Process Mining with Streaming Data

Sebastiaan J. van Zelst

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

van Zelst, Sebastiaan Johannes

Process Mining with Streaming Data by Sebastiaan J. van Zelst.
Eindhoven: Technische Universiteit Eindhoven, 2019. Proefschrift.

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

ISBN 978-90-386-4699-2

Keywords: Process Mining, Streaming Data

The work in this thesis has been funded by the Netherlands Organization for Scientific Research under the *Desire Lines in Big Data (DeLiBiDa)* project (612.001.351).

SIKS Dissertation Series No. 2019-05. The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Cover design by Remco van Dun.

Printed by Ipskamp Printing B.V., Auke Vleerstraat 145, 7547 PH Enschede, The Netherlands.

Process Mining with Streaming Data

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op
donderdag 14 maart 2019 om 16:00 uur

door

Sebastiaan Johannes van Zelst

geboren te Antwerpen, België

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

voorzitter:	prof.dr.ir. Johan J. Lukkien
1e promotor:	prof.dr.ir. Wil M.P. van der Aalst
2e promotor:	prof.dr.ir. Boudewijn F. van Dongen
leden:	prof.dr. Mark de Berg
	dr. João Gama (University of Porto)
	prof.dr. Mykola Pechenizkiy
	prof.dr. Thomas Seidl (Ludwig Maximilian University of Munich)

Abstract

Modern information systems allow us to track, often in great detail, the execution of processes within companies. Consider for example luggage handling in airports, manufacturing processes of products and goods, or processes related to service provision, all of these processes generate traces of valuable *event data*. Such event data are typically stored in a company's information system and describe the execution of the process at hand. In recent years, the field of *process mining* has emerged. Process mining techniques aim to translate the data captured during the process execution, i.e. the event data, into actionable insights. As such, we identify three main process mining types of analysis, i.e. *process discovery*, *conformance checking* and *process enhancement*. In process discovery, we aim to discover a process model, i.e. a formal behavioural description, which describes the process as captured by the event data. In conformance checking, we aim to assess to what degree the event data is in correspondence with a given reference model, i.e. a model describing how the process ought to be executed. Finally, within process enhancement, the main goal is to improve the view of the process, i.e. by enhancing process models on the basis of facts derived from event data.

Recent developments in information technology allow us to capture data at increasing rates, yielding enormous volumes of data, both in terms of size and velocity. In the context of process mining, this relates to the advent of real-time, online, streams of events that result in data sets that are no longer efficiently analysable by commodity hardware. Such types of data pose both opportunities and challenges. On the one hand, it allows us to get actionable insights into the process, *at the moment it is being executed*. On the other hand, conventional process mining techniques do not allow us to gain these insights, as they are not designed to cope with such a new type of data. As a consequence, new methods, techniques and tools are needed to allow us to apply process mining techniques and analyses on streams of event data of arbitrary size.

In this thesis, we explore, develop and analyse process mining techniques that are able to handle streaming event data. The premise of streaming event data, is the fact that we assume the stream of events under consideration to be of infinite size. As such, efficient techniques to temporarily store and use relevant recent subsets of event data

are needed. The techniques developed in the context of this thesis allow us to apply process mining techniques using potentially unbounded streams of data with arbitrary rates of emission. The ability to handle such data allows us to analyse the underlying process at the exact moment it is being executed. Such analysis paves the way for more advanced types of process mining techniques such as real-time process monitoring and real-time prediction. Since the techniques developed are able to handle data of arbitrary size, as a side-effect, they allow us to handle data sets that are beyond the size-limitation of conventional process mining techniques.

The contributions of this thesis can be categorized into four separate dimensions, all having a strong link with the main branches of process mining.

1. *The development of techniques for data storage and data quality.*

   We provide a general formalization for temporal storage of event data. We furthermore show that we are able to instantiate the proposed formalization using a vast array of existing data storage techniques. As such, we are able to lift all conventional process mining technique to the domain of streaming data. We furthermore present means to filter streaming event data, which allows us to increase the overall data quality considered.

2. *The development of techniques for event stream based process discovery.*

   We explicitly lift process discovery to the domain of event streams by means of designing a general purpose architecture, which describes a two-step discovery approach. The first step consists of constructing an algorithm-specific intermediate data representation, on the basis of the event stream, whereas the second step consists of translating the intermediate representation to a process model. We furthermore show that the proposed architecture covers a wide variety of process discovery algorithms. We additionally show advanced results for the class of language-based region theory-based process discovery algorithms, where we primarily focus on exploiting the intermediate representation of the algorithm to further improve process discovery results.

3. *The development of techniques for event stream based conformance checking.*

   We propose a greedy computational approach for the purpose of computing conformance checking statistics on the basis of event streams. We furthermore prove that the approach is to be seen as an under-estimator, which implies that when a deviation is observed, we are guaranteed that something went wrong during the execution of the process.

4. *The development of techniques for event stream based process enhancement.*

   Next to the control-flow perspective, we also provide support for the resource perspective. In particular, we examine the suitability of a set of existing resource network metrics in the context of event streams.

For each of the aforementioned contributions, a corresponding prototypical implementation is provided in both the process mining tool-kits `ProM` (`http://promtools.`

`org`) and `RapidProM` (`http://rapidprom.org`). As such, each technique discussed has an accompanying publicly available implementation that is used within the corresponding evaluation.

Contents

Abstract **v**

1 Introduction **1**
1.1 Process Mining . . . 3
1.1.1 Process Discovery . . . 5
1.1.2 Conformance Checking . . . 9
1.1.3 Process Enhancement . . . 11
1.1.4 Quality Dimensions in Process Mining . . . 12
1.2 Learning from Data Streams . . . 13
1.3 Streaming Data in the Context of Process Mining . . . 17
1.4 Research Goals and Contributions . . . 19
1.4.1 Research Goals . . . 19
1.4.2 Contributions . . . 20
1.5 Thesis Structure . . . 21

2 Preliminaries **23**
2.1 Basic Mathematical Concepts . . . 25
2.1.1 Sets, Tuples and Functions . . . 25
2.1.2 Multisets . . . 26
2.1.3 Sequences . . . 27
2.1.4 Matrices and Vectors . . . 28
2.1.5 Graphs . . . 29
2.1.6 Integer Linear Programming . . . 31
2.2 Process Models . . . 33
2.2.1 Petri nets . . . 33
2.2.2 Automata . . . 38
2.3 Event Data . . . 40
2.3.1 Event Logs . . . 40
2.3.2 Control-Flow Perspective . . . 42
2.3.3 Event Streams . . . 43

2.4 Process Mining 44
2.4.1 Process Discovery 44
2.4.2 Alignments 46
2.4.3 Computing (Prefix-)Alignments 51
2.4.4 Measuring Quality in Process Mining 52
2.5 Running Example 54

3 Efficient Event Storage: A Primer for General Purpose Process Mining 57
3.1 Introduction 59
3.2 Event Stores 59
3.3 Conventional Storage Techniques 63
3.3.1 Sliding Windows 63
3.3.2 Reservoir Sampling 65
3.3.3 Maintaining Frequent Cases 69
3.3.4 Time Decay 74
3.3.5 Guarantees 75
3.4 Exploiting Behavioural Similarity 75
3.4.1 Maintaining Control-Flow Oriented Prefix-Trees 77
3.4.2 Accounting for Event Removal 79
3.4.3 Reducing the Overall Memory Footprint 80
3.5 Evaluation 81
3.5.1 Data Quality 81
3.5.2 Memory Usage 83
3.6 Related Work 88
3.7 Conclusion 89
3.7.1 Contributions 90
3.7.2 Limitations 90
3.7.3 Open Challenges & Future Work 91

4 Filtering Infrequent Behaviour From Event Streams 93
4.1 Introduction 95
4.2 A Control-Flow-Oriented Taxonomy of Behaviour 96
4.3 Architecture 98
4.4 Automaton Based Filtering 99
4.4.1 Prefix-Based Automata 99
4.4.2 Incrementally Maintaining Collections of Automata 102
4.4.3 Filtering Events 104
4.5 Evaluation 105
4.5.1 Filtering Accuracy and Time Performance 105
4.5.2 Drift Detection Accuracy 111
4.6 Related Work 115
4.7 Conclusion 115
4.7.1 Contributions 115
4.7.2 Limitations 117
4.7.3 Open Challenges & Future Work 118

5 Avoiding Data Redundancy by Learning Intermediate Representations 119
5.1 Introduction 121
5.2 Architecture 122
5.3 Instantiating Intermediate Representation Based Discovery 123
5.3.1 The Directly Follows Abstraction 124
5.3.2 Region Theory 128
5.4 Evaluation 130
5.4.1 Structural Analysis 130
5.4.2 Behavioural Analysis 132
5.4.3 Concept Drift 136
5.4.4 Performance Analysis 139
5.5 Related Work 141
5.6 Conclusion 142
5.6.1 Contributions 142
5.6.2 Limitations 142
5.6.3 Open Challenges & Future Work 144

6 Improving Process Discovery Results by Exploiting Intermediate Representations 145
6.1 Introduction 147
6.2 ILP-Based Process Discovery 148
6.2.1 Regions and Integer Linear Programming 148
6.2.2 Guarantees on Discovering Minimal Regions 150
6.3 Discovering Relaxed Sound Workflow Nets 153
6.3.1 Discovering Multiple Places Based on Causal Relations 154
6.3.2 Discovering Workflow Nets 154
6.4 Dealing with Infrequent Behaviour 159
6.4.1 The Impact of Infrequent Exceptional Behaviour 159
6.4.2 Sequence Encoding Graphs 161
6.4.3 Filtering 164
6.5 Evaluation 165
6.5.1 Model Quality 165
6.5.2 Computation time 170
6.5.3 Application to Real-Life Event Logs 171
6.6 Related Work 175
6.7 Conclusion 175
6.7.1 Contributions 176
6.7.2 Limitations 176
6.7.3 Open Challenges & Future Work 177

7 Online Conformance Checking using Prefix-Alignments 179
7.1 Introduction 181
7.2 Online Prefix-Alignments Computation 182
7.2.1 An Incremental Framework 185
7.2.2 Parametrization 187

7.3 Evaluation 188
7.3.1 Experimental Set-up 189
7.3.2 Results 189
7.3.3 Evaluation using Real Event Data 195
7.4 Related Work 199
7.5 Conclusion 199
7.5.1 Contributions 199
7.5.2 Limitations 200
7.5.3 Open Challenges & Future Work 200

8 Discovering Social Networks **203**
8.1 Introduction 205
8.2 Social Networks 207
8.3 Architecture 208
8.3.1 Networks & Builders 208
8.3.2 Real-Time versus Right-Time Update Strategies 209
8.4 Evaluation 212
8.5 Related Work 214
8.6 Conclusion 217
8.6.1 Contributions 217
8.6.2 Limitations 217
8.6.3 Open Challenges & Future Work 217

9 Implementation **219**
9.1 Introduction 221
9.2 Event Stream Generation 222
9.3 Storage 235
9.4 Filtering 236
9.5 Discovery 238
9.5.1 Intermediate Representations 238
9.5.2 ILP-Based Process Discovery 241
9.6 Conformance Checking 245
9.7 Enhancement 246
9.8 Overview 247
9.9 Conclusion 249
9.9.1 Contributions 249
9.9.2 Limitations 249
9.9.3 Open Challenges & Future Work 249

10 Conclusions **251**
10.1 Contributions 253
10.2 Limitations 254
10.3 Open Challenges 255
10.4 Future Work 256

Bibliography 257

Index 269

Summary 273

Acknowledgements 275

Curriculum Vitae 277

List of Publications 279

SIKS Dissertations 283

Chapter 1

Introduction

Modern information systems allow us to track, often in great detail, the execution of processes within companies. Examples of such processes concern luggage handling in airports, manufacturing processes of products and goods and processes related to service provision. All of these processes generate traces of valuable *event data*. Such event data are typically stored in a company's information system and describe the execution of instances of the process at hand. *Process mining* [4] is a relatively young, purely data-driven, research discipline within computer science that is positioned in-between traditional data mining [15] on the one hand and business process management [51] on the other hand. The main goal of process mining is to gain insights in, and knowledge of, the behaviour of the processes executed within a company. In particular, we aim to attain such knowledge based on the event data that is generated during the execution of the process and stored in a company's underlying information system. As such, the diagrams and process models obtained by the application of process mining, represent, under the assumption that event data is recorded correctly, *what actually happened* during execution of the process.

Within process mining, we distinguish three main types of analysis, i.e. *process discovery*, *conformance checking* and *process enhancement*. These types of analysis mainly differ in the types of input elements they require, as well as their intended analysis result. In process discovery, the main goal is to discover a process model describing the process under study, based on the behaviour captured within the event data. In conformance checking, the main goal is to verify to what degree a given process model (possibly discovered) and the process under study, again as captured by the corresponding event data, conform to one another. In process enhancement, the main goal is to improve the overall view of the process by improving/extending a process model based on facts and figures deduced from behaviour captured within the event data, e.g. by adding performance statistics such as bottleneck information.

