discovery, a query language for partially ordered event data is of great importance. For example, consider comparative process mining [215]. Comparing multiple event logs is a common task when applying process mining. Techniques like process cubes [34, 210] rely on techniques, such as query languages, to split event data. All in all, querying variants is a critical task closely related to incremental process discovery but also crucial for various other process mining approaches.

This chapter proposes a textual query language for partially ordered event data, as considered in Chapter 8. Thus, as already in the previous chapter, we assume an event log as specified in Definition 3.18 (page 60) and exemplified in Table 3.1 (page 59). This chapter thus makes a contribution to the process mining area of *process querying*, which generally comprises techniques for querying any kind of process mining artifacts [156]. The proposed query language allows the specification of six essential control flow constraints,

which can be further restricted via cardinality constraints and arbitrarily combined via Boolean operators. The language design is based on standardized terms for control flow patterns in process mining. We formally specify the language's syntax and semantics to facilitate reuse in other tools. In the context of incremental process discovery, where the gradual selection of process behavior that is incorporated into a process model is central, a query language designed explicitly for querying based on control flow constraints is a valuable asset.

This chapter's remainder is organized as follows. First, Section 9.1 briefly elaborates on related work in the area of query languages in the field of process mining. Section 9.2 introduces the query language by defining its syntax and semantics. Subsequently, Section 9.4 presents an evaluation focusing on performance when evaluating queries. Finally, Section 9.5 concludes this chapter and outlines future work regarding potential extensions/continuations regarding query languages for partially ordered event data.

## 9.1. Related Work

In [157], Polyvyanyy et al. present a *process querying* framework that allows for comparing and organizing process querying techniques. Process querying techniques mainly differ in the input, for example, techniques assume event logs as input [21, 247]), while others assume process model repositories as input [20, 136]). Besides differences in the input, the specific capabilities and goals of the specific query methods, of course, differ. Various reviews of process querying techniques exist [155, 156, 157, 243]. Most identified process querying techniques focus on querying process models from process model repositories. In this chapter, we focus on querying event data; thus, we concentrate on process querying techniques operating on event logs in the following.

Celonis PQL [241] is a textual query language that supports event data and process models as input. It is a multi-purpose query language, i.e., it provides various query options. However, the language does not consider event data explicitly as partially ordered.<sup>1</sup> In [21], the authors present a query language that operates on a single graph that connects all events from an event log based on user-defined correlations among the events. This query language allows partitioning events according to specified constraints and querying paths that start and end with events meeting specific requirements. Compared to the query language proposed in this chapter, the event log is not transformed into a graph structure; instead, we operate on individual cases, respectively, variants composed of partially ordered event data, cf. Definition 3.18.

A natural language interface for querying event data is presented in [112]. This natural language interface uses a graph-based approach similar to the previously described approach [21]. The interface allows for the specification of queries such as "Who was involved in processing case X?" and "For which cases is the case attribute Y greater than Z?" However, control flow constraints for partially ordered event data are not supported compared to the query language proposed in this chapter. In [158], the authors propose an LTL-based query language to query cases consisting of totally ordered activities. In [247], an approach to query case fragments that involve a selected activity is proposed.

<sup>1</sup>Please note that Celonis PQL is a closed-source query language subject to continuous development.

The technique is intended to assist process designers by providing case fragments showing how each activity is executed in different cases. However, this approach also assumes cases comprising totally ordered activities.

In short, process querying is a research field within process mining. Various methods exist, most of them are designed for querying models from a process model repository [155, 156, 157, 243]. Considering querying techniques that operate on event data, most assume totally ordered event data. Thus, the query language proposed in this chapter differs in three main points from related work.

1. The proposed query language focuses on cases containing partially ordered activities.
2. The proposed query language focuses on cases rather than event data as a whole or individual events, i.e., executing a query returns cases satisfying the specified constraints.
3. The proposed query language focuses specifically on control flow patterns, i.e., it offers extensive options for specifying control flow patterns.

## 9.2. Query Language

This section introduces the proposed query language. Section 9.2.1 introduces its syntax, while Section 9.2.2 defines its semantics. Section 9.2.3 covers the query evaluation.

### 9.2.1. Syntax

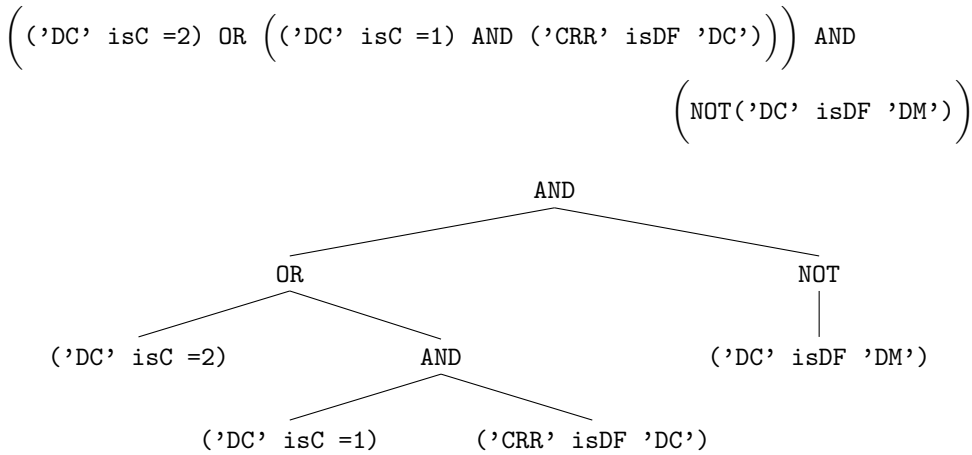


Figure 9.1: Example of a query and its Boolean tree structure; leaves in the tree represent individual query leaves as exemplified in Table 9.1) that are combined via Boolean operators (partly adapted from [181, Figure 2])

Here, we introduce the syntax of the proposed query language, which comprises six operators allowing to specify control flow constraints: three unary operators and three binary operators. Table 9.1 lists these six operators. Next to each operator, Table 9.1 presents exemplary queries and presents their semantics in natural language. The query examples show that cardinality constraints can be attached to each operator. We refer to a query as *leaf query* if the query contains only one operator. All queries  $Q_1, \dots, Q_{18}$  shown in Table 9.1 are leaf queries. Leaf queries can be arbitrarily combined using Boolean operators. For instance, Figure 9.1 shows a query composed of four leaf queries. Subsequently, Definition 9.1 formally specifies a query, i.e., the syntax of the query language.

**Definition 9.1** (Query Syntax)

Let  $l_1, \dots, l_{n-1}, l_n \in \mathcal{A}$  be activity labels,  $k \in \mathbb{N}_0$ ,  $\square \in \{\leq, \geq, =\}$ ,  $\circ \in \{isDF, isEF, isP\}$ ,  $\bullet \in \{isC, isS, isE\}$ , and  $\triangle \in \{ALL, ANY\}$ . We denote the universe of queries by  $\mathcal{Q}$  and recursively define a query  $Q \in \mathcal{Q}$  below.

**Leaf queries with an unary operator**

- $Q = 'l_1' \bullet$
- $Q = 'l_1' \bullet \square k$
- $Q = \triangle \{ 'l_1', \dots, 'l_{n-1}' \} \bullet$
- $Q = \triangle \{ 'l_1', \dots, 'l_{n-1}' \} \bullet \square k$

**Leaf queries with a binary operator**

- $Q = 'l_1' \circ 'l_n'$
- $Q = 'l_1' \circ 'l_n' \square k$
- $Q = \triangle \{ 'l_1', \dots, 'l_{n-1}' \} \circ 'l_n'$
- $Q = \triangle \{ 'l_1', \dots, 'l_{n-1}' \} \circ 'l_n' \square k$
- $Q = 'l_n' \circ \triangle \{ 'l_1', \dots, 'l_{n-1}' \}$
- $Q = 'l_n' \circ \triangle \{ 'l_1', \dots, 'l_{n-1}' \} \square k$

**Composed query using Boolean operators**

- If  $Q_1, \dots, Q_m \in \mathcal{Q}$  are  $m$  queries and  $\blacksquare \in \{AND, OR\}$ , then  $Q = (Q_1 \blacksquare \dots \blacksquare Q_m)$  is a query
- If  $Q_1 \in \mathcal{Q}$  is a query, then  $Q = NOT(Q_1)$  is a query

Note that ANY and ALL sets are only allowed on one side of a binary operator according to Definition 9.1.

Table 9.1.: Overview of the control flow constraints (partly adapted from [181, Table 2])

Type	Syntax	Example		
		Nr.	Query	Description of semantics
unary	isContained (isC)	$Q_1$	'A' isC	activity A is contained in the case
		$Q_2$	'A' isC $\geq 6$	activity A is contained at least 6 times in the case
		$Q_3$	ALL{'A', 'B'} isC $\geq 6$	activity A and B are both contained at least 6 times each in the case
	isStart (isS)	$Q_4$	'A' isS	there exists a start activity A <sup>(a)</sup>
		$Q_5$	'A' isS = 1	exactly one start activity of the case is an A activity <sup>(a)</sup>
		$Q_6$	ANY{'A', 'B'} isS = 1	case starts with exactly one A activity or/and with exactly one B activity <sup>(a)</sup>
	isEnd (isE)	$Q_7$	'A' isE	there exists an end activity A <sup>(a)</sup>
		$Q_8$	'A' isE $\geq 2$	at least two end activities of the case are an A activity <sup>(a)</sup>
		$Q_9$	ALL{'A', 'B'} isE	case ends with at least one A and one B activity <sup>(a)</sup>
binary	isDirectly Followed (isDF)	$Q_{10}$	'A' isDF 'B'	a B activity directly follows <i>each</i> A activity in the case
		$Q_{11}$	'A' isDF 'B' = 1	case contains exactly one A activity that is directly followed by B
		$Q_{12}$	'A' isDF ALL{'B', 'C'}	every A activity is directly followed by a B and C activity
	isEventually Followed (isEF)	$Q_{13}$	'A' isEF 'B'	after <i>each</i> A activity in the case a B activity eventually follows
		$Q_{14}$	'A' isEF 'B' $\geq 1$	case contains at least one A activity that is eventually followed by B
		$Q_{15}$	ALL{'A', 'B'} isEF 'C'	all A and B activities are eventually followed by a C activity
	isParallel (isP)	$Q_{16}$	'A' isP 'B'	each A activity in the case is in parallel to some B activity
		$Q_{17}$	'A' isP 'B' $\leq 4$	case contains at most four A activities that are in parallel to some B activity
		$Q_{18}$	'A' isP ANY{'B', 'C'} $\leq 2$	case contains at most two A activities that are parallel to a B or C activity

<sup>(a)</sup> Case may contain arbitrary further start respectively end activities.

### 9.2.2. Semantics

This section formally specifies the semantics of the proposed query language. First, we introduce notation conventions regarding existential quantification. Subsequently, we define the semantics for each control flow operator.

Let  $k \in \mathbb{N}$  and  $X$  be an arbitrary set. We write:

- $\exists^=^k x_1, \dots, x_k \in X (\dots)$  to denote that there exist *exactly*  $k$  pairwise distinct elements in set  $X$  satisfying a given formula (cf. Equation (9.1)),
- $\exists^{\geq}^k x_1, \dots, x_k \in X (\dots)$  to denote that there exist *at least*  $k$  pairwise distinct elements in set  $X$  satisfying a given formula (cf. Equation (9.2)), and
- $\exists^{\leq}^k x_1, \dots, x_k \in X (\dots)$  to denote that there exist *at most*  $k$  pairwise distinct elements in set  $X$  satisfying a given formula (cf. Equation (9.3)).

Equations (9.1) to (9.3) formally define the three existential quantifiers introduced above.

$$\begin{aligned} \exists^=^k x_1, \dots, x_k \in X \quad \forall 1 \leq i \leq k (P(x_i)) &\equiv \\ \exists x_1, \dots, x_k \in X \Big( \forall 1 \leq i < j \leq k (x_i \neq x_j) \wedge \forall 1 \leq i \leq k (P(x_i)) \wedge \\ &\quad \forall x \in X \setminus \{x_1, \dots, x_k\} (\neg P(x)) \Big) \end{aligned} \quad (9.1)$$

$$\begin{aligned} \exists^{\geq}^k x_1, \dots, x_k \in X \quad \forall 1 \leq i \leq k (P(x_i)) &\equiv \\ \exists x_1, \dots, x_k \in X \Big( \forall 1 \leq i < j \leq k (x_i \neq x_j) \wedge \forall 1 \leq i \leq k (P(x_i)) \Big) \end{aligned} \quad (9.2)$$

$$\begin{aligned} \exists^{\leq}^k x_1, \dots, x_k \in X \quad \forall 1 \leq i \leq k (P(x_i)) &\equiv \\ \exists x_1, \dots, x_k \in X \Big( \forall 1 \leq i \leq k (P(x_i)) \wedge \forall x \in X \setminus \{x_1, \dots, x_k\} (\neg P(x)) \Big) \end{aligned} \quad (9.3)$$

Note that variables  $x_1, \dots, x_k$  in Equation (9.3) must not be assigned *different* elements from the set  $X$ . Equation (9.3) specifies that at most  $k$  distinct elements in  $X$  exist satisfying  $P(\dots)$ . Below, we formally define the semantics of queries.

#### Definition 9.2 (Query Semantics)

Let  $C \subseteq \mathcal{C}$  be a case with corresponding high-level case view  $(C, \prec^{HL}, \Sigma, \lambda, )$ , and  $l_1, \dots, l_n \in \mathcal{A}$  be activity labels. Further let  $k \in \mathbb{N}_0$  and  $\square \in \{\leq, \geq, =\}$ . We recursively define the function

$$eval : \mathcal{Q} \times \mathcal{C} \rightarrow \mathbb{B}$$

that returns true iff case  $C$  satisfies the specified control flow constraints in  $Q$ .

**Unary operators (i.e.,  $isC$ ,  $isS$ , and  $isE$ )**

- If  $Q = 'l_1' \text{ } isC \square k$ , then  $eval(Q, C) \Leftrightarrow \exists^{\square k} e_1, \dots, e_k \in C \quad \forall 1 \leq i \leq k (e_i^l = l_1)$
- If  $Q = 'l_1' \text{ } isS \square k$ , then  $eval(Q, C) \Leftrightarrow$



$$\exists^{\square k} e_1, \dots, e_k \in C \quad \forall 1 \leq i \leq k \left( e_i^l = l_1 \wedge \nexists \tilde{e} \in C \left( \tilde{e} \prec^{HL} e_i \right) \right)$$

— If  $Q = 'l_1' \text{ isE } \square k$ , then  $\text{eval}(Q, C) \Leftrightarrow$

$$\exists^{\square k} e_1, \dots, e_k \in C \quad \forall 1 \leq i \leq k \left( e_i^l = l_1 \wedge \nexists \tilde{e} \in C \left( e_i \prec^{HL} \tilde{e} \right) \right)$$

**Binary operators (i.e., isDF, isEF, and isP)**

— If  $Q = 'l_1' \text{ isDF } 'l_2'$ , then  $\text{eval}(Q, C) \Leftrightarrow$

$$\forall e \in C \left( e^l = l_1 \Rightarrow \exists \tilde{e} \in T \left( \tilde{e}^l = l_2 \wedge e \prec^R \tilde{e} \right) \right)$$

— If  $Q = 'l_1' \text{ isDF } 'l_2' \square k$ , then  $\text{eval}(Q, C) \Leftrightarrow$

$$\exists^{\square k} e_1, \dots, e_k \in C \quad \forall 1 \leq i \leq k \left( e_i^l = l_1 \wedge \exists \tilde{e} \in C \left( \tilde{e}^l = l_2 \wedge e_i \prec^R \tilde{e} \right) \right)$$

— If  $Q = 'l_1' \text{ isDF ANY}\{'l_2', \dots, 'l_n'\}$ , then  $\text{eval}(Q, C) \Leftrightarrow$

$$\forall e \in C \left( e^l = l_1 \Rightarrow \exists \tilde{e} \in C \left( e \prec^R \tilde{e} \wedge \left( \bigvee_{j=2}^n \tilde{e}^l = l_j \right) \right) \right)$$

— If  $Q = 'l_1' \text{ isDF ANY}\{'l_2', \dots, 'l_n'\} \square k$ , then  $\text{eval}(Q, C) \Leftrightarrow$

$$\exists^{\square k} e_1, \dots, e_k \in C \quad \forall 1 \leq i \leq k \left( e_i^l = l_1 \wedge \exists \tilde{e} \in C \left( e_i \prec^{HL} \tilde{e} \wedge \bigvee_{j=2}^n \left( \tilde{e}^l = l_j \right) \right) \right)$$

— If  $Q = 'l_1' \text{ isDF ALL}\{'l_2', \dots, 'l_n'\} \square k$ , then  $\text{eval}(Q, C) \Leftrightarrow$

$$\exists^{\square k} e_1, \dots, e_k \in C \quad \forall 1 \leq i \leq k \left( e_i^l = l_1 \wedge \right. \\ \left. \exists \tilde{e}_2, \dots, \tilde{e}_n \in C \bigwedge_{j=2}^n \left( e_i \prec^{HL} \tilde{e}_j \wedge \tilde{e}_j^l = l_j \right) \right)$$

— If  $Q = 'l_1' \text{ isEF } 'l_2'$ , then  $\text{eval}(Q, C) \Leftrightarrow$

$$\forall e \in C \left( e^l = l_1 \Rightarrow \exists \tilde{e} \in C \left( \tilde{e}^l = l_2 \wedge e \prec^{HL} \tilde{e} \right) \right)$$

— If  $Q = 'l_1' \text{ iEF } 'l_2' \square k$ , then  $\text{eval}(Q, C) \Leftrightarrow$

$$\exists^{\square k} e_1, \dots, e_k \in C \quad \forall 1 \leq i \leq k \left( e_i^l = l_1 \wedge \exists \tilde{e} \in C \left( \tilde{e}^l = l_2 \wedge e_i \prec^{HL} \tilde{e} \right) \right)$$

— If  $Q = 'l_1' \text{ isEF ANY}\{'l_2', \dots, 'l_n'\}$ , then  $\text{eval}(Q, C) \Leftrightarrow$

$$\forall e \in C \left( e^l = l_1 \Rightarrow \exists \tilde{e} \in C \left( e \prec^{HL} \tilde{e} \wedge \bigvee_{i=2}^n \tilde{e}^l = l_i \right) \right)$$

— If  $Q = 'l_1' \text{ isEF ANY}\{'l_2', \dots, 'l_n'\} \square k$ , then  $\text{eval}(Q, C) \Leftrightarrow$

$$\exists^{\square k} e_1, \dots, e_k \in C \quad \forall 1 \leq i \leq k \left( e_i^l = l_1 \wedge \exists \tilde{e} \in C \left( e_i \prec^{HL} \tilde{e} \wedge \bigvee_{j=2}^n \tilde{e}^l = l_j \right) \right)$$

- If  $Q = 'l_1' \text{ isEF } ALL\{'l_2', \dots, 'l_n'\} \square k$ , then  $eval(Q, C) \Leftrightarrow$ 

$$\exists^{\square k} e_1, \dots, e_k \in C \quad \forall 1 \leq i \leq k \left( e_i^l = l_1 \wedge \right.$$

$$\left. \exists \tilde{e}_2, \dots, \tilde{e}_n \in C \left( \bigwedge_{j=2}^n (e_i \prec^{HL} \tilde{e}_j \wedge \tilde{e}_j^l = l_j) \right) \right)$$
- If  $Q = 'l_1' \text{ isP } 'l_2'$ , then  $eval(Q, C) \Leftrightarrow$ 

$$\forall e \in C \left( e^l = l_1 \Rightarrow \exists \tilde{e} \in C \left( \tilde{e}^l = l_2 \wedge e \not\prec \tilde{e} \wedge \tilde{e} \not\prec e \right) \right)$$
- If  $Q = 'l_1' \text{ isP } 'l_2' \square k$ , then  $eval(Q, C) \Leftrightarrow$ 

$$\exists^{\square k} e_1, \dots, e_k \in C \quad \forall 1 \leq i \leq k \left( e_i^l = l_1 \wedge \exists \tilde{e} \in C \left( \tilde{e}^l = l_2 \wedge e_i \not\prec \tilde{e} \wedge \tilde{e} \not\prec e_i \right) \right)$$
- If  $Q = 'l_1' \text{ isP } ANY\{'l_2', \dots, 'l_n'\}$ , then  $eval(Q, C) \Leftrightarrow$ 

$$\forall e \in C \left( e^l = l_1 \Rightarrow \exists \tilde{e} \in C \left( e \not\prec \tilde{e} \wedge \tilde{e} \not\prec e \wedge \bigvee_{j=2}^n \tilde{e}^l = l_j \right) \right)$$
- If  $Q = 'l_1' \text{ isP } ANY\{'l_2', \dots, 'l_n'\} \square k$ , then  $eval(Q, C) \Leftrightarrow$ 

$$\exists^{\square k} e_1, \dots, e_k \in C \quad \forall 1 \leq i \leq k \left( e_i^l = l_1 \wedge \exists \tilde{e} \in C \left( e_i \not\prec \tilde{e} \wedge \tilde{e} \not\prec e_i \wedge \bigvee_{j=2}^n \tilde{e}^l = l_j \right) \right)$$
- If  $Q = 'l_1' \text{ isP } ALL\{'l_2', \dots, 'l_n'\} \square k$ , then  $eval(Q, C) \Leftrightarrow$ 

$$\exists^{\square k} e_1, \dots, e_k \in C \quad \forall 1 \leq i \leq k \left( e_i^l = l_1 \wedge \right.$$

$$\left. \exists \tilde{e}_2, \dots, \tilde{e}_n \in C \left( \bigwedge_{j=2}^n (e_i \not\prec \tilde{e}_j \wedge \tilde{e}_j \not\prec e_i \wedge \tilde{e}_j^l = l_j) \right) \right)$$

### Boolean operators (i.e., AND, OR, and NOT)

Let  $Q_1, \dots, Q_n \in \mathcal{Q}$  be arbitrary queries.

- If  $Q = NOT(Q_1)$ , then  $eval(Q, C) \Leftrightarrow \neg eval(Q_1, C)$
- If  $Q = (Q_1 \text{ OR } \dots \text{ OR } Q_n)$ , then  $eval(Q, C) \Leftrightarrow eval(Q_1, C) \vee \dots \vee eval(Q_n, C)$
- If  $Q = (Q_1 \text{ AND } \dots \text{ AND } Q_n)$ , then  $eval(Q, C) \Leftrightarrow eval(Q_1, C) \wedge \dots \wedge eval(Q_n, C)$

Definition 9.2 does not cover all queries constructible using the syntax specified in Definition 9.1. However, we can rewrite any query into a *logically equivalent* one covered by Definition 9.1. If the condition below holds, we refer to two queries  $Q_1, Q_2 \in \mathcal{Q}$  as

logically equivalent queries, denoted  $Q_1 \equiv Q_2$ .

$$\forall C \in \mathcal{C} \left( eval(Q_1, C) \Leftrightarrow eval(Q_2, C) \right)$$

Subsequently, we list query rewriting rules that transform a given query into a logical equivalent query. Note that not all potential queries containing ANY or ALL sets can be transformed into a logically equivalent formula not containing ANY or ALL sets.

- $'l_1' \bullet \equiv 'l_1' \bullet \geq 1$
- $ANY\{'l_1', \dots, 'l_n'\} \bullet \equiv ('l_1' \bullet) \text{ OR } \dots \text{ OR } ('l_n' \bullet)$
- $ALL\{'l_1', \dots, 'l_n'\} \bullet \equiv ('l_1' \bullet) \text{ AND } \dots \text{ AND } ('l_n' \bullet)$
- $ANY\{'l_1', \dots, 'l_n'\} \bullet \square k \equiv ('l_1' \bullet \square k) \text{ OR } \dots \text{ OR } ('l_n' \bullet \square k)$
- $ALL\{'l_1', \dots, 'l_n'\} \bullet \square k \equiv ('l_1' \bullet \square k) \text{ AND } \dots \text{ AND } ('l_n' \bullet \square k)$
- $ANY\{'l_1', \dots, 'l_{n-1}'\} \circ 'l_n' \equiv ('l_1' \circ 'l_n') \text{ OR } \dots \text{ OR } ('l_{n-1}' \circ 'l_n')$
- $ALL\{'l_1', \dots, 'l_{n-1}'\} \circ 'l_n' \equiv ('l_1' \circ 'l_n') \text{ AND } \dots \text{ AND } ('l_{n-1}' \circ 'l_n')$
- $ANY\{'l_1', \dots, 'l_{n-1}'\} \circ 'l_n' \square k \equiv ('l_1' \circ 'l_n' \square k) \text{ OR } \dots \text{ OR } ('l_{n-1}' \circ 'l_n' \square k)$
- $ALL\{'l_1', \dots, 'l_{n-1}'\} \circ 'l_n' \square k \equiv ('l_1' \circ 'l_n' \square k) \text{ AND } \dots \text{ AND } ('l_{n-1}' \circ 'l_n' \square k)$
- $'l_1' \circ ALL\{'l_2', \dots, 'l_n'\} \equiv ('l_1' \circ 'l_2') \text{ AND } \dots \text{ AND } ('l_1' \circ 'l_n')$

According to Definition 9.2, the following queries are *not* logically equivalent. Hence, the operators ANY and ALL are *not* to be considered *syntactic sugar*.<sup>2</sup>

- $'l_1' \circ ANY\{'l_2', \dots, 'l_n'\} \not\equiv ('l_1' \circ 'l_2') \text{ OR } \dots \text{ OR } ('l_1' \circ 'l_n')$
- $'l_1' \circ ANY\{'l_2', \dots, 'l_n'\} \square k \not\equiv ('l_1' \circ 'l_2' \square k) \text{ OR } \dots \text{ OR } ('l_1' \circ 'l_n' \square k)$
- $'l_1' \circ ALL\{'l_2', \dots, 'l_n'\} \square k \not\equiv ('l_1' \circ 'l_2' \square k) \text{ AND } \dots \text{ AND } ('l_1' \circ 'l_n' \square k)$

For example, consider query  $Q_{18}$  in Table 9.1. The query states that there exist at most two A activities that are in parallel to B or C activities. Thus, a case containing four A activities, two parallel to an arbitrary number (greater than zero) of B activities, and two parallel to C activities, does not fulfill query  $Q_{18}$ . However, the described trace fulfills the query  $Q = ('A' \text{ isP } 'B' \leq 2) \text{ OR } ('A' \text{ isP } 'C' \leq 2)$ ; hence,  $E18 = 'A' \text{ isP } ANY\{'B', 'C'\} \leq 2 \neq Q$ .

<sup>2</sup>The term *syntactic sugar* was coined in [118] and refers to elements that extend a language's syntax to ease specifying certain constructs. However, these syntax extensions do not change the expressiveness and functionality of the language.

### 9.2.3. Query Evaluation

As exemplified in Figure 9.1, queries represent trees. Leaf vertices represent leaf queries that can be evaluated individually. Inner vertices and the root vertex represent Boolean operators. Thus, we can evaluate queries in a bottom-up fashion. First, the query leaves are evaluated for a given tree, i.e., a query. As a result, each leaf vertex can be assigned a Boolean value. Then, bottom-up, the given Boolean operators are applied recursively.

A complete evaluation is unnecessary for many queries to determine if a given case satisfies the specified constraints. For example, suppose one leaf vertex evaluates to false, and this leaf vertex's parent is a logical AND. In that case, its siblings must not be evaluated because its parent, representing a logical AND, can be immediately assigned the Boolean value *false*. Reconsider the query shown in Figure 9.1 and the case depicted in Figure 8.3b (page 222). The query comprises four query leaves. However, when evaluated in the correct order, only two queries are left to evaluate the entire query for the given case. Following a depth-first traversing strategy, the query leaf ('DC' isC =2) is evaluated first, satisfying the given case. Thus, we do not need to evaluate the right subtree of the OR, i.e., query leaves ('DC' isC =1) and ('CRR' isDF 'DC'). Finally, the query leaf ('DC' isDF 'DM') is evaluated. In short, we can evaluate the entire query by evaluating only two leaves. Furthermore, the order in which leaf queries are evaluated matters.

## 9.3. Illustrative Example

This section presents an illustrative example query. We use the BPI CH. 2012 event log, which contains partially ordered event data. Below, we depict an exemplary query.

```
'W_Completeren aanvraag' isP ALL {'A_ACCEPTED', 'O_SELECTED',
      'A_FINALIZED', 'O_CREATED', 'O_SENT'} AND
'A_ACCEPTED' isDF ALL {'A_FINALIZED', 'O_SELECTED'} AND
      'A_FINALIZED' isP 'O_SELECTED' AND
ALL {'A_FINALIZED', 'O_SELECTED'} isDF 'O_CREATED' AND
      'O_CREATED' isDF 'O_SENT';
```

Upon executing the above-specified query on the given event log, 1,167 out of 3,830 high-level variants satisfy the specified constraints. Figure 9.2 visualizes a few variants that satisfy the query. As specified in the query, all variants start with activity 'A\_SUBMITTED'. Furthermore, all variants contain an activity 'W\_Completeren aanvraag' that is in parallel to the activities 'A\_ACCEPTED', 'O\_SELECTED', 'A\_FINALIZED', 'O\_CREATED', and 'O\_SENT'. Finally, all further constraints specified in the example query are satisfied by the variants visualized in Figure 9.2. Besides specifying the start activity, the example query describes the activity pattern we see in the large middle chevron, which all variants contain.



Figure 9.2: Excerpt of variants from the BPI Ch. 2012 event log [228] that satisfy the query presented in Section 9.3

## 9.4. Evaluation

This section presents an evaluation of the proposed query language for partially ordered event data. The focus of this evaluation is on performance aspects. Section 9.4.1 presents the experimental setup. Next, Section 9.4.2 presents the results. Finally, Section 9.4.3 discusses the conducted experiments and elaborates threats to validity.

### 9.4.1. Experimental Setup

We use four real-life event logs, which are listed in Table 9.2. For each log, we automatically generated queries and then selected 1,000 queries that were not satisfied by all or by no case in the corresponding log. With this approach, we attempted to filter out trivial queries. We evaluated the performance-related statistics based on the 1,000 selected queries for each log.

Table 9.2.: Statistics about the event logs used

Event Log	#Cases	#Isomorphic high-level case views <sup>(a)</sup>
BPI Challenge 2012 [228]	13,087	3,830
BPI Challenge 2017 [229]	31,509	5,937
BPI Challenge 2020, Prepaid Travel Cost log [232]	2,099	213
Road Traffic Fine Management (RTFM) [56]	150,370	350

<sup>(a)</sup> Based on Definition 8.1 (page 221) using the lowest available bottom granularity.

### 9.4.2. Results

Each query is evaluated for all cases from the corresponding event log. However, not all leaf queries must be evaluated, as described in Section 9.2.3. Therefore, the number of leaves evaluated may vary when evaluating the same query for different cases. Thus, the case determines how many leaves of a given query must be evaluated. Figure 9.3 depicts the query evaluation runtime (in seconds) per event log for the median number of leaf nodes evaluated. Each plot summarizes 1,000 data points, that is, 1,000 queries evaluated on all cases from the corresponding log. A linear trend of increasing runtime is observed across all four event logs as the number of query leaves evaluated increases.

Figure 9.4 shows how queries are distributed according to their evaluation time. In addition, we can observe the ratio of leaves evaluated at the median. As before, each plot in the figure describes 1,000 queries. As in Figure 9.3, we derive that the number of evaluated leaf queries is the primary factor contributing to an increase in evaluation time, this trend is consistent across the different logs.

Finally, Figure 9.5 demonstrates the effect of early termination, i.e., not all leaf queries of a query are evaluated if not needed, as introduced in Section 9.2.3. Note that early termination was always used in the previous plots, i.e., Figure 9.3 and Figure 9.4. We

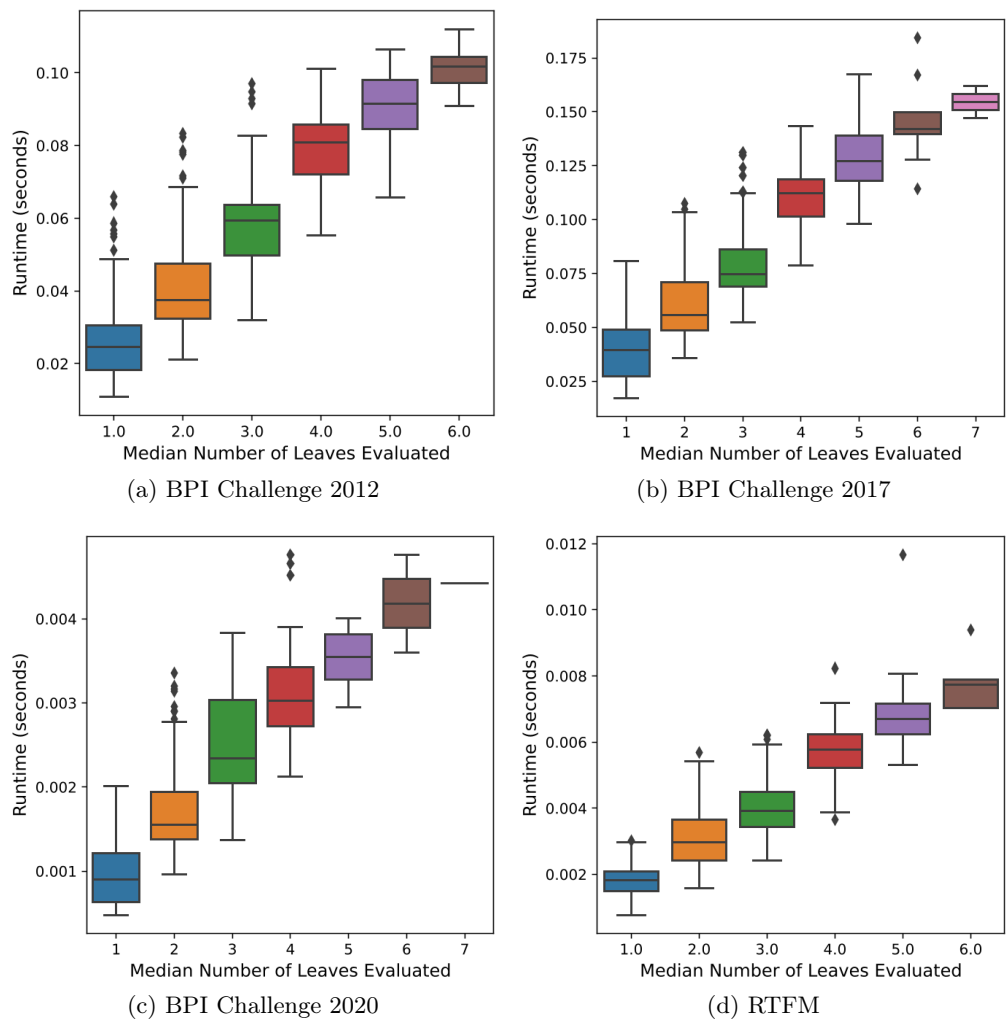


Figure 9.3: Query evaluation time; since the queries are applied to all cases, they are ordered by the median number of leaves evaluated per case (adapted from [181, Figure 5])

notice from the plots depicted in Figure 9.5 that early termination significantly impacts the evaluation time of the queries across all event logs. In conclusion, the results shown in this section indicate that the time required to evaluate queries increases linearly with the number of leaves evaluated.