Due to the ever increasing performance of computational systems and architectures, data are being generated at increasingly high rates, yielding data that are challenging in terms of *volume*, *velocity*, *variety* and *veracity*. For example, consider the fact that, with the rise of the use of mobile phones and their connection to the internet, virtually everything humans do, and their interaction with other humans and/or machines is being recorded. Moreover, more and more devices, e.g. televisions, washing machines and refrigerators, are being connected to the internet, abundantly generating traces of valuable operational data. This phenomenon, i.e. the generation of massive data sets containing potentially valuable information, is known as *Big Data* [57, 65] and poses several interesting opportunities and challenges, both from an academic- and a practical perspective.

From an operational point of view, the size of a data set that singular commodity hardware is able to process efficiently is limited by a computer's internal memory. In case a data set's size exceeds the available internal memory, costly swap operations need to be performed between internal memory and secondary storage, typically resulting in substandard, or even poor performance. Moreover, in case a data set exceeds secondary storage as well, techniques to efficiently distribute, manipulate and retrieve data across multiple computational entities are needed. Hence, we need tools, techniques and methodologies that are able to cope with data sets that exceed the computational capacity of commodity hardware. Moreover, most existing data analysis techniques are designed under the assumption that the data used for analysis, whether being huge or not, are of a static nature. As such, results obtained by the analysis performed represent a static, historical view on the data. However, there is a variety of application domains in which high-velocity sources of data need to be analysed *on the fly*, i.e. at the moment the data is observed. Such types of analysis require a fundamentally different view on the way we analyse and (temporarily) store the data, as well as the design of the underlying algorithms that we use in order to do so.

The majority of conventional process mining techniques do not explicitly take into account and/or envision that the data set(s) used are of large volume and/or velocity. As a consequence, when applying conventional process mining techniques on such data sets, often no results are obtained. The extreme volume and the velocity at which data are generated pose new requirements on the formalization, specification and design of process mining algorithms. In particular, the techniques need to be able to store and analyse data streams of potentially unbounded size. Hence, they need to either incorporate some mechanism to efficiently store, *and at some point forget*, event data, or, be able to approximate all behaviour observed on the stream as a whole. In this thesis we therefore explore, develop and analyse process mining techniques that are able to handle high-velocity data of arbitrary volume.

Consider Figure 1.1, in which we position the main focus of this thesis and its relation to conventional process mining. A stream of events, i.e. executed activities of the underlying process, is generated while executing the process. In the conventional setting, depicted on the right-hand side of Figure 1.1, this data is stored within the underlying information system. Subsequently, we perform process discovery, conformance checking and/or process enhancement on a *finite subset* of the event data. In this thesis, we primarily focus on the analysis of online streams of events, i.e. at the moment these events are generated. Opposed to conventional process mining, we moreover assume that the stream of events under consideration is unbounded, i.e. infinite, and is potentially of high-velocity. The techniques we present in this thesis therefore explicitly assume that the events published on the stream can only be stored

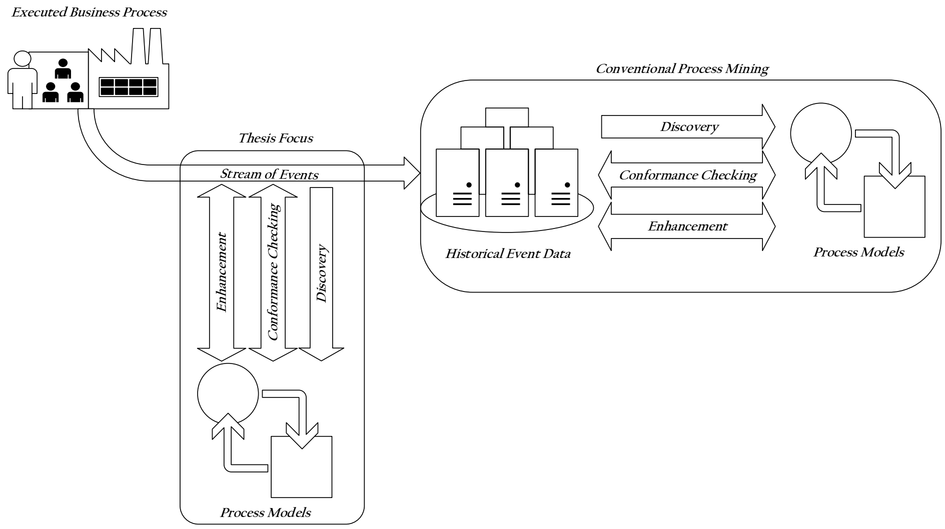


Figure 1.1: The focus of this thesis and its relation to conventional process mining.

temporarily and need to be processed efficiently. In particular, we aim to discover and/or interact with process models directly from the stream of events generated by the execution of the business process, rather than a-posteriori, as is the case in conventional process mining. As such, the techniques developed in the context of this thesis enable us to lift process mining to the domain of streaming data. Moreover, as we are able to generate a data stream out of data sets of arbitrary volume, the techniques presented in this thesis, as a consequence, additionally allow us to analyse data of arbitrary volume.

The remainder of this introductory chapter is organized as follows. In section 1.1, we introduce conventional process mining and its main sub-fields, i.e. process discovery, conformance checking and process enhancement, in more detail. In section 1.2, we introduce the concept of arbitrary data streams, which act as a basic underlying data model for the concepts discussed in this thesis, and define corresponding algorithmic requirements. In section 1.3, we introduce particular challenges of data streams in the context of process mining. In section 1.4, we explicitly quantify the main contributions of this thesis. Finally, in section 1.5, we present an overview of the structure of the remainder of this thesis.

1.1 Process Mining

The field of process mining revolves around the analysis of (business) processes. In the context of this thesis, we consider a *process* to describe a collection of activities,

Table 1.1: Simplified excerpt of a publicly available real-life event log containing events related to the treatment of patients suspected of having sepsis in the emergency department of a Dutch hospital [85].

Event-id	*Patient-id*	*Activity*	*Time-stamp*	*Resource*	*Leucocytes*	*CRP*	*Lactic Acid*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1533	1237	ER Registration	22-10-2014T11:15:41	122A1	∅	∅	∅
1534	1237	Measure Leucocytes	22-10-2014T11:27:00	122B1	9.6	∅	∅
1535	1237	Measure CRP	22-10-2014T11:27:00	122B1	∅	21.0	∅
1536	1237	Measure Lactic Acid	22-10-2014T11:27:00	122B1	∅	∅	2.2
1537	5427	ER Registration	22-10-2014T11:30:23	122A1	∅	∅	∅
1538	5427	Measure Leucocytes	22-10-2014T11:37:00	122C1	7.4	∅	∅
1539	5427	Measure CRP	22-10-2014T11:37:00	122C1	∅	24.2	∅
1540	5427	Measure Lactic Acid	22-10-2014T11:37:00	122C1	∅	∅	3.7
1541	1237	ER Triage	22-10-2014T11:42:12	122A1	∅	∅	∅
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

executed to achieve a certain (business) goal. Such a goal is, for example, the assembly of a product, the provision/rejection of a loan in a bank or the (successful) treatment of patients in a hospital. In particular, there is an associated (partial) order in which these activities are performed, e.g. a product first needs to be assembled after which it is ready to be packaged. The main aim of process mining is to increase the knowledge and understanding of a company's processes by analysing the *event data* generated during the execution of the process. As such, we consider event data as a *first class citizen* in any process mining technique.

The event data generated during the execution of a process is stored in the company's information system. Such data is often in the form of (or, easily translated to) an *event log*. Consider Table 1.1, in which we depict a simplified excerpt of a publicly available real-life event log containing events related to the treatment of patients suspected to have sepsis in a Dutch hospital [85]. The table shows some events recorded in the hospital's information system related to two patients, i.e. the patients identified by patient-id's 1237 and 5427 respectively. In the table, each row corresponds to an *event* that has been recorded in the context of the execution of an *activity* within the corresponding process. For example, the first row shows event 1533 which indicates that on October 22nd of 2014, at 11:15:41, the patient with patient-id 1237 was registered in the ER. The *Activity* column, i.e. describing the *ER Registration* activity (in the first row), refers to the activity that was performed as recorded by the event. In this case, resource *122A1* executed the activity. We observe three additional columns entitled *Leucocytes*, *CRP* and *Lactic Acid*. For event 1533 no value is recorded for these data attributes. However, for event 1534, which relates to the execution of the *Measure Leucocytes* activity, we observe that a value of 9.6 is recorded in the *Leucocytes* column. In this case, the value 9.6 relates to the result of the *Leucocytes Measurement* performed for patient 1237. Note that the three measurement activities are actually recorded at the same time (11:27) in the event log. There are several potential causes for this, e.g. the activities are actually performed at the same time, or, the events have been (manually) recorded at a later phase than their actual execution.

Observe that, after a sequence of events performed for patient 1237, we observe a similar sequence of events for patient 5427. Moreover, after the sequence of events observed for patient 5427, we again observe an event performed for patient 1273, i.e. event 1541. This phenomenon is inherently present in data originating from the execution of business processes, i.e. often multiple instances of the process are executed in parallel. As such, events related to the same patient potentially occur dispersed within the data. Observe that this represents a major difference with respect to conventional data mining techniques and applications. There, each row in a data table, i.e. each *data point*, is often assumed to be relatively independent of all other data points. However, in process mining, we specifically assume that several different data points are having a strong interrelation and are even dependent on each other.

Each execution of a process, e.g. the treatment of a patient in the context of Table 1.1, is referred to as a *process instance*. Within an event log, the events related to a process instance are usually tied together by some sort of identifier, which we refer to as the *case identifier*. In Table 1.1, the case-identifier is the column entitled *Patient-id*. A sequence of events executed in the context of a case identifier is referred to as a *trace* of event data. Observe that it is not uncommon, that within an event log, we are able to define multiple different case identifiers. In the context of medical data, e.g. Table 1.1, we are able to use the patient as a process instance, however, it is also possible to track the behavioural processes of the resources (e.g. doctors) active during the patient treatment process.

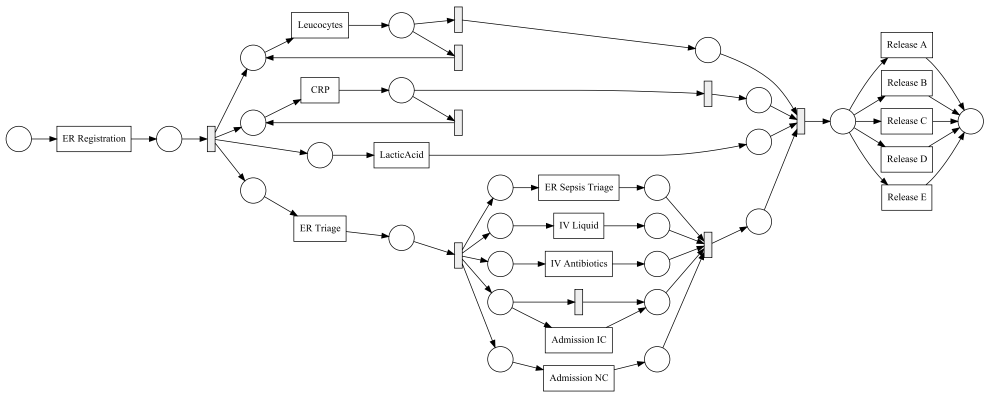
As indicated before, we distinguish three main branches of process mining, i.e. *process discovery*, *conformance checking* and *process enhancement*. In process discovery the main aim is to discover a process model that accurately describes the underlying process, based on the data observed in the event log. In conformance checking, the main aim is to assess to what degree a given process model, potentially discovered, and the event log conform to one another. Finally, in process enhancement, the main aim is to improve the view of the process, for example by applying process discovery, conformance checking, performance analysis and/or simulation techniques.

In the remainder of this section, we discuss the three main branches of process mining in more detail. We finish the section with a global discussion on typical process model quality criteria considered in process mining.

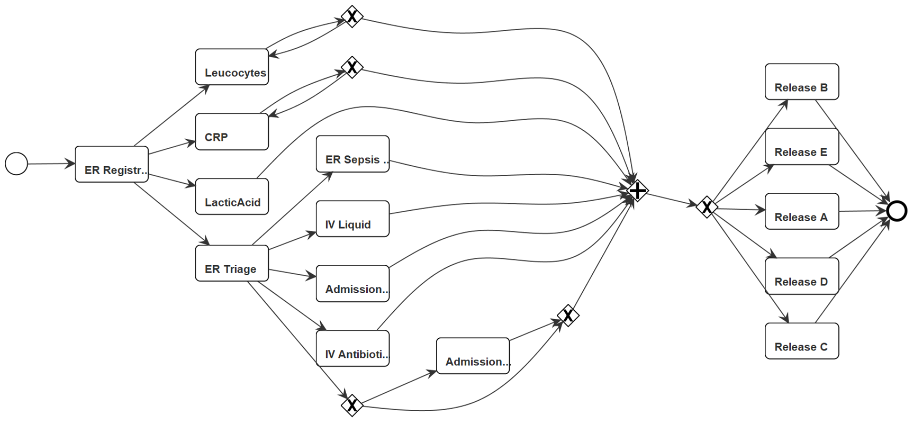
1.1.1 Process Discovery

The main aim of process discovery is to *discover* a process model which accurately describes the process as captured by an event log. We typically use an event log, extracted from the company's information system's database and try to discover a corresponding process model. For example, consider the two models depicted in Figure 1.2, on page 6, which are both discovered by applying a state-of-the-art process discovery algorithm [78] on the full event log related to the excerpt as presented in Table 1.1.