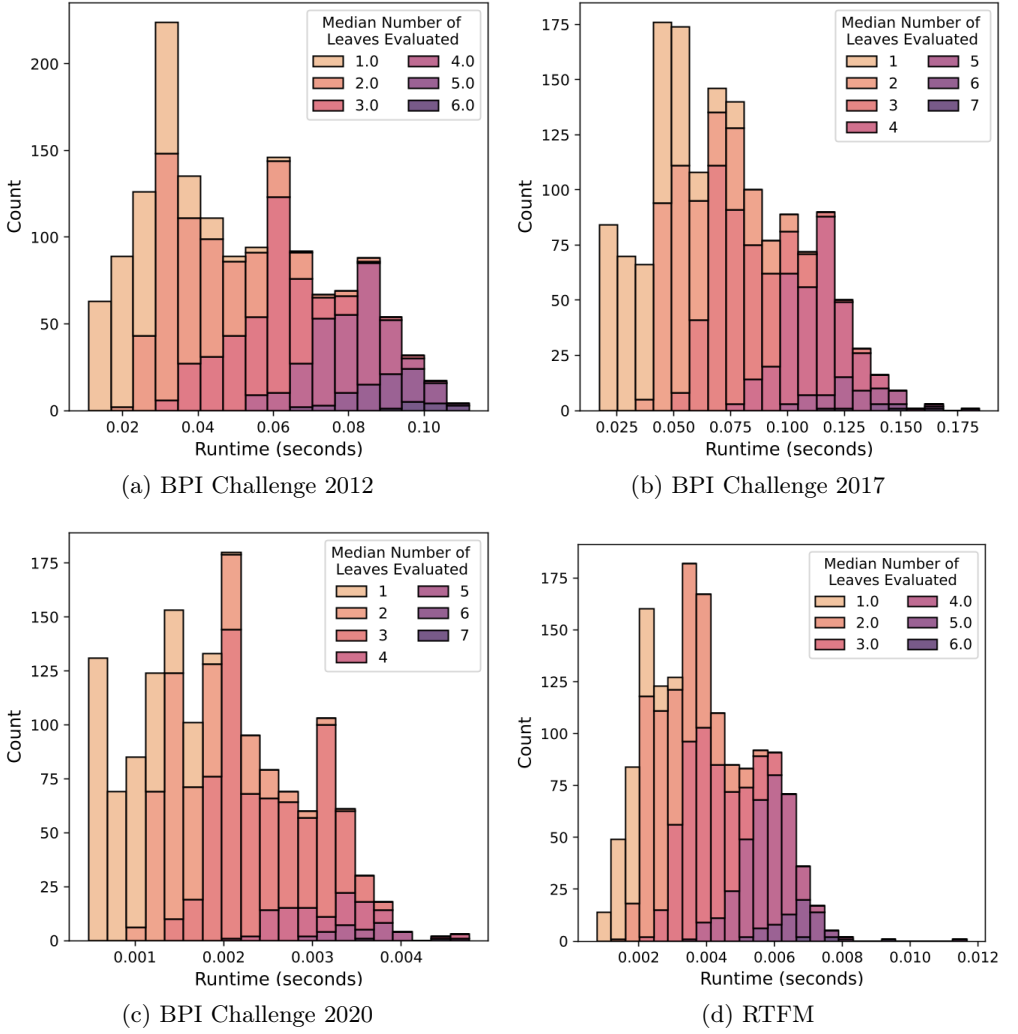


Figure 9.4: Query evaluation time distribution (adapted from [181, Figure 6])

### 9.4.3. Discussion & Threats to Validity

A potential threat to validity is the automatic query generation. Recall that we sample 1,000 queries from a large pool of automatically generated queries that at least evaluate for one case of a given event log to true. Thus, there is a high chance that many queries contain obvious activity constraints that are not satisfied by any case from a given event log. Thus, only a few query leaves might be evaluated to determine the overall result for a given case. Further, there is the threat that these automatically generated queries do not reflect queries that process analysts would specify in real-world scenarios. However,



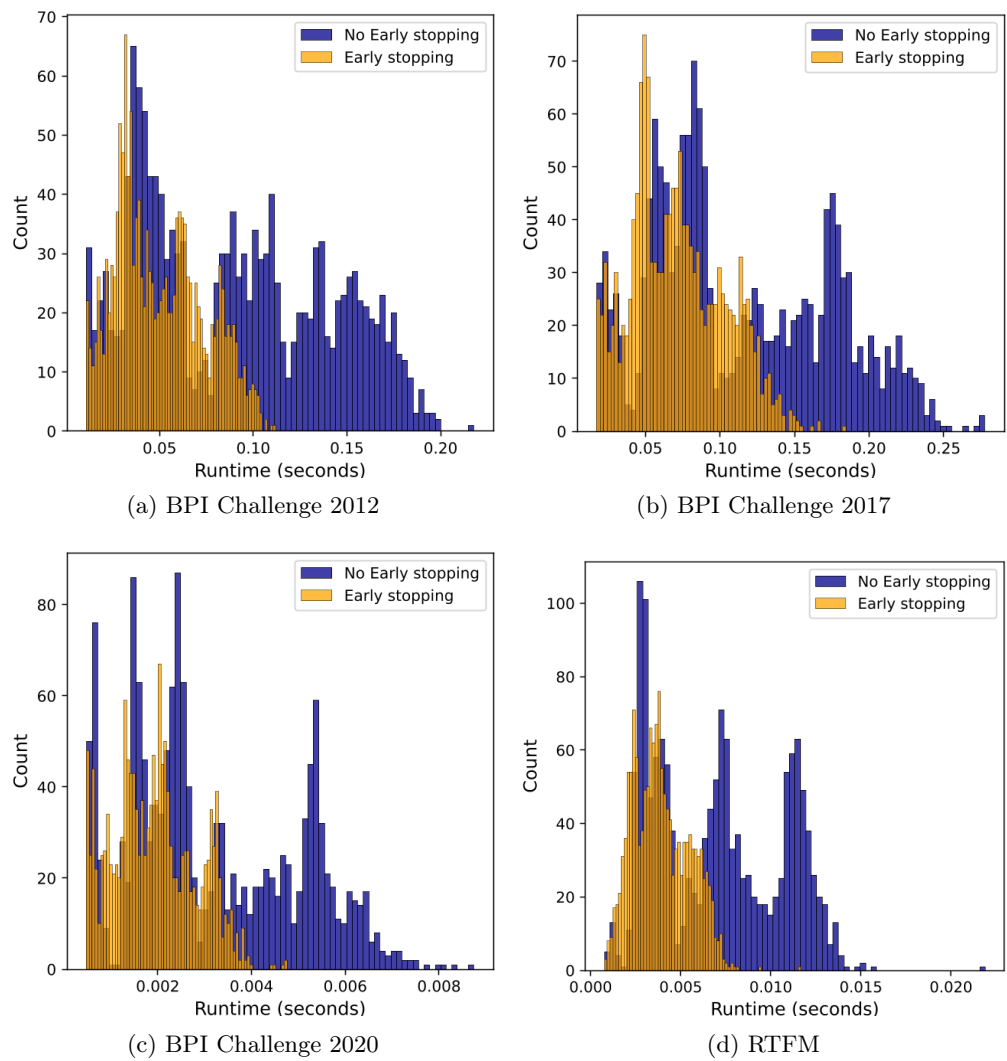


Figure 9.5: Impact of early termination on the query evaluation time (adapted from [181, Figure 7])

queries written by process analysts may also be heavily dependent on a particular event log and analysis task or question, making it difficult to generalize any potential results.

## 9.5. Conclusion

This chapter proposed a novel query language allowing query cases and variants containing partially ordered event data with heterogeneous temporal information. The proposed query language builds upon the high-level case view specified in Definition 8.1 (page 221). In the context of IPD, being able to query variants from an event log is critical to facilitate the user in incrementally selecting process behavior that is being incorporated into a process model.

The proposed query language could be extended to include operators to query cases based on their low-level case view, cf. Section 8.3.1 (page 231). Further, the query language could be extended to cover further aspects other than control flow constraints; for example, temporal constraints or resource constraints could be covered by the query language. Finally, the design of a visual query language using graphical elements similar the presented variant visualizations (cf. Sections 8.2.2 and 8.3.2) could be an interesting extension of the textual query language presented in this chapter.

## Part IV.

# Realization & Application



---

# Chapter 10.

## Tool Support: Cortado

---

This chapter is largely based on the following publications.

- D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Cortado: A dedicated process mining tool for interactive process discovery. *SoftwareX*, 22:101373, 2023. doi:10.1016/j.softx.2023.101373 [186]
- D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Cortado—an interactive tool for data-driven process discovery and modeling. In D. Buchs and J. Carmona, editors, *Application and Theory of Petri Nets and Concurrency*, volume 12734 of *Lecture Notes in Computer Science*, pages 465–475. Springer, 2021. doi:10.1007/978-3-030-76983-3\_23 [178]

Section 10.3.2 [Adding Behavior to a Process Model](#) is partly based on the following publication.

- D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Sub-model freezing during incremental process discovery in cortado. In *Proceedings of the ICPM Doctoral Consortium and Demo Track 2021*, pages 43–44. CEUR Workshop Proceedings, 2021. URL [https://ceur-ws.org/Vol-3098/demo\\_207.pdf](https://ceur-ws.org/Vol-3098/demo_207.pdf) [177]

Section 10.2.4 [Variant Frequent Pattern Mining](#) is largely based on the following publication.

- M. Martini, D. Schuster, and W. M. P. van der Aalst. Mining frequent infix patterns from concurrency-aware process execution variants. *Proceedings of the VLDB Endowment*, 16(10):2666–2678, 2023. doi:10.14778/3603581.3603603 [143]

Section 10.4.3 [Model-Based Performance Analysis](#) is largely based on the following publication.

- D. Schuster, L. Schade, S. J. van Zelst, and W. M. P. van der Aalst. Temporal performance analysis for block-structured process models in Cortado. In J. de Weerd and A. Polyvyanyy, editors, *Intelligent Information Systems*, volume 452 of *Lecture Notes in Business Information Processing*, pages 110–119. Springer, 2022. doi:10.1007/978-3-031-07481-3\_13 [182]

This chapter introduces the open-source software tool *Cortado*, which is a dedicated tool for incremental process discovery.<sup>1</sup> Cortado is an example of how the various algorithms and approaches proposed in this thesis can be seamlessly integrated into a comprehensive tool for incremental process discovery. This chapter's remainder is organized as follows. Section 10.1 provides an introduction to Cortado and presents its central functionalities and opportunities. In the subsequent sections, diverse functionality aspects are presented in detail as outlined in Figure 10.1. Section 10.2 introduces Cortado's various features for handling variants, which are an essential means of interaction within IPD. Next, Section 10.3 introduces Cortado's IPD functionality that comprises the algorithms proposed in Part II *Incremental Process Discovery* of this thesis. Section 10.4 presents the performance analysis features of Cortado. Section 10.5 lists the data exchange formats Cortado supports. Furthermore, Section 10.6 sketches Cortado's software architecture. Finally, Section 10.7 concludes this chapter.

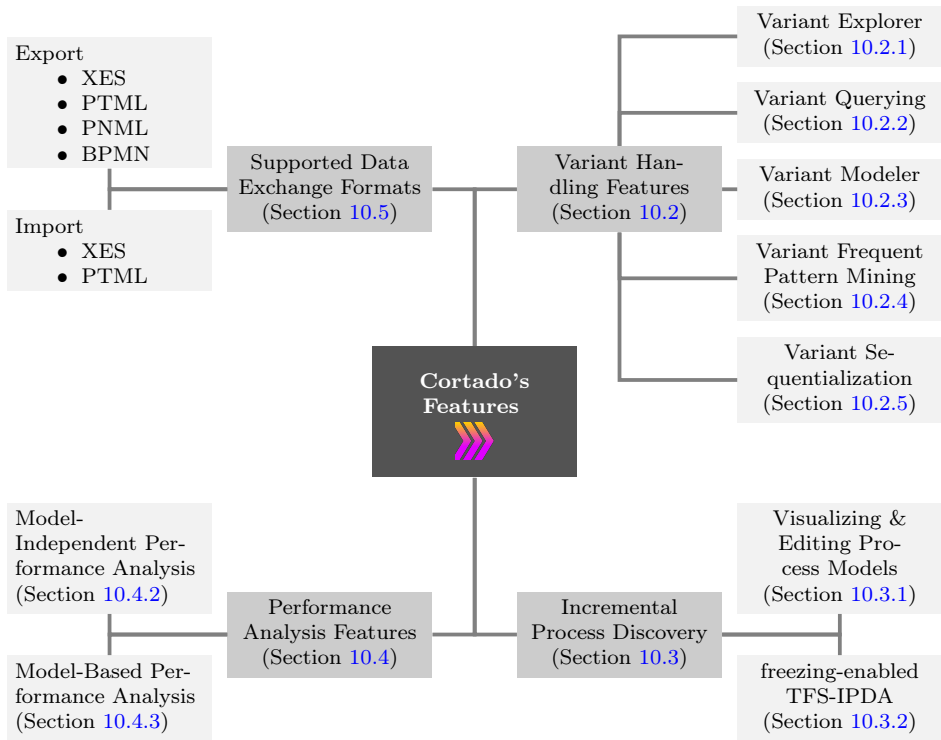


Figure 10.1: Overview of the most important features of Cortado and the structure of the following sections, which present the individual feature areas in detail

<sup>1</sup>Cortado's source code can be found online at <https://github.com/cortado-tool/cortado>.

# 10.1. Overview

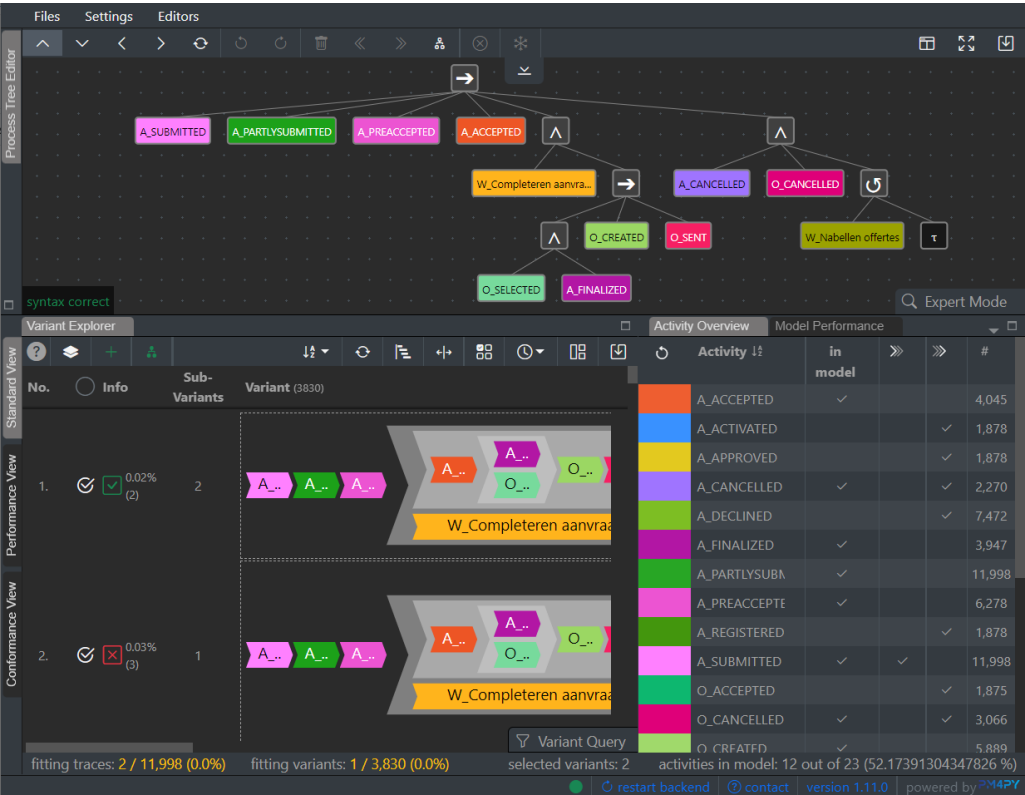


Figure 10.2: Screenshot of Cortado showing the process model editor at the top, the variant explorer in the lower left, and the activity overview table in the lower right (displayed variants originate from the BPI Ch. 2012 log [228])

This section introduces Cortado’s central functionality. Figure 10.3 (page 277) divides Cortado’s functionalities into four areas: (1) event data & variant handling, (2) conformance checking, (3) performance analysis, and (4) incremental process discovery. Feature areas (1) and (4) are visibly larger in Figure 10.3 because these are the core of Cortado. Figure 10.2 provides a first screenshot of Cortado showing the default User Interface (UI), consisting of the following components: the process model editor located in the upper part, the variant explorer in the lower left, and the activity overview table in the lower right.

Starting with an event log, Cortado detects high- and low-level variants shown in the *variant explorer*. The variant explorer is a central component of Cortado’s UI and allows users to explore the variants of an imported event log easily. As depicted in Figure 10.3, the variant explorer comprises various features such as variant querying, time granularity modification, and variant sequentialization rules. Further, users can add variants and

fragments to the variant explorer by either modeling them, extracting fragments from complete variants, or invoking frequent pattern mining for variants.

During incremental process discovery, users gradually select variants or fragments from the variant explorer that the IPDA algorithm implemented in Cortado incorporates into the provided process model. Users can import an initial process model or discover one from selected full variants invoking a conventional process discovery algorithm. Further, Cortado features a freezing-enabled IPDA; thus, model parts can be optionally frozen as indicated in Figure 10.3. The discovered process model can be viewed and edited in the process model editor anytime, providing maximal flexibility to users.

Besides the variant explorer and incremental process discovery, Cortado features conformance checking and temporal performance analysis. Conformance checking diagnostics provide valuable insights to the user about which variants are supported by the current process model; thus, conformance checking is essential within IPD. Temporal performance analysis allows linking the incrementally discovered model with the event log by calculating performance indicators.

While Figure 10.3 provides an overview of Cortado's functionality focusing on the interplay between the different functional areas, Figure 10.1 (page 274) provides a more structured overview of Cortado's features that are divided into four areas.

- *Variant handling* features are presented in Section 10.2. As variants are a crucial point of interaction within IPD, extensive support for handling variants is essential to realize an IPD software tool successfully. Cortado provides a variant explorer that allows users to assess high-level and low-level variants (cf. Chapter 8) of an imported event log. To handle large amounts of variants, the proposed query language (cf. Chapter 9) is implemented to enable querying variants. Cortado also features a variant modeler to manually incorporate process behavior not observed in the event log. Moreover, Cortado features frequent pattern mining for variants to identify frequent variant fragments that users might want to add to a process model incrementally (Section 10.2.4). Lastly, Cortado offers variant sequentialization rules that allow the modification of variants based on domain knowledge before incrementally adding them to a process model (Section 10.2.5).
- *Incremental process discovery* capabilities of Cortado are presented in Section 10.3. In short, Cortado implements a freezing-enabled TFS-IPDA. Thus, Cortado features all aspects of incremental process discovery as presented in Part II. Further, we also present Cortado's process model editor (Section 10.3.1).
- *Performance analysis* features of Cortado are presented in Section 10.4. The implemented performance analysis features can be further divided into *model-independent* and *model-based*. Model-independent performance analysis projects performance statistics onto variants in the variant explorer. In contrast, model-based performance analysis requires a process model as input besides an event log and project performance indicators onto a process model.
- Supported *data exchange formats* for process models and event logs are briefly discussed in Section 10.5.



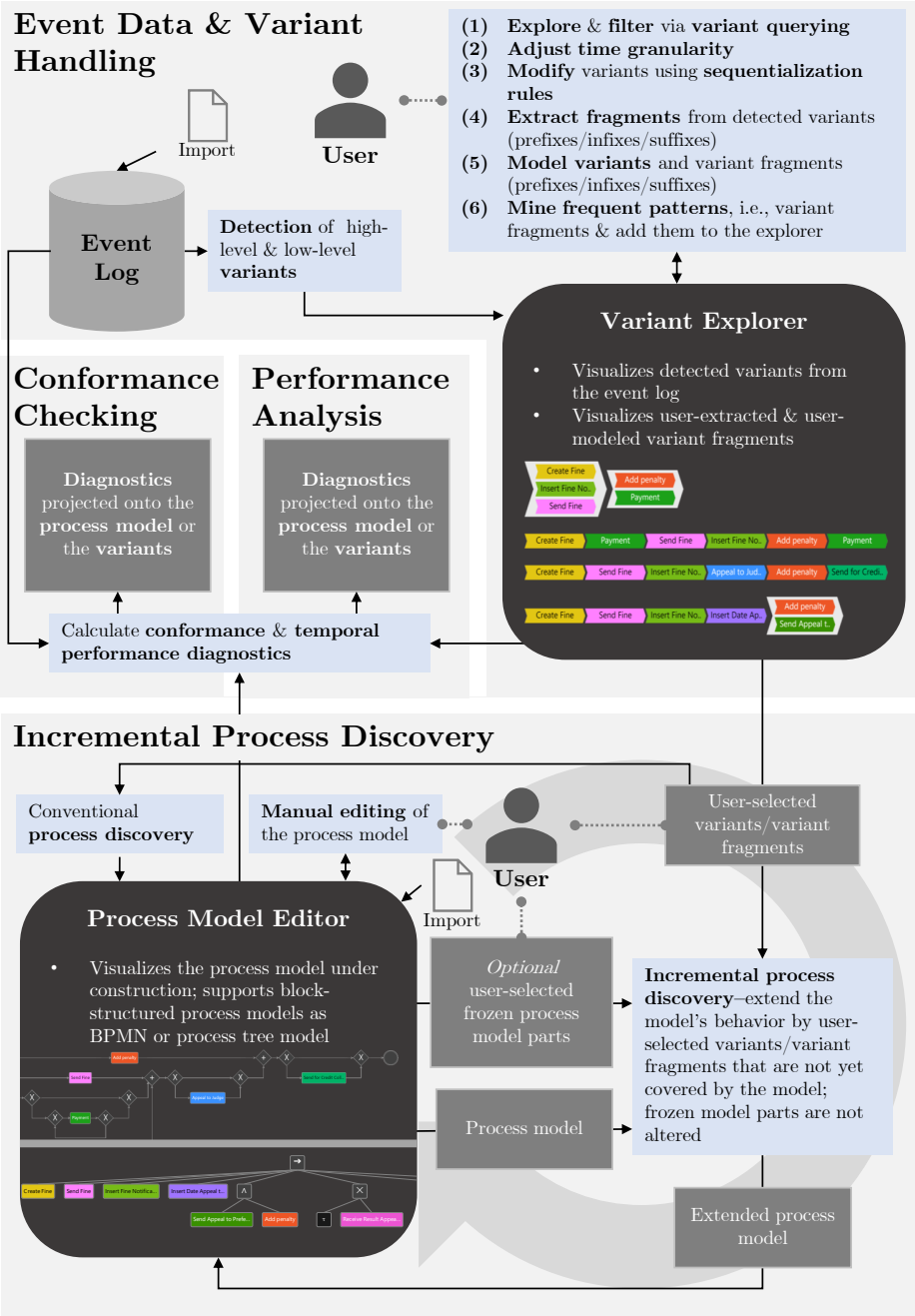


Figure 10.3: Overview of Cortado's central features and usage procedures within Cortado (partly adapted from [186, Figure 2])

## 10.2. Variant Handling

This section introduces the variant handling features of Cortado in detail. Starting from the variant explorer (Section 10.2.1), this section introduces variant querying (Section 10.2.2), the variant modeler (Section 10.2.3), frequent pattern mining functionality for variants (Section 10.2.4), and variant sequentialization techniques (Section 10.2.5).

### 10.2.1. Variant Explorer

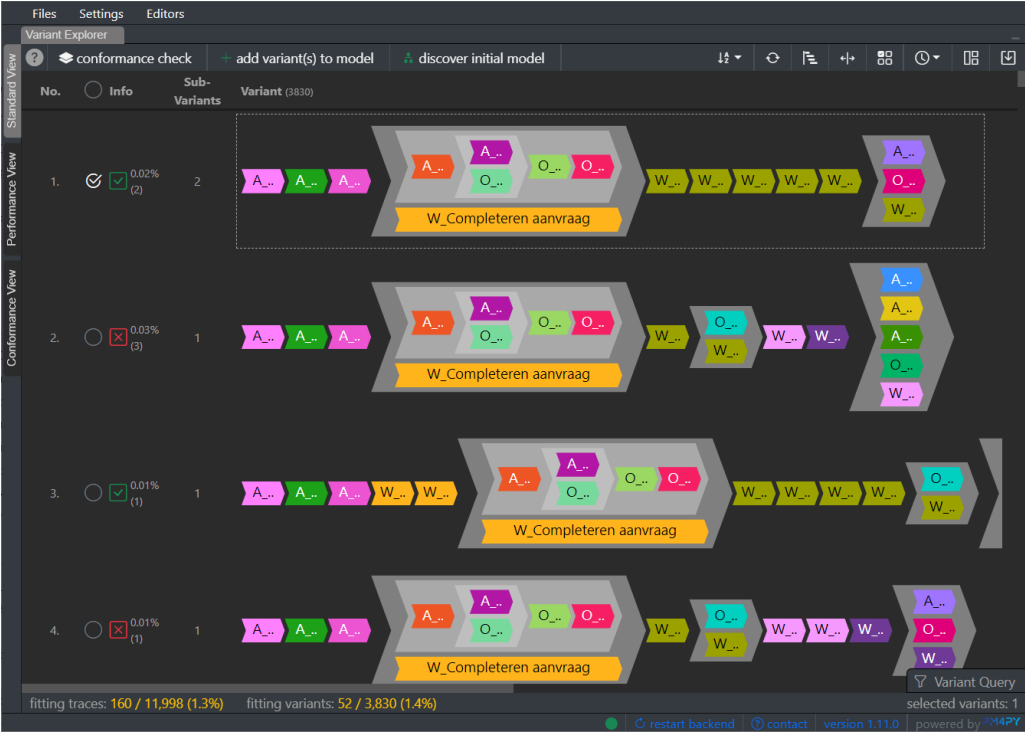


Figure 10.4: Full screen variant explorer showing high-level variants of the imported BPI Ch. 2012 [228] event log

The variant explorer is a central component in Cortado’s UI, cf. Figure 10.2. Upon starting Cortado, the variant explorer is located in the lower left by default. The variant explorer shows high-level variants from an imported event log. The activity overview table right to the variant explorer lists the various activities, assigns a unique color to each activity, and provides basic statistics, for instance, how often an activity occurs in the imported event log and if an activity is a start or end activity. Figure 10.4 shows the variant explorer in full-screen mode. Variants are displayed in a tabular manner, and each variant is enumerated; on the screenshot shown in Figure 10.4, four high-level variants are visible. Next to the enumeration, each variant can be selected. For instance,

the first variant is selected (cf. the circle icon containing a check mark left of the first high-level variant in Figure 10.4) while the three other ones are not (cf. the circle icons left of the second, third, and fourth high-level variant in Figure 10.4). Further, each variant is associated with a green check mark or a red cross mark. These icons indicate if the variant is supported by the currently loaded process model.<sup>2</sup>

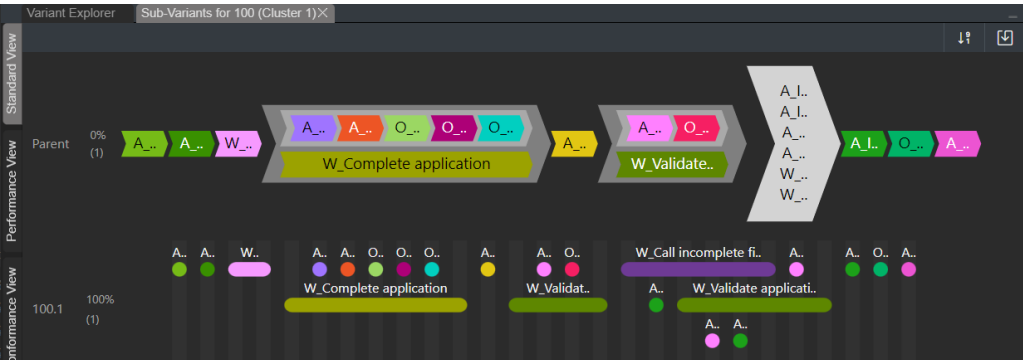


Figure 10.5: Variant explorer, which in this case displays the only low-level variant for a selected high-level variant that contains a chevron for which no other partitions could be found (cf. Section 8.2.3)

In the column sub-variants, a number for each displayed variant is shown. These numbers indicate the number of corresponding low-level variants per high-level variant. Recall a one-to-many relation generally exists between high-level and low-level variants, cf. Chapter 8. When clicking on the number of low-level variants, a tab with the corresponding low-level variants opens. Figure 10.5 shows a low-level variant for a corresponding high-level variant from the event log [229]. In the depicted case, the high-level variant contains exactly one low-level variant. Further note that the high-level variant contains a chevron for which no further partition could be found, i.e., the gray one holding six activities. Recall Section 8.2.3, explaining this phenomenon for high-level variants in detail. The low-level variant below can visualize all relations among the activities representing time intervals and time points.

The primary focus on high-level variants in the variant explorer stems from the fact that Cortado is primarily an interactive process discovery tool. The detail level of low-level variants is too high, so low-level variants' information often cannot be represented directly in process models. For instance, process model formalisms do not distinguish between activities being executed instantly, i.e., representing a time point, and activities being executed over time, i.e., representing a time interval. If separate entities in the process model represent the start and completion of activities, for example, two transitions in a Petri net represent the start and completion of an activity, low-level variants may be interesting for incremental process discovery. However, Cortado currently only considers low-level variants as a more detailed view of high-level variants.

<sup>2</sup>Note that the used process model is not visible in Figure 10.4.

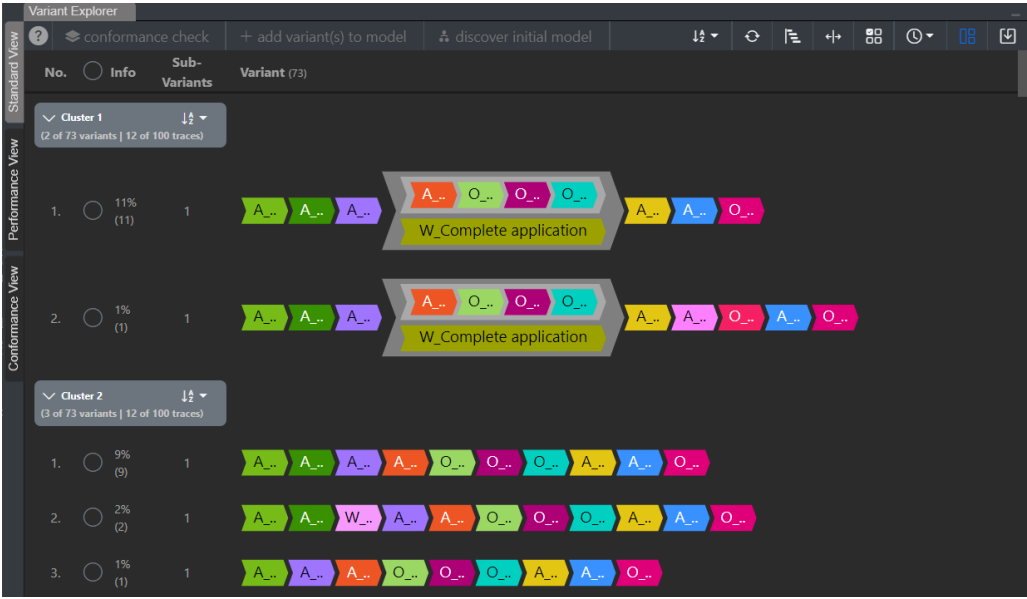
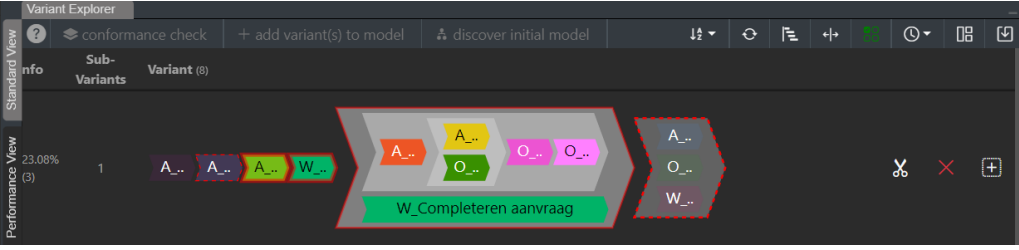


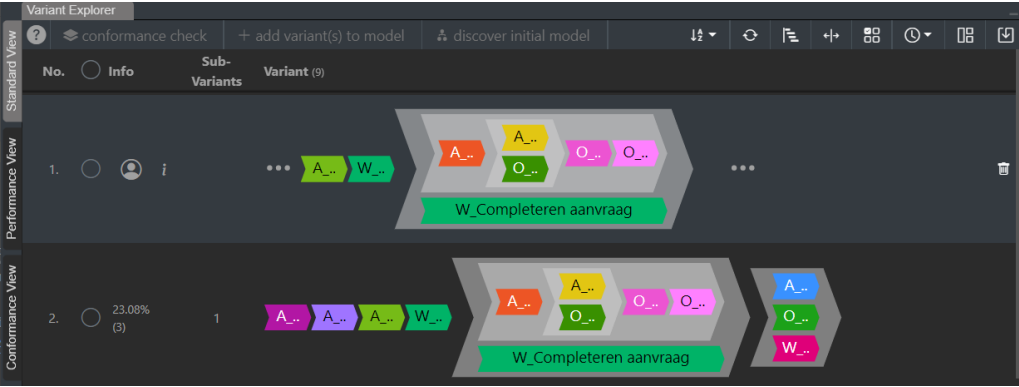
Figure 10.6: Variant explorer showing clusters using agglomerative clustering (max. edit distance: 3) on a BPI Ch. 2017 log [229] sample

## Variant Clustering

To cope with vast amounts of variants, Cortado offers clustering techniques for high-level variants. Two approaches are implemented in Cortado: *label vector clustering* and *agglomerative edit distance clustering*. The label vector clustering technique assumes a user-defined number of clusters. The high-level variants are transformed into vectors where each entry represents the occurrence of an activity label in the high-level variant. Next, these vector representations are used to construct clusters using the k-means clustering algorithm implemented in the machine learning library scikit-learn [84]. Agglomerative edit distance clustering uses the tree representation of high-level variants. Recall that each high-level variant can be represented as a tree; for example, consider Figure 8.4 on page 226. Cortado uses agglomerative clustering implemented in the machine learning library scikit-learn [84] to create the clusters. In this context, users must specify the maximum edit distance of the variants within a cluster. Figure 10.6 shows Cortado's variant explorer with applied clustering. In the screenshot, agglomerative clustering is applied to a sample of the event log BPI Ch. 2017 [229]. The first shown cluster consists of two high-level variants, while the second cluster contains three. In short, clustering high-level variants facilitates handling large amounts of variants.



- (a) Variant explorer in *fragment selection mode*; by clicking on chevrons, users can extract individual fragments (an infix in this case) from variants which are then displayed as fragments in the explorer together with complete variants



- (b) Variant explorer displaying the extracted variant fragment (cf. Figure 10.7a) along with other variants in a shared pool of complete variants derived from the imported event log

Figure 10.7: Extracting variant fragments from complete variants in Cortado

## Variant Fragment Extraction

Variant fragments are essential for incremental process discovery with fragments, cf. Chapter 6. To this end, Cortado allows to extract fragments from variants as exemplified in Figure 10.7. Figure 10.7a shows the variant explorer in *fragment selection mode*. This mode allows users to select fragments from a complete variant. In the depicted example, the large chevron is completely selected along with the two preceding activities. Note that Cortado only allows users to cut connected fragments. The selected fragment (cf. Figure 10.7a) can be added to the overall pool of variants. Upon extracting, the fragment is displayed in the variant explorer, i.e., a shared pool of variants originating from the imported event log and user-extracted variant fragments, cf. Figure 10.7b.

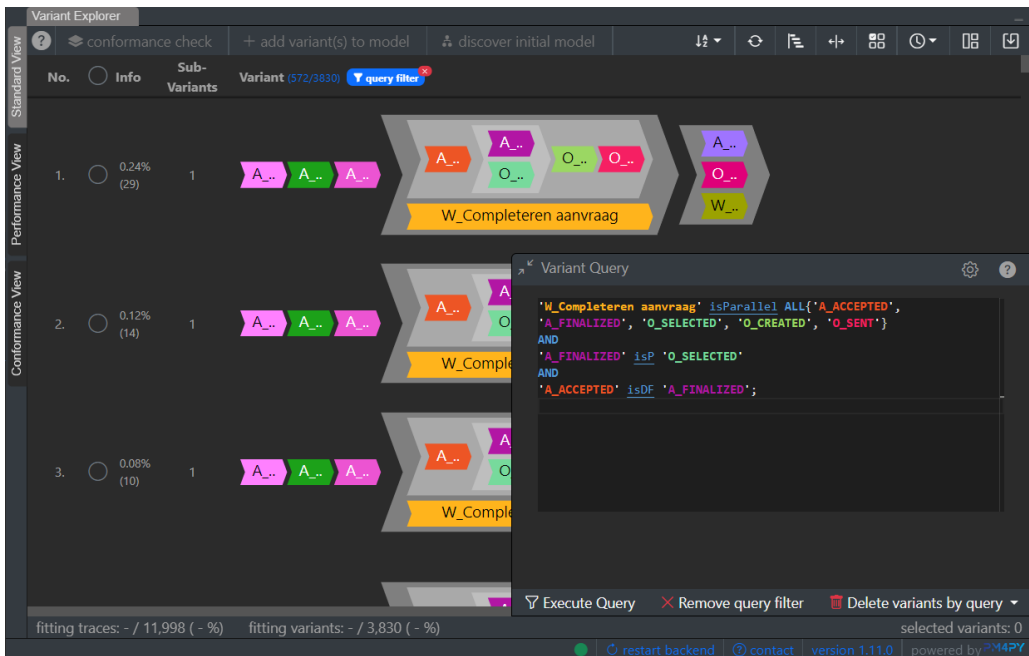


Figure 10.8: Variant explorer showing only variants satisfying the specified query

### 10.2.2. Variant Querying

Most event logs contain many variants, where the visual approach of the variant explorer reaches its limits, as it is challenging for users to find variants from a pool of thousands. To this end, Cortado implements the query language proposed in Chapter 9. The query language allows users to specify control-flow constraints that variants must satisfy. Figure 10.8 shows the variant explorer with an executed user-specified query executed on the imported event log; thus, all visible variants satisfy the specified query.

The query shown in Figure 10.8 specifies that activity 'W\_Completeren aanvraag' is parallel to: 'A\_ACCEPTED', 'A\_FINALIZED', 'O\_SELECTED', 'O\_CREATED', and 'O\_SENT'. Moreover, there exists activities 'A\_FINALIZED' and 'O\_SELECTED', as well as 'A\_ACCEPTED' and 'A\_FINALIZED' in parallel each. All variants visible in Figure 10.8 satisfy the provided query. In detail, the large chevron in the middle of these variants that contains activity 'W\_Completeren aanvraag' in parallel to a sequence of various other activities is the reason why these variants satisfy the specified query.

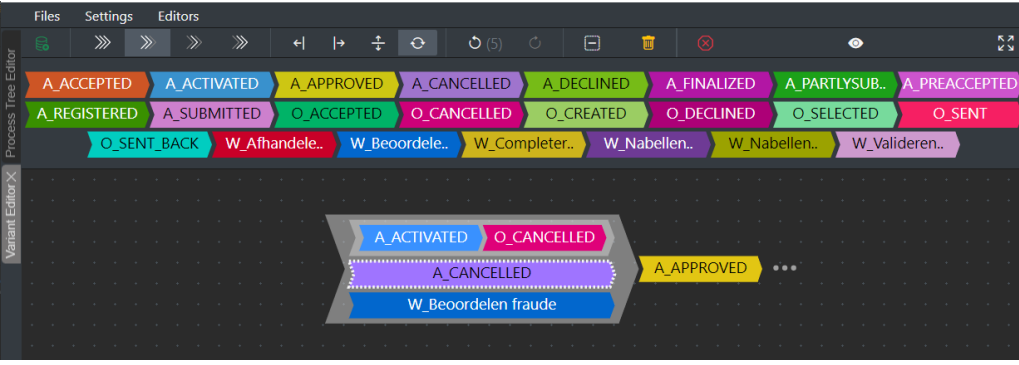


Figure 10.9: Screenshot of Cortado's variant modeler showing a variant prefix

### 10.2.3. Variant Modeler

During incremental process discovery, users may want to incorporate process behavior that has not been recorded in the event data. This is where Cortado's variant modeler comes in (cf. Figure 10.9), enabling users to model variants and variant fragments. In the screenshot of the variant modeler, users can access all activities from the imported event log, which are represented as chevrons at the top of the screen. The modeled variant fragment in the screenshot is a prefix since the variant ends with three gray dots that indicate that there may be further, unspecified process behavior. Once users have finished modeling their variants/variant fragments, they can add them to the pool, where they will be displayed and can be used within incremental process discovery.

### 10.2.4. Variant Frequent Pattern Mining

Next to manual variant fragment extraction (cf. Section 10.2.1) and variant/variant fragment modeling (cf. Section 10.2.3), Cortado features frequent pattern mining for variants. Figure 10.10 depicts a screenshot of the *frequent pattern miner*. On the left, diverse settings can be specified and on the right, the frequent variant patterns, i.e., variant fragments comprising variant prefixes, infixes, and suffixes, are displayed similar like in the variant explorer. For each frequent variant fragment, we can calculate conformance checking statistics, cf. red and green colored icons.<sup>3</sup> Further, we see per frequent pattern, its support value and whether it is closed or maximal.<sup>4</sup>

The frequent variant pattern approach implemented in Cortado exploits the inherent tree structure of variants; for instance, recall Figure 8.4 on page 226. Therefore, we can reduce the problem of finding frequent patterns in variants to *frequent subtree mining* over labeled rooted ordered trees [159]. Cortado implements three algorithms to mine

<sup>3</sup>To determine if a process model supports a given variant fragment, we compute all potential sequentializations of the given variant fragment and use alignments for trace fragments as introduced in Chapter 4.

<sup>4</sup>A frequent pattern is closed if there does not exist a super-pattern with the same support. A frequent pattern is maximal if there exists no super pattern that is frequent, too.

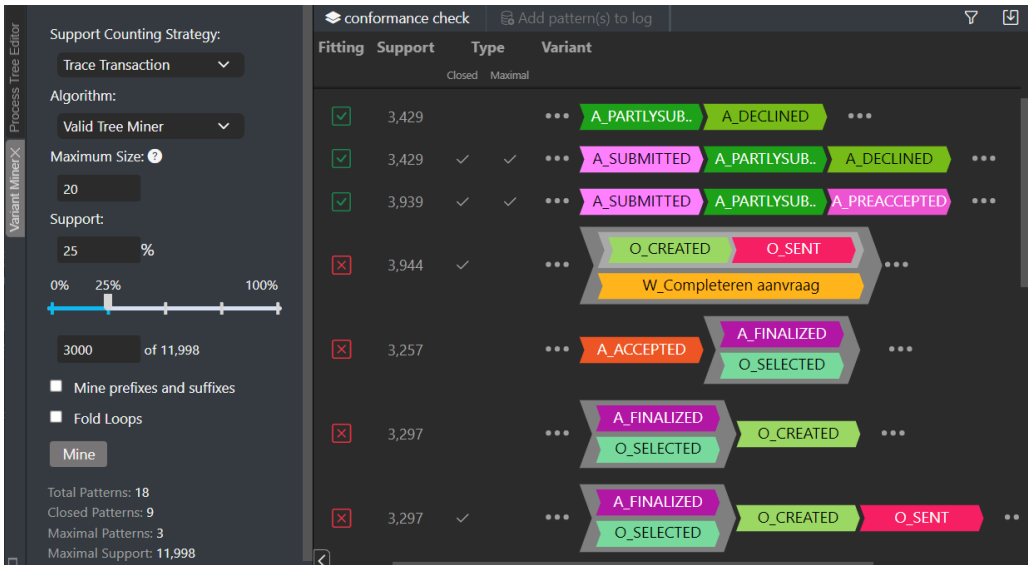


Figure 10.10: Screenshot of Cortado’s frequent pattern mining feature

frequent patterns that are eligible in the settings section on the left, cf. Figure 10.10. The default algorithm is the *Valid Tree Miner* [143]. This algorithm mines frequent *infix subtrees* that preserve sequential completeness, i.e., no activity within a sequence construct is skipped. Besides the algorithm, users must specify the support counting strategy. Two dimensions exist: root occurrence vs. transaction and trace vs. variant, resulting in four support counting strategies. Further, users must set a maximum pattern size and a support threshold.

Upon mining frequent variant patterns, users can explore and filter them based on size, support, closed/maximal properties, fragment type, and activities contained. Moreover, users can add frequent variant patterns, i.e., variant fragments, to the pool of variants and variant fragments. Once these variant fragments are added to the pool, they are displayed along with other variants and variant fragments as exemplified in Figure 10.7b. Therefore, all functions of the variant explorer are also available for these variant fragments.

### 10.2.5. Variant Sequentialization

The variant sequentialization feature allows users to ingest domain knowledge into variants. Recall the conceptual overview of domain knowledge utilization in the context of process discovery shown in Figure 2.1 (page 20). Variant sequentialization is an event data preprocessing feature that utilizes a priori domain knowledge to modify variants. Recall time granularity modification for variants as discussed in Section 8.5 (page 236 ff.). In short, moving to a coarser bottom granularity generally leads to more parallel behavior in variants; less sequentially oriented activities remain. The reasons for adjusting the bottom granularity lie in the questions that process analysts try to answer.



Whether an activity starts a few seconds before another may be irrelevant for analyzing many processes. It may even be a hindrance, as these few seconds between activities would cause the variant to show a sequential alignment between these activities, which are only a few seconds apart. Instead, a process analyst, for example, is more interested in analyzing the process daily, i.e., activities performed on the same day are considered parallel.

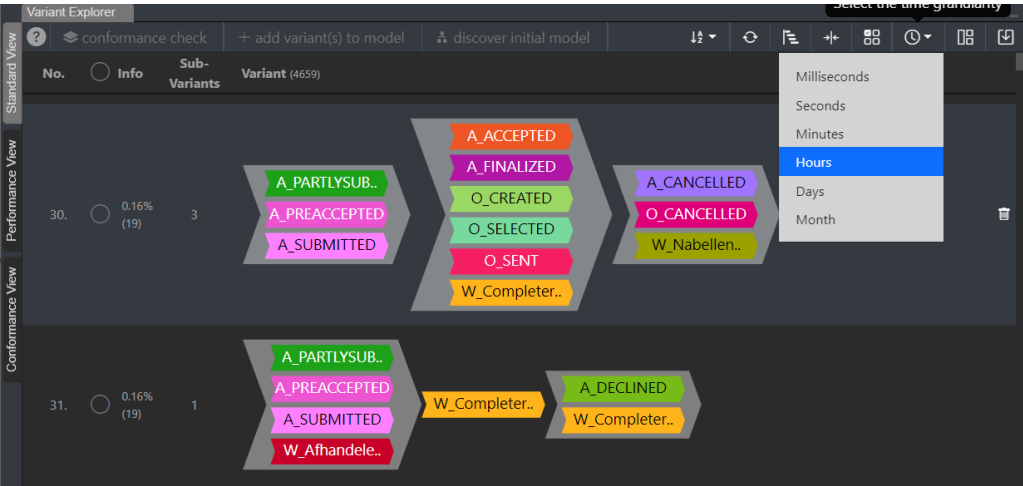


Figure 10.11: Variant explorer showing variants when the bottom granularity is set to hours instead of milliseconds (default bottom granularity) for the event log BPI Ch. 2012 [228]

As introduced in Section 8.5, Time Granularity Modifiers (TGMs) can be applied to modify the bottom granularity. Figure 10.11 depicts a screenshot of the variant explorer showing variants where the bottom granularity was set to hours instead of milliseconds, cf. Figure 10.4 (page 278). As exemplified in Figure 10.11 and discussed in Section 8.5, coarser bottom granularities generally lead to more parallel activities. Moreover, adjusting the temporal granularity might be necessary to adequately answer questions about the process being analyzed in many scenarios. Nevertheless, when switching to coarser time granularity, sequential dependencies between activities worth preserving may be lost, and these activities may become parallel. Therefore, Cortado offers the variant sequentializer functionality that allows the application of sequentialization rules consisting of source and target patterns. For instance, consider Figure 10.12 showing the source/target pattern modeler. In the example, the source pattern specifies that activities A\_ACCEPTED and A\_FINALIZED are in parallel. Moreover, further activities might be in parallel, as indicated by the chevron labeled with three dots. The source pattern specifies that A\_ACCEPTED should occur before A\_FINALIZED. Potential other activities, indicated by the chevron labeled with three dots, are executed in parallel to the sequence consisting of A\_ACCEPTED and A\_FINALIZED. Upon applying this sequentialization rule, all variants in the variant pool are checked for occurrences of the source pattern. If the source

pattern occurs, the corresponding occurrence in the variant/variant fragment is replaced by the target pattern. For instance, recall the variants shown in Figure 10.11. The first shown variant contains the source pattern specified in Figure 10.12 while the second variant does not. After applying the sequentialization rule consisting of the source/target pattern shown in Figure 10.12, the first variant changes as depicted in Figure 10.13.



Figure 10.12: Screenshot of Cortado's sequentializer functionality showing the source/-target pattern modeler

In summary, Cortado's variant sequentialization feature enables the incorporation of domain knowledge into variants that are eventually used in IPD. Variant sequentialization can be used to restore or create sequential order between activities running in parallel. Especially in combination with the option to change the time granularity (especially the change to coarse time granularity), variant sequentialization can be a helpful feature.

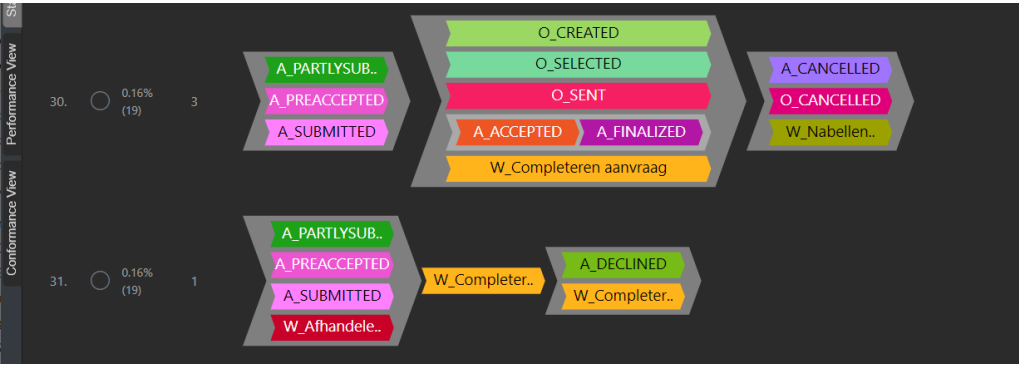


Figure 10.13: The two variants shown in Figure 10.11 *after* applying the source/target pattern rule shown in Figure 10.12; the specified source/target pattern rule applies to the first variant but not to the second one because the second one does not contain the source pattern, cf. Figure 10.11

## 10.3. Incremental Process Discovery

This section introduces the incremental process discovery features of Cortado. Section 10.3.1 introduces Cortado’s process tree editor and BPMN visualizer. Subsequently, Section 10.3.2 introduces the IPD features of Cortado from a user’s perspective.

### 10.3.1. Visualizing & Editing Process Models

The process model editor is an integral component of Cortado’s UI. Upon starting Cortado, the process model editor is by default located in the upper part, cf. Figure 10.2 (page 275). The core concept of Cortado is that users focus on a central process tree, which is presented in the process tree editor. The editor allows users to manipulate the tree at any time, cf. Figure 10.14a. Alternatively to the process tree editor, Cortado allows visualizing the process tree as a BPMN model, cf. Figure 10.14b. Moreover, recall freezing-enabled IPD, cf. Chapter 7. Users can freeze or unfreeze subtrees within the process tree editor and the BPMN visualizer. For instance, the right subtree of the process tree shown in Figure 10.14a is currently frozen. The process model editor indicates frozen subtrees with blue-colored vertices.

### 10.3.2. Adding Behavior to a Process Model

Cortado implements the LCA-IPDA proposed in Chapter 5 along with the extension for trace/variant fragments (cf. Chapter 6) and the extension to freeze subtrees (cf. Chapter 7) in an integrative fashion. From a users perspective, incremental process discovery boils down to selecting variants/variant fragments from the variant explorer and pressing the corresponding button for adding unsupported, selected process behavior to the model.



conventional process discovery algorithm.<sup>6</sup>

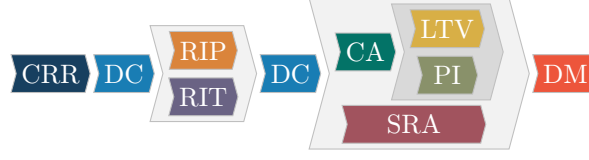
Upon having a process tree available, users can utilize the incremental process discovery feature in Cortado. To this end, users select variants/variant fragments from the variant explorer. For instance, the two variants in Figure 10.4 (page 278) are selected by a user. Note that the first variant is supported by the current process model (indicated by the green check mark icon left to the variant); however, the second variant is not (indicated by the red icon). Upon clicking the button labeled with the plus icon, Cortado's IPDA is invoked, and the process model is updated. Variants/variant fragments selected and supported by the current process model are considered previously added process behavior. Thus, according to the IPD framework, these variants remain supported by the resulting process model. Variants/variant fragments selected but not supported by the current process model are considered process behavior to be added next. Furthermore, users can additionally freeze subtrees in the process tree before invoking the IPDA, as exemplified in Figure 10.14.

Recall the formal specifications of the IPDAs presented in Part II, cf. Figures 5.1, 6.2 and 7.2 on pages 110, 159 and 180. All proposed frameworks and IPDAs assume traces consisting of totally ordered activities, i.e., sequences of activities (cf. Definition 3.20 on page 61). However, Cortado uses high-level variants consisting of partially ordered activities. To this end, Cortado generates in its backend all potential sequentializations of the selected variants/variant fragments to obtain traces/trace fragments. For example, consider Figure 10.15 showing two high-level variants and a specification of their corresponding set of all potential sequentializations each. The first depicted variant results in 16 sequentializations, cf. Figure 10.15a. Note that when calculating sequentializations, all potential sequential orderings of activities parallel to others are considered. Thus, the 16 sequentializations result from two options to order activities RIP and RIT, two to order LTV and PI, and four options to place SRA; hence,  $2 * 2 * 4 = 16$ . Similarly, the second variant is sequentialized, cf. Figure 10.15b. Note that Cortado treats the chevron containing the activities CA, DC, LTV, PI, and SRA as if all these activities were in parallel. Thus, the second variant (cf. Figure 10.15b) yields 240 sequentializations in total.

The described sequentializations are performed in the backend of Cortado. Subsequently, all generated sequentializations are added incrementally to the process model. From the user's point of view, these intermediate executions of the IPDA are not visible; after all sequentializations have been incrementally added, the process model is updated in the process model editor. Suppose the user has selected multiple variants in the variant explorer that the current process model does not support. In that case, each of these variants will be sequentialized, their sequentialized traces will be incrementally added by the IPDA, and finally, Cortado updates the process model in the process model editor.

---

<sup>6</sup>Note that variant fragments are not supported for initial process model discovery because the Inductive Miner, as well as most other conventional process discovery approaches, do not support variant fragments.



$$\begin{aligned}
 & \{ \langle CRR, DC \rangle \} \circ \{ \langle RIP, RIT \rangle, \langle RIT, RIP \rangle \} \circ \{ \langle DC \rangle \} \circ \\
 & \quad \left( \{ \langle CA, LTV, PI \rangle, \langle CA, PI, LTV \rangle \} \diamond \{ \langle SRA \rangle \} \right) \circ \{ \langle DM \rangle \} = \\
 & \{ \sigma_1 = \langle CRR, DC, RIP, RIT, DC, SRA, CA, LTV, PI, DM \rangle, \\
 & \sigma_2 = \langle CRR, DC, RIP, RIT, DC, CA, SRA, LTV, PI, DM \rangle, \\
 & \sigma_3 = \langle CRR, DC, RIP, RIT, DC, CA, LTV, SRA, PI, DM \rangle, \\
 & \sigma_4 = \langle CRR, DC, RIP, RIT, DC, CA, LTV, PI, SRA, DM \rangle, \\
 & \sigma_5 = \langle CRR, DC, RIP, RIT, DC, SRA, CA, PI, LTV, DM \rangle, \\
 & \sigma_6 = \langle CRR, DC, RIP, RIT, DC, CA, SRA, PI, LTV, DM \rangle, \\
 & \sigma_7 = \langle CRR, DC, RIP, RIT, DC, CA, PI, SRA, LTV, DM \rangle, \\
 & \sigma_8 = \langle CRR, DC, RIP, RIT, DC, CA, PI, LTV, SRA, DM \rangle, \\
 & \sigma_9 = \langle CRR, DC, RIT, RIP, DC, SRA, CA, LTV, PI, DM \rangle, \\
 & \sigma_{10} = \langle CRR, DC, RIT, RIP, DC, CA, SRA, LTV, PI, DM \rangle, \\
 & \sigma_{11} = \langle CRR, DC, RIT, RIP, DC, CA, LTV, SRA, PI, DM \rangle, \\
 & \sigma_{12} = \langle CRR, DC, RIT, RIP, DC, CA, LTV, PI, SRA, DM \rangle, \\
 & \sigma_{13} = \langle CRR, DC, RIT, RIP, DC, SRA, CA, PI, LTV, DM \rangle, \\
 & \sigma_{14} = \langle CRR, DC, RIT, RIP, DC, CA, SRA, PI, LTV, DM \rangle, \\
 & \sigma_{15} = \langle CRR, DC, RIT, RIP, DC, CA, PI, SRA, LTV, DM \rangle, \\
 & \sigma_{16} = \langle CRR, DC, RIT, RIP, DC, CA, PI, LTV, SRA, DM \rangle \}
 \end{aligned}$$

- (a) Specifying all potential 16 sequentializations derived from the depicted high-level variant (cf. Figure 8.4 on page 226); the variant's corresponding tree structure is depicted in Figure 8.5 on page 227



$$\begin{aligned}
 & \{ \langle CRR, DC \rangle \} \circ \{ \langle RIP, RIT \rangle, \langle RIT, RIP \rangle \} \circ \{ \langle DC \rangle \} \circ \\
 & \quad ( \langle CA \rangle \diamond \langle DC \rangle \diamond \langle LTV \rangle \diamond \langle PI \rangle \diamond \langle SRA \rangle ) \circ \{ \langle DM \rangle \}
 \end{aligned}$$

- (b) Specifying all  $2 * 5! = 240$  potential sequentializations (2 options for arranging RIP and RIT combined with  $5!$  options to arrange the activities CA, DC, LTV, PI, and SRA) derived from the high-level variant (cf. Figure 8.7 on page 229); the variant's corresponding tree structure is depicted in Figure 8.8 on page 229

Figure 10.15: Examples for the generation of all potential sequentializations of high-level variants

## 10.4. Temporal Performance Analysis

Analyzing the temporal performance of processes is crucial for many practical applications, such as bottleneck identification within processes. This section presents Cortado’s temporal performance analysis features that can be classified into model-independent (Section 10.4.2) and model dependent performance analysis (Section 10.4.3). Both approaches implemented in Cortado advance the state-of-the-art of temporal performance analysis.

### 10.4.1. Overview

Performance analysis is about gaining temporal knowledge about the process under consideration. For this purpose, the timestamps, which are an essential part of event data, are analyzed. Performance aspects are critical in process mining analysis as temporal optimizations of processes, for example, reduced cycle times and bottleneck detection, are of great importance to process owners. Generally, two types of performance analysis can be distinguished: model-based and model-independent performance analysis. Figure 10.16 provides a high-level overview of these two types.

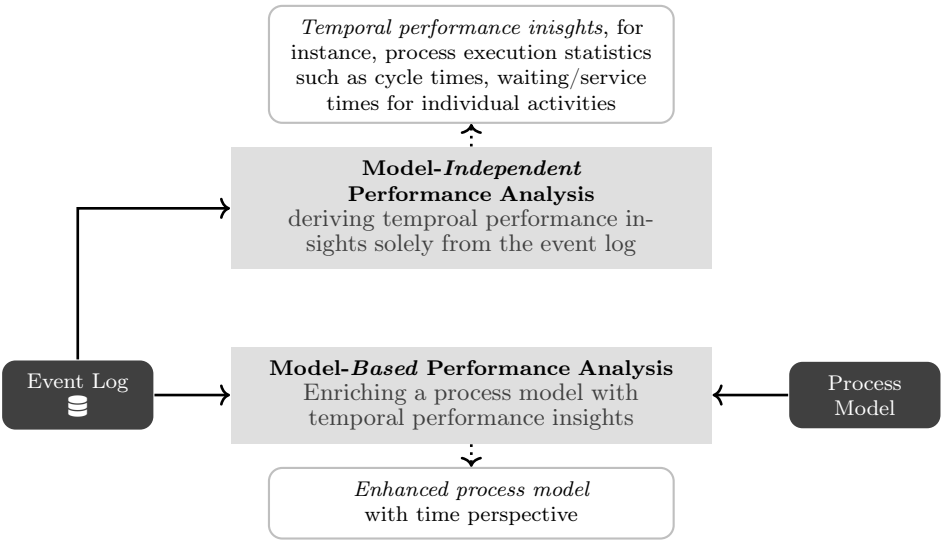


Figure 10.16: Overview of two types of performance analysis, i.e., model-independent and model-based temporal performance analysis

*Model-independent performance analysis* purely uses the recorded process behavior from the event log to provide temporal performance insights. Such insights comprise statistics on: process execution cycle times, execution time of individual activities, waiting, and idle times. To this end, standard statistical methods are often used. Moreover, timeline charts, cf. [211, Figure 9.12 and 9.13] and [78, Figure 11.16], and similar plot types are often used to visualize temporal behavior of individual process executions.

*Model-based performance analysis* enhances process models with performance statistics derived from the event log. Enriching process models with performance indicators is also referred to as *time perspective* [55]. Note that we have focused primarily on the control flow perspective so far. In order to provide performance statistics for a given process model, the recorded process executions from the event log must be aligned with the model, i.e., the process executions recorded in the event log must be replayed on the process model. Thus, model-based performance analysis techniques are tightly coupled with conformance checking techniques, especially with alignments [226]. Once process executions from the event log are linked with activities in the model, the process model can be enriched with performance indicators, for instance, showing which activities in the process model have high cycle times. In [6], the author describes how we can derive from alignments model-based temporal performance insights, i.e., enriching a model with temporal statistics.

### 10.4.2. Model-Independent Performance Analysis

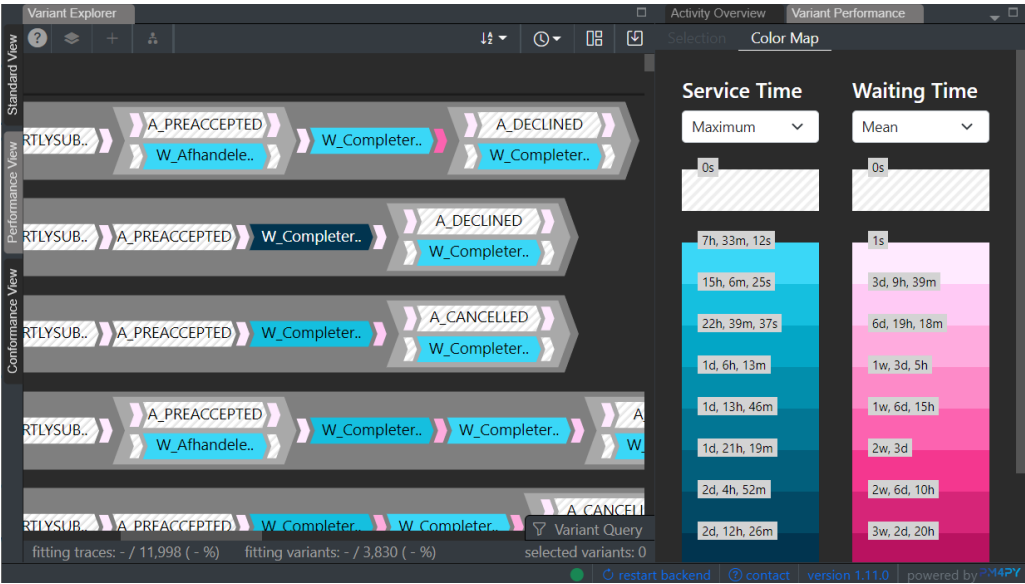


Figure 10.17: Performance view of the variant explorer; service and waiting time information is projected onto chevrons

Cortado offers model-independent performance analysis for variants. As presented before, variants are an essential concept within Cortado. Model-independent performance analysis allows projecting performance statistics onto variants independently of a process model. Recall that variants, as visualized in Cortado, do not contain temporal information; the variant visualization solely focuses on ordering relations among activities. For example, recall the variant shown in Figure 10.15a on page 290. Although the chevrons



representing activities RIP and RIT have equal widths, this does not imply that their execution time is identical.

Variants primarily show ordering information between activities. To additionally visualize temporal information, Cortado uses a color projection. Consider Figure 10.17 that shows a screenshot of Cortado's variant explorer in *performance view*. In the screenshot, Cortado projects service times onto chevrons representing activities, i.e., chevrons with process activity labels, using a blue color scale. Further, Cortado adds empty chevrons between any two chevrons to indicate waiting times as shown in Figure 10.17. Between two chevrons representing activities, the red color scale indicates the waiting time between these activities. Note that the statistical measures, i.e., minimum, maximum, mean, and standard deviation, can be chosen freely by users. Further, users can select variants or parts, i.e., any (nested) chevron, to see detailed performance statistics for the selection.

### 10.4.3. Model-Based Performance Analysis

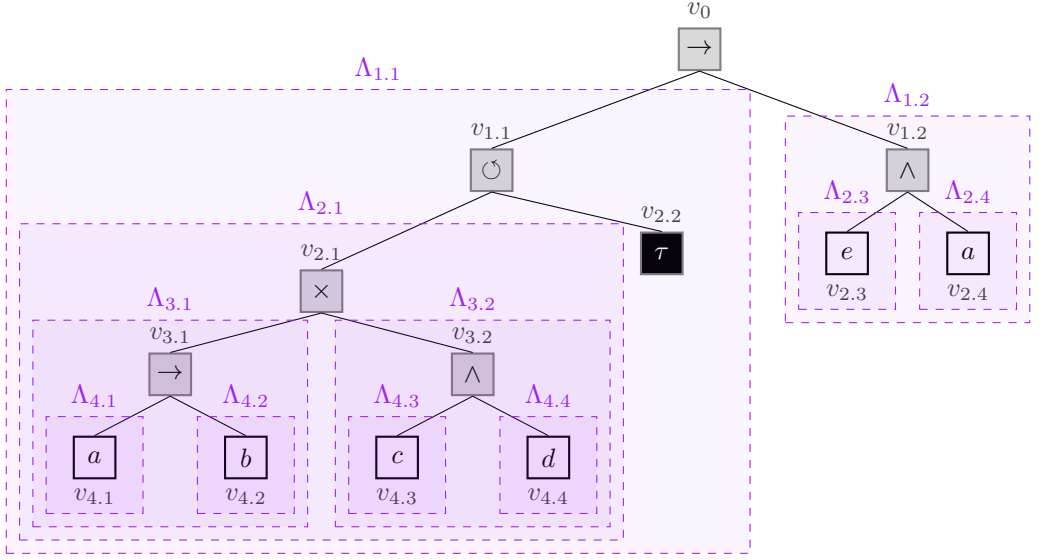
This section on model-based performance analysis is largely based on [182]. Next to model-independent performance analysis that solely considers the event data, model-based performance analysis requires a process model, tries to replay the recorded process behavior from the event log on the model, and eventually derives from the replay performance statistics for the model. Compared to existing model-based performance analysis approaches [6, 7], the approach implemented in Cortado focuses solely on process trees. Focusing on process trees makes it possible to calculate individual performance indicators for each subtree.

#### Defining Performance Indicators

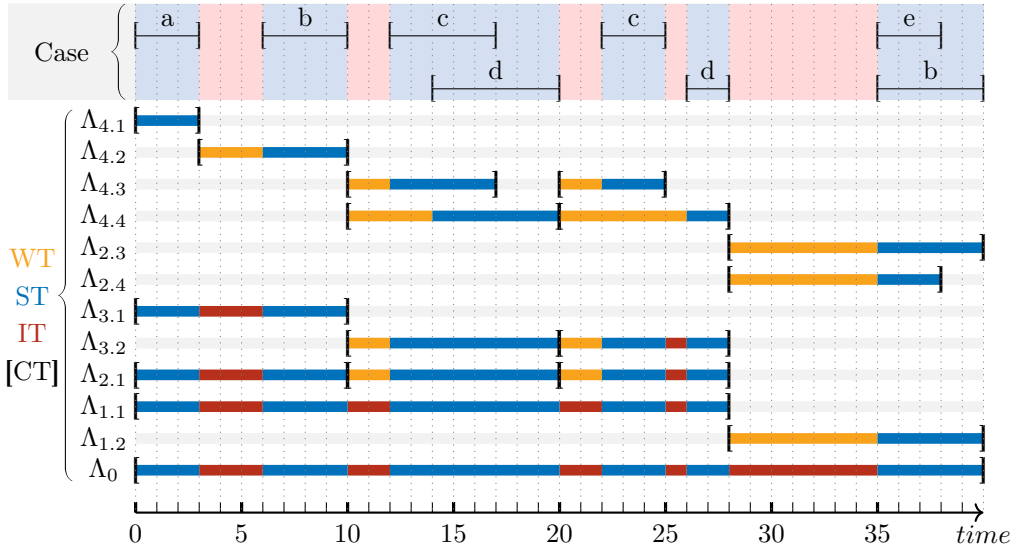
This section introduces four Performance Indicators (PIs) for process trees. Consider Figure 10.18 presenting an example to illustrate the four PIs. Figure 10.18a depicts the process tree  $\Lambda_0$ .

Figure 10.18b illustrates a case at the top and shows below the four PIs for each subtree including the entire tree  $\Lambda_0$ . Subsequently, we present and exemplify the four PIs.

- The *Service Time (ST)* specifies how long an individual activity is executed. For subtrees that contain several leaf vertices, i.e., several activities, the service time indicates the union of the service times of their leaf vertices. The ST of a leaf vertex corresponds to the recorded time of the aligned activity from the case. Note that leaf vertices labeled  $\tau$  are executed instantly and thus do not cause any ST. For example, the ST of the subtree  $\Lambda_{4,1}$  is three time units because the first activity  $a$  from the given case is replayed on leaf vertex  $v_{4,1}$ .
- The *Waiting Time (WT)* indicates how much time has passed between the earliest possible execution of a subtree and the actual execution according to the temporal information from the case. The WT of the entire process tree is always zero, as it is instantly activated when the first activity from the case is executed. Likewise, leaf vertices labeled  $\tau$  are assumed to be executed instantly and thus do not cause any WT. The WT of an inner vertex refers to the time that passes from its activation to the activation of its first executed leaf node. For example, the waiting time of



(a) Process tree  $\Lambda_0$ ; for simplicity, we label each subtree  $\Lambda_{1,1} = \Delta_{\Lambda_0}(v_{1,1}), \dots, \Lambda_{4,4} = \Delta_{\Lambda_0}(v_{4,4})$



(b) Cycle time (CT), service time (ST), waiting time (WT), and idle time (IT) based on a *fitting* case for process tree  $\Lambda_0$  (partly adapted from [182, Figure 4])

Figure 10.18: Example of PIs for a given process tree and case

subtree  $\Lambda_{1.2}$  is seven since after the closing of the previous subtree  $\Lambda_{1.1}$  at time 28, subtree  $\Lambda_{1.2}$  is opened at time 35.

- The *Idle Time (IT)* indicates for subtrees the time between activation and closing during which no leaf vertex is executed. Note that leaf vertices' IT always equals zero, as the execution of leaf vertices is non-interruptible. For example, the IT of the subtree  $\Lambda_{3.1}$  is three because three time units pass after the execution of the leaf vertex  $v_{4.1}$  and before the execution of the next vertex  $v_{4.2}$ .
- The *Cycle Time (CT)* is the sum of a subtree's ST, WT, and IT. For leaf vertices, the cycle time corresponds to the sum of WT and ST because leaf vertices' IT is always zero. As an example, the CT of subtree  $\Lambda_{3.1}$  is equal to ten, i.e., an overall ST of seven plus an IT of three.

In summary, four PIs can be calculated for a process tree and any of its subtrees, as exemplified in Figure 10.18.

### Calculating Performance Indicators

This section elaborates on Cortado's implementation for calculating the above-presented PIs for a given process tree and case. First, Cortado converts the given process tree into a WF-net.<sup>7</sup> Next, each visible transition of the WF-net is split into two, i.e., one transition representing the start, the other the completion of the respective activity. Figure 10.19 is illustrating the described splitting of visible transitions. Figure 10.19a depicts a visible transition  $t_1$  labeled  $a$  that has  $n$  input places, i.e.,  $p_1, \dots, p_n$ , and  $m$  output places, i.e.,  $p'_1, \dots, p'_m$ . When splitting  $t_1$ , two transitions  $t_{1,s}$  representing the start of activity  $a$  and  $t_{1,c}$  representing the completion of  $a$  are obtained, cf. Figure 10.19b. The input places of  $t_1$  are the input places of  $t_{1,s}$ , while the output places of  $t_1$  are the output places of  $t_{1,c}$ . Further, a new place  $p_0$  is added to connect  $t_{1,s}$  and  $t_{1,c}$ . The described splitting is applied to any visible transition. Figure 10.20 depicts the WF-net with split visible transitions representing process tree  $\Lambda_0$  (cf. Figure 10.18a). We also color-code parts of the WF-net that represent specific subtrees of  $\Lambda_0$ , cf. Figure 10.18a.

Likewise, any activity in a case is split into two, such that all events represent time points. Recall the case depicted at the top of Figure 10.18b. The split trace considered for temporal performance analysis is depicted below. Note that the fourth and third last activities, i.e.,  $(e, \blacktriangleright)$  and  $(a, \blacktriangleright)$ , could also occur in reverse order since both happen concurrently; however, for the sake of calculating PIs the order is irrelevant.

$$\left\langle (a, \blacktriangleright), (a, \blacksquare), (b, \blacktriangleright), (b, \blacksquare), (c, \blacktriangleright), (d, \blacktriangleright), (c, \blacksquare), (d, \blacksquare), (c, \blacktriangleright), (c, \blacksquare), (d, \blacktriangleright), (d, \blacksquare), \right. \\ \left. (e, \blacktriangleright), (a, \blacktriangleright), (e, \blacksquare), (a, \blacksquare) \right\rangle$$

Next, optimal alignments for the traces, which represent the cases from the imported event log sequentialized, are calculated. These alignments allow us to replay the traces on the model. During replay, we monitor the timing of the execution of each transition based

<sup>7</sup>Recall Table 3.2 on page 67 showing the conversion of process trees into WF-nets.

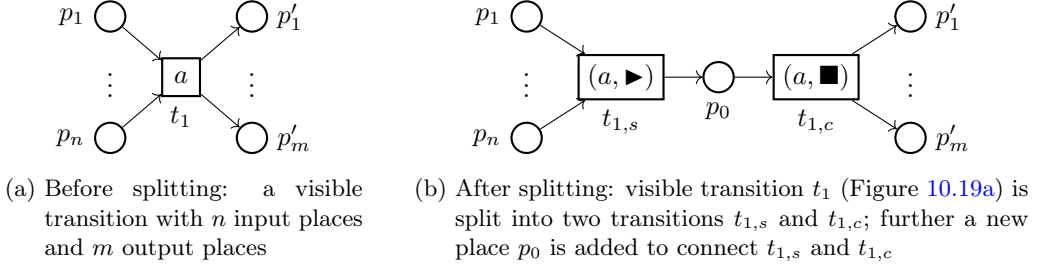


Figure 10.19: Illustration of splitting visible transitions into two in a WF-net (partly adapted from [182, Figure 6])

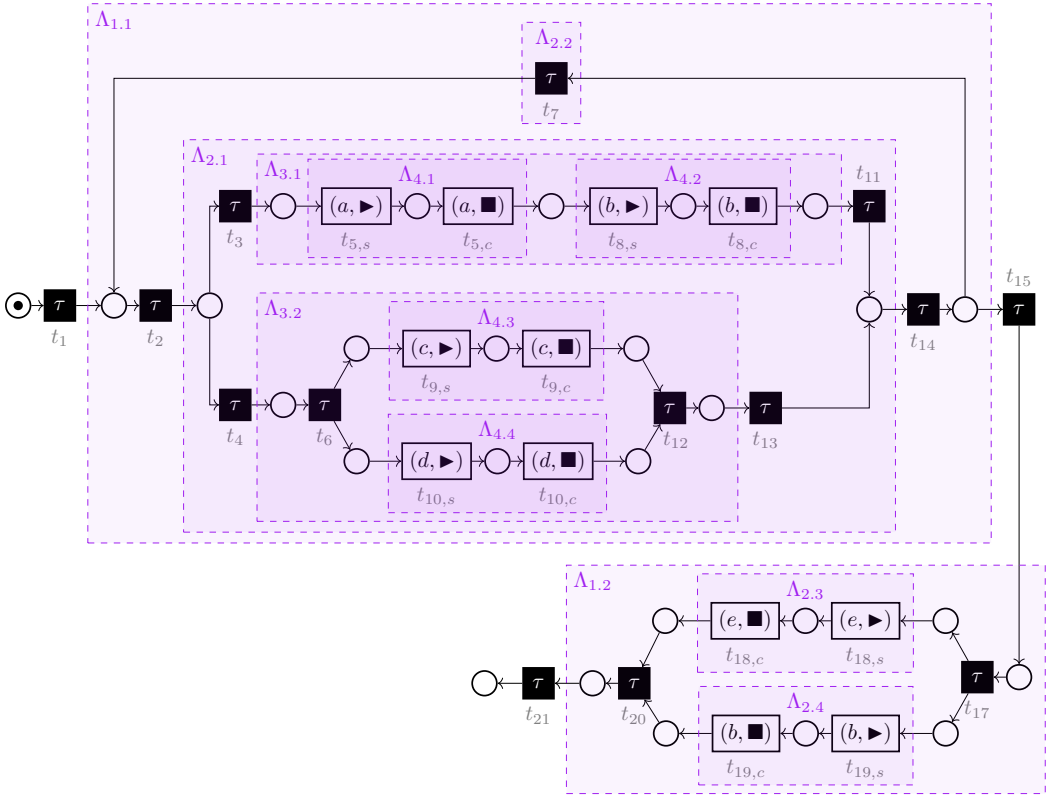


Figure 10.20: WF-net  $N_0$  representing process tree  $\Lambda_0$  (cf. Figure 10.18a) with split visible transitions; each subtree down to individual leaf vertices of  $\Lambda_0$  is highlighted in color

on the timing information from the trace. Finally, the timing information associated with the transitions allows us to calculate the defined PIs.

The reliability of model-based performance analysis depends on how well the used process tree represents reality [211]. Thus, if there is little similarity between the traces and the model, the significance of performance analysis is low because many temporal information from the trace cannot be used for calculating PIs of the process model. Furthermore, since there can be multiple optimal alignments for a given combination of a trace and a process model, and only one optimal alignment is used for each combination, the significance of the performance may be affected.<sup>8</sup>

Table 10.1.: Overview of calculable PIs per alignment combination representing the start and completion of an activity (partly adapted from [182, Table 2])

Alignment move combination	Designation	PI			
		WT	ST	IT	CT
Synchronous move on start & completion	Perfect activity instance	✓	✓	✓	✓
Synchronous move on start & model move on completion	Partial start	✓	–	✓ <sup>a</sup>	–
Model move on start & synchronous move on completion	Partial complete	✓ <sup>b</sup>	–	✓ <sup>c</sup>	✓ <sup>d</sup>
Model move on start & completion	Missing activity instance as per model	–	–	–	–
Log moves on start & completion	Missing activity instance as per log	–	–	–	–

<sup>a</sup> A partial start cannot be used for the IT of the currently considered activity instance since activities are atomic, i.e., their execution cannot be interrupted. However, the information on the start can be potentially used to determine the IT for subtrees containing the respective activity, i.e., the synchronous move on the activity's start could be the end of an idle time period.

<sup>b</sup> A partial complete cannot be used for calculating the WT of the actual activity instance. However, the completion information can be used to determine the WT for a subsequently executed activity and subtrees containing this subsequently executed activity.

<sup>c</sup> A partial complete cannot be used for the IT of the currently considered activity instance since activities are atomic, i.e., their execution cannot be interrupted. However, the information on the complete can be potentially used to determine the IT for subtrees containing the respective activity, i.e., the synchronous move on the activity's complete could be the start of an idle time period.

<sup>d</sup> A partial complete cannot be used for the CT of the considered activity; however, the completion information can be potentially used to determine the CT for subtrees containing the considered activity.