The models describe that the *Leucocytes*, *CRP*, *LacticAcid* activities, together with the *ER Triage* activity are executed in parallel. Furthermore, they indicate that it is possible to execute the *CRP* and *Leucocytes* activities more than once. The two modelling formalisms shown in Figure 1.2 are a Petri net [94], cf. Figure 1.2a, and a



(a) An example *Petri net* [94]-based model.



(b) An example BPMN [96]-based model.

Figure 1.2: Two example process models, in different process modelling notations, derived from the full event log related to the excerpt as presented in Table 1.1.

BPMN [96][1]-based model, cf. Figure 1.2b, respectively. Even though the two process modelling formalisms are different, the models describe the same behaviour. Petri nets are most commonly used within (process mining) research, whereas BPMN models are mostly used in business/industry for process modelling and documentation. There are several reasons for the use of Petri nets within research, e.g. the models are relatively easily translatable to more high-level business oriented process modelling formalisms such as BPMN, EPCs [90] and/or commercial vendor-specific modelling formalisms. Furthermore, due to their non-ambiguous formal nature, specifically when compared to industry oriented-standards, there is a vast body of literature covering formal foundational aspects and properties of Petri nets. Moreover, numerous analysis techniques exist for Petri nets as well. As indicated, modelling formalisms such as BPMN and EPCs are more often used in business, as their focus is more towards human understandability and interpretability rather than formal correctness. In the light of the aforementioned, in this thesis, we solely consider Petri nets, cf. section 2.2, and specific sub-classes thereof, as a main process modelling formalism.

A wide variety of process discovery techniques has been developed and studied in recent years [4, Chapters 6&7] [18, 49, 120]. Despite the numerous research efforts in the domain of process discovery, the task has proven to be far from trivial and, arguably, a definite process discovery algorithm that accurately discovers a process model based on arbitrary event data does not exist yet. One of the most challenging aspects of process discovery is the presence of *parallel behaviour* in most processes. In case we are able to execute two activities in parallel, we are able to execute the activities in any order, and moreover, their execution potentially overlaps. Even if we abstract from activity duration, the impact of parallelism on the behavioural variety of a process is enormous. The number of ways in which we are able to schedule parallel activities is factorial. Thus, if we have 3 possible activities in parallel, there are $3! = 3 \times 2 \times 1 = 6$ ways to arrange them. However, in case we have 10 possible activities in parallel, there are $10! = 10 \times 9 \cdots \times 1 = 3,628,800$ possible ways to arrange them.

For most existing process discovery algorithms, which are typically explicitly designed to cope with event data describing behaviour with underlying parallelism, the most prominent challenge is not necessarily the large variety caused by parallelism. The most challenging aspect, refers to the fact that the huge variety of possible behaviour in a process is often not fully represented within the event log. Namely, it is very common that the variety of behaviour is not uniformly distributed, in terms of the execution of a process. It is far more likely, that the variety of behaviour for example follows a Pareto distribution, cf. Figure 1.3, with a long tail. As an example, consider a bank providing loans to their (prospective) clients. A majority of the clients is likely to ask for a similar loan in terms of the amount, e.g. less than \$1.000.000, and therefore, for these clients, the bank roughly executes the same process in order to determine whether the client is eligible to obtain the loan or not. In some cases, due to parallelism, certain checks and/or activities are executed in a different order, yet in general, the vast majority of clients follows the same procedure, and hence, for multiple different clients the same sequence of activities is executed. Clients needing

[1]`https://www.omg.org/spec/BPMN/2.0/`

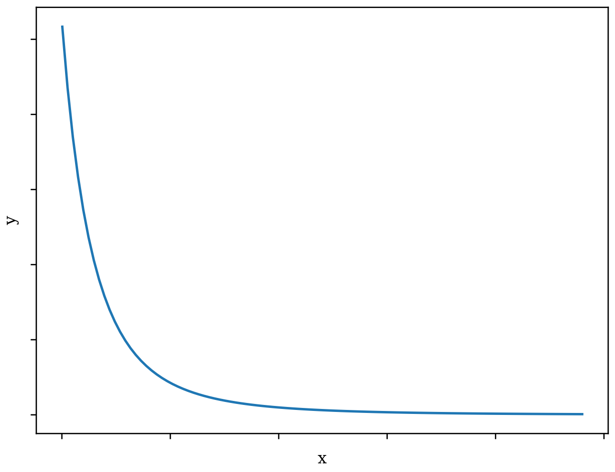


Figure 1.3: Example plot indicating the typical shape of a Pareto distribution. Event data captured during the execution of a process typically follows such a distribution, i.e. we observe a large share of similar behaviour and a “long tail” of relatively infrequently occurring behaviour.

a loan of higher amount are likely to be treated differently, and hence, alternative checks and/or activities are executed for them. The execution of the process for these different types of clients also belongs to the overall process executed by the bank to determine loan eligibility of their clients. Nonetheless, due to the relatively low amount of these types of customers, it is very unlikely that an event log contains all possible executions of the process for this type of clients.

Even with the advent of Big Data, which largely motivates this thesis, we are in no way guaranteed that all possible behaviour is witnessed within an event log and/or event stream. As indicated, this is due to the fact that it is common that several process instances share a great deal of similar behaviour and thus, border cases, which are theoretically possible within the process, are missed. As such, the event log typically represents a (marginal) fraction of the total variety of possible behaviour, which hampers the discovery algorithm to properly discover parallel constructs. Additionally, most process discovery algorithms are designed upon the presumption that the input event data actually describes a clear underlying process structure. However, data originating from existing information systems are often not of this form. There are multiple reasons for this phenomenon, e.g. data quality issues due to erroneous logging, concept drift, process flexibility and process deviation.

In summary, process discovery strives to discover a representation of a process, as described by the behaviour captured within an event log. This seemingly straightforward task has proven to be inherently complex and challenging. Specifically incompleteness and impurity of the event data hamper most existing process discovery algorithms, when applied on real event data.

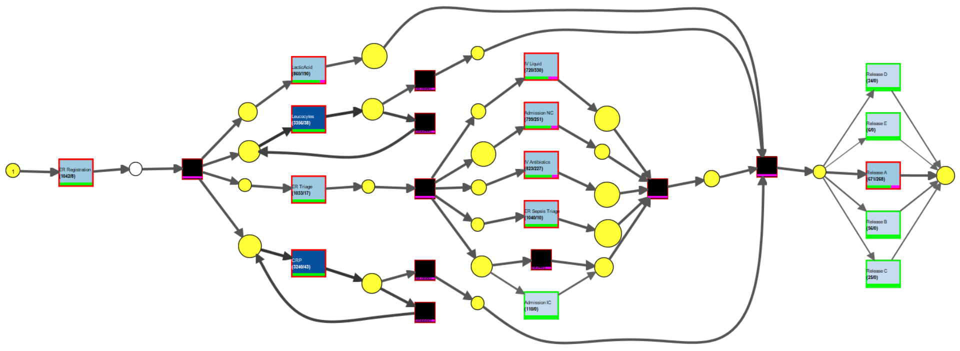
Process discovery in the context of this thesis Within this thesis we partially focus on the aforementioned challenges in process discovery, i.e. data impurity in particular, cf. chapter 4, yet the main aim is to lift process discovery, in general, to the streaming domain. In chapter 5, we present a general framework for the purpose of process discovery on the basis of event streams. The emphasis of the framework is on a common two-step characterization of process discovery algorithms which we use as a basis for online process discovery. We focus on learning intermediate data structures used by different algorithms in the context of event streams, which we subsequently translate to process models. Furthermore, in chapter 6, we show how to improve process discovery results by means of exploiting the internal intermediate data structure used by a state-of-the-art process discovery algorithm. In particular, we show that we are able to guarantee both structural and behavioural properties of the discovered process models. Furthermore, we tackle the problem of outlier behaviour caused by noise/and or rare behaviour by means of filtering directly on top of the aforementioned internal intermediate representation.

1.1.2 Conformance Checking

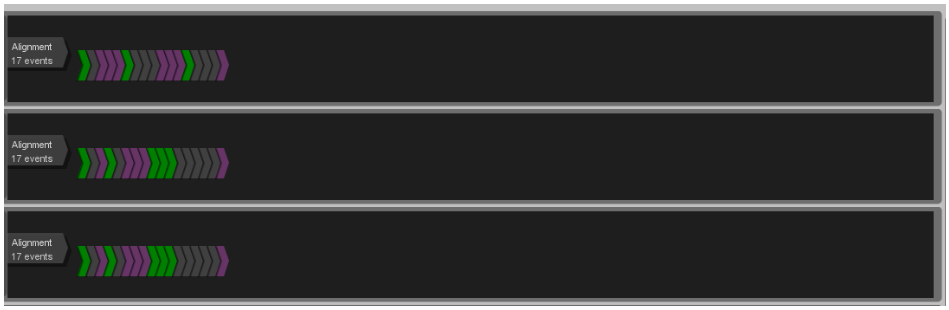
The main aim of conformance checking is to assess to what degree a given process model, supposed to describe the underlying process, and the behaviour as captured within an event log conform to one-another. Within conformance checking, we typically use both an event log and a process model as an input and compute their conformance. In some cases, these conformance statistics are projected back onto the model and/or the event log.

For example, consider Figure 1.4, in which we show the conformance checking results for the process model shown in Figure 1.2a and the corresponding event log.[2] In Figure 1.4a, we show a projection of the conformance checking results onto the process model. The size and the yellow colour of the places in the Petri net (visualized as circles) indicates that in some cases, an activity was performed according to the event data that was not possible in the state of the model, as represented by these places. Similarly, the intensity of the blue colour of the transitions (visualized as rectangles), indicates how often these activities were performed according the data. Moreover, the green/purple bars indicate how often the execution of an activity in the event data is aligned with an activity described by the model. In Figure 1.4b, we depict the conformance checking results, projected onto the event log. Each row in the figure represents the execution of an instance of the process. The chevrons in the figure, either green, grey or purple, correspond to the execution of an activity in the context of that case. The colour indicates whether or not it is possible to map the observed

[2]Note that the event log was explicitly filtered in order to obtain the models in Figure 1.2 and it therefore does not accurately describe all the cases present in the input event log.



(a) Projection onto the event log (trace-by-trace).



(b) Projection onto the event log (trace-by-trace).

Figure 1.4: Example visualizations of typical conformance checking results, as implemented in the ProM [113] framework.

execution, i.e. as present in the event log, to an activity execution in the given process model.

Early work in conformance checking literally *replayed* the process behaviour as captured within the event log in terms of the process model [102]. More recently, the concept of *alignments* was introduced [13], which quickly developed to the de-facto standard in conformance checking.[3] Essentially, an alignment explains the observed behaviour in the event log in terms of the model as good as possible. As such, alignments minimize the number of possible mismatches that can be found between the traces in an event log and the process model. Alignments conceptually resemble the replay techniques as mentioned earlier, however, by their sheer definition, they lead to less ambiguous results.

Observe that the data quality issues, mentioned briefly in subsection 1.1.1, i.e. representing one of the main challenges in process discovery, are also present in conformance checking, yet play a less significant role. In fact, conformance checking

[3]The results depicted in Figure 1.4 are based on alignments.

techniques allow us to track and/or find such problems, be it in terms of a predefined model. However, the danger exists that logging errors are falsely assumed to be erroneous executions of the process.

Conformance checking in the context of this thesis In this thesis, in chapter 7, we present means for online conformance checking, on the basis of the incremental computation of *prefix alignments*. Within this contribution, the main emphasis is on the computability of such prefix-alignments and their potential impact when used as an approximation scheme on the quality of the final conformance checking result.

1.1.3 Process Enhancement

Within process enhancement, the main aim is to improve the overall view of the process under study, based on event data recorded in an event log. Here, a process model, either discovered or designed by hand is combined with event data in order to obtain a better view of the process. Such a better view can for example be a repaired process model [44, 56] or the visualization of data-based decision points within the process model [80].

Not all work in process enhancement focusses on revising a given/discovered process model. Other approaches in process enhancement oriented studies focus on a particular category of data present in event logs. For example, a significant amount of work is devoted to the behaviour and interaction of resources within the process. In [9, 107] several techniques are presented to discover social networks of interacting resources as well as the automated identification of resource groups.