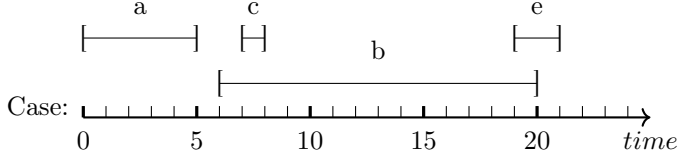
<sup>8</sup>Note that the described issue of multiple optimal alignments may also apply to traces that completely fit the given process model. However, only in case a process model contains an activity multiple times, there might be a chance that the calculated PIs are to a certain degree random, i.e., depending on the optimal alignment used in case multiple optimal ones exist.

Note that the trace used in the example shown in Figure 10.18 (page 294) is fully supported by process tree  $\Lambda_0$ . However, it is generally possible to also incorporate non-fitting traces for model-based performance analysis, i.e., traces that cannot be entirely replayed on a process model, cf. [6, Chapter 9]. Cortado implements the ideas presented in [6] and adapts them to the calculation of PIs for process trees. Table 10.1 shows an overview of five alignment move combinations representing the replay of an individual activity split into start and completion from a given trace. If an activity's start and completion are aligned via a synchronous move, we refer to it as a *perfect instance*. Perfect instances can be used for calculating all four PIs. If we observe only a synchronous move on the start but a model move on an activity's completion, we refer to it as a *partial start*. For partial starts, we can compute WTs and ITs because we know when an activity started; however, partial starts cannot be used to calculate STs and CTs. Analogously, if we observe a synchronous move on the completion but a model move on an activity's start, we refer to this as a *partial complete*. A partial complete may be used in certain situations for determining WTs, ITs, and CTs of subtrees containing the respective activity. For instance, if the partial complete, i.e., the synchronous move on an activity's completion, is the last activity of a subtree, we know that this subtree has a cycle time until the time point of the completion of the activity. Finally, suppose we have two model moves or two log moves on both the start and completion of an activity. In that case, we cannot derive any temporal information from these alignment moves to calculate PIs.

Although non-fitting cases can be used for performance analysis, they might lead to unreliable PI values. Figure 10.21 presents an example of a non-fitting case (cf. Figure 10.21a); its sequentialization with split activities is not supported by WF-net  $N_0$  (cf. Figure 10.20 on page 296). Figure 10.21b shows three optimal alignments for the trace shown in Figure 10.21a and  $N_0$ . Note that the shown alignments are incomplete; for the sake of simplicity, invisible model moves are not displayed. Below each synchronous and log move, the time information derived from the provided case is displayed (cf. Figure 10.21a).<sup>9</sup> The first alignment contains a perfect activity instance for activity  $a$  and  $e$ . Moreover, the alignment contains a partial start and a partial complete for activity  $b$ . Note that the partial start and the partial completion of the activity  $b$  relate to different transition pairs in  $N_0$  or leaf vertices in  $\Lambda_0$ . Thus, the temporal information of activity  $b$  from the case is split into two different instances of activity  $b$  in  $N_0$ , i.e., transitions  $t_{8,s}/t_{8,c}$  representing subtree  $\Lambda_{4,2}$  labeled  $b$  and transitions  $t_{19,s}/t_{19,c}$  representing subtree  $\Lambda_{2,4}$  labeled  $b$ . For calculating the ST of the entire tree, only the two perfect instances can be utilized (cf. Table 10.1). Hence, the CT of tree  $\Lambda_0$  using the first alignment is calculated as follows  $(5 - 0) + (21 - 19) = 7$ . The second depicted alignment contains three perfect activity instances of activities  $a$ ,  $c$ , and  $e$ . Computing the CT of  $\Lambda_0$  results in  $(5 - 0) + (8 - 7) + (21 - 19) = 8$ . Likewise, the third alignment contains three perfect activity instances; however, for the activities  $a$ ,  $b$ , and  $e$ . Computing the CT of  $\Lambda_0$  using the third alignment results in  $(5 - 0) + (21 - 6) = 20$ .<sup>10</sup> In conclusion, each depicted alignment leads to a different CT for  $\Lambda_0$ . The example demonstrates that non-fitting traces may lead to unreliable performance analysis results.

<sup>9</sup>Note that model moves originate from missing behavior not recorded in the event data; thus, no temporal information for model moves exists.

<sup>10</sup>Note that the time intervals of the two perfect activity instances of  $b$  and  $e$  overlap; thus, we calculate the ST of  $\Lambda_0$  as follows  $(5 - 0) + (\max\{20, 21\} - \min\{6, 19\}) = (5 - 0) + (21 - 6) = 20$ .



Trace with split activities:  $\langle (a, \blacktriangleright), (a, \blacksquare), (b, \blacktriangleright), (c, \blacktriangleright), (c, \blacksquare), (e, \blacktriangleright), (b, \blacksquare), (e, \blacksquare) \rangle$

(a) A non-fitting case and its sequential representation, used for alignment calculation

Perfect activity instance of a								Perfect activity instance of e		
$(a, \blacktriangleright)$	$(a, \blacksquare)$	$(b, \blacktriangleright)$	$(c, \blacktriangleright)$	$\gg$	$\gg$	$(c, \blacksquare)$	$(e, \blacktriangleright)$	$(b, \blacksquare)$	$(e, \blacksquare)$	
$t_{5,s}$	$t_{5,c}$	$t_{7,s}$	$t_{8,s}$	$t_{18,s}$	$t_{18,c}$	$t_{19,s}$	$t_{19,s}$	$t_{18,c}$	$t_{19,c}$	
$(a, \blacktriangleright)$	$(a, \blacksquare)$	$(b, \blacktriangleright)$	$\gg$	$(b, \blacksquare)$	$(b, \blacktriangleright)$	$\gg$	$(e, \blacktriangleright)$	$(b, \blacksquare)$	$(e, \blacksquare)$	
0	5	6	7	—	—	8	19	20	21	

ST of  $\Lambda_0$  derived from the above-shown alignment:  $(5 - 0) + (21 - 19) = 7$

Perfect activity instance of a								Perfect activity instance of c		Perfect activity instance of e		
$(a, \blacktriangleright)$	$(a, \blacksquare)$	$(b, \blacktriangleright)$	$\gg$	$\gg$	$(c, \blacktriangleright)$	$\gg$	$(c, \blacksquare)$	$\gg$	$(e, \blacktriangleright)$	$(b, \blacksquare)$	$(e, \blacksquare)$	
$t_{5,s}$	$t_{5,c}$	$t_{8,s}$	$t_{8,c}$	$t_{10,s}$	$t_{9,s}$	$t_{10,c}$	$t_{9,c}$	$t_{18,s}$	$t_{19,s}$	$t_{18,c}$	$t_{19,c}$	
$(a, \blacktriangleright)$	$(a, \blacksquare)$	$(b, \blacktriangleright)$	$(b, \blacksquare)$	$(d, \blacktriangleright)$	$(c, \blacktriangleright)$	$(d, \blacksquare)$	$(c, \blacksquare)$	$(b, \blacktriangleright)$	$(e, \blacktriangleright)$	$(b, \blacksquare)$	$(e, \blacksquare)$	
0	5	6	—	—	7	—	8	—	19	—	21	

ST of  $\Lambda_0$  derived from the above-shown alignment:  $(5 - 0) + (8 - 7) + (21 - 19) = 8$

Perfect activity instance of a								Perfect activity instance of b			Perfect activity instance of e	
$(a, \blacktriangleright)$	$(a, \blacksquare)$	$\gg$	$\gg$	$(b, \blacktriangleright)$	$(c, \blacktriangleright)$	$(c, \blacksquare)$	$(e, \blacktriangleright)$	$(b, \blacksquare)$	$(e, \blacksquare)$			
$t_{5,s}$	$t_{5,c}$	$t_{8,s}$	$t_{8,c}$	$t_{18,s}$	$\gg$	$\gg$	$t_{19,s}$	$t_{18,c}$	$t_{19,c}$			
$(a, \blacktriangleright)$	$(a, \blacksquare)$	$(b, \blacktriangleright)$	$(b, \blacksquare)$	$(b, \blacktriangleright)$	$(c, \blacktriangleright)$	$(c, \blacksquare)$	$(e, \blacktriangleright)$	$(b, \blacksquare)$	$(e, \blacksquare)$			
0	5	—	—	6	7	8	19	20	21			

ST of  $\Lambda_0$  derived from the above-shown alignment:  $(5 - 0) + (21 - 6) = 20$

(b) Three optimal alignments for the above trace (cf. Figure 10.21a) and WF-net  $N_0$  (cf. Figure 10.20); for simplicity, *invisible model moves* are not displayed in the above alignments

Figure 10.21: Example of unreliable performance analysis results when using a non-fitting trace, i.e., the ST of  $\Lambda_0$  depends on the optimal alignment used

## Presenting Performance Indicators

This section introduces Cortado’s approach to make the various PIs calculated for a process tree and cases accessible for users. The variant explorer (cf, Section 10.2.1) is the starting point for applying model-based performance analysis. Users can select which variants, and thus, the corresponding cases from the event log summarized by the chosen variants, are considered for model-based performance analysis.

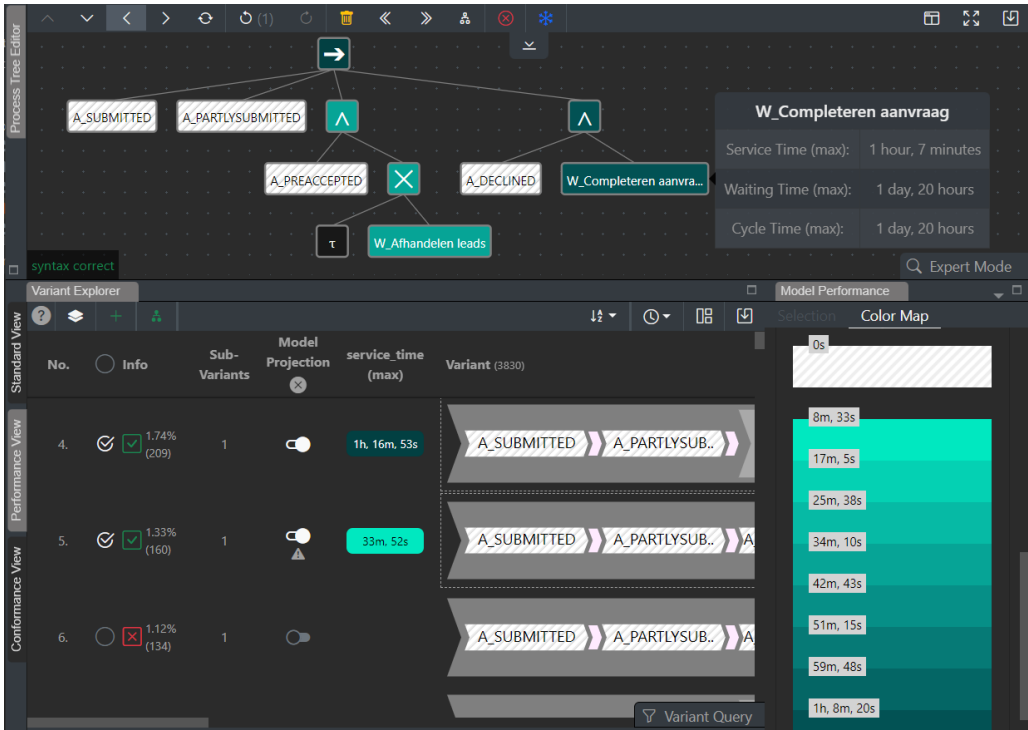


Figure 10.22: Projecting the maximal ST onto the process tree using selected variants

Figure 10.22 shows a screenshot where variants four and five are selected for model-based performance analysis. The color map indicates the max service time (setting not visible on the depicted screenshot); each vertex of the process tree shown in the editor at the top is colored accordingly. Since the event log imported contains event data with heterogeneous temporal information, the ST cannot be computed for all activities. Upon hovering over vertices in the tree, more temporal performance statistics are shown, cf. Figure 10.22. Note that the projected PI, as well as the statistical measure (i.e., min, max, mean, and standard deviation), can be freely configured by users. Moreover, Cortado offers detailed views on various performance statistics; Figure 10.23 shows statistics for the entire process tree shown in Figure 10.22. Next to the overall performance, including all selected variants, users can explore the different PIs summarized for each variant. The detailed statistics view is available for any subtree besides the entire tree.



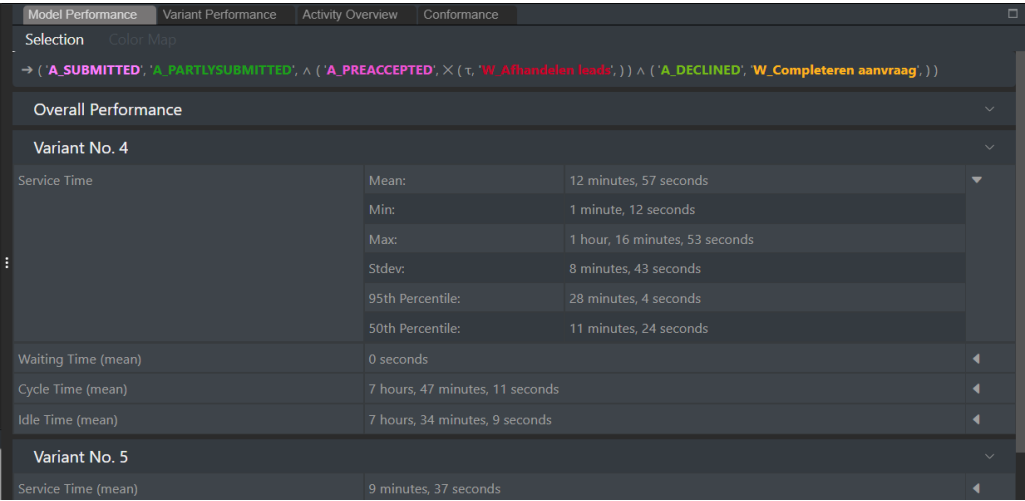


Figure 10.23: Detailed information on various PIs for the entire subtree

## 10.5. Supported Data Exchange Formats

Cortado supports standardized data exchange formats that make it easy to use Cortado in combination with other process mining tools. The XES standard [1, 2] specifies an XML schema for storing and exchanging event data. XES is a widely used standard in the process mining field; for instance, the widely used and extended open-source process mining tool ProM [234] supports XES. Cortado allows the import of event logs provided as XML files that adhere to the XML schema defined by the XES standard. Thus, other process mining tools can be easily used, for instance, to pre-process the event data eventually imported into Cortado.

Since the IPDA implemented in Cortado assumes process trees (cf. Part II), Cortado supports the PTML file format that allows storing and exchanging process trees. For instance, the open-source process mining library PM4Py [27] or the open-source process mining tool ProM [234] support PNML, too. Discovered/modeled process trees can be exported in three different formats: as process trees (PTML files), Petri nets (PNML files [32, 244]), and BPMN files [48]. In short, Cortado supports widely used data formats, which facilitates the integration of further process mining tools.

Although Cortado was primarily developed for incremental process discovery, as proposed in this thesis, users can also use Cortado's rich set of functionalities for various other process mining use cases. Figure 10.24 illustrates exemplary application scenarios of Cortado. Note that any input artifact may also originate from other process mining tools. Likewise, any output artifact may be used in a subsequently used process mining tool. Thus, Cortado can be easily integrated into process mining toolchains.

The central application use case is incremental process discovery as illustrated in Figure 10.24a. An event log is imported into Cortado, and a user (incrementally) discovers

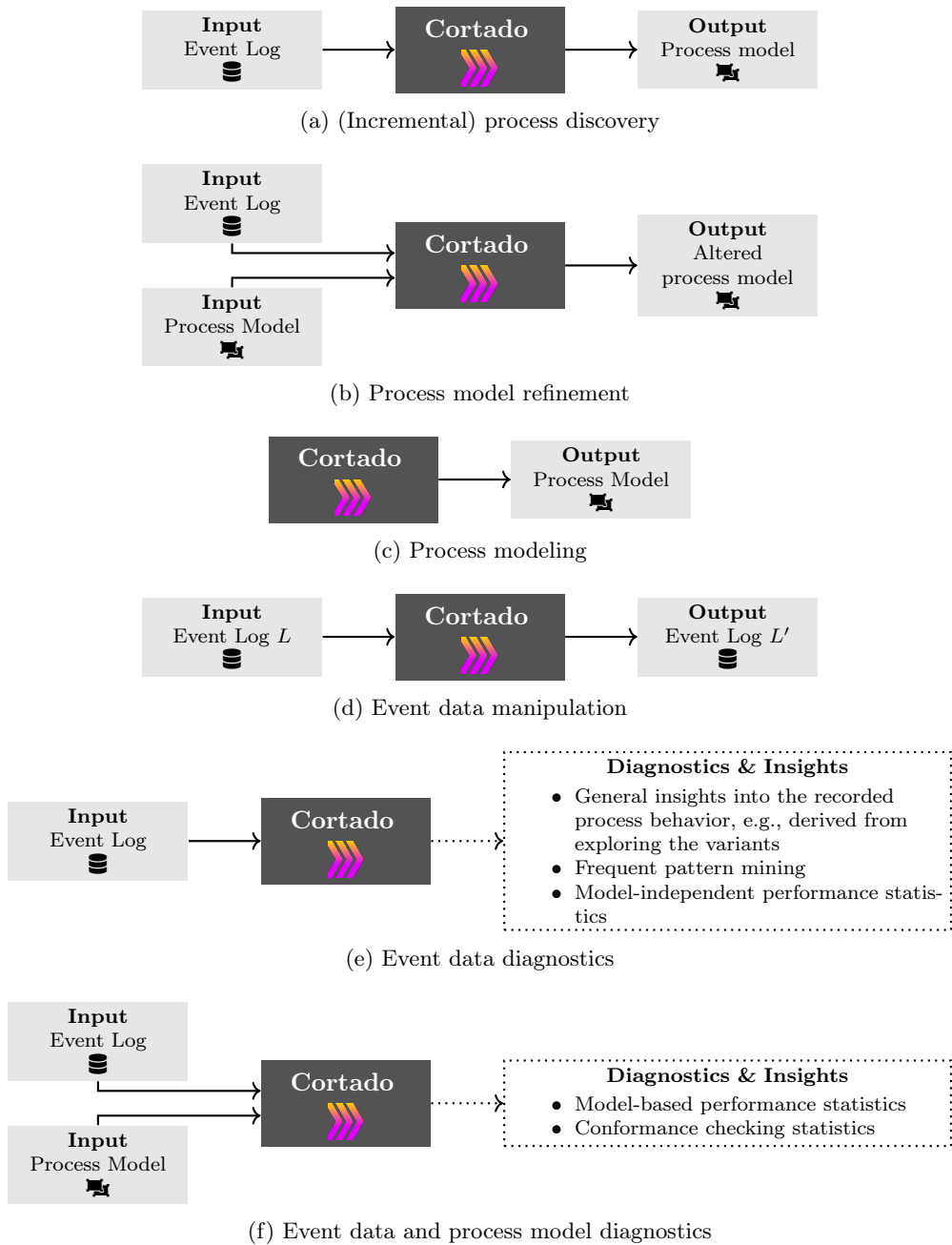


Figure 10.24: Selection of possible application scenarios for Cortado

a process model from the data and potential domain knowledge.<sup>11</sup> The output process model, for example, exported as a BPMN or PNML file, can be used in subsequent process mining tools. The *process model refinement* application scenario (cf. Figure 10.24b) can also be categorized as incremental process discovery with the difference that in this scenario, an initial process model is imported into Cortado. This process model can be enhanced by new functionality using Cortado's IPD approach. Besides discovering and refining a process model, Cortado can also be used as a classic process model editor, cf. application scenario *process modeling* illustrated in Figure 10.24c. Note that it is also possible to import a model first and manually edit it in this application scenario.

Besides exporting process models, cf. Figures 10.24a to 10.24c, Cortado can also be used as an event data manipulation tool. For example, users might use the query language (cf. Section 10.2.2) to filter the event log and export the filtered log for use in other tools. Moreover, users can also use the variant editor (cf. Section 10.2.3) to add behavior to the imported event log.

Cortado can also derive insights into the provided artifacts. In the *event data diagnostics* scenario (cf. Figure 10.24e), an event log is imported and analyzed using Cortado. For example, users browse through the variant explorer to understand the process behavior recorded in the event log. Users might also use the model-independent performance analysis functionality of Cortado, cf. Section 10.4.2. Besides solely analyzing the event log (cf. Figure 10.24e), users might also additionally consider a process model besides the imported event log, cf. the application scenario depicted in Figure 10.24f. Such derived diagnostics and insights (cf. Figures 10.24e and 10.24f) can already be used to make informed decisions about the process under study. Moreover, with these insights, users can decide which further analyses, possibly with other process mining tools, are necessary or promising to analyze the process under study further.

In conclusion, Cortado is a process mining tool with rich and unique functionalities. It supports import and export options, making integrating with other process mining tools easy. Combining Cortado with other tools is also often necessary because Cortado was not developed to holistically support existing process mining techniques and algorithms. Instead, it was explicitly developed for incremental process discovery, and all the functionalities it contains are designed with incremental process discovery in mind or serve some form of incremental process discovery.

## 10.6. Software Architecture & Distribution

This section delves into the architecture and implementation details of Cortado, focusing on its core components, which comprise the *cortado-core* Python library, the backend, and the frontend, i.e., the UI. Figure 10.25 illustrates Cortado's architecture.

The Python library *cortado-core* is based on the general-purpose process mining library PM4Py [27].<sup>12</sup> Cortado-core implements the IPDAs introduced in Part II. Further,

<sup>11</sup>Note that we do not illustrate domain knowledge in Figure 10.24 because no standardized data exchange formats exist to store and share domain knowledge. Domain knowledge is often only available implicitly.

<sup>12</sup>The source code of *cortado-core* can be found online at <https://github.com/cortado-tool/cortado-core>.

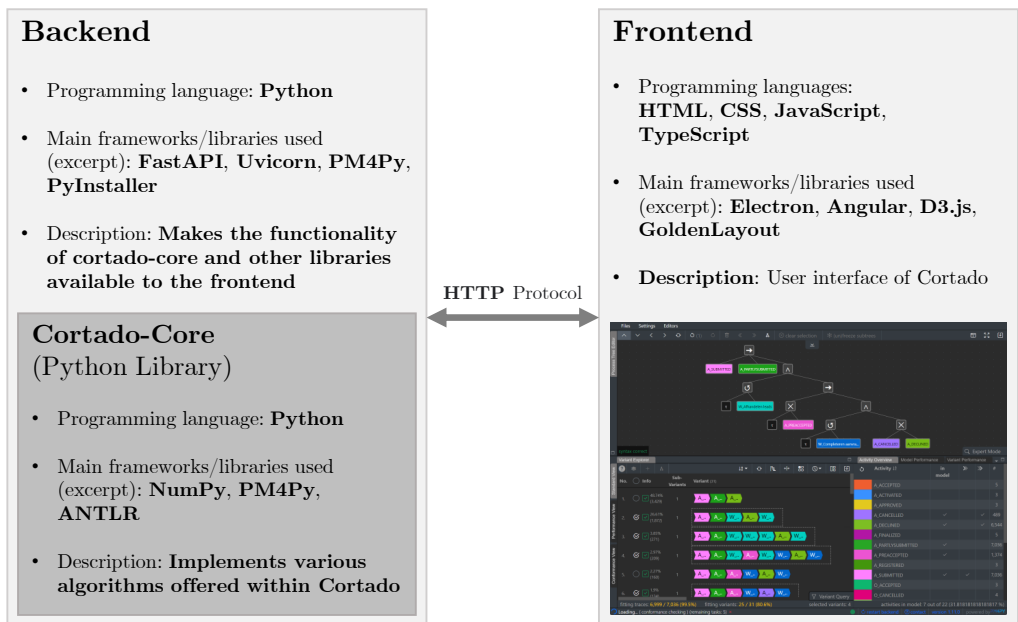


Figure 10.25: Illustration of Cortado’s software architecture consisting of a Python-based backend and a web technology-based frontend (figure partly adapted from [186, Figure 1])

cortado-core implements alignments for trace fragments as introduced in Chapter 4, a detection algorithm for high-level and low-level variants as introduced in Chapter 8, the query language as introduced in Chapter 9, algorithms for temporal performance analysis and many other functionalities. Outsourcing these various algorithms to a separate library was done intentionally to make the algorithms themselves easily accessible without embedding them in a complete software system, ultimately facilitating reuse, adaptation, and modification in research and other software tools. In short, cortado-core is a comprehensive software library that implements all the algorithms offered by the Cortado software tool.

The backend of Cortado is also written in Python and bundles various functions from cortado-core and PM4Py to make them accessible to the frontend. The backend uses the FastAPI framework<sup>13</sup> to build an application programming interface for the frontend. Further, PyInstaller bundles the backend into a single distributable package, including cortado-core and all their dependencies. Cortado’s backend and frontend communicate via the HTTP protocol.

Cortado’s frontend, referring to the UI that was described in detail in previous sections,

<sup>13</sup><https://github.com/tiangolo/fastapi>

is realized using web technologies. Angular<sup>14</sup> is the central framework on which Cortado's frontend is built. Furthermore, Cortado's frontend employs GoldenLayout<sup>15</sup> as a layout manager. Finally, to bundle the frontend and backend, the Electron framework<sup>16</sup> is utilized to generate standalone executable desktop applications. Cortado is available as a standalone desktop application for all major operating systems, i.e., Windows, macOS, and Linux.<sup>17</sup>

## 10.7. Conclusion

Cortado is a software tool for process mining specifically designed to enable incremental process discovery. In addition, it integrates the various contributions presented in this thesis into a single tool, demonstrating how the individual contributions proposed in the thesis work together towards the goal of incremental process discovery. Moreover, the source code for Cortado and its underlying software library, *cortado-core*, are open-source, which makes it possible to reuse and extend them.

Four functional areas can be distinguished in Cortado; recall Figure 10.3. Event data and especially variant handling, the first area, are central to the concept of Cortado. Users continuously interact with the visualized variants, for example, by selecting variants to conduct IPD or model-based performance analysis. Cortado offers various auxiliary functions for handling variants, such as frequent pattern mining, clustering, and querying. IPD is the second central functional area that implements the various approaches presented in Part II. Moreover, Cortado features conformance checking functionality as well as performance analysis.

---

<sup>14</sup><https://github.com/angular/angular>

<sup>15</sup><https://github.com/golden-layout/golden-layout>

<sup>16</sup><https://www.electronjs.org/>

<sup>17</sup>Builds of Cortado are available online at <https://github.com/cortado-tool/cortado>.



---

# Chapter 11.

## Case Study

---

This chapter is largely based on the following publication.

- D. Schuster, E. Benevento, D. Aloini, and W. M. P. van der Aalst. Analyzing healthcare processes with incremental process discovery: Practical insights from a real-world application. *Journal of Healthcare Informatics Research*, 2024. doi:[10.1007/s41666-024-00165-6](https://doi.org/10.1007/s41666-024-00165-6) [187]

This chapter presents a case study on applying Cortado in a real-life scenario. The subject of this case study is the healthcare sector, which is constantly striving to improve the quality of care and become more cost-effective at the same time. The healthcare sector's ongoing digitization, facilitated by technologies such as electronic health records [102, 107], is creating opportunities for process mining to be applied in healthcare processes [135].

Various studies have shown that process mining can be effectively utilized to analyze healthcare processes [146, 22, 23]. However, healthcare processes often have characteristic properties that need to be considered [160, 194]. Munoz et al. [146] highlight two considerable challenges. First, individual process executions, respectively, cases often have a high variability since healthcare processes are inherently complex, i.e., *knowledge-intensive* [64]. Since individual cases often represent patients, there is a high variability in cases due to differences in patient characteristics, responses to treatments, and the expertise of healthcare professionals involved [160]. Furthermore, several possible treatment pathways exist for a given medical condition [146]. As a result, most cases are often unique regarding the executed activities and their ordering. Second, event data quality is a significant challenge in healthcare, such as missing or incorrectly recorded events that are often caused by data entry or collection errors [135, 146, 137, 94, 239]. Moreover, the timestamps of recorded events are often imprecise. These event data quality issues can originate from excessive workload of healthcare staff, inadequate training, and extensive manual recording of performed activities [22, 93, 12]. The two challenges, i.e., high case variability and event data quality issues, may negatively impact process mining techniques' success when applied for analyzing healthcare processes [146].

Subsequently, Section 11.1 discusses related work on process mining in healthcare. Section 11.2 outlines the conducted case study. Afterward, Section 11.3 introduces the case study's analysis objectives and approach, followed by Section 11.4 presenting its

implementation and the obtained results. Section 11.5 discusses the conducted case study. Finally, Section 11.6 concludes this chapter.

## 11.1. Related Work

Analyzing processes in the healthcare sector using process mining techniques is receiving growing interest; multiple studies and literature reviews underscore the usefulness and opportunities of process mining in healthcare [52, 58, 146, 164]. For example, process mining techniques are used in healthcare to discover accurate patient flows [75], analyze process performance [193], assess compliance with clinical guidelines [245], and predict patient outcomes [145]. Below, we focus on related work on applying interactive process mining techniques in the healthcare sector.

Research highlights interactive process mining techniques utilizing event data and domain knowledge have been explored in healthcare [93]. However, despite the potential benefits of interactive process mining techniques, more evidence is needed to demonstrate the superiority of interactive approaches over automated and conventional approaches. To the best of our knowledge, there have been few samples where interactive process mining techniques have been practically and advantageously applied in complex real-world settings, such as healthcare.

Martin et al. [140] propose an interactive event data cleaning technique that involves three steps: data-based data quality assessment, discovery-based data quality assessment, and data cleaning heuristics. The proposed technique was evaluated in the context of a case study using an outpatient clinic's appointment system. The proposed technique allows users to guide event data cleaning by exploiting their domain knowledge. However, the proposed technique focuses solely on data cleaning, and it requires process stakeholders and analysts with enough experience in controlling interactive data cleaning.

An approach for using process mining over an interactive pattern recognition framework to support the iterative design of clinical pathways [54] for chronic diseases is proposed in [94]. In [22], the effectiveness of the interactive process discovery tool ProDiGy [74, 71] in modeling healthcare processes from event data is showcased. We refer to Chapter 2 for a detailed introduction to ProDiGy. Consequently, Cortado, which is applied in this case study, and its incremental process discovery approach is not directly comparable to existing interactive approaches applied in healthcare, cf. [22, 23, 74].

## 11.2. Overview

This section provides an overall overview of the conducted case study; specific analysis objectives and the concrete approach taken are presented subsequently in Section 11.2. The case study investigates the use of *incremental process discovery* to obtain a process model from a knowledge-intensive healthcare process. This case study analyzes an event log documenting the treatment of lung cancer patients given to hospitalized patients over a year. Each case details a patient's treatment, and activities represent individual procedures. Overall, lung cancer treatment is complex and requires the collaboration of various healthcare specialists. Each lung cancer patient undergoes diagnostic activities to



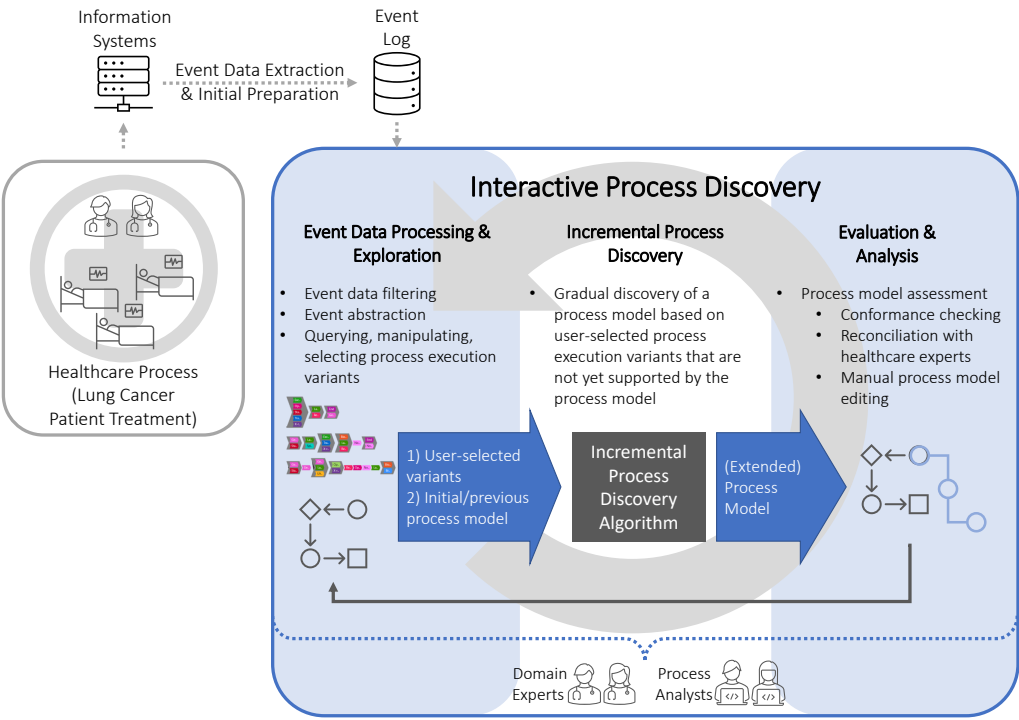


Figure 11.1: General approach of the conducted case study; starting from an event log describing the treatments of lung cancer patients, we interactively and incrementally discover a process model that summarizes the complex and highly individual sequence of the different treatment steps (adapted from [187, Figure 1])

confirm the diagnosis and assess the extent of the disease, followed by surgery and follow-up activities. Due to the complexity of lung cancer disease and the various treatments available, many unique and ad-hoc care pathways exist.

This case study was conducted in close collaboration with the hospital staff, including the head of the thoracic surgery department, an oncologist, a ward doctor, and a nurse from the pneumology department. All participants in the case study are specialized in caring for lung cancer patients.

Figure 11.1 depicts the overall approach used in this case study. Besides the event data extraction and initial preparation step, the outlined blue box represents the primary approach. We use Cortado to gradually discover a process model representing lung cancer treatments from the provided event log. In each iteration, domain experts and process analysts selectively choose individual variants to be added to the process model. Since the process model is incrementally discovered and intermediate models are displayed, domain experts can better understand and comprehend how the final process models

emerged from event data.

### 11.3. Analysis Objectives & Approach

Our main objective in this case study is to use Cortado to discover interactively a comprehensive process model for the complex lung cancer treatment procedure. Furthermore, we intend to blend the invaluable expertise of physicians with the event data describing patient treatments that have been extracted. In short, our goal is to develop a reliable process model that closely aligns with the physicians' perspective on the existing procedure and treatment options while also considering the vast number of recorded patient treatments reflected by the event data.

The case study consists of two phases: *event data extraction and initial preparation*, and *interactive process discovery*, as shown in Figure 11.1. Below, we detail these phases.

1. In the *event data extraction & preprocessing* phase event data is extracted from various hospital information systems and an event log is constructed. Thus, this phase involves merging event data, removing redundant or inconsistent information, and producing a single, coherent event log. We rely on established techniques in this phases [3, 59, 160, 236].
2. The *interactive process discovery* phase is essential within this case study. In this phase, interactive variant exploration, incremental process discovery, and analysis of the process model take place. We exclusively use Cortado during this phase. Exploiting domain knowledge is integral to the interactive discovery phase and crucial for reliable results. Furthermore, this phase is cyclically executed as required, and each cycle comprises three sub-phases, as indicated in Figure 11.1.
  - a) The *event data processing & exploration* subphase aims to explore, organize, and sort variants for the subsequent incremental process discovery phase. Cortado's functionalities, like variant visualizations (cf. Section 10.2.1) and variant querying (cf. Section 10.2.2), support users in this phase. In addition, this phase involves variant filtering and variant editing based on domain knowledge. For instance, the variant sequentialization feature introduced in Section 10.2.5 is applied in this subphase. Overall, process analysts take advantage of the domain knowledge from healthcare experts and incorporate this knowledge upon filtering, manipulating, and selecting variants for the subsequent stage of incremental process discovery.
  - b) In the *incremental process discovery* subphase, the process model specifying lung cancer patient treatment is enhanced. The variants selected in the previous subphase are incorporated into the model by the IPDA implemented in Cortado. Moreover, in this subphase, users can apply techniques such as freezing submodels (cf. Chapter 7) to guide the IPD. Additionally, users can manually modify parts of the process model based on their expertise.
  - c) In the *evaluation & analysis* subphase, users assess the incrementally extended process model resulting from the previous subphase. For example, the conformance checking functionalities offered in Cortado determine how well the

process model aligns with the provided event data and domain knowledge. Thereby, potential gaps or inconsistencies can be identified, which may affect the completeness of the process model and trigger further iterations of the interactive process discovery phase. As a result, users may decide to execute a further iteration of the overall interactive process discovery approach, cf. Figure 11.1.

## 11.4. Analysis Results

This section describes the steps taken in the previously mentioned phases and subphases. Section 11.4.1 describes the event data extraction and preparation phase, while Section 11.4.2 presents actions performed within the subsequent interactive process discovery phase.

### 11.4.1. Event Data Extraction & Initial Preparation

The event data subject to this study was extracted from two hospital information systems: the Electronic Medical Record (EMR) and the Radiology Information System (RIS). The Electronic Medical Record (EMR) is an information system that records all inpatient medical events in the hospital, including a patient's medical history, diagnoses, and treatments. Besides, the RIS is a specialized system for managing and organizing information related to radiology activities. The initial event log includes data from 998 patients, each of whom we consider a case. Furthermore, the log contains 45 distinct activities and about 40,000 events. Table 11.1 shows an excerpt of the event log constructed. All events contained are single-timestamped. Moreover, the bottom granularity is coarse, i.e., all events have days as bottom granularity.

We conducted initial event data cleaning to resolve quality issues, including the following steps.

- *Outliers and incomplete cases removal:* Eight cases were removed due to incorrect activity time records and missing relevant attribute values.
- *Elimination of less significant activities:* We have excluded activities that are not directly related to lung cancer treatment, such as eye lens surgery and bone excision. These activities were excluded due to the patient's comorbidity.
- *Event abstraction:* The event log contains many activities that we hierarchically categorized into different abstraction levels. The initial event log contains 45 different activity labels; all light blue highlighted activities depicted in Figure 11.2 correspond to these 45 activities. We hierarchically organize the various activities to achieve our analysis goals, as shown in Figure 11.2. The hierarchical structure allows us to generalize specific activities. For example, we relabeled activities related to laboratory tests such as glucose, potassium, and creatinine into a more general activity called "Lab test" because these low-level activities were considered too fine-grained for the process model we wanted to determine. The event abstraction approach mentioned above reduced the number of activity labels from 45 to 19, cf.