For example, consider Figure 1.5, in which we show an example social network, based on the complete event log corresponding to the excerpt in Table 1.1. The network represents a subcontracting network. The nodes/vertices, i.e. visualized as circles, are representing organizational units that are active within the process, i.e. its members perform certain (medical) activities. Whenever an arc exists between two organizational units, there is significant evidence in the underlying event data that subcontracting has taken place between the two units. Here, subcontracting relates to the situation where a certain unit hands over control of the process to another unit, i.e. by means of the execution of process activities, after which control is retained by the first unit. For example, there exists an arc between the node labelled with N, and the node labelled with Y, implying that at some point in the process, organisational unit N is subcontracting organisational unit Y. Note that, for example, resource X never participates in such subcontracting relation.

Process enhancement in the context of this thesis In this thesis, in chapter 8, we present a study towards computing social networks, i.e. networks of resources cooperating together during process execution, in the context of event streams. We show that there exist variations of such networks that lend themselves for incremental computation, which in turn allows us to adopt these variations in an online, event stream setting.

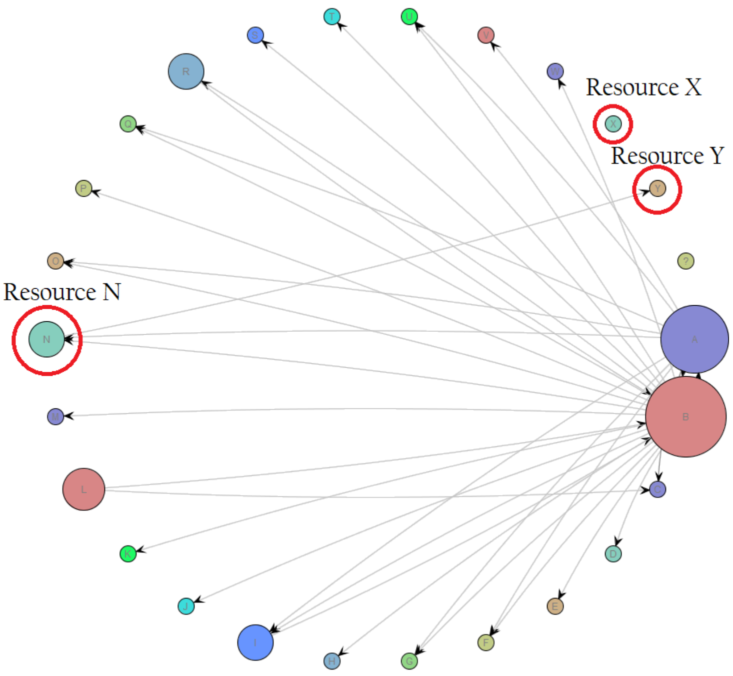


Figure 1.5: Example of a *social network* [9, 107], based on *subcontracting* behaviour as recorded within the event log of Table 1.1.

1.1.4 Quality Dimensions in Process Mining

Process models used within process mining, either discovered or designed by a process owner/expert, ideally describe the potentially complex behaviour of the process in a concise and compact manner. To what degree such model is *accurate*, is not easily captured in one definite quality metric. A given process model might describe all behaviour as captured within the event log, yet at the same time, it describes an abundance of behaviour not present in the event log. The opposite is possible as well, i.e. the model does not describe too much additional behaviour, yet, only covers a fraction of the behaviour observed in the event log.

Even if we discover or design a process model that strikes an adequate balance between coverage of the observed behaviour (*replay-fitness*) and additionally described behaviour (*precision*), it is probable that the model is too complex and incomprehensible for a human analyst. Therefore, a process model additionally needs to be as *simple* as possible, yet not too simple (Occam's razor). Moreover, the model should also be able to *generalize*, and allow for unseen, yet likely, behaviour.

In the light of the aforementioned considerations, within process mining, four quality dimensions are commonly considered that allow us to quantify the quality of a given process model with respect to the underlying process, as represented by the event log.

- *Replay-fitness*

  Quantifies to what degree a given process model describes the behaviour as captured in an event log. In case all behaviour present in the event log is also described by the process model, *replay-fitness is perfect*.

- *Precision*

  Quantifies to what degree a given process model describes behaviour that is not observed in the event log. In case the model does not describe any additional behaviour, i.e. all behaviour described by the model is also in the event log, *precision is perfect*.

- *Generalization*

  Quantifies to what degree a given process model generalizes beyond the behaviour observed in the event log. A model generalizes well in case it describes certain constructs, e.g. parallelism and/or looping behaviour, that allow us to deduce behaviour that is likely to be part of the process, yet, not necessarily observed in the event log.

- *Simplicity*

  Quantifies to what degree a given process model is interpretable by a human analyst. Ideally, a model is of such simplicity that removing more constructs than present in the model, jeopardizes the model quality, i.e. in terms of the other dimensions.

Ideally, a process model strikes an adequate balance between the four quality dimensions presented. In this regard, such adequate balance is somewhat subjective, as it to some degree depends on the context in which the model is used. However, maximizing all of the quality dimensions is often hard, or even impossible. In case we obtain a model with perfect replay-fitness and precision, it is likely that the model fails to generalize. Similarly, if we obtain a properly generalizing process model, it is likely that the precision of the model is not perfect.

1.2 Learning from Data Streams

In this thesis, we primarily focus on the application of process mining techniques in the context of streaming data, i.e. infinite streams of data elements that potentially arrive at unprecedented velocity. Such streaming data, as well as its corresponding formal model, are well-studied concepts in the field of data mining. Based on [64, 95], we define the *general data stream model* as follows.

We receive a sequence of input data elements $d_1, d_2, ..., d_i, ...$, that describe some, typically unknown, underlying generating function D. In general, two types of streaming models are recognized, i.e.

- *Insertion-Only/Cash Register*

  Once a data item d_i is received, it is not changed, i.e. it remains as-is.

- *Insert-Delete/Turnstile*

  After a data item d_i is received, it is potentially removed and/or updated.

Assume we maintain a bucket of marbles of all kinds of colours. Moreover, assume that we have a data stream in which the i^{th} data packet d_i describes the colour of a new marble that we put in the bucket. Let $D_i(\texttt{colour})$ denote the number of marbles of colour `colour` stored in the bucket, after receiving the i-th data packet. For example, $D_{120}(\texttt{blue})$ describes the number of blue-coloured marbles in the bucket after receiving the first 120 data packets. In the Insertion-Only model, once we observe a marble of a certain colour at time i, i.e. we observe d_i, the colour of the marble remains the same, i.e. it never changes. In case we adopt the Insert-Delete model, we are able to alter the colour of a marble that we previously observed on the stream. Moreover, we are even able to remove the marble of that colour at a later point in time from the bucket.

In general, the aim of any streaming data application and/or algorithm is to design a function on top of the underlying function D. Observe however that the stream is the only tangible representation of D. Such function is often a query related to the data structure, and, due to the nature of data streams, the answer to such query is likely to change over the course of time. In the context of our example, we are for example interested to keep track of the number of blue marbles, or, we want to know what are the most frequent colours of marbles that we observe on the stream. We assume to receive an infinite number of marbles of different colours, therefore, if we just keep on throwing the marbles we receive in the bucket, at some point, our bucket of marbles gets full and starts to overflow. Moreover, we assume that the marbles arrive on a relatively fast rate, and thus, we need a mechanism to efficiently place a newly arrived marble (temporarily) in our bucket and asses its colour.

In line with the aforementioned analogy, in terms of performance, we observe three main components in the data stream model:

- *Processing time*

  The time required to process a newly arrived data item. As we typically construct some representation of the underlying function D (alternatively referred to as $\hat{D}$), the item processing time can alternatively be seen as the time needed to update the internal representation after receiving a new data item. In the context of the aforementioned example, the processing time represents the time of placing a marble in (the internal representation of) the bucket.

- *Memory consumption*

  The memory consumption represents the amount of memory needed to store the internal representation of the underlying function D. In general, as we assume the stream to be potentially infinite, simply storing each each data item on the stream leads to infinite space consumption. In the context of the aforementioned example, the space consumption represents the amount of memory needed to describe (the internal representation of) the bucket.

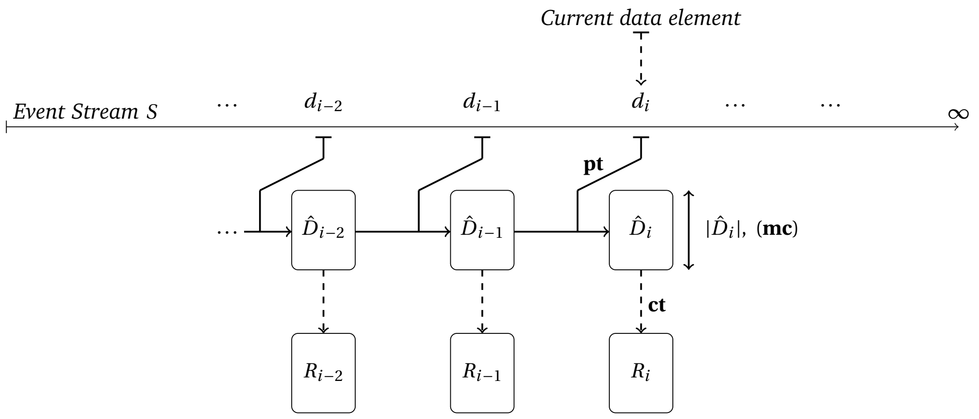


Figure 1.6: Schematic overview of the main performance dimensions of handling streaming data. Processing time (**pt**) concerns processing the new data element in the previous internal representation of the underling generating function D, i.e. $\hat{D}$, in order to generate an updated, new internal representation. Memory consumption (**mc**) represents the amount of memory needed to store the representation of the underlying generating function D, i.e. $\hat{D}$. Computation time (**ct**) represents the amount of time needed to translate the estimation of the underlying function to the desired result, i.e. translating $\hat{D}$ into R.

- *Computation time*

  Often, the function D is used to compute a derived function and/or object, that uses function D (more particularly its approximation $\hat{D}$) as a basic input. The computation time represents the time needed to compute the intended function of the algorithm on top of the underlying function D. In the context of the aforementioned example, assume that we aim to predict the number of blue marbles in the upcoming 500 marbles. The computation time represents the amount of time we need in order to translate the (internal representation of) the bucket in to such a prediction.

Consider Figure 1.6, in which we present a schematic overview of the different performance components of the data stream model. We maintain some representation of the underlying generating function D, represented by $\hat{D}$. We update the internal representation based on newly received events. As such, we have a sequence of representations of D, i.e. $\hat{D}_1$, $\hat{D}_2$, ..., $\hat{D}_i$, ... The time to incorporate the newly received data item, i.e. d_i, in order to transform $\hat{D}_{i-1}$ into $\hat{D}_i$, is referred to as *processing time*. The *memory consumption* represents the amount of physical memory required to store the maintained internal representation of the data stream. Finally, as we are often interested in some derivative result, computed on the basis of the current representation of the underlying function, we refer to *computation time* as the time needed to transform the current representation of the underlying function into such

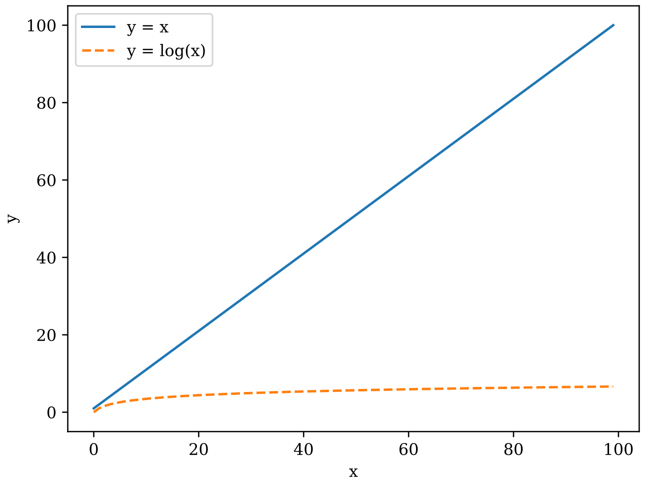


Figure 1.7: Two functions, i.e. $y{=}x$ and $y{=}\log_2(x)$, indicating the difference of rate-of-growth of linear functions versus logarithmic functions.

derivative.

Since we assume the input event stream to be potentially of high-velocity, we are required to limit the processing time. At the same time, we assume the amount of data items on the data stream to be potentially infinite. Let N denote the, potentially infinite, size of the data stream. A streaming algorithm ideally constitutes, at any point in time, to a processing time and space consumption that are simultaneously strictly less than linear in the size of N. Preferably, the simultaneous cost of processing time and space consumption is *polylogarithmic*, i.e. $O(\log^k(N))$. The implication of this bound on the processing time/space consumption is the fact that the algorithm's complexity growth is less than the rate at which packets arrive. For example, consider Figure 1.7, in which we plot functions $y = x$ and $y = \log_2(x)$ for $1{\leq}x{\leq}100$, which clearly illustrates the difference of complexity growth of logartihmic and linear functions. Hence, even after receiving a huge amount of events on the stream, we are guaranteed that memory usage is orders of magnitude smaller.

The cost of maintaining a representation of an infinite amount of data in ideally polylogarithmic time and/or space, often impacts the resulting function that the algorithm implements. In general, the function computed by the algorithm is an *approximation* of the actual function when applied on the data stream as observed thus-far. Therefore, a vast majority of the existing data stream algorithms implements the (ϵ,δ)-*approximation* scheme, i.e. the algorithm's result is correct within $1 \pm \epsilon$ of the actual result with a probability of at least $1-\delta$. Often, the complexity of algorithms implementing either one of the presented approximation schemes is specified in terms of ϵ and δ, e.g. $O\left(\frac{1}{\epsilon^2}\log(\frac{1}{\delta})\right)$. Hence, when we reduce the error margins of these algorithms, the algorithmic complexity increases.

1.3 Streaming Data in the Context of Process Mining

In the context of this thesis, we assume the notion of an *event stream*. As such, the data elements arriving on the stream refer to events, executed in the context of an underlying (business) process. Hence, a data element d_i represents an event, i.e. a row in Table 1.1, and the process itself represents the underlying function D. Events executed in context of a (business) process are in general executed once, and are irrevocable. Hence, we can assume the *Insertion-Only/Cash Register* streaming data model. Moreover, in the context of this thesis we assume that these events are executed in an *atomic* fashion, i.e. we abstract from activity duration.

When considering events originating from the execution of a process, in light of the general streaming data model, we identify the following non-trivial challenges:

- *Interrelated Data Elements*

  As the data elements present on the event stream originate from the execution of an underlying process, i.e. they relate to *events*, multiple events/data elements relate to the same process instance. Due to the inherent parallel execution of several process instances, e.g. multiple products are produced at the same time, batch processing is applied etc., we expect events related to the same process instance to arrive in a dispersed manner on the event stream. The fact that multiple data elements are interrelated adds an additional layer of complexity in terms of event storage, i.e. it is not possible to arbitrarily discard certain data elements from the data structures maintained.

- *Data Incompleteness*

  As a consequence of the interrelatedness of the different events that are emitted onto the stream, we identify the notion of *data incompleteness*. First of all, when we start observing a stream of events, it is likely that some process instances are already ongoing. Hence, the first few events observed are likely to relate to process instances of which the initial set of events has not been observed. Secondly, for newly started process instances, we observe the behaviour as a whole over the life-span of the corresponding process instance. Thus, whilst the process instance is still ongoing, we only possess partial behaviour of that specific process instance.

- *Implicit Start/End*

  Apart from data incompleteness, we assume that we have no explicit knowledge of process instance initialization and termination. As such, it is not clear when a process instance started and when it is safe to assume its termination. This implies that, when we observe an event on the event stream related to a case identifier that was not observed before, we have no guarantee that such an event is indeed the first event of the underlying process instance. Moreover,

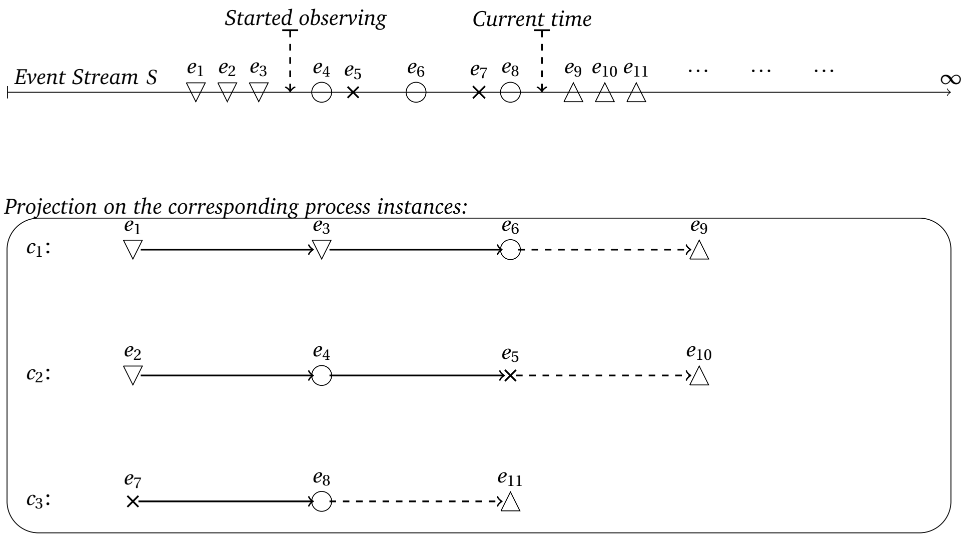

Started observing
Current time
Event Stream S
e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11
∞
Projection on the corresponding process instances:
c1:
c2:
c3:

Figure 1.8: Schematic overview of the challenges of handling streaming data in the context of process mining. A ▽-symbol indicates an unobserved event, i.e. executed prior to observing the stream, a ○-symbol indicates a stored event, a ×-symbol indicates a dropped event, i.e. it is not stored, and, a △-symbol indicates a future event.

since we have no explicit knowledge related to process instance termination, we potentially remove events related to such an instance, prior to its termination. As we are likely to receive future behaviour for such a process instance, we may falsely assume that newly arriving behaviour relates to the start of the process instance.

- *Noise*

  Most existing data stream algorithms are designed under the assumption that the input data is free of noise, or, noise plays a less significant role. For example, techniques allow us to find the most frequent items on the stream simply do not report noise as the noise is likely to be infrequent. Within process mining however, during the execution of the process, events are generated that are potentially spurious, i.e. they did not actually happen. Moreover, some events that are executed are potentially never received on the event stream, e.g. due to network issues. Translating the raw data directly into a process model therefore potentially leads to results of inferior quality.

Consider Figure 1.8, in which we present a schematic overview of (some of) the different challenges of adopting streaming data in the context of process mining. We start observing the stream at some point in time, which does not allow us to observe the full behaviour of each process instance, i.e. some instances are already running (represented by the ▽-symbol). For some events, given some strategy, we decide to store them in our internal data structure, whereas for other events, we decide to ignore them (represented by the ○-symbol and the ×-symbol respectively). Since we consider running instances, some events will only become available in the future, i.e. as represented by the △-symbol.

In section 1.4, we characterize the research goals and main contributions of this thesis, which are partially based on/related to the aforementioned challenges. However, the majority of these challenges relates to the design of (temporal) data storage, i.e. what items do we store and for what period in time, which is specifically covered in one of the research goals.

1.4 Research Goals and Contributions

The main goal of the work performed in the context of this thesis is to enable process mining techniques to deal with streaming data. As such, we develop algorithms, methods, tools and techniques that explicitly take into account the requirements as defined by the general data stream model, cf. section 1.2.

1.4.1 Research Goals

We formalize the main research goals of this thesis along the lines of the main components of process mining, i.e. *process discovery*, *conformance checking* and *process enhancement*. Additionally, we focus on efficient storage of event data. We characterize the main research goals as follows.

- ***Research Goal I***

  Development/design of general purpose techniques for high-quality, efficient event data storage.

  We develop efficient techniques for the storage of events originating from event streams. These techniques act as a primer for any process mining technique. We moreover develop techniques that allow us to increase the overall quality of the data considered in subsequent process mining analyses.

- ***Research Goal II***

  Development/design of specialized techniques for efficient, event stream based, process discovery.

  We develop techniques that go beyond efficient event storage and decrease the memory consumption need for the purpose of process discovery.

- ***Research Goal III***

  Development/design of specialized techniques for efficient, event stream based, conformance checking.

  State-of-the-art conformance checking techniques are inherently complex, i.e. they are of a combinatorial nature. We, therefore, develop techniques that allow us to compute and/or approximate conformance checking results in an event stream based context.

- ***Research Goal IV***

  Development/design of specialized techniques for efficient, event stream based, process enhancement.

  We develop and investigate the application of process enhancement techniques in a streaming context, i.e. by explicitly taking the requirements dictated by the streaming data model into account.

1.4.2 Contributions

In line with the defined research goals, we present the main contributions of this thesis here.

- *Data Engineering*

  In chapter 3, we present and formalize the notion of an *event store*. In essence, an event store represents a finite view of the event stream, i.e. a subsequence of the stream as a whole. As such, we are able to apply any conventional process mining algorithm in the context of event streams. We primarily focus on showing that we are able to instantiate event stores using a variety of existing data storage techniques. Moreover, we assess to what degree these storage techniques are suitable for the purpose of process mining.

  In chapter 4, we present means to filter *infrequent behaviour* from event streams. The technique presented in chapter 4 acts as a *stream processor*, i.e. both its input and output are a stream of events. As such, we are able to apply it prior to constructing event stores, in order to achieve higher quality event data.

- *Process discovery*

  In chapter 5, we present a general framework for the purpose of online process discovery. Conceptually, the framework describes a high-level architecture on the basis of the internal data structures used by the most common process discovery algorithms. The main aim is to design these algorithms in such a way that we require a minimal memory footprint, i.e. we store the least amount of data needed to reconstruct the algorithm's internal data structure. Moreover, we show that the proposed architecture covers several different classes of process discovery algorithms, and, we provide several instantiations of the framework.

  In chapter 6, we show that, for a specific class of process discovery algorithms, we are able to exploit the internal data structure to such extent that this allows us to guarantee both structural- and behavioural properties of the discovered process models. We moreover present an internal filtering method, built on top of the algorithm's data structure, that allows us to increase the overall quality of the discovered process models.

- *Conformance checking*

  In chapter 7, we present a greedy algorithm, alongside different parametrization options, that allows us to perform online conformance checking. The technique presented computes *prefix-alignments*, i.e. explanations of observed behaviour in terms of a given reference model, whilst accounting for future behaviour. We primarily focus on the quality of the conformance checking results using different instantiations of the proposed parametrization of the greedy algorithm. Furthermore, we show that, under certain conditions, the prefix-alignments that we compute are an under-estimator for the final deviation costs, i.e. upon completion of the process instance.

- *Process Enhancement*

  In chapter 8, we present an assessment of the computation of social networks in streaming settings. We primarily focus on computational feasibility in terms of incremental network updates of *handover-of-work networks*. We moreover show that some of these network variants lend themselves for incremental computation, which allows us to adopt these variants in an event stream setting.

1.5 Thesis Structure

In line with the research questions and associated contributions identified in section 1.4, the outline of this thesis, as visualized in Figure 1.9, is as follows. In chapter 2, we present basic preliminaries that aid the reader in understanding the basic notations and concepts used throughout the thesis. We furthermore present (two variants of) a running example, which we use throughout the thesis to exemplify concepts and/or algorithms, where necessary. We discuss *Data Engineering* related issues in chapter 3 and chapter 4. In particular, in chapter 3, we present several means to efficiently store events emitted onto an event stream, which effectively allows us to perform any existing process mining algorithm in the context of event streams. In chapter 4, we present means to filter out infrequent behaviour from event streams. The area of *process discovery* is covered in two chapters, i.e. chapter 5 and chapter 6. In chapter 5, we present a generic architecture that enables us to decompose process discovery into two sub-parts, i.e. learning and maintaining an intermediate representation and intermediate representation based discovery. In chapter 6, we show that, in some cases, we are able to explicitly exploit the nature of intermediate representations for the purpose of improving process discovery results. We cover *conformance checking* in chapter 7, where we examine the use of incrementally computed prefix-alignments. We cover *process enhancement* in chapter 8, where we study the incremental computation of social networks in an online setting. We detail on associated implementations of all techniques presented in this thesis in chapter 9. Finally, in chapter 10, we conclude the work presented in this thesis.

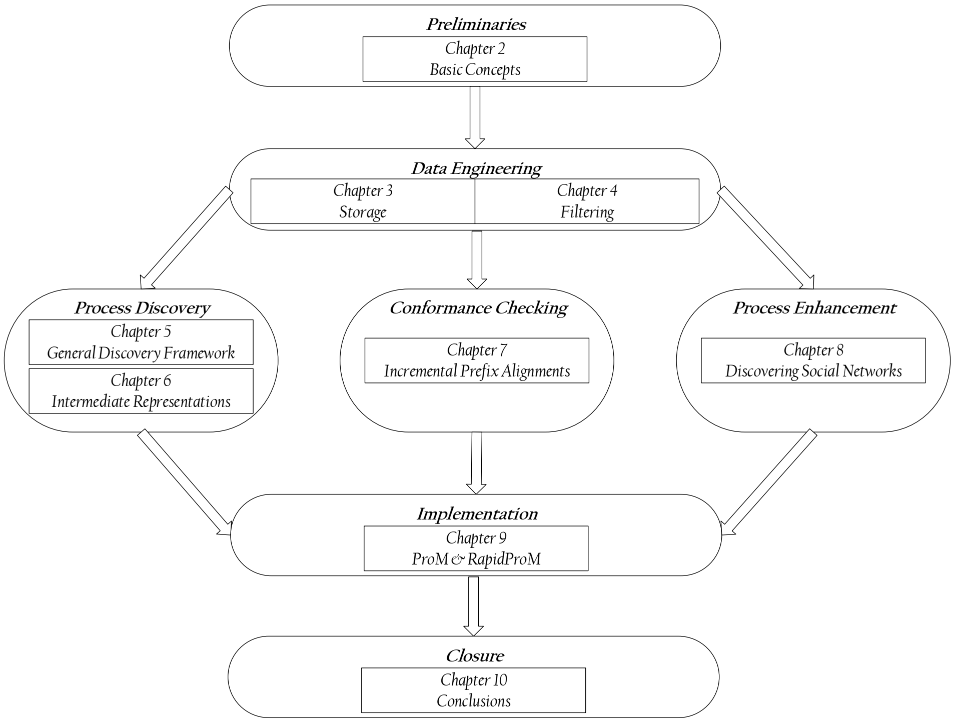


Figure 1.9: The general structure of this thesis.