Table 11.1.: Excerpt of the extracted event data covering the treatment of lung cancer patients (adapted from [187, Table 1])

Case ID	Activity Label	Activity Category	Timestamp	...
25480	General Physical Examination (GPE)	-	15/05/2017	...
25480	Creatinine (Cre)	Examination	15/05/2017	...
25480	Calcium (Cal)	Examination	15/05/2017	...
25480	Glucose (Glu)	Examination	15/05/2017	...
25480	Magnesium (Mag)	Examination	15/05/2017	...
25480	Chest X-ray (ChX)	Examination	15/05/2017	...
25480	Spirometry (Spi)	Examination	25/05/2017	...
25480	General Physical Examination (GPE)	-	01/06/2017	...
47777	General Physical Examination (GPE)	-	17/07/2017	...
47777	CT Chest (CTC)	Examination	25/07/2017	...
47777	Calcium (Cal)	Examination	25/07/2017	...
47777	Glucose (Glu)	Examination	25/07/2017	...
47777	Magnesium (Mag)	Examination	25/07/2017	...
47777	Creatinine (Cre)	Examination	25/07/2017	...
47777	Liver Biopsy (LiB)	Examination	25/07/2017	...
47777	Electrocardiogram (Elc)	Examination	18/08/2017	...
47777	Spirometry (Spi)	Examination	18/08/2017	...
47777	Excision of lung and bronchus (ELB)	Surgery	01/09/2017	...
47777	Computer aided surgery (CAS)	Surgery	01/09/2017	...
47777	Other non-operative procedure (ONOP)	Treatment	10/09/2017	...
47777	Other non-operative procedure (ONOP)	Treatment	10/09/2017	...
47777	General Physical Examination (GPE)	-	15/09/2017	...
47777	Calcium (Cal)	Examination	15/09/2017	...
47777	Glucose (Glu)	Examination	15/09/2017	...
47777	Magnesium (Mag)	Examination	15/09/2017	...
47777	Creatinine (Cre)	Examination	15/09/2017	...
40036	General Physical Examination (GPE)	-	01/07/2017	...
40036	Calcium (Cal)	Examination	01/07/2017	...
...	...	...	...	...

red highlighted activities in Figure 11.2. Together with the medical professionals, these 19 activities were identified as the optimal level of abstraction.

The preprocessing described above yielded an event log with over 14,000 events, 990 patient cases, and 19 types of activities. As anticipated and explained at the beginning of this chapter, there is a high level of heterogeneity in the process, with 934 different variants out of 990 cases. These factors complicate process discovery and analysis, as each patient's treatment is almost unique regarding the ordering of performed activities.

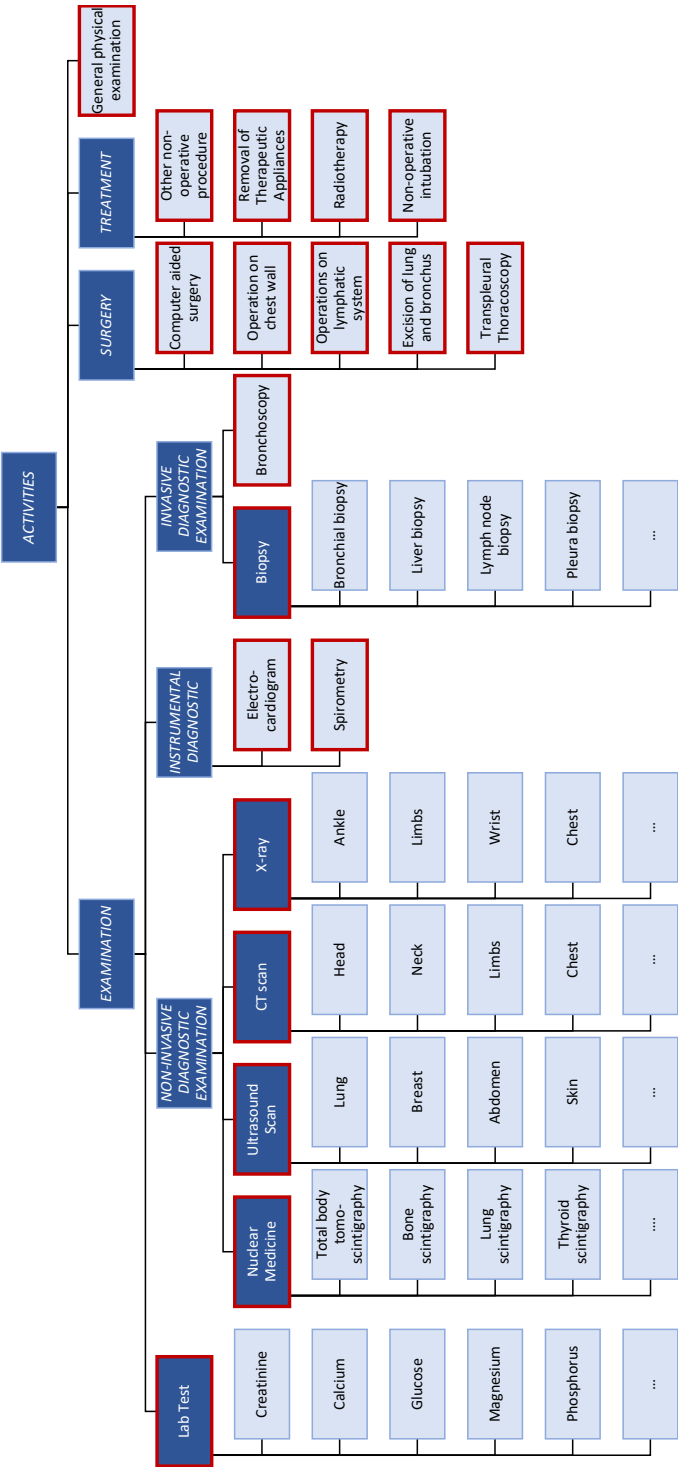


Figure 11.2: Overview activity hierarchy; activities highlighted in light blue originate from the extracted event data, activities highlighted in dark blue represent abstracted activities, i.e., activities below dark blue activities are a specialization, and activities outlined in red have been identified as the correct abstraction level for the analysis objectives (adapted from [187, Figure 6])

### 11.4.2. Interactive Process Discovery

This section presents the actions performed during three main subphases of the *Interactive Process Discovery* phase, cf. Figure 11.1, and the corresponding results.

#### Event Data Exploration & Processing

Using variant querying (cf. Section 10.2.2), we explored the diverse variants to investigate their activity relations. Moreover, we removed variant outliers that could not be detected in the previous phase, i.e., the *event data extraction & preprocessing* phase, cf. Figure 11.1. We identified truncated and incomplete traces resulting from errors in data entry or extraction. Removing these variants allowed us to reduce the event data’s complexity partly.

We investigated variants based on their frequency and the activities they contained. During our analysis, we observed that the two most frequent variants, with 11 and 9 cases, respectively, consisted of only one repeated activity. We also found that 0.4% of all variants had a maximum of three activities, while 0.2% had up to six activities of only two types. The hospital team identified these variants as problematic since they do not reflect the actual execution of the process, i.e., the lung cancer treatment. Instead, these variants result from poor data quality; hence, we filtered them. The hospitalization process typically involves a series of tests, treatments, visits, and, if necessary, surgery. Therefore, cases where a patient is hospitalized for only one or two examinations are not plausible and indicate missing data. Overall, most of the outliers we observed are most likely due to registration errors or non-registration by hospital staff.

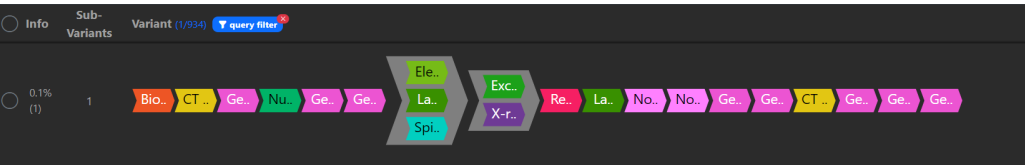


Figure 11.3: Cortado’s variant explorer showing a variant that indicates an inconsistent ordering of activities, i.e., “Biopsy” (i.e., the first activity of the visualized variant) is executed before “CT scan”, i.e., yellow highlighted activities (adapted from [187, Figure 7])

We have also identified and removed incorrect variants that do not comply with clinical guidelines and medical expertise. For example, during the diagnostic phase of lung cancer treatment, non-invasive diagnostic exams, such as X-rays and CT scans, must be performed before invasive diagnostic procedures like bronchoscopy or biopsy occur. However, we found some variants in the event log where this relationship was not respected. One example is shown in Figure Figure 11.3, where a biopsy is performed before a non-invasive diagnostic procedure, i.e., a CT scan. These incorrect variants could be due to registration errors or delayed registration of the performed activities in the information systems. As these behaviors can affect the reliability of the results and subsequent process analysis, we decided to filter such variants.

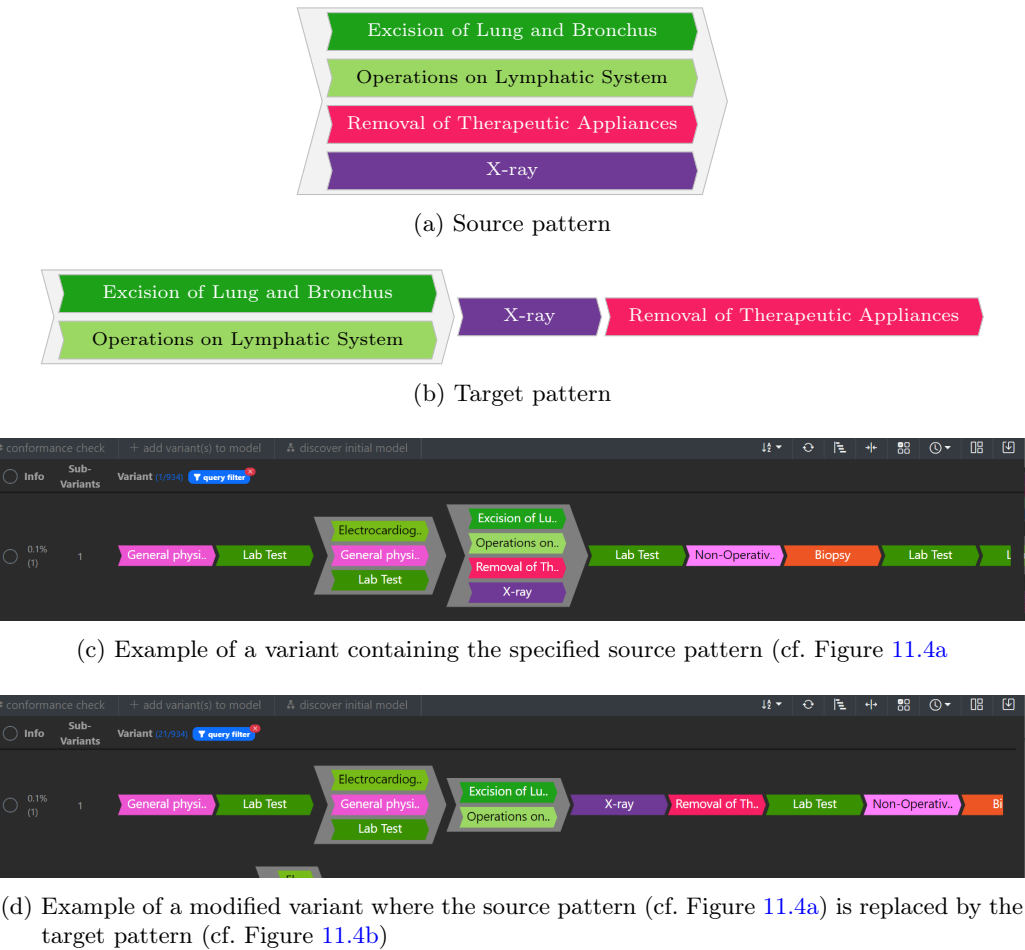


Figure 11.4: Example of a source/target pattern pair for sequentializing variants and its application to a variant (adapted from [187, Figure 8])

We used Cortado’s *variant sequentialization* functionality (cf. Section 10.2.5) to integrate domain knowledge into imprecise variants. These variants contained several activities that were performed in parallel but should follow a specific execution order as per the guidelines and understanding of the medical staff. For example, during the surgery phase of lung cancer treatment, the patient usually undergoes instrumental examinations to assess their operability, followed by surgery, an X-ray, and, if necessary, removal of the therapeutic device. These activities should occur sequentially. However, due to the coarse bottom granularity of days, we see these activities executed in parallel since every event has the same timestamp.

Figure 11.4 depicts an example of such sequentialization. The variant depicted in Fig-

ure 11.4c displays that removing the therapy device could be performed before its installation during the surgery, as both activities are parallel. Thus, using this variant for process discovery results in an imprecise process model, allowing for incorrect behavior. We alter these variants using the variant sequentialization functionality in Cortado (cf. Section 10.2.5) and the domain experts' knowledge. Figure 11.4a shows the described source pattern, while Figure 11.4b shows the corresponding target pattern. When applying this sequentialization rule, consisting of the above-mentioned source and target pattern, to the variant shown in Figure 11.4c, we obtain the sequentialized one depicted in Figure 11.4d. In addition to this example, we applied many more sequentialization rules we designed with the medical experts to mitigate the limitations of the coarse bottom granularity of the event data.

### Incremental Process Discovery

After filtering and refining variants, we gradually discover a normative model for the treatment of lung cancer patients that captures the wide variety of patient trajectories recorded. Therefore, we apply Cortado's incremental discovery approach. We resolved any inaccuracies in intermediate process models during each iteration, i.e., after adding few selected variants, by editing the model manually together with the medical experts if needed. Thus, we blend process discovery with process modeling by using Cortado. When necessary, we revisited the *event data processing & exploration* phase to further refine the variants, cf. Figure 11.1.

Figure 11.5 depicts the initial model, which only describes variants without diagnostic activities in the initial phase. Note that these particular variants represent the most frequent variants. Thus, the initial model describes the journey of hospitalized patients undergoing diagnostic procedures outside the hospital, either in outpatient settings or private healthcare facilities. The initial process model can be divided into three process stages.

1. The diagnostic stage includes only the *general physical examination*, which describes the initial visit with the physician.
2. The *surgery* stage involves the following activities: *Excision of Lung and Bronchus*, *Operations on Lymphatic System*, *Computer Aided Surgery*, *Lab Test*, *X-ray*, and *Removal of Therapeutic Appliances*.
3. The final stage, referred to as *follow-up*, includes activities such as *Other Non-Operative Procedures*, *Non-Operative Intubation*, and further *Lab Test* activities.

Building upon the initial model, we started to incorporate variants encompassing diagnostic activities and involved one surgical procedure per variant in the model. The addition of these variants enabled us to gain insights into the impact of diagnostic tests on the overall process and at which point these changes occur. We manually applied the following changes to the model to improve its comprehensibility and reliability. We inserted a loop for diagnostic and follow-up activities to account for various patients having potentially multiple examinations. Besides, we put the activities *general physical examination* and *lab test* in parallel with the diagnostic and surgical activities since *general physical examination* and *lab test* are essential activities that may be executed repeatedly



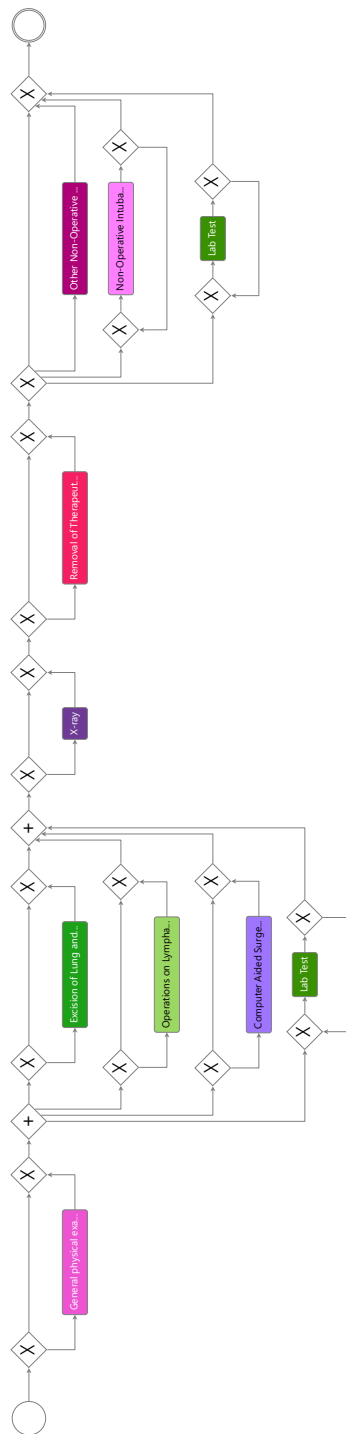


Figure 11.5: Initial process model visualized in BPMN describing the 15 most frequent variants (adapted from [187, Figure 9])

during treatment. Next, we added the remaining intricate variants, encompassing diagnostic activities and various surgical procedures. Figure 11.6 depicts the intermediate process model obtained. The model has a sequential relation between the activities of the diagnostic phase and those of the surgical phase. In contrast, the follow-up phase involves diverse activities that can be repeated multiple times, depending on the patient's health condition and reaction to the surgery.

Before incrementally adding the remaining variants related to non-surgical hospitalized patients, we froze all submodels concerning the surgical phase in the model. We applied freezing to ensure no changes in the surgical phase while continuing with the incremental process discovery. Figure 11.7 depicts the finally obtained process model that we derived from the provided event log and the healthcare experts' domain knowledge. The resulting model describes all suitable variants of the event log and shows that:

- the diagnostic and surgical phases are more structured, as recommended by the clinical guidelines,
- the follow-up phase relies on the experience of the physicians and the condition of the patients, and
- a patient may receive several treatments during the entire treatment period.

## Evaluation & Analysis

During the incremental process discovery phase, we constantly assessed the degree of conformance between the discovered model and the event log in Cortado. Especially after we applied manual changes to the model to ensure these changes did not remove already incorporated variants from the process model. Over time, we eventually reached full fitness, indicating a high level of conformance to the incrementally processed event log. Recall that we modified the given event log in various iterations; for example, we applied event abstraction and transformed various variants by applying sequentialization rules.

We organized a follow-up meeting with the entire medical team after completing multiple iterations of the interactive process discovery phase (cf. Figure 11.1). The meeting lasted for two hours and took place online via video conference. Its purpose was to obtain feedback from medical experts regarding the new interactive process discovery approach enabled by Cortado and the output process model obtained. During the meeting, we presented and discussed Cortado's capabilities in interactive data exploration/manipulation and incremental process discovery and showed two process models: one produced by Cortado (cf. Figure 11.7), and one produced by an automated process mining technique, i.e., the Inductive Miner [122] (cf. Figure 11.8), which is commonly used in healthcare [135]. As input for the IM, we used the event log obtained at the end of the *Event Data Processing and Exploration* phase to make the qualitative comparison of process models fairer. The model discovered by the IM places the main activities in parallel, which results in a loss of sequentiality between the central process stages: diagnosis, surgery, and follow-up. Since the IM cannot discover process models with duplicate labels compared to Cortado, the model discovered by the IM is imprecise. A lack of sequencing

of key activities leads to unrealistic process behavior, including incorrect positioning of non-invasive and invasive diagnostic tests.

The medical team found Cortado's interactive data exploration and manipulation helpful. Two members mentioned how difficult it can be to work with large amounts of patient data that often have quality issues. Extracting valuable insights from such data without dedicated tools can be challenging. Thus, they found it invaluable to use Cortado to explore the event data, identify patient journeys, and filter the event logs to obtain reliable and valuable information. Also, the incremental process discovery approach was considered as a positive aspect. A healthcare professional said that *"[...] what is even more value-added is to be able to incrementally model the process and decide what to include, freeze or modify in the map [...], exploiting our knowledge"*. To summarize, team members emphasized the crucial role of medical experts' active participation in filling data gaps and providing insights to enhance the model's quality. However, during the follow-up meeting, a critical point was raised about using interactive data exploration and incremental process discovery functionality. It has been emphasized that prior knowledge of process mining concepts and modeling languages is essential to fully utilizing Cortado's features. Alternatively, having the assistance of an experienced process analyst would ensure that Cortado's capabilities are fully employed. For instance, a team member pointed out: *"[...] I know neither Petri nets nor decision trees, and I would not be able to conduct the analysis alone [...]"*.

Regarding the process models, the medical team found the one discovered gradually with Cortado to be more accurate and consistent with the guidelines. However, it is more complex in terms of the number of elements due to duplicate labels. The model produced by the IM was deemed illogical. For instance, one team member even expressed uncertainty regarding the interpretation of the model, saying *"Is this the process for lung cancer patients?"*), and presented incorrect process behavior allowed by the model. Overall, the follow-up meeting provided helpful insights into the medical team's perspectives on the incremental approach enabled by Cortado.

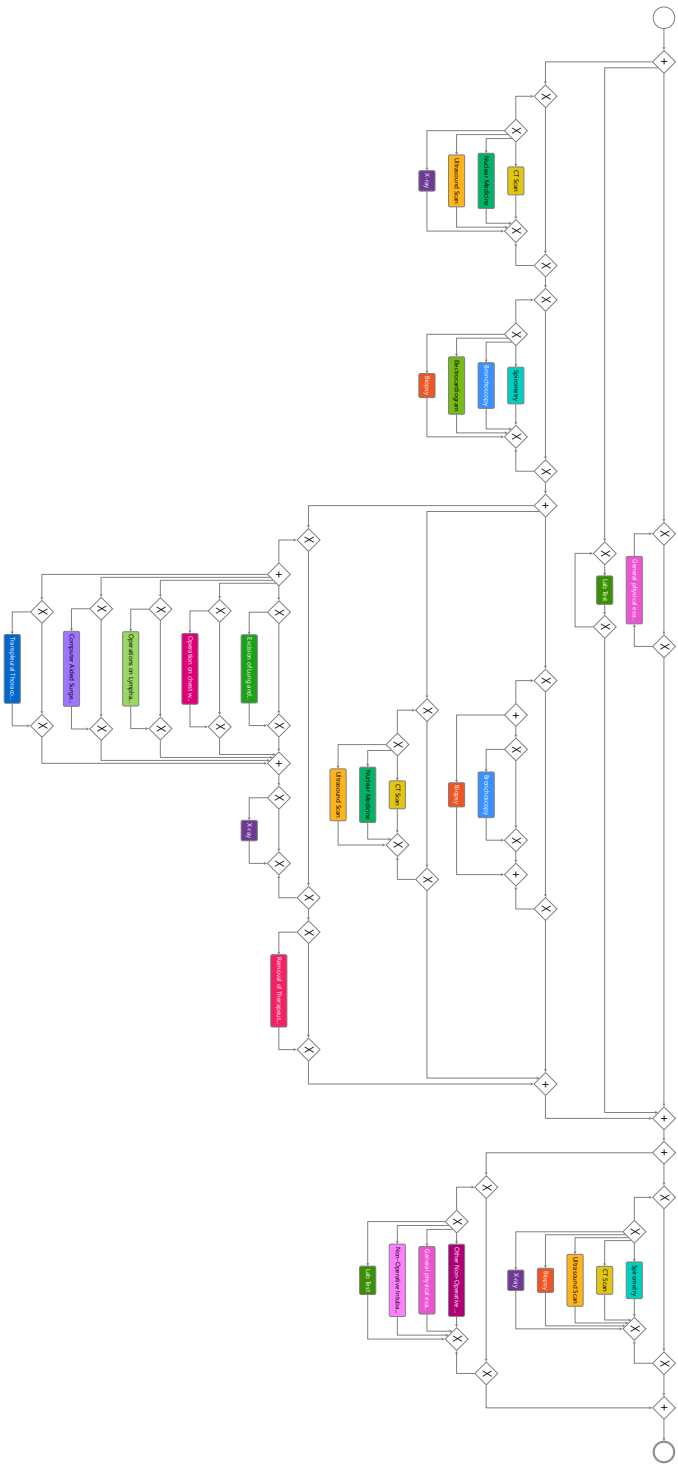


Figure 11.6: Intermediate process model visualized in BPMN depicting treatment options for lung cancer patients undergoing surgical treatment; the model illustrates the sequential relationship between diagnostic and surgical phases, while highlighting the diverse activities involved in the follow-up phase, which may be repeated based on individual patient reactions and health conditions (adapted from [187, Figure 10])

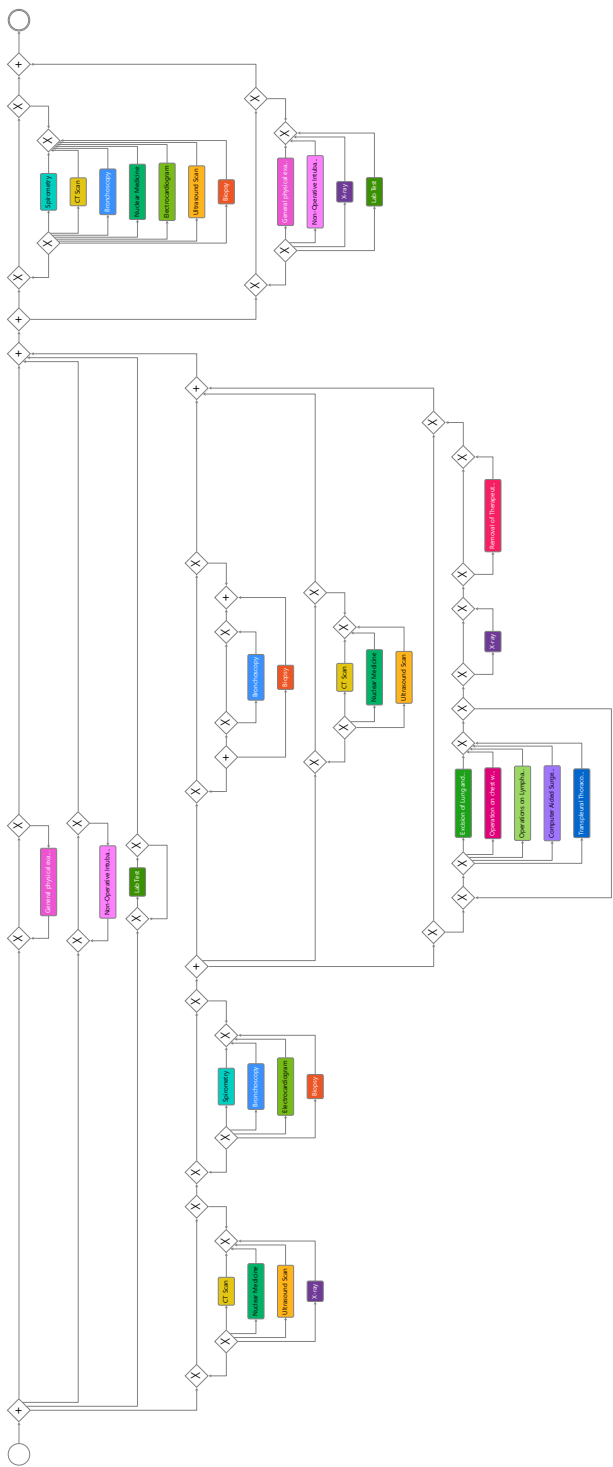


Figure 11.7: Final process model that describes the various options for lung cancer patient treatments obtained using Cortado; compared to Figure 11.8, this model is more precise and clearly structured because it can make case distinctions within different branches of the model since the IPDA approach in Cortado supports duplicate labels (adapted from [187, Figure 11])

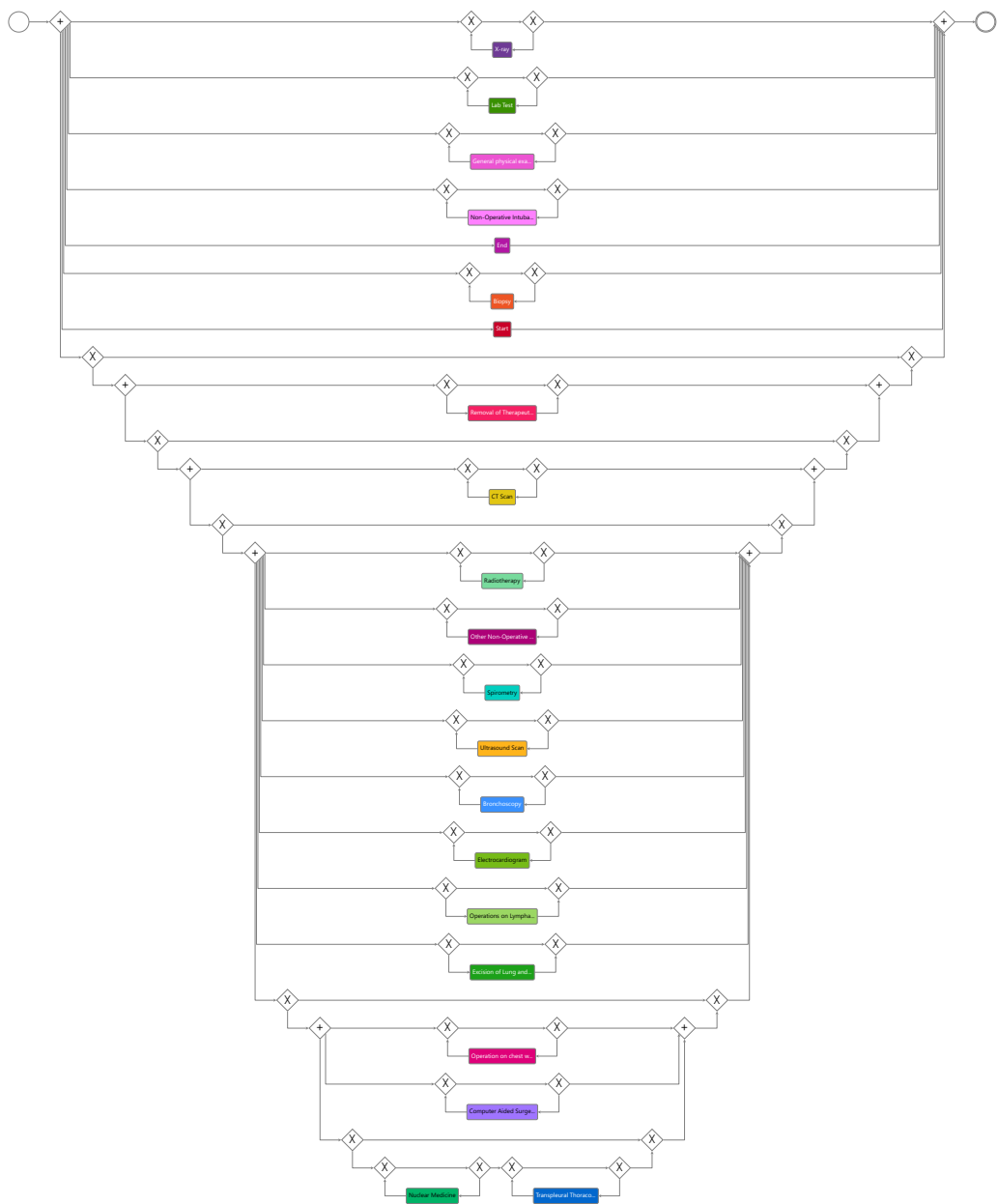


Figure 11.8: Process model visualized in BPMN discovered by the conventional process discovery algorithm Inductive Miner [122]; note that almost every activity is any times executable, resulting in an imprecise model (adapted from [187, Figure 12])

## 11.5. Discussion

This section discusses the presented case study, points out the lessons learned (cf. Section 11.5.1), derives practical implications (cf. Section 11.5.2), and indicates limitations and potential future research directions (cf. Section 11.5.3). In short, this case study indicates that the usual obstacles encountered in healthcare, such as a large variety of variants and event data quality issues, can be mitigated by using interactive process discovery techniques.

### 11.5.1. Lessons Learned

Below, we present three lessons learned from the presented case study.

#### **Incorporating Domain Experts**

Involving domain experts in any data-driven process modeling project is essential. Domain experts can add valuable information not found in event logs. Utilizing the domain experts' knowledge improves the finally obtained process models. In addition, the involvement of experts plays a crucial role in detecting and ultimately resolving event data quality issues. Assume, for instance, the scenario highlighted in this case study where clinical guidelines prescribe a specific sequence of examinations. However, domain experts can intervene and resolve this issue if the order of these examinations is not identifiable in the event data due to the lack of temporal information. It is essential to note that this is not an anomaly in the process but highlights a challenge related to the event data quality. In such situations, domain expertise becomes indispensable and highlights the general need for process mining techniques to allow the incorporation of domain knowledge. In short, this case study contributes to the open problem of utilizing domain knowledge [19] in process mining by showcasing a concrete application of Cortado to discover a healthcare process from event data and domain knowledge.

#### **Blending of Event Data Processing & Process Discovery**

Event data exploration and manipulation is integral to the process discovery phase and cannot be entirely automated. In the case study, many data quality issues became apparent during the *event data processing and exploration* phase. Some of these issues were easily recognizable, such as partial traces of only one activity. However, others required domain knowledge, such as traces with activity sequences that do not comply with medical guidelines. In this context, Cortado's strong focus on variants helps domain experts better understand the behavior recorded in an event log. As stated in [78], technical experts usually think about a process on a case-by-case basis rather than holistically. Therefore, the use of variant visualizations that summarize cases is of great benefit in communicating with domain experts and fosters them to apply their domain knowledge, for instance, by the variant sequentialization functionality (cf. Section 10.2.5).

Further, we learned that data exploration, process discovery, and analysis could not be performed sequentially, as often happens in traditional process mining methodologies [236, 160]. Sometimes, it becomes necessary to revisit and repeat intermediate steps

to improve or correct any inconsistencies in the event data. In our case study, while incrementally discovering the process model, we faced additional data issues that required revisiting the previous phase to carry out the *variant sequentialization* task.

### Incremental Process Discovery

Finally, it is essential to note that while incremental process discovery may seem more time-consuming than automated process discovery, IPD has been demonstrated in this case study to be more effective in producing accurate process models. Particularly when faced with event data quality issues, IPD allows, due to its incremental fashion, to switch between event data preparation and discovery. Thus, event data quality issues, often only partially visible initially, can be resolved during discovery. In contrast, automated discovery techniques often produce illogical and unreliable process models, as revealed in the follow-up meeting with the medical team. In addition, these models require further manual modifications, making the task laborious and time-consuming.

Using interactive and incremental discovery techniques requires fundamental process analysis and modeling knowledge to exploit their potential. The same, of course, also applies to automated discovery techniques. However, since IPD generally offers users more options, we consider it to be advanced in terms of user knowledge needed compared to automated process discovery. This challenge confirms the findings by [141], where the authors identify poor analytical skills from people as a critical challenge in applying process mining in organizations. Further, in [78], the authors identify the need for more process model formalism skills of domain experts as a challenge. The team involved in the study, but also medical teams in general, do not have the necessary expertise. Therefore, supporting the medical team with process analysts and providing the medical team with brief training was essential. The incremental approach made it easier for medical staff to understand the process model because it evolved gradually as more variations were added. As a result, incremental changes are more accessible to domain experts in the process model. These changes also provide a foundation for discussion, as only a few parts of the process model change in each iteration.

#### 11.5.2. Practical Implications

This case study demonstrates how Cortado, providing an incremental and interactive approach to process discovery, can help domain experts model healthcare processes effectively despite challenges posed by high process variability and poor event data quality. Specifically, Cortado enables users to quickly identify the variants that should be incorporated into a process model, eliminate incorrect ones, and obtain comprehensible process models to make informed decisions. The findings presented are of significant interest to healthcare managers and practitioners who seek to improve processes using data-driven approaches.

#### 11.5.3. Limitations & Future Work

Although this study yielded encouraging outcomes, it is crucial to consider certain limitations. The case study's findings are context-specific and may not be generalizable to



other healthcare settings and domains besides healthcare. Furthermore, the results were only validated through a single follow-up meeting with a small group of medical experts. While this meeting offered valuable feedback on the discovered models, a more comprehensive validation approach would be required to ensure the strength and reliability of the incremental discovery approach. A structured user study involving a broader, more diverse group of healthcare professionals would be needed to address these limitations and further develop the IPD approach developed in Cortado. Such a user study would allow us to quantitatively evaluate the interactive approach, gain additional insights, and investigate potential challenges.

## 11.6. Conclusion

In this case study, we showcased the usefulness of Cortado for analyzing a knowledge-intensive healthcare process. We used a real-life dataset from an Italian hospital documenting the treatment of lung cancer patients. We demonstrated how to use Cortado's functionalities to obtain a process model that is valuable to medical experts. Unlike other interactive and traditional process discovery techniques [23, 211], Cortado integrates event data exploration, processing, and manipulation with the process discovery/modeling phase, resulting in a more streamlined and integrated process discovery approach. In short, this case study showcased one solution regarding more effective approaches to discovering/modeling healthcare processes [146] by providing evidence on the effectiveness of applying incremental process discovery in healthcare.



Part V.

Closure



---

## Chapter 12.

# Conclusion

---

Process discovery—the central topic of this thesis—is a central research field within process mining. Several process discovery algorithms exist. Most can be classified as conventional, assuming event data as input and automatically discovering a process model. Thus, the actual process discovery phase is an automated black box from a user’s perspective. In contrast to conventional fully-automated discovery approaches, incremental process discovery allows users to utilize and integrate their knowledge and expertise about the process under consideration during the process discovery phase besides event data. IPD allows users to assess and manipulate the intermediate incrementally discovered process models and, through interaction, control the further process discovery phase.

Recall that there is not a single most-suited process model for a given event log but rather a multitude of possible models that all describe the event log to a certain degree. Selecting the best process model from this multitude depends on many factors, especially the user’s requirements regarding the purpose of the process model. For what is the process model used? For example, should it only contain the most important or frequently recorded process executions? Should it be combined with normative process behavior not included in the event log? Should it exclude particular process behavior? Should it exclusively describe the behavior in the event log, or should it also generalize and allow for behavior that does not occur in the event log? Since traditional process discovery is fully automated, obtaining a process model that meets a user’s requirements can be challenging. If a discovered model does not meet the requirements, users must discover an entirely new process model from scratch. Therefore, users must understand the various process discovery algorithms to select a suitable one that leads to the desired process model. Furthermore, users must understand the possibly diverse parameter settings of an applied process discovery algorithm to discover the targeted process model. IPD provides a unique approach to process discovery by enabling users to discover process models gradually. Thereby, IPD allows users to control and monitor the process discovery phase. Furthermore, the incremental discovery of a process model allows users to understand better where specific control flow patterns in the process model originate. In short, IPD, as proposed in this thesis, marks a novel approach to process discovery.

The remainder of this section is structured as follows. Section 12.1 summarizes the central contributions of this thesis. Subsequently, Section 12.2 discusses limitations and remaining challenges related to the contributions of this thesis. Finally, Section 12.3 outlines future work opportunities that build upon the proposed contributions.

## 12.1. Contributions

This section summarizes the four substantial contributions of this thesis: the review of domain-knowledge-utilizing process discovery (Section 12.1.1), IPD (Section 12.1.2), variants for partially ordered event data (Section 12.1.3), and Cortado (Section 12.1.4).

### 12.1.1. Review of Domain-Knowledge-Utilizing Process Discovery

Conventional process discovery approaches are commonly used in process mining, with various techniques available [15, 60, 235]. However, there are also a few non-conventional approaches to process discovery, which, from the user's point of view, can differ significantly more than conventional approaches, for example, concerning the interaction and required inputs. Since IPD is classified as a non-conventional approach, we presented a literature review on non-conventional process discovery approaches (cf. Chapter 2). We propose a taxonomy to classify non-conventional approaches based on several distinguishing features in this context. Due to the general nature of the taxonomy, we see the literature review as a central contribution, as the taxonomy can also be applied to future non-conventional discovery approaches. In addition, the literature review has revealed various challenges that exceed those addressed in this thesis. The review identified only twelve discovery approaches that can be considered non-conventional compared to many conventional process discovery approaches. IPD, as proposed in this thesis, was identified as a research gap on which hardly any research existed.

### 12.1.2. Incremental Process Discovery

This thesis proposed an incremental process discovery framework (cf. Chapter 5) that allows users to gradually discover a process model from event data by selecting the process behavior to be incorporated. Incrementally selecting process behavior that an IPDA subsequently incorporates into a process model is one central form of interaction within IPD. As such, the proposed IPD idea breaks with prevailing conventional process discovery, which operates fully automated and, therefore, does not allow for any form of user interaction. The overall framework, introduced in Chapter 5, is enhanced with support for incomplete process behavior, cf. Chapter 6. The extension of the IPD framework to support incomplete behavior is a practical application example for the infix and postfix alignments proposed in Chapter 4. For example, in the case of processes that span large parts of an organization or even several organizations, process execution fragments can be a helpful means of discovering a process model not only gradually but also by process stage. Finally, the IPD framework is enhanced with support for freezing, cf. Chapter 7. The freezing option in the context of IPD allows users to influence the IPDA by limiting the potential outputs. Freezing can be applied to any subtree of a given process tree. Upon conducting an IPD iteration, the frozen subtrees remain unchanged, i.e., the utilized IPDA is not allowed to alter frozen subtrees. Thus, submodel freezing represents a further novel form of user interaction with a process discovery algorithm. In short, an extensive IPD framework with two major extensions was proposed in this thesis.

### 12.1.3. Variants for Partially Ordered Event Data

Interacting with recorded process behavior is paramount within IPD. To enable users to incrementally select process behavior in the context of IPD, the event data must be adequately presented. To this end, this thesis proposed novel definitions for variants and corresponding visualizations in Chapter 8. Existing variant definitions and visualizations assume time-point-based events that are either totally or partially ordered. In comparison, we assume partially ordered event data with heterogeneous temporal information, i.e., time-point-based and time-interval-based events. Moreover, we proposed a textual query language for partially ordered event data, cf. Chapter 9. The query language allows to define control flow structures among activities that are assumed to be partially ordered.

### 12.1.4. Cortado

The comprehensive software tool Cortado (cf. Chapter 10), developed in the context of this thesis, demonstrates how IPD can be realized such that process analysts and process mining practitioners can apply it. Cortado features all approaches and algorithms proposed in this thesis. Thus, Cortado illustrates how the different contributions of this thesis are interrelated and interact in the larger context of IPD and serve a common goal. Cortado is open-source, which allows for reuse, customization, and further development. Moreover, we conducted a case study to evaluate Cortado and, thus, the various contributions of this work as a whole. Within the case study, we analyzed the treatment process of lung cancer patients. The results of the study showed that Cortado and IPD can be successfully applied in industrial contexts and have significant value compared to conventional process discovery approaches.

## 12.2. Limitations & Remaining Challenges

This section examines and discusses some of the limitations and challenges associated with the proposed contributions.

### 12.2.1. Nondeterminism of the LCA-IPDA

The LCA-IPDA proposed in Chapter 5 and extended in the two subsequent chapters by freezing and trace fragment support (cf. Chapters 6 and 7) utilizes alignments. Generally, there might exist more than one optimal alignment for a given trace and process tree; moreover, the optimal alignment returned by a search algorithm is usually random.<sup>1</sup> Since LCA-IPDA uses an optimal alignment to calculate the subtree that needs to be changed in the provided process tree, a different optimal alignment can lead to a different subtree and eventually to a different final process tree. Thus, even when using the same initial process model, the same previously added traces, and the same trace to be added next, different process models might be returned when executing LCA-IPDA multiple times.

---

<sup>1</sup>Note that computing all optimal alignments is possible [6, Section 4.6], however, it is exhaustive and may not even be possible to compute for large model trace combinations due to limited memory, cf. [6, Section 10.2].

Note that the guarantees of the LCA-IPDA hold, i.e., both the previously added traces and the trace to be added next are supported by the returned process tree. However, non-determinism in an algorithm, i.e. the same input may lead to different outputs in different executions, can be confusing for users, especially if they do not know the algorithmic details of the implemented IPDA.

### 12.2.2. Representational Bias

The IPDAs proposed in this thesis all discover process trees. The process tree formalism is crucial for the proposed IPDAs because they exploit the hierarchical structure of process trees to narrow down subtrees that must be altered. However, process trees are a subclass of sound WF-nets and are not as expressive as sound WF-nets, referred to as *representational bias* [206, 227]. A well-known control flow pattern that cannot be directly modeled using process trees is a *long-term dependency*.<sup>2</sup> In short, a long-term dependency describes control flow constraints where a choice at a branch within a model influences a choice at a subsequent branch. In short, IPDAs supporting larger subclasses of sound WF-nets other than process trees is a remaining challenge.

### 12.2.3. Support for Partially Ordered Event Data

Most existing process mining techniques assume totally ordered events per case [211, 218]. Partial-order-based process mining algorithms and approaches are still small in number compared to overall process mining [123]. In Part III, we propose novel variant definitions and visualizations as well as a corresponding query language for partially ordered event data. However, the proposed IPDAs (cf. Part II) assume totally ordered event data. By calculating all sequentializations for high-level variants in Cortado, high-level variants become compatible with the implemented IPDA. Since most process discovery algorithms, such as the Inductive Miner [122], which we use within the IPDA implemented in Cortado, can detect parallelism if most or all orderings are present in the provided traces, the applied sequentialization is feasible. However, a more sophisticated way would be to assume partially ordered event data natively also within the IPD framework and corresponding IPDAs. However, native support for partially ordered event data within IPD requires several changes and adaptations; for instance, alignments as introduced in Section 3.5 and Chapter 4 could not be used; alignments for partially ordered event data must be employed, for instance, [127]. Further, the mechanism to detect subtrees and compute sublogs must be adapted to be compatible with partially ordered alignments. Finally, the conventional process discovery algorithm utilized within the LCA-IPDA must also support partially ordered event data.

<sup>2</sup>However, it should be noted that such long-term dependencies can also be modeled in process trees using duplicate labels. However, depending on the situation, this approach can lead to huge process trees compared to sound WF-net, which can model long-term dependencies much simpler using fewer elements.



#### 12.2.4. Lack of Thorough User Evaluation

This thesis proposed Cortado—a significant contribution of this thesis—that combines the various algorithmic approaches proposed in this thesis into a comprehensive tool for process analysts. A dedicated user study has been performed for the proposed variant visualizations focused on ease of use and usefulness, cf. Chapter 8. For an overall evaluation of Cortado, we have conducted a case study utilizing Cortado to analyze a process from the healthcare domain. However, the obtained results from the case study are limited regarding generalizability. Thus, more systematic evaluations are needed to quantify better the value added of Cortado and IPD in general. Therefore, Cortado and its functionality should be considered as one comprehensive tool that is evaluated accordingly, for example, using controlled experiments focusing on tasks that require various functionalities of Cortado with a large number of participants to obtain reliable results.

#### 12.2.5. Incorporating Low-Level Variants

Chapter 8 introduced high-level and low-level variants. Although both variant types have been implemented in Cortado, only high-level variants are supported for IPD. Low-level variants are only perceived as a detailed view on individual high-level variants in Cortado, cf. Section 10.2.1. Thus, users cannot select individual low-level variants for using them within IPD; only high-level variants can be used. The main reason that low-level variants are not supported for IPD is the fact that process trees cannot represent the level of detail that can be expressed using low-level variants. However, for some processes, the abstraction level of low-level variants might be the intended one, so IPD should also be supported for low-level variants.

### 12.3. Future Research Directions

The proposed contributions in this thesis offer various new opportunities for future research. The limitations and remaining challenges discussed above in Section 12.2 provide promising avenues for further research. This section discusses further exemplary opportunities that do not directly originate from the abovementioned limitations and challenges.

#### 12.3.1. Beyond Adding Individual Traces in IPD

IPD as specified in the introduced framework, cf. Chapter 5, allows to gradually incorporate individual traces to an existing model. However, selecting individual traces might become tedious and impractical for users. Therefore, Cortado allows several variants to be selected at once and added to a model in one iteration from a user's perspective. However, Cortado executes internally several iterations of the implemented IPDA, i.e., traces derived from the selected variants are added individually.<sup>3</sup> Adding traces individually is computational complex because, for example, sublogs must be calculated many times, cf. Chapter 5. Moreover, ordering effects might occur, cf. Section 5.6. Thus, the proposed

---

<sup>3</sup>Recall that Cortado first computes all potential sequentializations for any selected variant. Next, each variant's sequentializations, i.e., traces, are added to the process model incrementally.

IPD framework (cf. Figure 5.1 on page 110) should be extended to allow adding more than one trace in a single iteration.

Thinking one step further, artifacts other than variants or traces could be considered as a medium to select process behavior that should be incorporated into a process model using IPD. An example of such alternative artifacts could be DECLARE constraints [151] that allow specifying certain dependencies among activities. Such constraints could be an additional input to the proposed IPD framework and would provide users another form of incorporating their domain knowledge during process discovery. Another example are local process models [198, 199, 201]. Local process models are represented in established process model formalisms like Petri nets or process trees. However, local process models do not capture the process from start to end; instead, they describe frequent patterns of the overall process model. An interesting direction for future work on IPD would be to investigate to which extent such local process models could be used to extend a given process model. Thus, users can select both variants/traces and local process models. Since local process models summarize incomplete parts of the overall process from different cases, their use in IPD could provide added value.

### 12.3.2. Incremental Process Reduction

IPD, as proposed in this thesis, is concerned with adding process behavior to an existing model. Further, the selected trace to be added next is always guaranteed to fit the resulting model. However, incrementally adding process behavior to a model might also end in imprecise process models, i.e., further process behavior is supported by the process model over time that has never been selected. An important direction for future work is to investigate techniques to remove selected process behavior supported by a process model. For instance, a framework for incremental process reduction could be designed similarly to the IPD framework proposed in Chapter 5. As input a process tree  $\Lambda$ , a trace to be removed  $\sigma_{remove} \in \mathbb{L}(\Lambda)$ , and traces to be kept  $A \subseteq \mathbb{L}(\Lambda)$  are provided to an incremental process reduction algorithm. The algorithm returns a process tree  $\Lambda'$  that does not support trace  $\sigma_{remove} \notin \mathbb{L}(\Lambda')$  but still supports traces contained in  $A \subseteq \mathbb{L}(\Lambda')$ .

### 12.3.3. Enhanced Interaction & Assistance

The central interaction between users and an IPDA lies in the incremental selection of process behavior and, optionally, the freezing of subtrees. However, further interaction between users and IPDAs is conceivable. For instance, an IPDA could involve users more by providing different solutions to alter the process model such that the new model supports the selected process behavior and users decide which solution is taken. In addition, IPD could benefit from assistance techniques to facilitate decision-making on the part of users. For example, users might be interested in recommendations on which traces to add to a given model; for instance, recommendations can be made based on the level of conformity between a variant and the current model. In this context, recommendation systems could indicate suitable subtrees for freezing to users.

#### 12.3.4. Incremental Discovery Beyond Control Flow

Process discovery, as considered in process mining, primarily focuses on learning control flow structures [15, 16, 60, 235]. This observation applies to process mining generally and is not limited to IPD. However, process model formalisms like BPMN also offer elements to model organizational aspects and data/information exchange. Especially in the context of IPDA, where users are assumed to be involved in the process discovery phase, incorporating additional perspectives to the process model based on event data and domain knowledge is a clear opportunity to enhance the prevailing control-flow-oriented process discovery. The central question is how IPD can combine event data and process experts' knowledge to discover process models that unite different perspectives.

#### 12.3.5. Supporting Object-Centric Event Data

Moving from traditional event data to object-centric event data [213], also referred to as multi-dimensional event data [83], is an approaching topic within process mining. Object-centric event logs do not contain case identifiers compared to the event logs assumed in this thesis. Thus, individual process executions do not exist within object-centric event logs. Events are instead through individual objects they are interacting with connected. Hence, identifying individual process executions and variants, central for IPD as proposed in this thesis, is already not trivial, and various approaches are possible [5]. Due to the increased complexity of the event data, process models can also become significantly more complex. Moreover, few process discovery approaches exist for object-centric event data [217]. Therefore, transferring the ideas of incremental process discovery, as proposed in this thesis, is an exciting research opportunity.



---

# References

---

- [1] IEEE standard for eXtensible Event Stream (XES) for achieving interoperability in event logs and event streams. *IEEE Std 1849-2016*, pages 1–50, 2016. doi:[10.1109/IEEESTD.2016.7740858](https://doi.org/10.1109/IEEESTD.2016.7740858).
- [2] G. Acampora, A. Vitiello, B. Di Stefano, W. M. P. van der Aalst, C. W. Günther, and H. M. W. Verbeek. IEEE 1849: The XES standard: The second IEEE standard sponsored by IEEE computational intelligence society [society briefs]. *IEEE Computational Intelligence Magazine*, 12(2):4–8, 2017. doi:[10.1109/MCI.2017.2670420](https://doi.org/10.1109/MCI.2017.2670420).
- [3] R. Accorsi and J. Leberherz. A practitioner’s view on process mining adoption, event log engineering and data challenges. In W. M. P. van der Aalst and J. Carmona, editors, *Process Mining Handbook*, volume 448 of *Lecture Notes in Business Information Processing*, pages 212–240. Springer, 2022. doi:[10.1007/978-3-031-08848-3\\_7](https://doi.org/10.1007/978-3-031-08848-3_7).
- [4] J. N. Adams, G. Park, S. Levich, D. Schuster, and W. M. P. van der Aalst. A framework for extracting and encoding features from object-centric event data. In J. Troya, B. Medjahed, M. Piattini, L. Yao, P. Fernández, and A. Ruiz-Cortés, editors, *Service-Oriented Computing*, volume 13740 of *Lecture Notes in Computer Science*, pages 36–53. Springer, 2022. doi:[10.1007/978-3-031-20984-0\\_3](https://doi.org/10.1007/978-3-031-20984-0_3).
- [5] J. N. Adams, D. Schuster, S. Schmitz, G. Schuh, and W. M. P. van der Aalst. Defining cases and variants for object-centric event data. In *2022 4th International Conference on Process Mining (ICPM)*, pages 128–135. IEEE, 2022. doi:[10.1109/ICPM57379.2022.9980730](https://doi.org/10.1109/ICPM57379.2022.9980730).
- [6] A. Adriansyah. *Aligning observed and modeled behavior*. Ph.D. thesis, Eindhoven University of Technology, Department of Mathematics and Computer Science, 2014.
- [7] A. Adriansyah, B. F. van Dongen, D. A. M. Piessens, M. T. Wynn, and M. Adams. Robust performance analysis on yawl process models with advanced constructs. *Journal of Information Technology Theory and Application (JITTA)*, 12(3):5–26, 2011. URL <https://aisel.aisnet.org/jitta/vol12/iss3/2>.
- [8] A. Adriansyah, J. Munoz-Gama, J. Carmona, B. F. van Dongen, and W. M. P. van der Aalst. Measuring precision of modeled behavior. *Information Systems and e-Business Management*, 13(1):37–67, 2015. doi:[10.1007/s10257-014-0234-7](https://doi.org/10.1007/s10257-014-0234-7).
- [9] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972. doi:[10.1137/0201008](https://doi.org/10.1137/0201008).
- [10] W. Aigner, S. Miksch, H. Schumann, and C. Tominski. *Visualization of Time-Oriented Data*. Springer, 2011. doi:[10.1007/978-0-85729-079-3](https://doi.org/10.1007/978-0-85729-079-3).

- [11] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983. doi:[10.1145/182.358434](https://doi.org/10.1145/182.358434).
- [12] R. Andrews, C. van Dun, M. T. Wynn, W. Kratsch, M. Röglinger, and A. ter Hofstede. Quality-informed semi-automated event log generation for process mining. *Decision Support Systems*, 132:113265, 2020. doi:[10.1016/j.dss.2020.113265](https://doi.org/10.1016/j.dss.2020.113265).
- [13] A. Armas-Cervantes, N. van Beest, M. La Rosa, M. Dumas, and S. Raboczi. Incremental and interactive business process model repair in apromore. *Proceedings of the BPM Demo Track and BPM Dissertation Award*, 1920, 2017. URL [https://ceur-ws.org/Vol-1920/BPM\\_2017\\_paper\\_206.pdf](https://ceur-ws.org/Vol-1920/BPM_2017_paper_206.pdf).
- [14] A. Armas-Cervantes, N. R. T. P. van Beest, M. La Rosa, M. Dumas, and L. García-Bañuelos. Interactive and incremental business process model repair. In H. Panetto, C. Debruyne, W. Gaaloul, M. Papazoglou, A. Paschke, C. A. Ardagna, and R. Meersman, editors, *On the Move to Meaningful Internet Systems. OTM 2017 Conferences*, volume 10573 of *Lecture Notes in Computer Science*, pages 53–74. Springer, 2017. doi:[10.1007/978-3-319-69462-7\\_5](https://doi.org/10.1007/978-3-319-69462-7_5).
- [15] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, F. M. Maggi, A. Marrella, M. Mecella, and A. Soo. Automated discovery of process models from event logs: Review and benchmark. *IEEE Transactions on Knowledge and Data Engineering*, 31(4): 686–705, 2019. doi:[10.1109/TKDE.2018.2841877](https://doi.org/10.1109/TKDE.2018.2841877).
- [16] A. Augusto, J. Carmona, and H. M. W. Verbeek. Advanced process discovery techniques. In W. M. P. van der Aalst and J. Carmona, editors, *Process Mining Handbook*, volume 448 of *Lecture Notes in Business Information Processing*, pages 76–107. Springer, 2022. doi:[10.1007/978-3-031-08848-3\\_3](https://doi.org/10.1007/978-3-031-08848-3_3).
- [17] P. Badakhshan, B. Wurm, T. Grisold, J. Geyer-Klingenberg, J. Mendling, and J. vom Brocke. Creating business value with process mining. *The Journal of Strategic Information Systems*, 31(4):101745, 2022. doi:[10.1016/j.jsis.2022.101745](https://doi.org/10.1016/j.jsis.2022.101745).
- [18] J. Barwise. An introduction to first-order logic. In *HANDBOOK OF MATHEMATICAL LOGIC*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 5–46. Elsevier, 1977. doi:[10.1016/S0049-237X\(08\)71097-8](https://doi.org/10.1016/S0049-237X(08)71097-8).
- [19] I. Beerepoot, C. Di Ciccio, H. A. Reijers, S. Rinderle-Ma, W. Bandara, A. Burattin, D. Calvanese, T. Chen, I. Cohen, B. Depaire, G. Di Federico, M. Dumas, C. van Dun, T. Fehrer, D. A. Fischer, A. Gal, M. Indulska, V. Isahagian, C. Klinkmüller, W. Kratsch, H. Leopold, A. van Looy, H. Lopez, S. Lukumbuzya, J. Mendling, L. Meyers, L. Moder, M. Montali, V. Muthusamy, M. Reichert, Y. Rizk, M. Rosemann, M. Röglinger, S. Sadiq, R. Seiger, T. Slaats, M. Simkus, I. A. Someh, B. Weber, I. Weber, M. Weske, and F. Zerbato. The biggest business process management problems to solve before we die. *Computers in Industry*, 146:103837, 2023. doi:[10.1016/j.compind.2022.103837](https://doi.org/10.1016/j.compind.2022.103837).
- [20] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes with bp-ql. *Information Systems*, 33(6):477–507, 2008. doi:[10.1016/j.is.2008.02.005](https://doi.org/10.1016/j.is.2008.02.005).

- [21] S.-M.-R. Beheshti, B. Benatallah, H. R. Motahari-Nezhad, and S. Sakr. A query language for analyzing business processes execution. In S. Rinderle-Ma, F. Toumani, and K. Wolf, editors, *Business Process Management*, volume 6896 of *Lecture Notes in Computer Science*, pages 281–297. Springer, 2011. doi:[10.1007/978-3-642-23059-2\\_22](https://doi.org/10.1007/978-3-642-23059-2_22).
- [22] E. Benevento, P. M. Dixit, M. F. Sani, D. Aloini, and W. M. P. van der Aalst. Evaluating the effectiveness of interactive process discovery in healthcare: A case study. In C. Di Francescomarino, R. Dijkman, and U. Zdun, editors, *Business Process Management Workshops*, volume 362 of *Lecture Notes in Business Information Processing*, pages 508–519. Springer, 2019. doi:[10.1007/978-3-030-37453-2\\_41](https://doi.org/10.1007/978-3-030-37453-2_41).
- [23] E. Benevento, D. Aloini, and W. M. P. van der Aalst. How can interactive process discovery address data quality issues in real business settings? evidence from a case study in healthcare. *Journal of Biomedical Informatics*, 130:104083, 2022. doi:[10.1016/j.jbi.2022.104083](https://doi.org/10.1016/j.jbi.2022.104083).
- [24] G. Bernard and P. Andritsos. Truncated trace classifier. removal of incomplete traces from event logs. In S. Nurcan, I. Reinhartz-Berger, P. Soffer, and J. Zdravkovic, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 387 of *Lecture Notes in Business Information Processing*, pages 150–165. Springer, 2020. doi:[10.1007/978-3-030-49418-6\\_10](https://doi.org/10.1007/978-3-030-49418-6_10).
- [25] A. Berti and W. M. P. van der Aalst. A novel token-based replay technique to speed up conformance checking and process enhancement. In M. Koutny, F. Kordon, and L. Pomello, editors, *Transactions on Petri Nets and Other Models of Concurrency XV*, volume 12530 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2021. doi:[10.1007/978-3-662-63079-2\\_1](https://doi.org/10.1007/978-3-662-63079-2_1).
- [26] A. Berti, C.-Y. Li, D. Schuster, and S. J. van Zelst. The process mining toolkit (PMTK): Enabling advanced process mining in an integrated fashion. In *Proceedings of the ICPM Doctoral Consortium and Demo Track 2021*, pages 43–44. CEUR Workshop Proceedings, 2021. URL [https://ceur-ws.org/Vol-3098/demo\\_206.pdf](https://ceur-ws.org/Vol-3098/demo_206.pdf).
- [27] A. Berti, S. J. van Zelst, and D. Schuster. PM4Py: A process mining library for Python. *Software Impacts*, 17:100556, 2023. doi:[10.1016/j.simpa.2023.100556](https://doi.org/10.1016/j.simpa.2023.100556).
- [28] A. Berti, H. Kourani, H. Häfke, C.-Y. Li, and D. Schuster. Evaluating large language models in process mining: Capabilities, benchmarks, and evaluation strategies. In H. van der Aa, D. Bork, R. Schmidt, and A. Sturm, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 511 of *Lecture Notes in Business Information Processing*, pages 13–21. Springer, 2024. doi:[10.1007/978-3-031-61007-3\\_2](https://doi.org/10.1007/978-3-031-61007-3_2).
- [29] A. Berti, D. Schuster, and W. M. P. van der Aalst. Abstractions, scenarios, and prompt definitions for process mining with LLMs: A case study. In J. de Weerd and L. Pufahl, editors, *Business Process Management Workshops*, volume 492 of

- Lecture Notes in Business Information Processing*, pages 427–439. Springer, 2024. doi:[10.1007/978-3-031-50974-2\\_32](https://doi.org/10.1007/978-3-031-50974-2_32).
- [30] P. Bertoli, C. Di Francescomarino, M. Dragoni, and C. Ghidini. Reasoning-based techniques for dealing with incomplete business process execution traces. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, M. Baldoni, C. Baroglio, G. Boella, and R. Micalizio, editors, *AI\*IA 2013: Advances in Artificial Intelligence*, volume 8249 of *Lecture Notes in Computer Science*, pages 469–480. Springer, 2013. doi:[10.1007/978-3-319-03524-6\\_40](https://doi.org/10.1007/978-3-319-03524-6_40).
- [31] F. Bezerra, J. Wainer, and W. M. P. van der Aalst. Anomaly detection using process mining. In T. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, and R. Ukor, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 29 of *Lecture Notes in Business Information Processing*, pages 149–161. Springer, 2009. doi:[10.1007/978-3-642-01862-6\\_13](https://doi.org/10.1007/978-3-642-01862-6_13).
- [32] J. Billington, S. Christensen, K. M. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The petri net markup language: Concepts, technology, and tools. In G. Goos, J. Hartmanis, J. van Leeuwen, W. M. P. van der Aalst, and E. Best, editors, *Applications and Theory of Petri Nets 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 483–505. Springer, 2003. doi:[10.1007/3-540-44919-1\\_31](https://doi.org/10.1007/3-540-44919-1_31).
- [33] A. Bogarín, R. Cerezo, and C. Romero. A survey on educational process mining. *WIREs Data Mining and Knowledge Discovery*, 8(1), 2018. doi:[10.1002/widm.1230](https://doi.org/10.1002/widm.1230).
- [34] A. Bolt and W. M. P. van der Aalst. Multidimensional process mining using process cubes. In K. Gaaloul, R. Schmidt, S. Nurcan, S. Guerreiro, and Q. Ma, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 214 of *Lecture Notes in Business Information Processing*, pages 102–116. Springer, 2015. doi:[10.1007/978-3-319-19237-6\\_7](https://doi.org/10.1007/978-3-319-19237-6_7).
- [35] R. P. J. C. Bose, R. S. Mans, and W. M. P. van der Aalst. Wanna improve process mining results? In *2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, pages 127–134. IEEE, 2013. doi:[10.1109/CIDM.2013.6597227](https://doi.org/10.1109/CIDM.2013.6597227).
- [36] A. Bottrighi, L. Canensi, G. Leonardi, S. Montani, and P. Terenziani. Trace retrieval for business process operational support. *Expert Systems with Applications*, 55:212–221, 2016. doi:[10.1016/j.eswa.2015.12.002](https://doi.org/10.1016/j.eswa.2015.12.002).
- [37] C. Breshears. *The art of concurrency: A thread monkey’s guide to writing parallel applications*. Theory in practice. O’Reilly, 1. ed. edition, 2009.
- [38] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. On the role of fitness, precision, generalization and simplicity in process discovery. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar,



- M. Y. Vardi, G. Weikum, R. Meersman, H. Panetto, T. Dillon, S. Rinderle-Ma, P. Dadam, X. Zhou, S. Pearson, A. Ferscha, S. Bergamaschi, and I. F. Cruz, editors, *On the Move to Meaningful Internet Systems: OTM 2012*, volume 7565 of *Lecture Notes in Computer Science*, pages 305–322. Springer, 2012. doi:[10.1007/978-3-642-33606-5\\_19](https://doi.org/10.1007/978-3-642-33606-5_19).
- [39] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. A genetic algorithm for discovering process trees. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2012. doi:[10.1109/CEC.2012.6256458](https://doi.org/10.1109/CEC.2012.6256458).
- [40] 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. *International Journal of Cooperative Information Systems*, 23(01): 1440001, 2014. doi:[10.1142/S0218843014400012](https://doi.org/10.1142/S0218843014400012).
- [41] A. Burattin. Streaming process discovery and conformance checking. In S. Sakr and A. Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*, pages 1–8. Springer, 2018. doi:[10.1007/978-3-319-63962-8\\_103-1](https://doi.org/10.1007/978-3-319-63962-8_103-1).
- [42] D. M. Campbell and D. Radford. Tree isomorphism algorithms: Speed vs. clarity. *Mathematics Magazine*, 64(4):252–261, 1991. doi:[10.1080/0025570X.1991.11977616](https://doi.org/10.1080/0025570X.1991.11977616).
- [43] L. Canensi, G. Leonardi, S. Montani, and P. Terenziani. Multi-level interactive medical process mining. In A. ten Teije, C. Popow, J. H. Holmes, and L. Sacchi, editors, *Artificial Intelligence in Medicine*, volume 10259 of *Lecture Notes in Computer Science*, pages 256–260. Springer, 2017. doi:[10.1007/978-3-319-59758-4\\_28](https://doi.org/10.1007/978-3-319-59758-4_28).
- [44] J. Carmona and R. Gavalda. Online techniques for dealing with concept drift in process mining. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, J. Hollmén, F. Klawonn, and A. Tucker, editors, *Advances in Intelligent Data Analysis XI*, volume 7619 of *Lecture Notes in Computer Science*, pages 90–102. Springer, 2012. doi:[10.1007/978-3-642-34156-4\\_10](https://doi.org/10.1007/978-3-642-34156-4_10).
- [45] J. Carmona, B. F. van Dongen, A. Solti, and M. Weidlich. *Conformance Checking*. Springer, 2018. doi:[10.1007/978-3-319-99414-7](https://doi.org/10.1007/978-3-319-99414-7).
- [46] J. Carmona, B. F. van Dongen, and M. Weidlich. Conformance checking: Foundations, milestones and challenges. In W. M. P. van der Aalst and J. Carmona, editors, *Process Mining Handbook*, volume 448 of *Lecture Notes in Business Information Processing*, pages 155–190. Springer, 2022. doi:[10.1007/978-3-031-08848-3\\_5](https://doi.org/10.1007/978-3-031-08848-3_5).
- [47] J. A. Carrasco-Ochoa, J. F. Martínez-Trinidad, J. H. Sossa-Azuela, J. A. Olvera López, and F. Famili, editors. *Pattern Recognition*. Lecture Notes in Computer Science. Springer, 2015. doi:[10.1007/978-3-319-19264-2](https://doi.org/10.1007/978-3-319-19264-2).
- [48] M. Chinosi and A. Trombetta. BPMN: An introduction to the standard. *Computer Standards & Interfaces*, 34(1):124–134, 2012. doi:[10.1016/j.csi.2011.06.002](https://doi.org/10.1016/j.csi.2011.06.002).

- [49] A. C. Choueiri, D. M. V. Sato, E. E. Scalabrin, and E. A. P. Santos. An extended model for remaining time prediction in manufacturing systems using process mining. *Journal of Manufacturing Systems*, 56:188–201, 2020. doi:[10.1016/j.jmsy.2020.06.003](https://doi.org/10.1016/j.jmsy.2020.06.003).
- [50] P. Ciancarini, A. Fantechi, and R. Gorrieri, editors. *Formal Methods for Open Object-Based Distributed Systems*. Springer, 1999. doi:[10.1007/978-0-387-35562-7](https://doi.org/10.1007/978-0-387-35562-7).
- [51] D. Dakic, D. Stefanovic, T. Lolic, D. Narandzic, and N. Simeunovic. Event log extraction for the purpose of process mining: A systematic literature review. In G. Prosteau, J. J. Lavios Villahoz, L. Brancu, and G. Bakacsi, editors, *Innovation in Sustainable Management and Entrepreneurship*, Springer Proceedings in Business and Economics, pages 299–312. Springer, 2020. doi:[10.1007/978-3-030-44711-3\\_22](https://doi.org/10.1007/978-3-030-44711-3_22).
- [52] M. R. Dallagassa, C. dos Santos Garcia, E. E. Scalabrin, S. O. Ioshii, and D. R. Carvalho. Opportunities and challenges for applying process mining in healthcare: a systematic mapping study. *Journal of Ambient Intelligence and Humanized Computing*, 13(1):165–182, 2022. doi:[10.1007/s12652-021-02894-7](https://doi.org/10.1007/s12652-021-02894-7).
- [53] F. D. Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, 13(3):319–340, 1989. doi:[10.2307/249008](https://doi.org/10.2307/249008).
- [54] L. de Bleser, R. Depreitere, K. de Waele, K. Vanhaecht, J. Vlayen, and W. Sermeus. Defining pathways. *Journal of Nursing Management*, 14(7):553–563, 2006. doi:[10.1111/j.1365-2934.2006.00702.x](https://doi.org/10.1111/j.1365-2934.2006.00702.x).
- [55] M. de Leoni. Foundations of process enhancement. In W. M. P. van der Aalst and J. Carmona, editors, *Process Mining Handbook*, volume 448 of *Lecture Notes in Business Information Processing*, pages 243–273. Springer, 2022. doi:[10.1007/978-3-031-08848-3\\_8](https://doi.org/10.1007/978-3-031-08848-3_8).
- [56] M. de Leoni and F. Mannhardt. Road traffic fine management process - event log. 4TU.Centre for Research Data, 2015. URL <https://doi.org/10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5>.
- [57] H. de Oliveira, V. Augusto, B. Jouaneton, L. Lamarsalle, M. Prodel, and X. Xie. Optimal process mining of timed event logs. *Information Sciences*, 528:58–78, 2020. doi:[10.1016/j.ins.2020.04.020](https://doi.org/10.1016/j.ins.2020.04.020).
- [58] E. de Roock and N. Martin. Process mining in healthcare - an updated perspective on the state of the art. *Journal of Biomedical Informatics*, 127:103995, 2022. doi:[10.1016/j.jbi.2022.103995](https://doi.org/10.1016/j.jbi.2022.103995).
- [59] J. de Weerd and M. T. Wynn. Foundations of process event data. In W. M. P. van der Aalst and J. Carmona, editors, *Process Mining Handbook*, volume 448 of *Lecture Notes in Business Information Processing*, pages 193–211. Springer, 2022. doi:[10.1007/978-3-031-08848-3\\_6](https://doi.org/10.1007/978-3-031-08848-3_6).

- [60] J. de Weerd, M. de Backer, J. Vanthienen, and B. Baesens. A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. *Information Systems*, 37(7):654–676, 2012. doi:[10.1016/j.is.2012.02.004](https://doi.org/10.1016/j.is.2012.02.004).
- [61] J. de Weerd, A. Schupp, Vanderloock, and B. Baesens. Process mining for the multi-faceted analysis of business processes—a case study in a financial services organization. *Computers in Industry*, 64(1):57–67, 2013. doi:[10.1016/j.compind.2012.09.010](https://doi.org/10.1016/j.compind.2012.09.010).
- [62] N. Deo. *Graph theory with applications to engineering and computer science*. Prentice-Hall series in automatic computation. Prentice-Hall, 1974.
- [63] J. Desel and J. Esparza. *Free choice Petri nets*, volume 40 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 1 edition, 1995. doi:[10.1017/CBO9780511526558](https://doi.org/10.1017/CBO9780511526558).
- [64] C. Di Ciccio, A. Marrella, and A. Russo. Knowledge-intensive processes: Characteristics, requirements and analysis of contemporary approaches. *Journal on Data Semantics*, 4(1):29–57, 2015. doi:[10.1007/s13740-014-0038-4](https://doi.org/10.1007/s13740-014-0038-4).
- [65] C. Di Francescomarino and C. Ghidini. Predictive process monitoring. In W. M. P. van der Aalst and J. Carmona, editors, *Process Mining Handbook*, volume 448 of *Lecture Notes in Business Information Processing*, pages 320–346. Springer, 2022. doi:[10.1007/978-3-031-08848-3\\_10](https://doi.org/10.1007/978-3-031-08848-3_10).
- [66] K. Diba, K. Batoulis, M. Weidlich, and M. Weske. Extraction, correlation, and abstraction of event data for process mining. *WIREs Data Mining and Knowledge Discovery*, 10(3), 2020. doi:[10.1002/widm.1346](https://doi.org/10.1002/widm.1346).
- [67] R. M. Dijkman, M. Dumas, and C. Ouyang. Semantics and analysis of business process models in bpmn. *Information and Software Technology*, 50(12):1281–1294, 2008. doi:[10.1016/j.infsof.2008.02.006](https://doi.org/10.1016/j.infsof.2008.02.006).
- [68] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. doi:[10.1007/BF01386390](https://doi.org/10.1007/BF01386390).
- [69] P. M. Dixit. *Interactive Process Mining*. Dissertation, Technische Universiteit Eindhoven., 2019. URL [https://research.tue.nl/files/127274756/20190619\\_Dixit.pdf](https://research.tue.nl/files/127274756/20190619_Dixit.pdf).
- [70] P. M. Dixit, J. C. A. M. Buijs, W. M. P. van der Aalst, B. F. A. Hompes, and J. Burman. Using domain knowledge to enhance process mining results. In P. Ceravolo and S. Rinderle-Ma, editors, *Data-Driven Process Discovery and Analysis*, volume 244 of *Lecture Notes in Business Information Processing*, pages 76–104. Springer, 2017. doi:[10.1007/978-3-319-53435-0\\_4](https://doi.org/10.1007/978-3-319-53435-0_4).
- [71] P. M. Dixit, J. C. A. M. Buijs, and W. M. P. van der Aalst. Prodigy: Human-in-the-loop process discovery. In *12th International Conference on Research Challenges in Information Science (RCIS)*, pages 1–12. IEEE, 2018. doi:[10.1109/RCIS.2018.8406657](https://doi.org/10.1109/RCIS.2018.8406657).