Chapter 2

Preliminaries

The vast majority of process mining concepts, techniques and algorithms, build on a small set of basic mathematical concepts. In this chapter, we present these concepts, after which we formally define specific process mining concepts such as event data and commonly used process modelling formalisms.

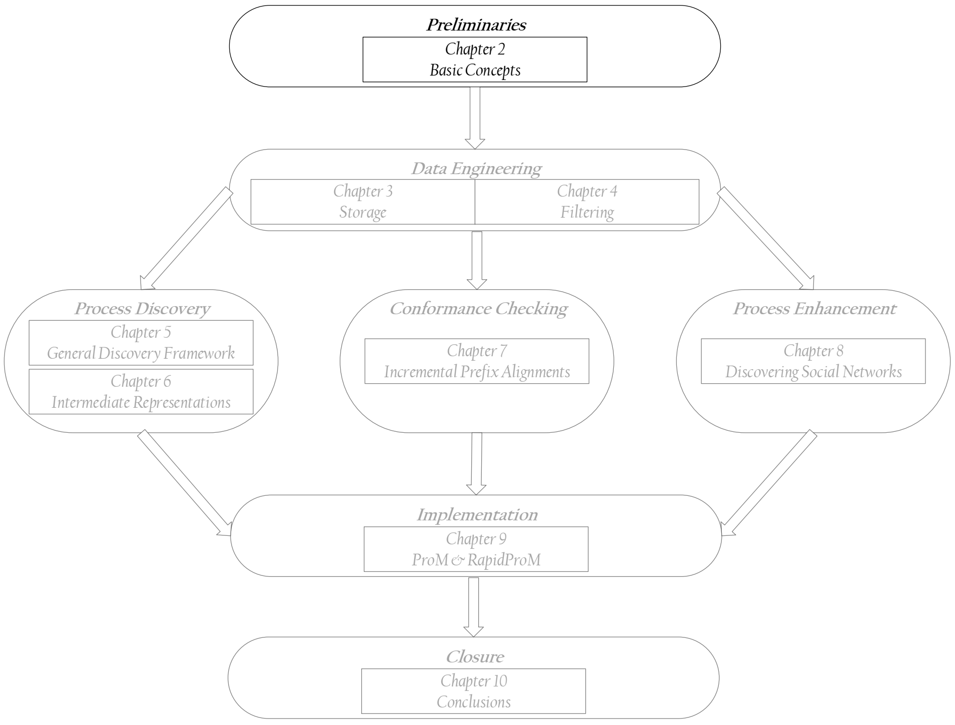


Figure 2.1: The contents of this chapter, i.e. preliminaries, highlighted in the context of the general structure of this thesis.

2.1 Basic Mathematical Concepts

In this section, we present some basic, well understood, mathematical concepts in the field of process mining as well as computer science in general. The main goal of this section is to familiarize the reader with the notation used throughout this thesis.

2.1.1 Sets, Tuples and Functions

Let $X = \{x_1, x_2, ..., x_n\}$ denote a set consisting of n different elements. The power set of a set X, i.e. $\mathscr{P}(X)$, denotes the set containing all possible subsets of X, i.e. $\mathscr{P}(X) = \{X' \mid X' \subseteq X\}$. We let $\mathbb{Z}$ denote the set of integers, i.e. numbers that we can write without a fractional component $(..., -2, -1, 0, 1, 2, ...)$. We let $\mathbb{N} = \{1, 2, ...\}$ denote the set of positive integers, i.e. the *natural numbers*, $\mathbb{N}_0 = \{0, 1, 2, ...\}$ additionally includes 0. We let $\mathbb{B} = \{0, 1\}$ denote the set of boolean values. Observe that a boolean value of 0 corresponds to value `false`, whereas a value of 1 corresponds to value `true`. Let $n_1, n_2 \in \mathbb{N}_0$ s.t. $n_1 < n_2$ we let $\{n_1, ..., n_2\}$ denote the interval of integers between (and including) n_1 and n_2. Given n arbitrary sets, i.e. $X_1, ..., X_n$, we define the n-ary Cartesian product of these n sets as $X_1 \times \cdots \times X_n = \{(x_1, ..., x_n) \mid \forall 1 \leq i \leq n\,(x_i \in X_i)\}$. We refer to an element in an n-ary Cartesian product as a *n-ary tuple*. In case $n = 2$, the Cartesian product defines the set of all ordered *pairs* $(x_1, x_2) \in X_1 \times X_2$. Given set X and Cartesian product $X_1 \times \cdots \times X_n$, if $\forall\ 1 \leq i \leq n(X_i = X)$, we simply write X^n. In some cases, we are interested in a particular element of a tuple. To this end we define a projection function, i.e. given $1 \leq i \leq n$, $\pi^i \colon X_1 \times \cdots \times X_n \to X_i$, s.t. $\pi^i((x_1, ..., x_i, ..., x_n)) = x_i$, e.g. for $(x, y, z) \in X \times Y \times Z$, we have $\pi^1((x, y, z)) = x$, $\pi^2((x, y, z)) = y$ and $\pi^3((x, y, z)) = z$. In the remainder of this thesis, we omit the explicit surrounding braces of tuples, when applying a projection on top of them, i.e. we write $\pi^1(x, y, z)$ rather than $\pi^1((x, y, z))$.

Any arbitrary subset $R \subseteq X_1 \times \cdots X_n$ is an *n-ary relation*. In case $n = 2$ we refer to a *binary relation*. Given such a binary relation, if $(x_1, x_2) \in R$ we alternatively write $x_1 R x_2$. Consider a binary relation R on sets X and Y, i.e. $R \subseteq X \times Y$. In the context of this thesis, we formulate the following properties on a binary relation R:

- R is *functional*, if and only if, $\forall x \in X, y, y' \in Y\left(xRy \wedge xRy' \implies y = y'\right)$, i.e. if a pair of the form $(x, ...)$ exists in R, it is the only pair of that form.
- R is *injective*, if and only if, $\forall x, x' \in X, y \in Y\left(xRy \wedge x'Ry \implies x = x'\right)$, i.e. there exist only one pair of the form $(..., y)$ in R.
- R is *left-total*, if and only if, $\forall x \in X, \exists y \in Y\left(xRy\right)$, i.e. for each element in X, there exists a counter-part in Y.

A functional relation is also referred to as a *partial function* f from X to Y, written as $f \colon X \nrightarrow Y$. Instead of xfy, we alternatively write $f(x) = y$. The domain of f represents all elements on which f is defined, i.e. $dom(f) = \{x \in X \mid \exists y \in Y\left(f(x) = y\right)\}$ whereas Y denotes the *codomain* of f, i.e. $codom(f) = Y$. The *range* of f, represents the values in Y that actually have a counter-part in X i.e. $rng(f) = \{y \in Y \mid \exists x \in X\left(f(x) = y\right)\}$. A binary relation that is both left-total and functional is referred to as a *total function* f from

X to Y, written as $f \colon X \to Y$. In the remainder, when we use the term function, we refer to a total function. Observe that a function, i.e. non-partial, is defined on every element in X, and each element of X has exactly one corresponding function value in Y. As such, the domain of a function $f \colon X \to Y$ is equal to X. Similar to projections, given a function f with characterization $f \colon X \times Y \to Z$, we write $f(x, y)$ rather than $f((x, y))$.

A binary function $R{\in}X \times X$ is referred to as an *endorelation* on X. For such endorelations, the following properties are of interest in the context of this thesis.

- R is *reflexive*, if and only if, $\forall x{\in}X\,(xRx)$.
- R is *irreflexive*, if and only if, $\nexists x{\in}X\,(xRx)$.
- R is *symmetric*, if and only if, $\forall x, y{\in}X\,\big(xRy \implies yRx\big)$.
- R is *antisymmetric*, if and only if, $\forall x, y{\in}X\,\big(xRy \implies \neg yRx\big)$.
- R is *transitive*, if and only if, $\forall x, y, z{\in}X\,\big(xRy \wedge yRz \implies xRz\big)$.

A relation $\preceq\, \subseteq X \times X$, alternatively written $(X, \preceq)$, is a *partial order*, if and only if, it is *reflexive*, *antisymmetric* and *transitive*. A relation $\prec\, \subseteq X \times X$, alternatively written $(X, \prec)$, is a *strict partial order*, if and only if, it is *irreflexive*, *antisymmetric* and *transitive*. Finally, given a strict partial order $(X, \prec)$ and a function $f \colon X \to X$, f is *strictly increasing*, if and only if, $x < x' \Leftrightarrow f(x) < f(x')$.

2.1.2 Multisets

A *multiset* (or bag) generalizes the concept of a set and allows elements to have a multiplicity, i.e. degree of membership, exceeding one. Let $X = \{x_1, x_2, ..., x_n\}$ be a set, a multiset B over X is a function $B \colon X \to \mathbb{N}_0$. We write a multiset as $B = [x_1^{k_1}, x_2^{k_2}, ..., x_n^{k_n}]$, where for each $i{\in}\{1, ..., n\}$ we have $B(x_i) = k_i$, however, if $B(x_i) = 0$, we omit x_i^0 from multiset notation, and, if $B(x_i) = 1$ we simply write x_i in multiset notation, i.e. we omit its superscript. The empty multiset is written as $[\,]$. If for some $x{\in}X$ we have $B(x) > 0$, we write $x{\in}_+B$. We define $B_+ = \{x{\in}X \mid x \in_+ B\}{\subseteq}X$. If for some $x{\in}X$ and $k{\in}\mathbb{N}$, we have $B(x) \geq k$, we write $x{\in}_+^k B$. Finally, if for some $x{\in}X$ and $k{\in}\mathbb{N}$, we have $B(x) = k$, we write $x{\in}^k B$. The universe of multisets over some set X, i.e. all possible multisets over X, being the multiset equivalence of the notion of the power set, is written as $\mathcal{B}(X)$.

Given multisets B_1 and B_2 over set X, we write $B_1{\subseteq}B_2$ if and only if $\forall x{\in}X\,(B_1(x) \leq B_2(x))$. The *union* of two multisets, i.e. $B_1{\cup}B_2$, yields a resulting multiset B' with $B'(x) = \max(B_1(x), B_2(x))$. The *intersection* of two multisets, i.e. $B_1{\cap}B_2$, yields a resulting multiset B' with $B'(x) = \min(B_1(x), B_2(x))$. The *sum* of two multisets, i.e. $B_1{\uplus}B_2$, yields a resulting multiset B' with $B'(x) = B_1(x) + B_2(x)$. Finally, the *difference* between two multisets, i.e. $B_1 - B_2$, yields a resulting multiset B' with $B'(x) = \max(0, B_1(x) - B_2(x))$.

Observe that multisets are defined in terms of a base set X. In some cases we need to compute operations on multisets defined over different domains, i.e. given some multiset B_X over X and B_Y over Y, s.t. $X \neq Y$, we want to compute $B_X{\uplus}B_Y$. Observe that we are able to extend any multiset $B_{X'}$ over $X'{\subseteq}X$ to a multiset over X

by assigning $B_{X'}(x) = 0, \forall x \in X \setminus X'$. Hence, to compute $B_X \uplus B_Y$, we first extend both B_X and B_Y to be multisets over $X \cup Y$, after which we apply the $\uplus$-operator. In case we apply a multiset operator on a multiset B_X and arbitrary and set Y, we, implicitly, first convert Y into a multiset $B_Y \colon Y \to \mathbb{N}_0$, where $B_Y(y) = 1$ if $y \in Y$ and 0 otherwise. Subsequently we perform the operator of choice on B_X and B_Y.

2.1.3 Sequences

Sequences represent enumerated collections of elements which additionally, like multisets, allow its elements to appear multiple times. However, within sequences we explicitly keep track of the *order* of an element. Given an arbitrary set X, a *sequence* of length n over X is defined as a function $\sigma \colon \{1, ..., n\} \to X$. Thus, σ assigns an element of X to each index $i \in \{1, ..., n\}$. We write a sequence as $\sigma = \langle \sigma(1), \sigma(2), ..., \sigma(n) \rangle$. The length of a sequence is written $|\sigma|$. We let ϵ denote the empty sequence, i.e. $|\epsilon| = 0$. The set of all possible sequences over set X, including infinite sequences, is written as X^*.