- [72] P. M. Dixit, J. C. A. M. Buijs, H. M. W. Verbeek, and W. M. P. van der Aalst. Fast incremental conformance analysis for interactive process discovery. In W. Abramowicz and A. Paschke, editors, *Business Information Systems*, volume 320 of *Lecture Notes in Business Information Processing*, pages 163–175. Springer, 2018. doi:[10.1007/978-3-319-93931-5\\_12](https://doi.org/10.1007/978-3-319-93931-5_12).
- [73] P. M. Dixit, S. Suriadi, R. Andrews, M. T. Wynn, A. H. M. ter Hofstede, J. C. A. M. Buijs, and W. M. P. van der Aalst. Detection and interactive repair of event ordering imperfection in process logs. In J. Krogstie and H. A. Reijers, editors, *Advanced Information Systems Engineering*, volume 10816 of *Lecture Notes in Computer Science*, pages 274–290. Springer, 2018. doi:[10.1007/978-3-319-91563-0\\_17](https://doi.org/10.1007/978-3-319-91563-0_17).
- [74] P. M. Dixit, H. M. W. Verbeek, J. C. A. M. Buijs, and W. M. P. van der Aalst. Interactive data-driven process model construction. In J. C. Trujillo, K. C. Davis, X. Du, Z. Li, T. W. Ling, G. Li, and M. L. Lee, editors, *Conceptual Modeling*, volume 11157 of *Lecture Notes in Computer Science*, pages 251–265. Springer, 2018. doi:[10.1007/978-3-030-00847-5\\_19](https://doi.org/10.1007/978-3-030-00847-5_19).
- [75] D. Duma and R. Aringhieri. Mining the patient flow through an emergency department to deal with overcrowding. In P. Cappanera, J. Li, A. Matta, E. Sahin, N. J. Vandaele, and F. Visintin, editors, *Health Care Systems Engineering*, volume 210 of *Springer Proceedings in Mathematics & Statistics*, pages 49–59. Springer, 2017. doi:[10.1007/978-3-319-66146-9\\_5](https://doi.org/10.1007/978-3-319-66146-9_5).
- [76] M. Dumas, W. M. P. van der Aalst, and A. ter Hofstede, editors. *Process-aware information systems: Bridging people and software through process technology*. Wiley-Interscience, 2005. doi:[10.1002/0471741442](https://doi.org/10.1002/0471741442).
- [77] M. Dumas, M. La Rosa, J. Mendling, and H. A. Reijers. *Fundamentals of Business Process Management*. Springer, 2013. doi:[10.1007/978-3-642-33143-5](https://doi.org/10.1007/978-3-642-33143-5).
- [78] M. Dumas, M. La Rosa, J. Mendling, and H. A. Reijers. *Fundamentals of Business Process Management*. Springer, 2 edition, 2018. doi:[10.1007/978-3-662-56509-4](https://doi.org/10.1007/978-3-662-56509-4).
- [79] S. Dunzer, M. Stierle, M. Matzner, and S. Baier. Conformance checking: A state-of-the-art literature review. In S. Betz, editor, *Proceedings of the 11th International Conference on Subject-Oriented Business Process Management - S-BPM ONE '19*, pages 1–10. ACM, 2019. doi:[10.1145/3329007.3329014](https://doi.org/10.1145/3329007.3329014).
- [80] R. Dybowski, K. B. Laskey, J. W. Myers, and S. Parsons. Introduction to the special issue on the fusion of domain knowledge with data for decision support. *The Journal of Machine Learning Research*, 4:293–294, 2003. URL <https://www.jmlr.org/papers/volume4/dybowski03a/dybowski03a.pdf>.
- [81] G. Engels, A. Förster, R. Heckel, and S. Thöne. Process modeling using uml. In M. Dumas, W. M. P. van der Aalst, and A. H. M. ter Hofstede, editors, *Process-Aware Information Systems*, pages 83–117. John Wiley & Sons, Inc, 2005. doi:[10.1002/0471741442.ch5](https://doi.org/10.1002/0471741442.ch5).

- [82] M. ER, N. Arsad, H. M. Astuti, R. P. Kusumawardani, and R. A. Utami. Analysis of production planning in a global manufacturing company with process mining. *Journal of Enterprise Information Management*, 31(2):317–337, 2018. doi:[10.1108/JEIM-01-2017-0003](https://doi.org/10.1108/JEIM-01-2017-0003).
- [83] S. Esser and D. Fahland. Multi-dimensional event data in graph databases. *Journal on Data Semantics*, 10(1-2):109–141, 2021. doi:[10.1007/s13740-021-00122-1](https://doi.org/10.1007/s13740-021-00122-1).
- [84] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011. URL <http://jmlr.org/papers/v12/pedregosa11a.html>.
- [85] D. Fahland and W. M. P. van der Aalst. Repairing process models to reflect reality. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, A. Barros, A. Gal, and E. Kindler, editors, *Business Process Management*, volume 7481 of *Lecture Notes in Computer Science*, pages 229–245. Springer, 2012. doi:[10.1007/978-3-642-32885-5\\_19](https://doi.org/10.1007/978-3-642-32885-5_19).
- [86] D. Fahland and W. M. P. van der Aalst. Model repair — aligning process models to reality. *Information Systems*, 47:220–243, 2015. doi:[10.1016/j.is.2013.12.007](https://doi.org/10.1016/j.is.2013.12.007).
- [87] D. Fahland, D. Lübke, J. Mendling, H. Reijers, B. Weber, M. Weidlich, and S. Zugal. Declarative versus imperative process modeling languages: The issue of understandability. In T. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, and R. Ukor, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 29 of *Lecture Notes in Business Information Processing*, pages 353–366. Springer, 2009. doi:[10.1007/978-3-642-01862-6\\_29](https://doi.org/10.1007/978-3-642-01862-6_29).
- [88] D. Fahland, J. Mendling, H. A. Reijers, B. Weber, M. Weidlich, and S. Zugal. Declarative versus imperative process modeling languages: The issue of maintainability. In S. Rinderle-Ma, S. Sadiq, and F. Leymann, editors, *Business Process Management Workshops*, volume 43 of *Lecture Notes in Business Information Processing*, pages 477–488. Springer, 2010. doi:[10.1007/978-3-642-12186-9\\_45](https://doi.org/10.1007/978-3-642-12186-9_45).
- [89] D. Falessi, N. Juristo, C. Wohlin, B. Turhan, J. Münch, A. Jedlitschka, and M. Oivo. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering*, 23(1):452–489, 2018. doi:[10.1007/s10664-017-9523-3](https://doi.org/10.1007/s10664-017-9523-3).
- [90] S. Ferilli. Incremental declarative process mining with woman. In *2020 IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS)*, pages 1–8. IEEE, 2020. doi:[10.1109/EAIS48028.2020.9122700](https://doi.org/10.1109/EAIS48028.2020.9122700).
- [91] S. Ferilli and F. Esposito. A logic framework for incremental learning of process models. *Fundamenta Informaticae*, 128:413–443, 2013. doi:[10.3233/FI-2013-951](https://doi.org/10.3233/FI-2013-951).

- [92] S. Ferilli, D. Redavid, and F. Esposito. Logic-based incremental process mining. In A. Bifet, M. May, B. Zadrozny, R. Gavalda, D. Pedreschi, F. Bonchi, J. Cardoso, and M. Spiliopoulou, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 9286 of *Lecture Notes in Computer Science*, pages 218–221. Springer, 2015. doi:[10.1007/978-3-319-23461-8\\_17](https://doi.org/10.1007/978-3-319-23461-8_17).
- [93] C. Fernández-Llatas, editor. *Interactive Process Mining in Healthcare*. Health Informatics. Springer, 2021. doi:[10.1007/978-3-030-53993-1](https://doi.org/10.1007/978-3-030-53993-1).
- [94] C. Fernández-Llatas, J. L. Bayo, A. Martinez-Romero, J. M. Benedi, and V. Traver. Interactive pattern recognition in cardiovascular disease management. a process mining approach. In *2016 IEEE-EMBS International Conference on Biomedical and Health Informatics (BHI)*, pages 348–351. IEEE, 2016. doi:[10.1109/BHI.2016.7455906](https://doi.org/10.1109/BHI.2016.7455906).
- [95] F. Friedrich, J. Mendling, and F. Puhlmann. Process model generation from natural language text. In R. King, editor, *Active Flow and Combustion Control 2018*, volume 141 of *Notes on Numerical Fluid Mechanics and Multidisciplinary Design*, pages 482–496. Springer, 2019. doi:[10.1007/978-3-642-21640-4\\_36](https://doi.org/10.1007/978-3-642-21640-4_36).
- [96] A. Ghose, G. Koliadis, and A. Chueng. Process discovery from model and text artefacts. In *2007 IEEE Congress on Services (Services 2007)*, pages 167–174. IEEE, 2007. doi:[10.1109/SERVICES.2007.52](https://doi.org/10.1109/SERVICES.2007.52).
- [97] A. Gianluigi Greco and L. P. Guzzo. Process discovery via precedence constraints. In Luc De Raedt, Christian Bessiere, Didier Dubois, Patrick Doherty, Paolo Frasconi, Fredrik Heintz, and Peter Lucas, editors, *Frontiers of Artificial Intelligence and Applications*, volume 242, pages 366–371. IOS Press, 2012. doi:[10.3233/978-1-61499-098-7-366](https://doi.org/10.3233/978-1-61499-098-7-366).
- [98] S. Goedertier, D. Martens, B. Baesens, R. Haesen, and J. Vanthienen. Process mining as first-order classification learning on logs with negative events. In D. Hutchinson, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, A. ter Hofstede, B. Benatallah, and H.-y. Paik, editors, *Business Process Management Workshops*, volume 4928 of *Lecture Notes in Computer Science*, pages 42–53. Springer, 2008. doi:[10.1007/978-3-540-78238-4\\_6](https://doi.org/10.1007/978-3-540-78238-4_6).
- [99] G. Greco, A. Guzzo, F. Lupia, and L. Pontieri. Process discovery under precedence constraints. *ACM Transactions on Knowledge Discovery from Data*, 9(4):1–39, 2015. doi:[10.1145/2710020](https://doi.org/10.1145/2710020).
- [100] M. Hammori, J. Herbst, and N. Kleiner. Interactive workflow mining. In T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, J. Desel, B. Pernici, and M. Weske, editors, *Business Process Management*, volume 3080 of *Lecture Notes in Computer Science*, pages 211–226. Springer, 2004. doi:[10.1007/978-3-540-25970-1\\_14](https://doi.org/10.1007/978-3-540-25970-1_14).

- [101] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi:[10.1109/tssc.1968.300136](https://doi.org/10.1109/tssc.1968.300136).
- [102] K. Häyrynen, K. Saranto, and P. Nykänen. Definition, structure, content, use and impacts of electronic health records: a review of the research literature. *International Journal of Medical Informatics*, 77(5):291–304, 2008. doi:[10.1016/j.ijmedinf.2007.09.001](https://doi.org/10.1016/j.ijmedinf.2007.09.001).
- [103] J. Hopcroft and R. Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973. doi:[10.1145/362248.362272](https://doi.org/10.1145/362248.362272).
- [104] D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, M. Li, B. Boehm, and L. J. Osterweil, editors. *Unifying the Software Process Spectrum*. Lecture Notes in Computer Science. Springer, 2006. doi:[10.1007/11608035](https://doi.org/10.1007/11608035).
- [105] Jane Webster and Richard T. Watson. Analyzing the past to prepare for the future: Writing a literature review. *MIS Quarterly*, 26(2):xiii–xxiii, 2002. URL <http://www.jstor.org/stable/4132319>.
- [106] M. Jans, M. Alles, and M. Vasarhelyi. The case for process mining in auditing: Sources of value added and areas of application. *International Journal of Accounting Information Systems*, 14(1):1–20, 2013. doi:[10.1016/j.accinf.2012.06.015](https://doi.org/10.1016/j.accinf.2012.06.015).
- [107] A. Janssen, J. Kay, S. Talic, M. Pusic, R. J. Birnbaum, R. Cavalcanti, D. Gasevic, and T. Shaw. Electronic health records that support health professional reflective practice: a missed opportunity in digital health. *Journal of Healthcare Informatics Research*, 6(4):375–384, 2022. doi:[10.1007/s41666-022-00123-0](https://doi.org/10.1007/s41666-022-00123-0).
- [108] C. S. Jensen and R. T. Snodgrass. Time instant. In L. LIU and M. T. ÖZSU, editors, *Encyclopedia of Database Systems*, page 3112. Springer, 2009. doi:[10.1007/978-0-387-39940-9\\_1516](https://doi.org/10.1007/978-0-387-39940-9_1516).
- [109] C. S. Jensen and R. T. Snodgrass. Time interval. In L. LIU and M. T. ÖZSU, editors, *Encyclopedia of Database Systems*, pages 3112–3113. Springer, 2009. doi:[10.1007/978-0-387-39940-9\\_1423](https://doi.org/10.1007/978-0-387-39940-9_1423).
- [110] Joos Buijs. Receipt phase of an environmental permit application process event log. 4TU.Centre for Research Data, 2014. URL <https://doi.org/10.4121/uuid:a07386a5-7be3-4367-9535-70bc9e77dbe6>.
- [111] E. Kindler, V. Rubin, and W. Schäfer. Incremental workflow mining based on document versioning information. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, M. Li, B. Boehm, and L. J. Osterweil, editors, *Unifying the Software Process Spectrum*,



- volume 3840 of *Lecture Notes in Computer Science*, pages 287–301. Springer, 2006. doi:[10.1007/11608035\\_25](https://doi.org/10.1007/11608035_25).
- [112] M. Kobeissi, N. Assy, W. Gaaloul, B. Defude, and B. Haidar. An intent-based natural language interface for querying process execution data. In *2021 3rd International Conference on Process Mining (ICPM)*, pages 152–159. IEEE, 2021. doi:[10.1109/ICPM53251.2021.9576850](https://doi.org/10.1109/ICPM53251.2021.9576850).
- [113] A. Koschmider, T. Hornung, and A. Oberweis. Recommendation-based editor for business process modeling. *Data & Knowledge Engineering*, 70(6):483–503, 2011. doi:[10.1016/j.datak.2011.02.002](https://doi.org/10.1016/j.datak.2011.02.002).
- [114] H. Kourani, D. Schuster, and W. M. P. van der Aalst. Scalable discovery of partially ordered workflow models with formal guarantees. In *2023 5th International Conference on Process Mining (ICPM)*, pages 89–96. IEEE, 2023. doi:[10.1109/ICPM60904.2023.10271941](https://doi.org/10.1109/ICPM60904.2023.10271941).
- [115] H. Kourani, A. Berti, D. Schuster, and W. M. P. van der Aalst. Process modeling with large language models. In H. van der Aa, D. Bork, R. Schmidt, and A. Sturm, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 511 of *Lecture Notes in Business Information Processing*, pages 229–244. Springer, 2024. doi:[10.1007/978-3-031-61007-3\\_18](https://doi.org/10.1007/978-3-031-61007-3_18).
- [116] M. La Rosa, M. Dumas, R. Uba, and R. Dijkman. Merging business process models. In R. Meersman, T. Dillon, and P. Herrero, editors, *On the Move to Meaningful Internet Systems: OTM 2010*, volume 6426 of *Lecture Notes in Computer Science*, pages 96–113. Springer, 2010. doi:[10.1007/978-3-642-16934-2\\_10](https://doi.org/10.1007/978-3-642-16934-2_10).
- [117] M. La Rosa, H. A. Reijers, W. M. P. van der Aalst, R. M. Dijkman, J. Mendling, M. Dumas, and L. García-Bañuelos. Apromore: An advanced process model repository. *Expert Systems with Applications*, 38(6):7029–7040, 2011. doi:[10.1016/j.eswa.2010.12.012](https://doi.org/10.1016/j.eswa.2010.12.012).
- [118] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964. doi:[10.1093/comjnl/6.4.308](https://doi.org/10.1093/comjnl/6.4.308).
- [119] S. J. J. Leemans. Automated process discovery. In S. Sakr and A. Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*, pages 121–130. Springer, 2019. doi:[10.1007/978-3-319-77525-8\\_88](https://doi.org/10.1007/978-3-319-77525-8_88).
- [120] S. J. J. Leemans. *Robust Process Mining with Guarantees*, volume 440 of *Lecture Notes in Business Information Processing*. Springer, 2022. doi:[10.1007/978-3-030-96655-3](https://doi.org/10.1007/978-3-030-96655-3).
- [121] S. J. J. Leemans and H. Leopold, editors. *Process Mining Workshops*. Lecture Notes in Business Information Processing. Springer, 2021. doi:[10.1007/978-3-030-72693-5](https://doi.org/10.1007/978-3-030-72693-5).
- [122] 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*, volume 7927, pages 311–329. Springer, 2013. doi:[10.1007/978-3-642-38697-8\\_17](https://doi.org/10.1007/978-3-642-38697-8_17).



- [123] S. J. J. Leemans, S. J. van Zelst, and X. Lu. Partial-order-based process mining: a survey and outlook. *Knowledge and Information Systems*, 65(1):1–29, 2023. doi:[10.1007/s10115-022-01777-3](https://doi.org/10.1007/s10115-022-01777-3).
- [124] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.
- [125] N. Lohmann, H. M. W. Verbeek, and R. Dijkman. Petri net transformations for business processes – a survey. In K. Jensen and W. M. P. van der Aalst, editors, *Transactions on Petri Nets and Other Models of Concurrency II*, volume 5460 of *Lecture Notes in Computer Science*, pages 46–63. Springer, 2009. doi:[10.1007/978-3-642-00899-3\\_3](https://doi.org/10.1007/978-3-642-00899-3_3).
- [126] G. Lomidze, D. Schuster, C.-Y. Li, and S. J. van Zelst. Enhanced transformation of BPMN models with cancellation features. In J. P. A. Almeida, D. Karastoyanova, G. Guizzardi, M. Montali, F. M. Maggi, and C. M. Fonseca, editors, *Enterprise Design, Operations, and Computing*, volume 13585 of *Lecture Notes in Computer Science*, pages 128–144. Springer, 2022. doi:[10.1007/978-3-031-17604-3\\_8](https://doi.org/10.1007/978-3-031-17604-3_8).
- [127] X. Lu, D. Fahland, and W. M. P. van der Aalst. Conformance checking based on partially ordered event data. In F. Fournier and J. Mendling, editors, *Business Process Management Workshops*, volume 202 of *Lecture Notes in Business Information Processing*, pages 75–88. Springer, 2015. doi:[10.1007/978-3-319-15895-2\\_7](https://doi.org/10.1007/978-3-319-15895-2_7).
- [128] X. Lu, D. Fahland, F. J. H. M. van den Biggelaar, and W. M. P. van der Aalst. Handling duplicated tasks in process discovery by refining event labels. In M. La Rosa, P. Loos, and O. Pastor, editors, *Business Process Management*, volume 9850 of *Lecture Notes in Computer Science*, pages 90–107. Springer, 2016. doi:[10.1007/978-3-319-45348-4\\_6](https://doi.org/10.1007/978-3-319-45348-4_6).
- [129] X. Lu, D. Fahland, R. Andrews, S. Suriadi, M. T. Wynn, A. H. M. ter Hofstede, and W. M. P. van der Aalst. Semi-supervised log pattern detection and exploration using event concurrence and contextual information. In H. Panetto, C. Debruyne, W. Gaaloul, M. Papazoglou, A. Paschke, C. A. Ardagna, and R. Meersman, editors, *On the Move to Meaningful Internet Systems. OTM 2017 Conferences*, volume 10573 of *Lecture Notes in Computer Science*, pages 154–174. Springer, 2017. doi:[10.1007/978-3-319-69462-7\\_11](https://doi.org/10.1007/978-3-319-69462-7_11).
- [130] F. M. Maggi, A. J. Mooij, and W. M. P. van der Aalst. User-guided discovery of declarative process models. In *2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, pages 192–199. IEEE, 2011. doi:[10.1109/CIDM.2011.5949297](https://doi.org/10.1109/CIDM.2011.5949297).
- [131] F. Mannhardt, M. de Leoni, H. A. Reijers, W. M. P. van der Aalst, and P. J. Tous-saint. From low-level events to activities - a pattern-based approach. In M. La Rosa, P. Loos, and O. Pastor, editors, *Business Process Management*, volume 9850 of *Lecture Notes in Computer Science*, pages 125–141. Springer, 2016. doi:[10.1007/978-3-319-45348-4\\_8](https://doi.org/10.1007/978-3-319-45348-4_8).

- [132] F. Mannhardt, M. de Leoni, H. A. Reijers, W. M. P. van der Aalst, and P. J. Toussaint. Guided process discovery – a pattern-based approach. *Information Systems*, 76:1–18, 2018. doi:[10.1016/j.is.2018.01.009](https://doi.org/10.1016/j.is.2018.01.009).
- [133] Mannhardt, Felix. Hospital billing - event log. 4TU.Centre for Research Data, 2012. URL <https://doi.org/10.4121/uuid:76c46b83-c930-4798-a1c9-4be94df741>.
- [134] R. S. Mans, M. H. Schonenberg, M. Song, W. M. P. van der Aalst, and P. J. M. Bakker. Application of process mining in healthcare – a case study in a dutch hospital. In A. Fred, J. Filipe, and H. Gamboa, editors, *Biomedical Engineering Systems and Technologies*, volume 25 of *Communications in Computer and Information Science*, pages 425–438. Springer, 2009. doi:[10.1007/978-3-540-92219-3\\_32](https://doi.org/10.1007/978-3-540-92219-3_32).
- [135] R. S. Mans, W. M. P. van der Aalst, and R. J. B. Vanwersch. *Process Mining in Healthcare*. Springer, 2015. doi:[10.1007/978-3-319-16071-9](https://doi.org/10.1007/978-3-319-16071-9).
- [136] I. Markovic, A. Costa Pereira, D. de Francisco, and H. Muñoz. Querying in business process modeling. In E. Di Nitto and M. Ripeanu, editors, *Service-Oriented Computing - ICSOC 2007 Workshops*, volume 4907 of *Lecture Notes in Computer Science*, pages 234–245. Springer, 2009. doi:[10.1007/978-3-540-93851-4\\_23](https://doi.org/10.1007/978-3-540-93851-4_23).
- [137] N. Martin. Using indoor location system data to enhance the quality of healthcare event logs: Opportunities and challenges. In F. Daniel, Q. Z. Sheng, and H. Motahari, editors, *Business Process Management Workshops*, volume 342 of *Lecture Notes in Business Information Processing*, pages 226–238. Springer, 2019. doi:[10.1007/978-3-030-11641-5\\_18](https://doi.org/10.1007/978-3-030-11641-5_18).
- [138] N. Martin, B. Depaire, and Caris. The use of process mining in a business process simulation context: Overview and challenges. In *2014 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, pages 381–388. IEEE, 2014. doi:[10.1109/CIDM.2014.7008693](https://doi.org/10.1109/CIDM.2014.7008693).
- [139] N. Martin, B. Depaire, and Caris. The use of process mining in business process simulation model construction: Structuring the field. *Business & Information Systems Engineering*, 58(1):73–87, 2016. doi:[10.1007/s12599-015-0410-4](https://doi.org/10.1007/s12599-015-0410-4).
- [140] N. Martin, A. Martinez-Millana, B. Valdivieso, and C. Fernández-Llatas. Interactive data cleaning for process mining: A case study of an outpatient clinic’s appointment system. In C. Di Francescomarino, R. Dijkman, and U. Zdun, editors, *Business Process Management Workshops*, volume 362 of *Lecture Notes in Business Information Processing*, pages 532–544. Springer, 2019. doi:[10.1007/978-3-030-37453-2\\_43](https://doi.org/10.1007/978-3-030-37453-2_43).
- [141] N. Martin, D. A. Fischer, G. D. Kerpedzhiev, K. Goel, S. J. J. Leemans, M. Röglinger, W. M. P. van der Aalst, M. Dumas, M. La Rosa, and M. T. Wynn. Opportunities and challenges for process mining in organizations: Results of a Delphi study. *Business & Information Systems Engineering*, 63(5):511–527, 2021. doi:[10.1007/s12599-021-00720-0](https://doi.org/10.1007/s12599-021-00720-0).

- [142] N. Martin, N. Wittig, and J. Munoz-Gama. Using process mining in healthcare. In W. M. P. van der Aalst and J. Carmona, editors, *Process Mining Handbook*, volume 448 of *Lecture Notes in Business Information Processing*, pages 416–444. Springer, 2022. doi:[10.1007/978-3-031-08848-3\\_14](https://doi.org/10.1007/978-3-031-08848-3_14).
- [143] M. Martini, D. Schuster, and W. M. P. van der Aalst. Mining frequent infix patterns from concurrency-aware process execution variants. *Proceedings of the VLDB Endowment*, 16(10):2666–2678, 2023. doi:[10.14778/3603581.3603603](https://doi.org/10.14778/3603581.3603603).
- [144] J. Mendling, B. F. van Dongen, and G. Neumann. Detection and prediction of errors in eps of the sap reference model. *Data & Knowledge Engineering*, 64(1): 312–329, 2008. doi:[10.1016/j.datak.2007.06.019](https://doi.org/10.1016/j.datak.2007.06.019).
- [145] O. Metsker, S. Kesarev, E. Bolgova, K. Golubev, A. Karsakov, A. Yakovlev, and S. Kovalchuk. Modelling and analysis of complex patient-treatment process using graphminer toolbox. In J. M. F. Rodrigues, P. J. S. Cardoso, J. Monteiro, R. Lam, V. V. Krzhizhanovskaya, M. H. Lees, J. J. Dongarra, and P. M. Sloot, editors, *Computational Science – ICCS 2019*, volume 11540 of *Lecture Notes in Computer Science*, pages 674–680. Springer, 2019. doi:[10.1007/978-3-030-22750-0\\_65](https://doi.org/10.1007/978-3-030-22750-0_65).
- [146] J. Munoz-Gama, N. Martin, C. Fernández-Llatas, O. A. Johnson, M. Sepúlveda, E. Helm, V. Galvez-Yanjari, E. Rojas, A. Martinez-Millana, D. Aloini, I. A. Aman-tea, R. Andrews, M. Arias, I. Beerepoot, E. Benevento, A. Burattin, D. Capurro, J. Carmona, M. Comuzzi, B. Dalmas, R. de La Fuente, C. Di Francesco-marino, C. Di Ciccio, R. Gatta, C. Ghidini, F. Gonzalez-Lopez, G. Ibanez-Sanchez, H. B. Klasky, A. Prima Kurniati, X. Lu, F. Mannhardt, R. Mans, M. Marcos, R. Medeiros de Carvalho, M. Pegoraro, S. K. Poon, L. Pufahl, H. A. Reijers, S. Remy, S. Rinderle-Ma, L. Sacchi, F. Seoane, M. Song, A. Stefanini, E. Sulis, A. H. M. ter Hofstede, P. J. Toussaint, V. Traver, Z. Valero-Ramon, I. de van Weerd, W. M. P. van der Aalst, R. J. B. Vanwersch, M. Weske, M. T. Wynn, and F. Zerbato. Process mining for healthcare: Characteristics and challenges. *Journal of Biomedical Informatics*, 127:103994, 2022. doi:[10.1016/j.jbi.2022.103994](https://doi.org/10.1016/j.jbi.2022.103994).
- [147] R. C. Nickerson, U. Varshney, and J. Muntermann. A method for taxonomy development and its application in information systems. *European Journal of Information Systems*, 22(3):336–359, 2013. doi:[10.1057/ejis.2012.26](https://doi.org/10.1057/ejis.2012.26).
- [148] G. Park and W. M. P. van der Aalst. Action-oriented process mining: bridging the gap between insights and actions. *Progress in Artificial Intelligence*, 2022. doi:[10.1007/s13748-022-00281-7](https://doi.org/10.1007/s13748-022-00281-7).
- [149] G. Park, D. Schuster, and W. M. P. van der Aalst. Pattern-based action engine: Generating process management actions using temporal patterns of process-centric problems. *Computers in Industry*, 153:104020, 2023. doi:[10.1016/j.compind.2023.104020](https://doi.org/10.1016/j.compind.2023.104020).
- [150] M. Park, M. Song, T. H. Baek, S. Son, S. J. Ha, and S. W. Cho. Workload and delay analysis in manufacturing process using process mining. In J. Bae, S. Suriadi, and L. Wen, editors, *Asia Pacific Business Process Management*, volume 219 of

- Lecture Notes in Business Information Processing*, pages 138–151. Springer, 2015. doi:[10.1007/978-3-319-19509-4\\_11](https://doi.org/10.1007/978-3-319-19509-4_11).
- [151] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst. Declare: Full support for loosely-structured processes. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, page 287. IEEE, 2007. doi:[10.1109/EDOC.2007.14](https://doi.org/10.1109/EDOC.2007.14).
- [152] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, 1977. doi:[10.1145/356698.356702](https://doi.org/10.1145/356698.356702).
- [153] P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, and H. A. Reijers. Imperative versus declarative process modeling languages: An empirical investigation. In F. Daniel, K. Barkaoui, and S. Dustdar, editors, *Business Process Management Workshops*, volume 99 of *Lecture Notes in Business Information Processing*, pages 383–394. Springer, 2012. doi:[10.1007/978-3-642-28108-2\\_37](https://doi.org/10.1007/978-3-642-28108-2_37).
- [154] G. Polančič and B. Cegnar. Complexity metrics for process models – a systematic literature review. *Computer Standards & Interfaces*, 51:104–117, 2017. doi:[10.1016/j.csi.2016.12.003](https://doi.org/10.1016/j.csi.2016.12.003).
- [155] A. Polyvyanyy. Business process querying. In S. Sakr and A. Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*, pages 1–9. Springer, 2019. doi:[10.1007/978-3-319-77525-8\\_108](https://doi.org/10.1007/978-3-319-77525-8_108).
- [156] A. Polyvyanyy. *Process Querying Methods*. Springer, 2022. doi:[10.1007/978-3-030-92875-9](https://doi.org/10.1007/978-3-030-92875-9).
- [157] A. Polyvyanyy, C. Ouyang, A. Barros, and W. M. P. van der Aalst. Process querying: Enabling business intelligence through query-based process analytics. *Decision Support Systems*, 100:41–56, 2017. doi:[10.1016/j.dss.2017.04.011](https://doi.org/10.1016/j.dss.2017.04.011).
- [158] M. Räm, C. Di Ciccio, F. M. Maggi, M. Mecella, and J. Mendling. Log-based understanding of business processes through temporal logic query checking. In R. Meersman, H. Panetto, T. Dillon, M. Missikoff, L. Liu, O. Pastor, A. Cuzzocrea, and T. Sellis, editors, *On the Move to Meaningful Internet Systems: OTM 2014 Conferences*, volume 8841 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2014. doi:[10.1007/978-3-662-45563-0\\_5](https://doi.org/10.1007/978-3-662-45563-0_5).
- [159] T. Ramraj and R. Prabhakar. Frequent subgraph mining algorithms – a survey. *Procedia Computer Science*, 47:197–204, 2015. doi:[10.1016/j.procs.2015.03.198](https://doi.org/10.1016/j.procs.2015.03.198).
- [160] Á. Rebuge and D. R. Ferreira. Business process analysis in healthcare environments: A methodology based on process mining. *Information Systems*, 37(2):99–116, 2012. doi:[10.1016/j.is.2011.01.003](https://doi.org/10.1016/j.is.2011.01.003).
- [161] A. J. Rembert, A. Omokpo, P. Mazzoleni, and R. T. Goodwin. Process discovery using prior knowledge. In S. Basu, C. Pautasso, L. Zhang, and X. Fu, editors, *Service-Oriented Computing*, pages 328–342. Springer, 2013. doi:[10.1007/978-3-642-45005-1\\_23](https://doi.org/10.1007/978-3-642-45005-1_23).

- [162] K. Revoredo. On the use of domain knowledge for process model repair. *Software and Systems Modeling*, 2022. doi:[10.1007/s10270-022-01067-0](https://doi.org/10.1007/s10270-022-01067-0).
- [163] J. Ribeiro, J. Carmona, M. Mısıır, and M. Sebag. A recommender system for process discovery. In S. Sadiq, P. Soffer, and H. Völzer, editors, *Business Process Management*, volume 8659 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2014. doi:[10.1007/978-3-319-10172-9\\_5](https://doi.org/10.1007/978-3-319-10172-9_5).
- [164] E. Rojas, J. Munoz-Gama, M. Sepúlveda, and D. Capurro. Process mining in healthcare: A literature review. *Journal of Biomedical Informatics*, 61:224–236, 2016. doi:[10.1016/j.jbi.2016.04.007](https://doi.org/10.1016/j.jbi.2016.04.007).
- [165] A. Rozinat and W. M. P. van der Aalst. Conformance checking of processes based on monitoring real behavior. *Information Systems*, 33(1):64–95, 2008. doi:[10.1016/j.is.2007.07.001](https://doi.org/10.1016/j.is.2007.07.001).
- [166] A. Rozinat, I. de Jong, C. W. Günther, and W. M. P. van der Aalst. Process mining applied to the test process of wafer scanners in asml. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 39(4):474–479, 2009. doi:[10.1109/TSMCC.2009.2014169](https://doi.org/10.1109/TSMCC.2009.2014169).
- [167] V. Rubin, C. W. Günther, W. M. P. van der Aalst, E. Kindler, B. F. van Dongen, and W. Schäfer. Process mining framework for software processes. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. P. Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, Q. Wang, D. Pfahl, and D. M. Raffo, editors, *Software Process Dynamics and Agility*, volume 4470 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 2007. doi:[10.1007/978-3-540-72426-1\\_15](https://doi.org/10.1007/978-3-540-72426-1_15).
- [168] K. Salimifard and M. Wright. Petri net-based modelling of workflow systems: An overview. *European Journal of Operational Research*, 134(3):664–676, 2001. doi:[10.1016/S0377-2217\(00\)00292-7](https://doi.org/10.1016/S0377-2217(00)00292-7).
- [169] M. F. Sani, S. J. van Zelst, and W. M. P. van der Aalst. Improving process discovery results by filtering outliers using conditional behavioural probabilities. In E. Teniente and M. Weidlich, editors, *Business Process Management Workshops*, volume 308 of *Lecture Notes in Business Information Processing*, pages 216–229. Springer, 2018. doi:[10.1007/978-3-319-74030-0\\_16](https://doi.org/10.1007/978-3-319-74030-0_16).
- [170] M. F. Sani, A. Berti, S. J. van Zelst, and W. M. P. van der Aalst. Filtering toolkit: Interactively filter event logs to improve the quality of discovered models. In B. Depaire, J. de Smedt, M. Dumas, D. Fahland, A. Kumar, H. Leopold, M. Reichert, S. Rinderle-Ma, S. Schulte, S. Seidel, and W. M. P. van der Aalst, editors, *BPM 2019 Dissertation Award, Doctoral Consortium, and Demonstration Track*, CEUR Workshop Proceedings, pages 134–138. CEUR Workshop Proceedings, 2019. URL <https://ceur-ws.org/Vol-2420/paperDT4.pdf>.
- [171] A.-W. Scheer, O. Thomas, and O. Adam. Process modeling using event-driven process chains. In M. Dumas, W. M. P. van der Aalst, and A. H. M. ter Hofstede,

- editors, *Process-Aware Information Systems*, pages 119–145. John Wiley & Sons, Inc, 2005. doi:[10.1002/0471741442.ch6](https://doi.org/10.1002/0471741442.ch6).
- [172] D. Schuster and G. J. Kolhof. Scalable online conformance checking using incremental prefix-alignment computation. In H. Hacid, F. Outay, H.-y. Paik, A. Alloum, M. Petrocchi, M. R. Bouadjenek, A. Beheshti, X. Liu, and A. Maaradji, editors, *Service-Oriented Computing – ICSOC 2020 Workshops*, volume 12632 of *Lecture Notes in Computer Science*, pages 379–394. Springer, 2021. doi:[10.1007/978-3-030-76352-7\\_36](https://doi.org/10.1007/978-3-030-76352-7_36).
- [173] D. Schuster and S. J. van Zelst. Online process monitoring using incremental state-space expansion: An exact algorithm. In D. Fahland, C. Ghidini, J. Becker, and M. Dumas, editors, *Business Process Management*, volume 12168 of *Lecture Notes in Computer Science*, pages 147–164. Springer, 2020. doi:[10.1007/978-3-030-58666-9\\_9](https://doi.org/10.1007/978-3-030-58666-9_9).
- [174] D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Incremental discovery of hierarchical process models. In F. Dalpiaz, J. Zdravkovic, and P. Loucopoulos, editors, *Research Challenges in Information Science*, volume 385 of *Lecture Notes in Business Information Processing*, pages 417–433. Springer, 2020. doi:[10.1007/978-3-030-50316-1\\_25](https://doi.org/10.1007/978-3-030-50316-1_25).
- [175] D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Alignment approximation for process trees. In S. J. J. Leemans and H. Leopold, editors, *Process Mining Workshops*, volume 406 of *Lecture Notes in Business Information Processing*, pages 247–259. Springer, 2021. doi:[10.1007/978-3-030-72693-5\\_19](https://doi.org/10.1007/978-3-030-72693-5_19).
- [176] D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Freezing sub-models during incremental process discovery. In A. Ghose, J. Horkoff, V. E. Silva Souza, J. Parsons, and J. Evermann, editors, *Conceptual Modeling*, volume 13011 of *Lecture Notes in Computer Science*, pages 14–24. Springer, 2021. doi:[10.1007/978-3-030-89022-3\\_2](https://doi.org/10.1007/978-3-030-89022-3_2).
- [177] D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Sub-model freezing during incremental process discovery in cortado. In *Proceedings of the ICPM Doctoral Consortium and Demo Track 2021*, pages 43–44. CEUR Workshop Proceedings, 2021. URL [https://ceur-ws.org/Vol-3098/demo\\_207.pdf](https://ceur-ws.org/Vol-3098/demo_207.pdf).
- [178] D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Cortado—an interactive tool for data-driven process discovery and modeling. In D. Buchs and J. Carmona, editors, *Application and Theory of Petri Nets and Concurrency*, volume 12734 of *Lecture Notes in Computer Science*, pages 465–475. Springer, 2021. doi:[10.1007/978-3-030-76983-3\\_23](https://doi.org/10.1007/978-3-030-76983-3_23).
- [179] D. Schuster, E. Domnitsch, S. J. van Zelst, and W. M. P. van der Aalst. A generic trace ordering framework for incremental process discovery. In T. Bouadi, E. Fromont, and E. Hüllermeier, editors, *Advances in Intelligent Data Analysis XX*, volume 13205 of *Lecture Notes in Computer Science*, pages 264–277. Springer, 2022. doi:[10.1007/978-3-031-01333-1\\_21](https://doi.org/10.1007/978-3-031-01333-1_21).



- [180] D. Schuster, N. Föcking, S. J. van Zelst, and W. M. P. van der Aalst. Conformance checking for trace fragments using infix and postfix alignments. In M. Sellami, P. Ceravolo, H. A. Reijers, W. Gaaloul, and H. Panetto, editors, *Cooperative Information Systems*, volume 13591 of *Lecture Notes in Computer Science*, pages 299–310. Springer, 2022. doi:[10.1007/978-3-031-17834-4\\_18](https://doi.org/10.1007/978-3-031-17834-4_18).
- [181] D. Schuster, M. Martini, S. J. van Zelst, and W. M. P. van der Aalst. Control-flow-based querying of process executions from partially ordered event data. In J. Troya, B. Medjahed, M. Piattini, L. Yao, P. Fernández, and A. Ruiz-Cortés, editors, *Service-Oriented Computing*, volume 13740 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2022. doi:[10.1007/978-3-031-20984-0\\_2](https://doi.org/10.1007/978-3-031-20984-0_2).
- [182] D. Schuster, L. Schade, S. J. van Zelst, and W. M. P. van der Aalst. Temporal performance analysis for block-structured process models in Cortado. In J. de Weerd and A. Polyvyanyy, editors, *Intelligent Information Systems*, volume 452 of *Lecture Notes in Business Information Processing*, pages 110–119. Springer, 2022. doi:[10.1007/978-3-031-07481-3\\_13](https://doi.org/10.1007/978-3-031-07481-3_13).
- [183] D. Schuster, L. Schade, S. J. van Zelst, and W. M. P. van der Aalst. Visualizing trace variants from partially ordered event data. In J. Munoz-Gama and X. Lu, editors, *Process Mining Workshops*, volume 433 of *Lecture Notes in Business Information Processing*, pages 34–46. Springer, 2022. doi:[10.1007/978-3-030-98581-3\\_3](https://doi.org/10.1007/978-3-030-98581-3_3).
- [184] D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Utilizing domain knowledge in data-driven process discovery: A literature review. *Computers in Industry*, 137: 103612, 2022. doi:[10.1016/j.compind.2022.103612](https://doi.org/10.1016/j.compind.2022.103612).
- [185] D. Schuster, N. Föcking, S. J. van Zelst, and W. M. P. van der Aalst. Incremental discovery of process models using trace fragments. In C. Di Francescomarino, A. Burattin, C. Janiesch, and S. Sadiq, editors, *Business Process Management*, volume 14159 of *Lecture Notes in Computer Science*, pages 55–73. Springer, 2023. doi:[10.1007/978-3-031-41620-0\\_4](https://doi.org/10.1007/978-3-031-41620-0_4).
- [186] D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Cortado: A dedicated process mining tool for interactive process discovery. *SoftwareX*, 22:101373, 2023. doi:[10.1016/j.softx.2023.101373](https://doi.org/10.1016/j.softx.2023.101373).
- [187] D. Schuster, E. Benevento, D. Aloini, and W. M. P. van der Aalst. Analyzing healthcare processes with incremental process discovery: Practical insights from a real-world application. *Journal of Healthcare Informatics Research*, 2024. doi:[10.1007/s41666-024-00165-6](https://doi.org/10.1007/s41666-024-00165-6).
- [188] D. Schuster, F. Zerbato, S. J. van Zelst, and W. M. P. van der Aalst. Defining and visualizing process execution variants from partially ordered event data. *Information Sciences*, 657:119958, 2024. doi:[10.1016/j.ins.2023.119958](https://doi.org/10.1016/j.ins.2023.119958).
- [189] E. Serral, D. Schuster, and Y. Bertrand. Supporting users in the continuous evolution of automated routines in their smart spaces. In A. Marrella and B. Weber, editors, *Business Process Management Workshops*, volume 436 of *Lecture Notes in*

- Business Information Processing*, pages 391–402. Springer, 2022. doi:[10.1007/978-3-030-94343-1\\_30](https://doi.org/10.1007/978-3-030-94343-1_30).
- [190] S. Shaikh, M. F. Sani, and Jans M. Janssenswillen G. Kalenkova A. Maggi F.M. Process model simplification based on probabilities in process tree. *CEUR Workshop Proceedings*, 3098, 2021. URL [https://ceur-ws.org/Vol-3098/demo\\_210.pdf](https://ceur-ws.org/Vol-3098/demo_210.pdf).
- [191] R. Shraga, A. Gal, D. Schumacher, A. Senderovich, and M. Weidlich. Process discovery with context-aware process trees. *Information Systems*, 106:101533, 2022. doi:[10.1016/j.is.2020.101533](https://doi.org/10.1016/j.is.2020.101533).
- [192] T. Slaats, S. Debois, and C. O. Back. Weighing the pros and cons: Process discovery with negative examples. In A. Polyvyanyy, M. T. Wynn, A. van Looy, and M. Reichert, editors, *Business Process Management*, volume 12875 of *Lecture Notes in Computer Science*, pages 47–64. Springer, 2021. doi:[10.1007/978-3-030-85469-0\\_6](https://doi.org/10.1007/978-3-030-85469-0_6).
- [193] A. Stefanini, D. Aloini, E. Benevento, R. Dulmin, and V. Mininno. Performance analysis in emergency departments: a data-driven approach. *Measuring Business Excellence*, 22(2):130–145, 2018. doi:[10.1108/MBE-07-2017-0040](https://doi.org/10.1108/MBE-07-2017-0040).
- [194] A. Stefanini, D. Aloini, E. Benevento, R. Dulmin, and V. Mininno. A process mining methodology for modeling unstructured processes. *Knowledge and Process Management*, 27(4):294–310, 2020. doi:[10.1002/kpm.1649](https://doi.org/10.1002/kpm.1649).
- [195] V. Stein Dani, H. Leopold, J. M. E. M. van der Werf, X. Lu, I. Beerepoot, J. J. Koorn, and H. A. Reijers. Towards understanding the role of the human in event log extraction. In A. Marrella and B. Weber, editors, *Business Process Management Workshops*, volume 436 of *Lecture Notes in Business Information Processing*, pages 86–98. Springer, 2022. doi:[10.1007/978-3-030-94343-1\\_7](https://doi.org/10.1007/978-3-030-94343-1_7).
- [196] Stijn Goedertier, David Martens, Jan Vanthienen, and Bart Baesens. Robust process discovery with artificial negative events. *Journal of Machine Learning Research*, 10(44):1305–1340, 2009. URL <http://jmlr.org/papers/v10/goedertier09a.html>.
- [197] M. Svahnberg, A. Aurum, and C. Wohlin. Using students as subjects - an empirical evaluation. In D. Rombach, S. Elbaum, and J. Münch, editors, *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '08*, pages 288–290. ACM, 2008. doi:[10.1145/1414004.1414055](https://doi.org/10.1145/1414004.1414055).
- [198] N. Tax, N. Sidorova, R. Haakma, and W. M. P. van der Aalst. Mining local process models. *Journal of Innovation in Digital Ecosystems*, 3(2):183–196, 2016. doi:[10.1016/j.jides.2016.11.001](https://doi.org/10.1016/j.jides.2016.11.001).
- [199] N. Tax, N. Sidorova, W. M. P. van der Aalst, and R. Haakma. Heuristic approaches for generating local process models through log projections. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE, 2016. doi:[10.1109/SSCI.2016.7849948](https://doi.org/10.1109/SSCI.2016.7849948).



- [200] N. Tax, I. Verenich, M. La Rosa, and M. Dumas. Predictive business process monitoring with lstm neural networks. In E. Dubois and K. Pohl, editors, *Advanced Information Systems Engineering*, volume 10253 of *Lecture Notes in Computer Science*, pages 477–492. Springer, 2017. doi:[10.1007/978-3-319-59536-8\\_30](https://doi.org/10.1007/978-3-319-59536-8_30).
- [201] N. Tax, B. Dalmas, N. Sidorova, W. M. P. van der Aalst, and S. Norre. Interest-driven discovery of local process models. *Information Systems*, 77:105–117, 2018. doi:[10.1016/j.is.2018.04.006](https://doi.org/10.1016/j.is.2018.04.006).
- [202] N. Tax, X. Lu, N. Sidorova, D. Fahland, and W. M. P. van der Aalst. The imprecisions of precision measures in process mining. *Information Processing Letters*, 135: 1–8, 2018. doi:[10.1016/j.ipl.2018.01.013](https://doi.org/10.1016/j.ipl.2018.01.013).
- [203] F. Taymouri, M. La Rosa, M. Dumas, and F. M. Maggi. Business process variant analysis: Survey and classification. *Knowledge-Based Systems*, 211:106557, 2021. doi:[10.1016/j.knosys.2020.106557](https://doi.org/10.1016/j.knosys.2020.106557).
- [204] W. M. P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 08(01):21–66, 1998. doi:[10.1142/S0218126698000043](https://doi.org/10.1142/S0218126698000043).
- [205] W. M. P. van der Aalst. Formalization and verification of event-driven process chains. *Information and Software Technology*, 41(10):639–650, 1999. doi:[10.1016/S0950-5849\(99\)00016-6](https://doi.org/10.1016/S0950-5849(99)00016-6).
- [206] W. M. P. van der Aalst. On the representational bias in process mining. In *2011 IEEE 20th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 2–7. IEEE, 2011. doi:[10.1109/WETICE.2011.64](https://doi.org/10.1109/WETICE.2011.64).
- [207] W. M. P. van der Aalst. Process mining: discovering and improving spaghetti and lasagna processes. In *2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, pages 1–7. IEEE, 2011. doi:[10.1109/CIDM.2011.6129461](https://doi.org/10.1109/CIDM.2011.6129461).
- [208] W. M. P. van der Aalst. Process mining. *Communications of the ACM*, 55(8): 76–83, 2012. doi:[10.1145/2240236.2240257](https://doi.org/10.1145/2240236.2240257).
- [209] W. M. P. van der Aalst. Process mining: Overview and opportunities. *ACM Transactions on Management Information Systems*, 3(2):1–17, 2012. doi:[10.1145/2229156.2229157](https://doi.org/10.1145/2229156.2229157).
- [210] W. M. P. van der Aalst. Process cubes: Slicing, dicing, rolling up and drilling down event data for process mining. In M. Song, M. T. Wynn, and J. Liu, editors, *Asia Pacific Business Process Management*, volume 159 of *Lecture Notes in Business Information Processing*, pages 1–22. Springer, 2013. doi:[10.1007/978-3-319-02922-1\\_1](https://doi.org/10.1007/978-3-319-02922-1_1).
- [211] W. M. P. van der Aalst. *Process Mining: Data Science in Action*. Springer, 2016. doi:[10.1007/978-3-662-49851-4](https://doi.org/10.1007/978-3-662-49851-4).

- [212] W. M. P. van der Aalst. A practitioner's guide to process mining: Limitations of the directly-follows graph. *Procedia Computer Science*, 164:321–328, 2019. doi:[10.1016/j.procs.2019.12.189](https://doi.org/10.1016/j.procs.2019.12.189).
- [213] W. M. P. van der Aalst. Object-centric process mining: Dealing with divergence and convergence in event data. In P. C. Ölveczky and G. Salaün, editors, *Software Engineering and Formal Methods*, volume 11724 of *Lecture Notes in Computer Science*, pages 3–25. Springer, 2019. doi:[10.1007/978-3-030-30446-1\\_1](https://doi.org/10.1007/978-3-030-30446-1_1).
- [214] W. M. P. van der Aalst. On the pareto principle in process mining, task mining, and robotic process automation. In *Proceedings of the 9th International Conference on Data Science, Technology and Applications*, pages 5–12. SCITEPRESS - Science and Technology Publications, 2020. doi:[10.5220/00099792000500012](https://doi.org/10.5220/00099792000500012).
- [215] W. M. P. van der Aalst. Process mining: A 360 degree overview. In W. M. P. van der Aalst and J. Carmona, editors, *Process Mining Handbook*, volume 448 of *Lecture Notes in Business Information Processing*, pages 3–34. Springer, 2022. doi:[10.1007/978-3-031-08848-3\\_1](https://doi.org/10.1007/978-3-031-08848-3_1).
- [216] W. M. P. van der Aalst. Foundations of process discovery. In W. M. P. van der Aalst and J. Carmona, editors, *Process Mining Handbook*, volume 448 of *Lecture Notes in Business Information Processing*, pages 37–75. Springer, 2022. doi:[10.1007/978-3-031-08848-3\\_2](https://doi.org/10.1007/978-3-031-08848-3_2).
- [217] W. M. P. van der Aalst and A. Berti. Discovering object-centric Petri nets. *Fundamenta Informaticae*, 175(1-4):1–40, 2020. doi:[10.3233/FI-2020-1946](https://doi.org/10.3233/FI-2020-1946).
- [218] W. M. P. van der Aalst and J. Carmona, editors. *Process Mining Handbook*. Lecture Notes in Business Information Processing. Springer, 2022. doi:[10.1007/978-3-031-08848-3](https://doi.org/10.1007/978-3-031-08848-3).
- [219] W. M. P. van der Aalst and L. Santos. May I take your order? In A. Marrella and B. Weber, editors, *Business Process Management Workshops*, volume 436 of *Lecture Notes in Business Information Processing*, pages 99–110. Springer, 2022. doi:[10.1007/978-3-030-94343-1\\_8](https://doi.org/10.1007/978-3-030-94343-1_8).
- [220] W. M. P. van der Aalst and A. ter Hofstede. YAWL: yet another workflow language. *Information Systems*, 30(4):245–275, 2005. doi:[10.1016/j.is.2004.02.002](https://doi.org/10.1016/j.is.2004.02.002).
- [221] W. M. P. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003. doi:[10.1023/A:1022883727209](https://doi.org/10.1023/A:1022883727209).
- [222] W. M. P. van der Aalst, K. M. van Hee, J. M. E. M. van der Werf, and M. Verdonk. Auditing 2.0: Using process mining to support tomorrow's auditor. *Computer*, 43(3):90–93, 2010. doi:[10.1109/MC.2010.61](https://doi.org/10.1109/MC.2010.61).
- [223] W. M. P. van der Aalst, A. Adriansyah, and B. F. van Dongen. Causal nets: A modeling language tailored towards process discovery. In J.-P. Katoen and B. König, editors, *CONCUR 2011 – Concurrency Theory*, volume 6901 of *Lecture Notes in Computer Science*, pages 28–42. Springer, 2011. doi:[10.1007/978-3-642-23217-6\\_3](https://doi.org/10.1007/978-3-642-23217-6_3).

- [224] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, and M. T. Wynn. Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3):333–363, 2011. doi:[10.1007/s00165-010-0161-4](https://doi.org/10.1007/s00165-010-0161-4).
- [225] W. M. P. van der Aalst, A. Adriansyah, A. K. A. de Medeiros, F. Arcieri, T. Baier, T. Blickle, J. C. Bose, P. van den Brand, R. Brandtjen, J. C. A. M. Buijs, A. Burattin, J. Carmona, M. Castellanos, J. Claes, J. Cook, N. Costantini, F. Curbera, E. Damiani, M. de Leoni, P. Delias, B. F. van Dongen, M. Dumas, S. Dustdar, D. Fahland, D. R. Ferreira, W. Gaaloul, F. van Geffen, S. Goel, C. W. Günther, A. Guzzo, P. Harmon, A. ter Hofstede, J. Hoogland, J. E. Ingvaldsen, K. Kato, R. Kuhn, A. Kumar, M. La Rosa, F. Maggi, D. Malerba, R. S. Mans, A. Manuel, M. McCreesh, P. Mello, J. Mendling, M. Montali, H. R. Motahari-Nezhad, M. zur Muehlen, J. Munoz-Gama, L. Pontieri, J. Ribeiro, A. Rozinat, H. Seguel Pérez, R. Seguel Pérez, M. Sepúlveda, J. Sinur, P. Soffer, M. Song, A. Sperduti, G. Stilo, C. Stoel, K. Swenson, M. Talamo, W. Tan, C. Turner, J. Vanthienen, G. Varvaressos, H. M. W. Verbeek, M. Verdonk, R. Vigo, J. Wang, B. Weber, M. Weidlich, T. Weijters, L. Wen, M. Westergaard, and M. T. Wynn. Process mining manifesto. In F. Daniel, K. Barkaoui, and S. Dustdar, editors, *Business Process Management Workshops*, volume 99 of *Lecture Notes in Business Information Processing*, pages 169–194. Springer, 2012. doi:[10.1007/978-3-642-28108-2\\_19](https://doi.org/10.1007/978-3-642-28108-2_19).
- [226] W. M. P. van der Aalst, A. Adriansyah, and B. F. van Dongen. Replaying history on process models for conformance checking and performance analysis. *WIREs Data Mining and Knowledge Discovery*, 2(2):182–192, 2012. doi:[10.1002/widm.1045](https://doi.org/10.1002/widm.1045).
- [227] W. M. P. van der Aalst, J. C. A. M. Buijs, and B. F. van Dongen. Towards improving the representational bias of process mining. In K. Aberer, E. Damiani, and T. Dillon, editors, *Data-Driven Process Discovery and Analysis*, volume 116 of *Lecture Notes in Business Information Processing*, pages 39–54. Springer, 2012. doi:[10.1007/978-3-642-34044-4\\_3](https://doi.org/10.1007/978-3-642-34044-4_3).
- [228] B. F. van Dongen. BPI challenge 2012 – event log. 4TU.Centre for Research Data, 2012. URL <https://doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f>.
- [229] B. F. van Dongen. BPI challenge 2017 – event log. 4TU.Centre for Research Data, 2017. URL <https://doi.org/10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b>.
- [230] B. F. van Dongen. Efficiently computing alignments. In M. Weske, M. Montali, I. Weber, and J. vom Brocke, editors, *Business Process Management*, volume 11080 of *Lecture Notes in Computer Science*, pages 197–214. Springer, 2018. doi:[10.1007/978-3-319-98648-7\\_12](https://doi.org/10.1007/978-3-319-98648-7_12).
- [231] B. F. van Dongen. BPI challenge 2019 – event log. 4TU.Centre for Research Data, 2020. URL <https://doi.org/10.4121/uuid:d06aff4b-79f0-45e6-8ec8-e19730c248f1>.

- [232] B. F. van Dongen. BPI challenge 2020 – event log. 4TU.Centre for Research Data, 2020. URL <https://doi.org/10.4121/uuid:52fb97d4-4588-43c9-9d04-3604d4613b51>.
- [233] B. F. van Dongen and F. Borchert. BPI challenge 2018 – event log. 4TU.Centre for Research Data, 2018. URL <https://doi.org/10.4121/uuid:3301445f-95e8-4ff0-98a4-901f1f204972>.
- [234] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst. The ProM framework: A new era in process mining tool support. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, G. Ciardo, and P. Darondeau, editors, *Applications and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer, 2005. doi:[10.1007/11494744\\_25](https://doi.org/10.1007/11494744_25).
- [235] B. F. van Dongen, A. K. Alves de Medeiros, and L. Wen. Process mining: Overview and outlook of Petri net discovery algorithms. In K. Jensen and W. M. P. van der Aalst, editors, *Transactions on Petri Nets and Other Models of Concurrency II*, volume 5460 of *Lecture Notes in Computer Science*, pages 225–242. Springer, 2009. doi:[10.1007/978-3-642-00899-3\\_13](https://doi.org/10.1007/978-3-642-00899-3_13).
- [236] M. L. van Eck, X. Lu, S. J. J. Leemans, and W. M. P. van der Aalst. PM<sup>2</sup>: A process mining project methodology. In J. Zdravkovic, M. Kirikova, and P. Johannesson, editors, *Advanced Information Systems Engineering*, volume 9097 of *Lecture Notes in Computer Science*, pages 297–313. Springer, 2015. doi:[10.1007/978-3-319-19069-3\\_19](https://doi.org/10.1007/978-3-319-19069-3_19).
- [237] S. J. van Zelst, A. Bolt, M. Hassani, B. F. van Dongen, and W. M. P. van der Aalst. Online conformance checking: relating event streams to process models using prefix-alignments. *International Journal of Data Science and Analytics*, 8(3): 269–284, 2019. doi:[10.1007/s41060-017-0078-6](https://doi.org/10.1007/s41060-017-0078-6).
- [238] S. J. van Zelst, F. Mannhardt, M. de Leoni, and A. Koschmider. Event abstraction in process mining: literature review and taxonomy. *Granular Computing*, 6(3): 719–736, 2021. doi:[10.1007/s41066-020-00226-2](https://doi.org/10.1007/s41066-020-00226-2).
- [239] L. Vanbrabant, N. Martin, K. Ramaekers, and K. Braekers. Quality of input data in emergency department simulations: Framework and assessment techniques. *Simulation Modelling Practice and Theory*, 91:83–101, 2019. doi:[10.1016/j.simpat.2018.12.002](https://doi.org/10.1016/j.simpat.2018.12.002).
- [240] B. Vázquez-Barreiros, M. Mucientes, and M. Lama. Enhancing discovered processes with duplicate tasks. *Information Sciences*, 373:369–387, 2016. doi:[10.1016/j.ins.2016.09.008](https://doi.org/10.1016/j.ins.2016.09.008).
- [241] T. Vogelgesang, J. Ambrosy, D. Becher, R. Seilbeck, J. Geyer-Klingenberg, and M. Klenk. Celonis pql: A query language for process mining. In A. Polyvyanyy, editor, *Process Querying Methods*, pages 377–408. Springer, 2022. doi:[10.1007/978-3-030-92875-9\\_13](https://doi.org/10.1007/978-3-030-92875-9_13).

- [242] J. vom Brocke, A. Simons, B. Niehaves, B. Niehaves, K. Reimer, R. Plattfaut, and A. Cleven. Reconstructing the giant: On the importance of rigour in documenting the literature search process. In *ECIS 2009 Proceedings*. AIS Electronic Library (AISeL), 2009. URL <https://aisel.aisnet.org/ecis2009/161>.
- [243] J. Wang, T. Jin, R. K. Wong, and L. Wen. Querying business process model repositories. *World Wide Web*, 17(3):427–454, 2014. doi:[10.1007/s11280-013-0210-z](https://doi.org/10.1007/s11280-013-0210-z).
- [244] M. Weber and E. Kindler. The petri net markup language. In G. Goos, J. Hartmanis, J. van Leeuwen, H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technology for Communication-Based Systems*, volume 2472 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 2003. doi:[10.1007/978-3-540-40022-6\\_7](https://doi.org/10.1007/978-3-540-40022-6_7).
- [245] H. Xu, J. Pang, X. Yang, L. Ma, H. Mao, and D. Zhao. Applying clinical guidelines to conformance checking for diagnosis and treatment: a case study of ischemic stroke. In *2020 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 2125–2130. IEEE, 2020. doi:[10.1109/BIBM49941.2020.9313532](https://doi.org/10.1109/BIBM49941.2020.9313532).
- [246] B. N. Yahya, H. Bae, S.-o. Sul, and J.-Z. Wu. Process discovery by synthesizing activity proximity and user’s domain knowledge. In M. Song, M. T. Wynn, and J. Liu, editors, *Asia Pacific Business Process Management*, volume 159 of *Lecture Notes in Business Information Processing*, pages 92–105. Springer, 2013. doi:[10.1007/978-3-319-02922-1\\_7](https://doi.org/10.1007/978-3-319-02922-1_7).
- [247] K. Yongsiriwit, N. N. Chan, and W. Gaaloul. Log-based process fragment querying to support process design. In *2015 48th Hawaii International Conference on System Sciences*, pages 4109–4119. IEEE, 2015. doi:[10.1109/HICSS.2015.493](https://doi.org/10.1109/HICSS.2015.493).
- [248] I. Yürek, D. Birant, and K. U. Birant. Interactive process miner: a new approach for process mining. *Turkish Journal of Electrical Engineering and Computer Sciences*, 26(3):1314–1328, 2018. doi:[10.3906/elk-1708-112](https://doi.org/10.3906/elk-1708-112).
- [249] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42:31–60, 2001. doi:[10.1023/A:1007652502315](https://doi.org/10.1023/A:1007652502315).
- [250] R. Zaman, M. Hassani, and B. F. van Dongen. Efficient memory utilization in conformance checking of process event streams. In J. Hong, M. Bures, J. W. Park, and T. Cerny, editors, *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 437–440. ACM, 2022. doi:[10.1145/3477314.3507217](https://doi.org/10.1145/3477314.3507217).
- [251] R. Zaman, M. Hassani, and B. F. van Dongen. Conformance checking of process event streams with constraints on data retention. *Information Systems*, 117:102228, 2023. doi:[10.1016/j.is.2023.102228](https://doi.org/10.1016/j.is.2023.102228).
- [252] F. Zerbato, P. Soffer, and B. Weber. Initial insights into exploratory process mining practices. In A. Polyvyanyy, M. T. Wynn, A. van Looy, and M. Reichert, editors,

- Business Process Management Forum*, volume 427 of *Lecture Notes in Business Information Processing*, pages 145–161. Springer, 2021. doi:[10.1007/978-3-030-85440-9\\_9](https://doi.org/10.1007/978-3-030-85440-9_9).
- [253] Z. Zhang, R. Hildebrant, F. Asgarinejad, N. Venkatasubramanian, and S. Ren. Improving process discovery results by filtering out outliers from event logs with hidden markov models. In *2021 IEEE 23rd Conference on Business Informatics (CBI)*, pages 171–180. IEEE, 2021. doi:[10.1109/CBI52690.2021.00028](https://doi.org/10.1109/CBI52690.2021.00028).
- [254] L. Zimmermann, F. Zerbato, and B. Weber. Process mining challenges perceived by analysts: An interview study. In A. Augusto, A. Gill, D. Bork, S. Nurcan, I. Reinhartz-Berger, and R. Schmidt, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 450 of *Lecture Notes in Business Information Processing*, pages 3–17. Springer, 2022. doi:[10.1007/978-3-031-07475-2\\_1](https://doi.org/10.1007/978-3-031-07475-2_1).

---

# List of Publications

---

## Publications authored by Daniel Schuster that are related to this thesis

- D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Utilizing domain knowledge in data-driven process discovery: A literature review. *Computers in Industry*, 137: 103612, 2022. doi:[10.1016/j.compind.2022.103612](https://doi.org/10.1016/j.compind.2022.103612)
- D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Cortado: A dedicated process mining tool for interactive process discovery. *SoftwareX*, 22:101373, 2023. doi:[10.1016/j.softx.2023.101373](https://doi.org/10.1016/j.softx.2023.101373)
- D. Schuster, F. Zerbato, S. J. van Zelst, and W. M. P. van der Aalst. Defining and visualizing process execution variants from partially ordered event data. *Information Sciences*, 657:119958, 2024. doi:[10.1016/j.ins.2023.119958](https://doi.org/10.1016/j.ins.2023.119958)
- D. Schuster, E. Benevento, D. Aloini, and W. M. P. van der Aalst. Analyzing healthcare processes with incremental process discovery: Practical insights from a real-world application. *Journal of Healthcare Informatics Research*, 2024. doi:[10.1007/s41666-024-00165-6](https://doi.org/10.1007/s41666-024-00165-6)
- D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Cortado—an interactive tool for data-driven process discovery and modeling. In D. Buchs and J. Carmona, editors, *Application and Theory of Petri Nets and Concurrency*, volume 12734 of *Lecture Notes in Computer Science*, pages 465–475. Springer, 2021. doi:[10.1007/978-3-030-76983-3\\_23](https://doi.org/10.1007/978-3-030-76983-3_23)
- D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Incremental discovery of hierarchical process models. In F. Dalpiaz, J. Zdravkovic, and P. Loucopoulos, editors, *Research Challenges in Information Science*, volume 385 of *Lecture Notes in Business Information Processing*, pages 417–433. Springer, 2020. doi:[10.1007/978-3-030-50316-1\\_25](https://doi.org/10.1007/978-3-030-50316-1_25)
- D. Schuster, N. Föcking, S. J. van Zelst, and W. M. P. van der Aalst. Incremental discovery of process models using trace fragments. In C. Di Francescomarino, A. Burattin, C. Janiesch, and S. Sadiq, editors, *Business Process Management*, volume 14159 of *Lecture Notes in Computer Science*, pages 55–73. Springer, 2023. doi:[10.1007/978-3-031-41620-0\\_4](https://doi.org/10.1007/978-3-031-41620-0_4)
- D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Freezing sub-models during incremental process discovery. In A. Ghose, J. Horkoff, V. E. Silva Souza, J. Parsons, and J. Evermann, editors, *Conceptual Modeling*, volume 13011 of *Lecture Notes in Computer Science*, pages 14–24. Springer, 2021. doi:[10.1007/978-3-030-89022-3\\_2](https://doi.org/10.1007/978-3-030-89022-3_2)



- D. Schuster, M. Martini, S. J. van Zelst, and W. M. P. van der Aalst. Control-flow-based querying of process executions from partially ordered event data. In J. Troya, B. Medjahed, M. Piattini, L. Yao, P. Fernández, and A. Ruiz-Cortés, editors, *Service-Oriented Computing*, volume 13740 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2022. doi:[10.1007/978-3-031-20984-0\\_2](https://doi.org/10.1007/978-3-031-20984-0_2)
- D. Schuster, N. Föcking, S. J. van Zelst, and W. M. P. van der Aalst. Conformance checking for trace fragments using infix and postfix alignments. In M. Sellami, P. Ceravolo, H. A. Reijers, W. Gaaloul, and H. Panetto, editors, *Cooperative Information Systems*, volume 13591 of *Lecture Notes in Computer Science*, pages 299–310. Springer, 2022. doi:[10.1007/978-3-031-17834-4\\_18](https://doi.org/10.1007/978-3-031-17834-4_18)
- D. Schuster, E. Domnitsch, S. J. van Zelst, and W. M. P. van der Aalst. A generic trace ordering framework for incremental process discovery. In T. Bouadi, E. Fromont, and E. Hüllermeier, editors, *Advances in Intelligent Data Analysis XX*, volume 13205 of *Lecture Notes in Computer Science*, pages 264–277. Springer, 2022. doi:[10.1007/978-3-031-01333-1\\_21](https://doi.org/10.1007/978-3-031-01333-1_21)
- M. Martini, D. Schuster, and W. M. P. van der Aalst. Mining frequent infix patterns from concurrency-aware process execution variants. *Proceedings of the VLDB Endowment*, 16(10):2666–2678, 2023. doi:[10.14778/3603581.3603603](https://doi.org/10.14778/3603581.3603603)
- D. Schuster, L. Schade, S. J. van Zelst, and W. M. P. van der Aalst. Visualizing trace variants from partially ordered event data. In J. Munoz-Gama and X. Lu, editors, *Process Mining Workshops*, volume 433 of *Lecture Notes in Business Information Processing*, pages 34–46. Springer, 2022. doi:[10.1007/978-3-030-98581-3\\_3](https://doi.org/10.1007/978-3-030-98581-3_3)
- D. Schuster, L. Schade, S. J. van Zelst, and W. M. P. van der Aalst. Temporal performance analysis for block-structured process models in Cortado. In J. de Weerdt and A. Polyvyanyy, editors, *Intelligent Information Systems*, volume 452 of *Lecture Notes in Business Information Processing*, pages 110–119. Springer, 2022. doi:[10.1007/978-3-031-07481-3\\_13](https://doi.org/10.1007/978-3-031-07481-3_13)
- D. Schuster, S. J. van Zelst, and W. M. P. van der Aalst. Alignment approximation for process trees. In S. J. J. Leemans and H. Leopold, editors, *Process Mining Workshops*, volume 406 of *Lecture Notes in Business Information Processing*, pages 247–259. Springer, 2021. doi:[10.1007/978-3-030-72693-5\\_19](https://doi.org/10.1007/978-3-030-72693-5_19)

### Further publications authored by Daniel Schuster

- A. Berti, S. J. van Zelst, and D. Schuster. PM4Py: A process mining library for Python. *Software Impacts*, 17:100556, 2023. doi:[10.1016/j.simpaa.2023.100556](https://doi.org/10.1016/j.simpaa.2023.100556)
- G. Park, D. Schuster, and W. M. P. van der Aalst. Pattern-based action engine: Generating process management actions using temporal patterns of process-centric problems. *Computers in Industry*, 153:104020, 2023. doi:[10.1016/j.compind.2023.104020](https://doi.org/10.1016/j.compind.2023.104020)



- D. Schuster and S. J. van Zelst. Online process monitoring using incremental state-space expansion: An exact algorithm. In D. Fahland, C. Ghidini, J. Becker, and M. Dumas, editors, *Business Process Management*, volume 12168 of *Lecture Notes in Computer Science*, pages 147–164. Springer, 2020. doi:[10.1007/978-3-030-58666-9\\_9](https://doi.org/10.1007/978-3-030-58666-9_9)
- D. Schuster and G. J. Kolhof. Scalable online conformance checking using incremental prefix-alignment computation. In H. Hacid, F. Outay, H.-y. Paik, A. Alloum, M. Petrocchi, M. R. Bouadjenek, A. Beheshti, X. Liu, and A. Maaradji, editors, *Service-Oriented Computing – ICSOC 2020 Workshops*, volume 12632 of *Lecture Notes in Computer Science*, pages 379–394. Springer, 2021. doi:[10.1007/978-3-030-76352-7\\_36](https://doi.org/10.1007/978-3-030-76352-7_36)
- J. N. Adams, D. Schuster, S. Schmitz, G. Schuh, and W. M. P. van der Aalst. Defining cases and variants for object-centric event data. In *2022 4th International Conference on Process Mining (ICPM)*, pages 128–135. IEEE, 2022. doi:[10.1109/ICPM57379.2022.9980730](https://doi.org/10.1109/ICPM57379.2022.9980730)
- G. Lomidze, D. Schuster, C.-Y. Li, and S. J. van Zelst. Enhanced transformation of BPMN models with cancellation features. In J. P. A. Almeida, D. Karastoyanova, G. Guizzardi, M. Montali, F. M. Maggi, and C. M. Fonseca, editors, *Enterprise Design, Operations, and Computing*, volume 13585 of *Lecture Notes in Computer Science*, pages 128–144. Springer, 2022. doi:[10.1007/978-3-031-17604-3\\_8](https://doi.org/10.1007/978-3-031-17604-3_8)
- H. Kourani, D. Schuster, and W. M. P. van der Aalst. Scalable discovery of partially ordered workflow models with formal guarantees. In *2023 5th International Conference on Process Mining (ICPM)*, pages 89–96. IEEE, 2023. doi:[10.1109/ICPM60904.2023.10271941](https://doi.org/10.1109/ICPM60904.2023.10271941)
- A. Berti, D. Schuster, and W. M. P. van der Aalst. Abstractions, scenarios, and prompt definitions for process mining with LLMs: A case study. In J. de Weerd and L. Pufahl, editors, *Business Process Management Workshops*, volume 492 of *Lecture Notes in Business Information Processing*, pages 427–439. Springer, 2024. doi:[10.1007/978-3-031-50974-2\\_32](https://doi.org/10.1007/978-3-031-50974-2_32)
- H. Kourani, A. Berti, D. Schuster, and W. M. P. van der Aalst. Process modeling with large language models. In H. van der Aa, D. Bork, R. Schmidt, and A. Sturm, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 511 of *Lecture Notes in Business Information Processing*, pages 229–244. Springer, 2024. doi:[10.1007/978-3-031-61007-3\\_18](https://doi.org/10.1007/978-3-031-61007-3_18)
- A. Berti, H. Kourani, H. Häfke, C.-Y. Li, and D. Schuster. Evaluating large language models in process mining: Capabilities, benchmarks, and evaluation strategies. In H. van der Aa, D. Bork, R. Schmidt, and A. Sturm, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 511 of *Lecture Notes in Business Information Processing*, pages 13–21. Springer, 2024. doi:[10.1007/978-3-031-61007-3\\_2](https://doi.org/10.1007/978-3-031-61007-3_2)

- E. Serral, D. Schuster, and Y. Bertrand. Supporting users in the continuous evolution of automated routines in their smart spaces. In A. Marrella and B. Weber, editors, *Business Process Management Workshops*, volume 436 of *Lecture Notes in Business Information Processing*, pages 391–402. Springer, 2022. doi:[10.1007/978-3-030-94343-1\\_30](https://doi.org/10.1007/978-3-030-94343-1_30)
- J. N. Adams, G. Park, S. Levich, D. Schuster, and W. M. P. van der Aalst. A framework for extracting and encoding features from object-centric event data. In J. Troya, B. Medjahed, M. Piattini, L. Yao, P. Fernández, and A. Ruiz-Cortés, editors, *Service-Oriented Computing*, volume 13740 of *Lecture Notes in Computer Science*, pages 36–53. Springer, 2022. doi:[10.1007/978-3-031-20984-0\\_3](https://doi.org/10.1007/978-3-031-20984-0_3)
- A. Berti, C.-Y. Li, D. Schuster, and S. J. van Zelst. The process mining toolkit (PMTK): Enabling advanced process mining in an integrated fashion. In *Proceedings of the ICPM Doctoral Consortium and Demo Track 2021*, pages 43–44. CEUR Workshop Proceedings, 2021. URL [https://ceur-ws.org/Vol-3098/demo\\_206.pdf](https://ceur-ws.org/Vol-3098/demo_206.pdf)

---

# Acknowledgment

---

This thesis would not have been possible without the efforts of so many people. Therefore, I want to thank everyone who has supported me along the way.

First, I want to thank the committee, consisting of Wil van der Aalst (1<sup>st</sup> Reviewer), Boudewijn van Dongen (2<sup>nd</sup> Reviewer), Gerhard Lakemeyer (Chairperson), and Horst Lichter (Examiner), for participating in my defense and the evaluation of my Ph.D. thesis with summa cum laude. I am honored to receive your recognition of the originality and quality of my research.

I especially want to thank Wil for his commitment, guidance, and support throughout my entire Ph.D. journey. Your dedication to process mining is genuinely inspiring. I appreciate that you always had an open ear to discuss new ideas, cared about the tiniest definitions and theorems in every paper we co-authored, and supported me wherever possible. Thank you, Wil.

I would also like to thank my daily supervisor, Sebastiaan J. (Bas) van Zelst, for all the guidance and cooperation throughout my PhD. As Wil, you were always eager to discuss ideas, co-authored most of my papers, and supported me in my journey. Thanks, Bas. Many thanks also to Boudewijn, the second reviewer of my Ph.D. thesis.

I want to thank all my further close co-authors with whom I have collaborated over the past years. Many, many thanks to you: Niklas Adams, Davide Aloini, Elisabetta Benevento, Alessandro Berti, Yannis Bertrand, Emanuel Domnitsch, Niklas Föcking, Hannes Häfke, Gero Kolhof, Humam Kourani, Chiao-Yun Li, Giorgi Lomidze, Michael Martini, Gyunam Park, Lukas Schade, Estefanía Serral Asensio, and Francesca Zerbato.

I want to thank all the Bachelor's and Master's students I have supervised over the years who contributed to incremental process discovery with their theses. I would also like to thank all student assistants who contributed to the development of Cortado. Without your efforts, Cortado would not be at its current level. Thanks to: Ahmad Arslan, Lars Dietrich, Emanuel Domnitsch, Niklas Föcking, Edgar Holzmann, Gero Kolhof, Giorgi Lomidze, Michael Martini, Minh-Nghia Phan, Lukas Schade, Ariba Siddiqui, and Weiran Yang. In this context, I would also like to thank the Software Campus program, which supported the development of Cortado with 100,000 euros from the Federal Ministry of Education and Research.

I would like to thank all my colleagues and friends at Fraunhofer FIT and the Chair of Process & Data Science at RWTH Aachen University for a fantastic and unforgettable time as a doctoral student. I have had many remarkable, enjoyable, fun, and exciting experiences with you and made true friends over the years. I will miss you all and always remember our countless moments together.

Last but not least, I would like to thank my family and friends for all their love, understanding, and support throughout my Ph.D. journey.

Thank you all!



---

# Curriculum Vitae

---

## Education

- Master of Science (M. Sc.), **Management, Business and Economics**  
RWTH Aachen University  
Oct 2018 - Feb 2024
- Master of Science (M. Sc.), **Computer Science**  
RWTH Aachen University  
Oct 2016 - Jun 2019
- Bachelor of Science (B. Sc.), **Computer Science**  
RWTH Aachen University  
Oct 2012 - Feb 2016

## Experience

- **Fraunhofer Institute for Applied Information Technology FIT**  
Dept. Data Science & Artificial Intelligence — Process Mining Research Group  
Aug 2019 - Jul 2024
  - **Research Group Lead**  
Apr 2023 - Jul 2024
  - **Research Associate**  
Aug 2019 - Mar 2023
- Ph.D. candidate at **RWTH Aachen University**  
Chair of Process and Data Science  
Aug 2019 - Jul 2024
- Student Assistant at **RWTH Aachen University**  
Chair of Information Management in Mechanical Engineering  
Oct 2017 - Jul 2019
- Internship at **Porsche AG**  
May 2017 - Sep 2017
- Student Assistant at **RWTH Aachen University**  
Chair of Information Management in Mechanical Engineering  
Sep 2016 - Apr 2017
- Internship at **T-Systems International**  
Mar 2016 - Aug 2016