Given a sequence $\sigma \in X^*$ and $x \in X$, we write $x \in_* \sigma$ if and only if $\exists 1 \leq i \leq |\sigma| \, (\sigma(i) = x)$. Furthermore, we define $elem \colon X^* \to \mathscr{P}(X)$, with $elem(\sigma) = \{x \in X \mid x \in_* \sigma\}$. We additionally define the *Parikh abstraction*, which counts the multiplicity of a certain element within a sequence, i.e. $parikh \colon X^* \to \mathscr{B}(X)$, where:

$$parikh(\sigma) = \left[x^n \mid x \in X \wedge n = \sum_{i=1}^{|\sigma|} \left(\begin{cases} 1 & \text{if } \sigma(i) = x \\ 0 & \text{otherwise} \end{cases} \right) \right] \tag{2.1}$$

For simplicity, given $\sigma \in X^*$, we write $\vec{\sigma}$ to denote its Parikh abstraction, i.e. $\vec{\sigma} = parikh(\sigma)$.

Given two sequences $\sigma_1, \sigma_2 \in X^*$ the concatenation of sequence σ_1 and σ_2, written as $\sigma_1 \cdot \sigma_2$, yields sequence $\langle \sigma_1(1), \sigma_1(2), ..., \sigma_1(|\sigma_1|), \sigma_2(1), \sigma_2(2), ..., \sigma_2(|\sigma_2|) \rangle$. Similar to multisets, if $\sigma_1 \in X^*$ and $\sigma_2 \in Y^*$, then $\sigma_1 \cdot \sigma_2 \in (X \cup Y)^*$.

Given two sequences $\sigma_1, \sigma_2 \in X^*$, σ_1 is a *subsequence* of σ_2, written $\sigma_1 \subseteq_* \sigma_2$, if there exists a strictly increasing function:

$$\varphi \colon \{1, ..., |\sigma_1|\} \to \{1, ..., |\sigma_2|\} \text{ s.t. } \forall i \in \{1, ..., |\sigma_1|\} \left(\sigma_1(i) = \sigma_2(\varphi(i)) \right)$$

Consider for example $\langle a, b, c \rangle$, which is a subsequence of $\langle a, a, d, b, e, c \rangle$ as we are able to construct $\varphi = \{(1,1), (2,4), (3,6)\}$. A subsequence is a *strict subsequence* if the mapping is consecutive. Given a sequence $\sigma \in X^*$ and $1 \leq i \leq j \leq |\sigma|$ we denote its strict subsequence starting at index i and ending at index j as $\sigma_{i...j}$, e.g. $\langle a, a, b, c, b, d \rangle_{2...4} = \langle a, b, c \rangle$. A subsequence is a *prefix* if and only if the mapping function φ is an identity function, i.e. $\varphi(1) = 1, \varphi(2) = 2, ..., \varphi(|\sigma|) = |\sigma|$, e.g. $\langle a, a, d \rangle$ is a prefix of $\langle a, a, d, b, e, c \rangle$, as we have $\varphi = \{(1,1), (2,2), (3,3)\}$. A set of sequences $X \in \mathscr{P}(Y^*)$ is *prefix-closed* if $\epsilon \in X \wedge \forall \sigma \in X \left(\exists \sigma' \in X \left(\sigma' = \sigma_{1...|\sigma|-1} \right) \right)$. Furthermore, we define the *prefix-closure* of set $X \in \mathscr{P}(Y^*)$ as $\overline{X}$, with $X \subseteq \overline{X}$, and, recursively:

$$\overline{X} = \{\epsilon\} \cup \bigcup_{\sigma \in \overline{X}} \left(\bigcup_{i=1}^{|\sigma|} (\sigma_{1...i}) \right) \tag{2.2}$$

Observe that, by definition, a prefix-closure is prefix-closed. We overload notation here, and also define the prefix-closure for multisets of sequences, i.e. given a multiset $B\colon X^* \to \mathbb{N}_0$, we define $\overline{B}\colon X^* \to \mathbb{N}_0$, where:

$$\overline{B}(\sigma) = B(\sigma) + \sum_{\sigma\cdot\langle x\rangle \in B_+} \overline{B}(\sigma \cdot \langle x\rangle)$$

For example, for $B = [\langle a,b\rangle^5, \langle a,c\rangle^3]$, we obtain $\overline{B} = [\epsilon^8, \langle a\rangle^8, \langle a,b\rangle^5, \langle a,c\rangle^3]$. A subsequence is a *suffix* of a sequence if it is a strict subsequence, and its last element is the last element of the other sequence, i.e. $\langle b,e,c\rangle$ is a suffix of $\langle a,a,d,b,e,c\rangle$.

In some cases, we need to project a sequence of tuples on a sequence of elements of a specific set within the Cartesian product. To this end we extend projection to the level of sequences, i.e. given a sequence $\sigma \in (X_1 \times X_2 \cdots X_n)^*$ and $1 \leq i \leq n$, we define a projection function $\pi^i_*\colon (X_1 \times X_2 \cdots X_n)^* \to X_i^*$ s.t. $\pi^i_*(\sigma) = \langle \pi^i(\sigma(1)), \pi^i(\sigma(2), ..., \pi^i(\sigma(|\sigma|)))\rangle$, e.g. for $\sigma \in (X \times Y \times Z)^*$, we have $\pi^1_*(\sigma) \in X^*$.

Given $\sigma \in X^*$ and $Y \subseteq X$ we define $\sigma_{\downarrow_Y} \in Y^*$ recursively with $\epsilon_{\downarrow_Y} = \epsilon$ and $(\langle x\rangle \cdot \sigma')_{\downarrow_Y} = \langle x\rangle \cdot \sigma'_{\downarrow_Y}$ if $x \in Y$ and $\sigma'_{\downarrow_Y}$ if $x \notin Y$. Finally, given a function $f\colon X \to Y$, we define $f_*\colon X^* \to Y^*$, such that, given a sequence $\sigma \in X^*$, we have $f_*(\sigma) = \langle f(\sigma(1), f(\sigma(2)), ..., f(\sigma(|\sigma|))\rangle$.

2.1.4 Matrices and Vectors

We occasionally use the notions of matrices and vectors in this thesis. We assume *vectors* to be n-ary tuples in $\mathbb{R}^n$. We define such vector of size n as $\vec{x} \in \mathbb{R}^n$, e.g. $(1, \frac{1}{4}, 2) \in \mathbb{R}^3$. Furthermore, a vector $\vec{x} \in \mathbb{R}^n$ represents a *column vector* whereas $\vec{x}^\intercal \in \mathbb{R}^n$ represents a *row vector*:

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \vec{x}^\intercal = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix} \tag{2.3}$$

To access the i^{th} element of a vector $\vec{x}$, we write $\vec{x}(i)$, rather than $\pi^i(\vec{x})$.

A *matrix* is considered a rectangular array of values in $\mathbb{R}$. A matrix consisting of n rows and m columns, i.e. an $n \times m$ matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$, is written as:

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,m} \\ a_{2,1} & a_{2,2} & \ldots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \ldots & a_{n,m} \end{pmatrix} \tag{2.4}$$

We write $\mathbf{A}_{i,j}$ to refer to element $a_{i,j}$, i.e. the value at row i, column j. Observe that a row vector of size n corresponds to an $1 \times n$ matrix, whereas a column vector of size n corresponds to an $n \times 1$ matrix.

Multiplication of two matrices is possible if the number of columns of the left argument coincides with the number of rows of the right argument. Given an $n \times m$

matrix $\mathbf{A}$ and an $m \times k$ matrix $\mathbf{A}'$, multiplying the two, i.e. $\mathbf{AA}'$ results in an $n \times k$ sized matrix where $(\mathbf{AA}')_{i,j} = \sum\limits_{l=1}^{m} \mathbf{A}_{i,l}\mathbf{A}'_{l,j}$. Observe that we are thus able to multiply a row vector $\vec{x}^\intercal$ of size n with a matrix $\mathbf{A}$ of the form $n \times m$, i.e. $\vec{x}^\intercal\mathbf{A}$, for arbitrary m. In case $m = 1$ we are multiplying a row vector with a column vector of size n. Similarly, we are able to multiply a matrix $\mathbf{A}$ of the form $n \times m$ with an n-sized column vector, i.e. $\mathbf{A}\vec{x}$.

In some cases, we define a vector as $\vec{x} \in \mathbb{N}_0^n$, which merely indicates that we are only interested in assigning values to the vector that are in $\mathbb{N}_0$. Moreover, in several application scenario's, given an arbitrary set X, we assume the existence of some function $f \colon X \to \mathbb{R}$ and define a corresponding vector $\vec{x} \in \mathbb{R}^{|X|}$. In these cases we assume that there exists an injective index function $\iota \colon X \to \{1, ..., |X|\}$ s.t. $\vec{x}(\iota(x)) = f(x)$. We furthermore assume that only one such index function exists and is used, i.e. as such given two vectors $\vec{x}, \vec{y} \in \mathbb{R}^{|X|}$, we assume that $\vec{x}$ and $\vec{y}$ agree on their indices. This allows us to directly write and reason about operations such as $\vec{x}^\intercal \vec{y}$. For convenience, we write $\vec{x}(x)$ as a place-holder for $\vec{x}(\iota(x))$. Note that the concept of indices and sets extends to matrices. As such, given $\sigma \in X^*$, when working with vectors, we overload notation and refer to $\overrightarrow{\sigma}$ as the *Parikh vector* of σ, rather than the Parikh abstraction.

2.1.5 Graphs

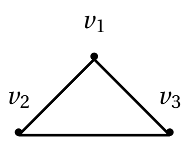
A graph is an ordered pair $G = (V, E)$ where V is a set of vertices and E is a set of edges with $E \subseteq V \times V$, i.e. an endorelation on V. A graph is undirected if E is *irreflexive* and *symmetric*. As such, in an undirected graph, we write edges as sets rather than pairs, i.e. $\{v_1, v_2\}$ represents the fact that $(v_1, v_2), (v_2, v_1) \in E$. A directed graph does not have the aforementioned property and in case of a directed graph we alternatively refer to the E as a set of arcs. As an example, consider Figure 2.2 in which we depict an example undirected- and directed graph. Graph G_1 in Figure 2.2a represents an undirected graph with three vertices v_1, v_2 and v_3, which are graphically represented as a black dot. The edges, graphically, connect the vertices by means of a line. In graph G_2, in Figure 2.2b, we depict a directed graph. The main difference is within the visualization of the arcs, which are represented by arrows, rather than lines.

The degree of a vertex in an undirected graph is a function $deg \colon V \to \mathbb{N}_0$ which represents the number of edges that are connected to it, i.e.

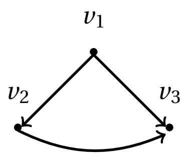
$$deg(v) = |\{\, v' \in V \mid \{v, v'\} \in E\}| \tag{2.5}$$

For example, observe that all vertices in G_1 have a degree of 2. For directed graphs we take the orientation of an arc into account, and thus we have an indegree deg^- and outdegree deg^+, i.e. $deg^-(v) = |\{v' \in V \mid (v', v) \in E\}|$ and $deg^+(v) = |\{v' \in V \mid (v, v') \in E\}|$. In case a vertex has no indegree, i.e. $deg^-(v) = 0$ it is a *source*. If it has no outdegree, i.e. $deg^+(v) = 0$ it is a *sink*. Note that in G_2, v_1 is a source whereas v_3 is a sink, and, $deg^-(v_2) = deg^+(v_2) = 1$.

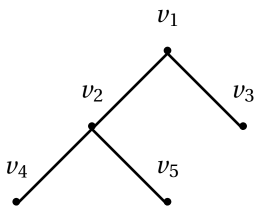
In an (un)directed graph, a *path* is a non-empty sequence of edges, which connects a corresponding sequence of vertices. An undirected graph is *connected* if there exists a path between any pair of vertices. For example, graph G_1 in Figure 2.2a is a connected undirected graph. An undirected graph contains a cycle



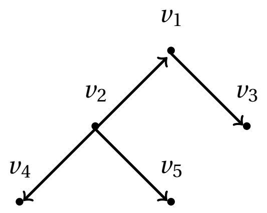
(a) Example undirected graph $G_1 = (\{v_1, v_2, v_3\}, \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}\})$



(b) Example directed graph $G_2 = (\{v_1, v_2, v_3\}, \{(v_1, v_2), (v_1, v_3), (v_2, v_3)\}$



(c) Example tree $G_1^T = (\{v_1, v_2, v_3, v_4, v_5\}, \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_4\}, \{v_2, v_5\}\})$



(d) Example rooted tree $G_2^T = (\{v_1, v_2, v_3, v_4, v_5\}, \{(v_1, v_3), (v_2, v_1), (v_2, v_4), (v_2, v_5)\})$, where v_2 is the tree's root.

Figure 2.2: Two example graphs, i.e. G_1 and G_2, and two example trees, i.e. G_1^T and G_2^T.

if $\exists v \in V\, (\exists \sigma \in E^*\, (\sigma = \langle (v, v_1), (v_1, v_2), ..., (v_{n-1}, v_n), (v_n, v) \rangle))$. Observe that graph G_1 does contain a cycle.

If an undirected graph is connected yet does not contain any cycle, such graph is a *tree*. As a consequence, given a tree $G^T = (V, E)$, we have $|E| = |V| - 1$. Observe that, G_1 in Figure 2.2a contains a cycle, and hence, is not a tree. However, the graph depicted in Figure 2.2c, i.e. G_1^T, is in fact a tree. In some cases we assign a specific vertex $v^r \in V$ as the root of the tree. We call such tree a *rooted tree*, cf. Figure 2.2d. The depth of a vertex present in a rooted tree is a function $depth\colon V \to \mathbb{N}_0$, s.t. $depth(v^r) = 0$ and for any other $v \in V$, $depth(v)$ is defined as the length of the path from v^r to v. To visualize a rooted tree, we use arrows instead of lines to connect the vertices, even though a tree is always an undirected graph. We use arrows to visualize the depth of the tree. The vertex that has no incoming arcs in such visualization is the root vertex. Observe that the tree in Figure 2.2d is similar to the tree in Figure 2.2c, yet has vertex v_2 as its root. A vertex v in a tree, not equal to the root of the tree, for which we have $deg(v) = 1$ is a *leaf* vertex. Other vertices, again except the root, are called *internal* vertices. Finally, in Figure 2.2d, observe that the depth of vertices v_1, v_4 and v_5 is 1, whereas the depth of vertex v_3 is 2.

A special type of tree that allows us to efficiently encode sets of sequences is a *prefix tree*. Given an alphabet, i.e. set of characters Σ, and a set of sequences over Σ, i.e. $\mathfrak{L} \subseteq \Sigma^*$. A prefix tree is a rooted tree with an additional labelling function $\lambda\colon E \to \Sigma$ and a boolean terminal function $term\colon V \to \mathbb{B}$ which allows us to encode $\mathfrak{L}$. The projection of each vertex on a path starting from the root vertex and ending in a vertex

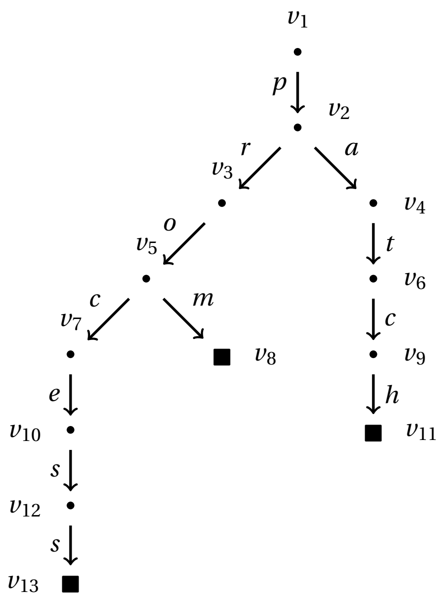


Figure 2.3: Example prefix-tree G_3^T encoding $\mathfrak{L}$ = {process, prom, patch}. Vertices v with $term(v) = \texttt{true}$ are visualized as ■ symbols.

v with $term(v) = 1$ corresponds to an element of $\mathfrak{L}$. As an example of a prefix-tree, consider prefix-tree G_3^T depicted in Figure 2.3. The prefix-tree encodes the set of sequences $\mathfrak{L}$ = {process, prom, patch}. Within prefix-tree G_3^T we depict non-terminal vertices, i.e. $term(v) = 0$, as •-symbols and terminal vertices as ■-symbols. In this example, only leaves are terminal, however, in general, this need not be the case. For example, if we add the sequence $\langle p, a \rangle$ to the language, v_4 becomes a terminal vertex. In particular, if $\epsilon \in \mathfrak{L}$ then the root is a terminal vertex. Also, observe that an internal vertex being a terminal node implies that some strict prefixes of sequences in $\mathfrak{L}$ are in $\mathfrak{L}$ to. Moreover, if all vertices are terminal (including the root vertex), the prefix-tree describes a prefix-closed language.

2.1.6 Integer Linear Programming

An Integer Linear Programming (ILP)-formulation, allows us to specify an optimization problem, i.e. either a minimization or maximization problem, defined in terms of a set of integer-valued decision variables. The goal is to find an assignment of the decision variables that minimizes/maximizes the corresponding *objective function* and at the same time adheres to an additionally defined set of *constraints*. These constraints are in turn linear (in)equalities, expressed in terms of the decision variables. We further exemplify the general notion of an ILP-formulation in 2.1.

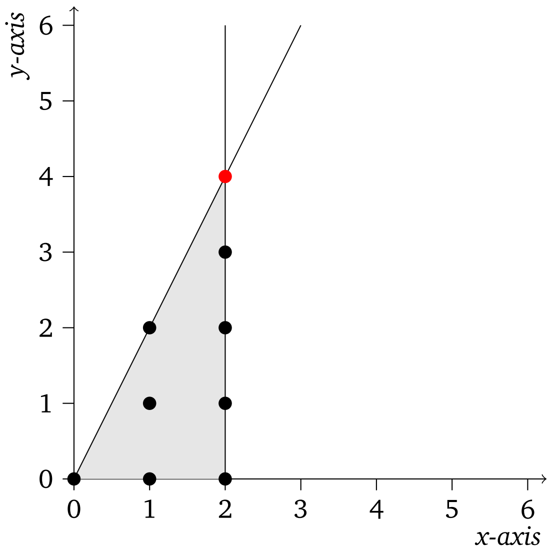


Figure 2.4: Visualization of the example ILP-formulation. The grey area highlights the region in which integer-valued solutions exist, i.e. the feasible region. The black dots represent (some of the) solutions according to the body of constraints, i.e. feasible solutions, whereas the red dot represents the optimal solution.

Example 2.1 (Example ILP-formulation). *Consider that we have two integer variables, i.e. $x, y \in \mathbb{Z}$. We aim at maximizing the sum of these two variables, i.e. $x+y$. Furthermore, we want to constrain the value of x to be at most 2, i.e. $x \leq 2$. We constrain y in terms of x, i.e. $y \leq 2x$. We are able to construct a corresponding ILP-formulation of the following form:*

$$\begin{array}{rrl} \textbf{\textit{maximize}} & x+y & \textit{objective function} \\ \textbf{\textit{such that}} & x \leq 2 & \textit{constraint on } x \\ \textbf{and} & 2x+y \leq 0 & \textit{constraint on } y \textit{ in terms of } x \\ & x, y \in \mathbb{Z} & x \textit{ and } y \textit{ are integers} \end{array} \tag{2.6}$$

In case of the simple ILP-formulation listed in Equation 2.6, it is easy to see that the corresponding solution is $x = 2$, $y = 4$. For convenience, a corresponding graphical representation of the example is provided in Figure 2.4. In Figure 2.4, the grey area indicates all possible variable assignments in $\mathbb{R}_{\geq 0}$ (we omit negative values in the figure) that adhere to the two constraints. The black dots indicate the actual integer values *that adhere to the constraints. The total set of solutions to the given body of constraints is also referred to as the set of* feasible solutions. *In the example visualization, the black dots are part of the set of feasible solutions. Finally, the solution that maximizes the objective function (or, in minimization is minimizing the objective function) is also referred to as the* optimal solution. *In Figure 2.4, the optimal solution is visualized by means of a red dot.*

In the example ILP-formulation presented in 2.1, finding an optimal variable assignment is rather easy. In the general sense, however, the complexity of finding such a solution is proven to be *NP-complete*. In the context of this thesis, we do not go into detail with respect to the mechanics of finding an optimal solution to a given ILP-formulation, i.e. we refer to [105]. We merely assume that we are able to *solve* a given ILP-formulation, using an *ILP-solver*, i.e. a dedicated software library that provides us with an optimal variable assignment given an ILP-formulation. Examples of such solvers are `LPSolve`[1], `Gurobi`[2] and `IBM ILOG CPLEX`[3]. Finally, it is important to note that, in general, it is possible that no solution exists for an ILP (due to contradicting constraints), or, a multitude of optimal solutions exist.

2.2 Process Models

Since process mining revolves around event data originating from the execution of processes, we need a, preferably formal, means of representing and reasoning about these processes. Hence, within process mining, process models, i.e. formalisms that describe the behaviour of processes, are, like event data, considered as being a first class citizen. In essence, a process model describes the intended behaviour of a process. Reconsider Figure 1.2 (in chapter 1, subsection 1.1.1 on page 6), in which we depict process models using different formalisms, i.e. a Petri net (Figure 1.2a) and a BPMN model (Figure 1.2b). In the context of this thesis, we define $\mathscr{M}$ as the universe of process models, i.e. any process model, regardless of its formalism, is a member of $\mathscr{M}$. Thus, both the Petri net in Figure 1.2a and the BPMN model in Figure 1.2b are members of $\mathscr{M}$. However, even though many other process modelling formalisms exist as well, within this thesis we primarily focus on (labelled) Petri nets [94], which allow us to explicitly model concurrency in a concise and compact manner. In the upcoming section we introduce Petri nets, in subsequent sections we discuss alternative, process oriented modelling formalisms which are explicitly used in this thesis.

2.2.1 Petri nets

Consider Figure 2.5, in which we depict an example Petri net. The Petri net describes a simplified online ID-document verification process. The model describes that the first activity to be performed should always be *receive ID*. Subsequently, the *scan picture* and *scan watermark* activities can be performed concurrently. However, we are also allowed to only perform the scan watermark activity and subsequently perform a *verification*, i.e. we are allowed to skip scanning the picture. If a verification results in a negative verification result, we are able to redo the scan of the watermark/picture. However, such a decision, i.e. to redo these activities, is not explicitly captured by the system. This unobservable action is represented by the grey rectangle with label t_6. Eventually, either the *activate account* or the *block account* activity is performed.

[1] `http://lpsolve.sourceforge.net/`

[2] `http://gurobi.com`

[3] `https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer`

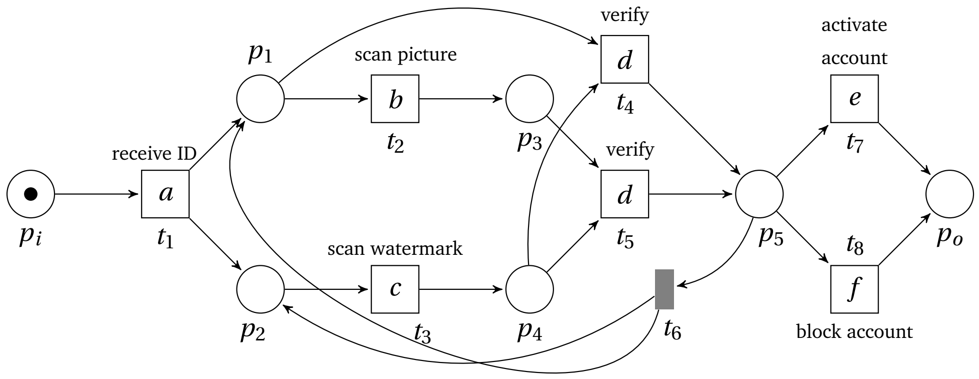


Figure 2.5: Example process related to an ID verification process, modelled as a labelled Petri net with initial marking $[p_i]$.

A Petri net consists of *places* and *transitions*. Places are used to represent the *state* of the described process whereas transitions represent possible executable activities, subject to the state. Graph-theoretically, a Petri net is a bipartite graph in which places are only connected to transitions, and, consequently, transitions are only connected to places. The Petri net in Figure 2.5 consists of 7 places (denoted P), i.e. $P = \{p_i, p_1, ..., p_5, p_o\}$, visualized as circles. The Petri net furthermore contains 8 transitions (denoted T), i.e. $\{t_1, ..., t_8\}$, visualized as boxes. Observe that, indeed, in the example Petri net depicted in Figure 2.5, places only connect to transitions whereas transitions only connect to places.

Definition 2.1 (Petri net). *Let P denote a set of places and let T denote a set of transitions s.t. $P \cap T = \emptyset$. Let $F = (P \times T) \cup (T \times P)$ denote the flow relation. Furthermore, let Σ denote the universe of labels, let $\tau \notin \Sigma$ and let $\lambda \colon T \rightarrow \Sigma \cup \{\tau\}$ denote the transition labelling function. A Petri net N, is a tuple $N = (P, T, F, \lambda)$.*

Observe that the labelling function maps transitions to a corresponding label, e.g. the label of transition t_1 in Figure 2.5 is *receive ID*, or a in short-hand notation. In case a transition t is unobservable, e.g. transition t_6 in Figure 2.5, we have $\lambda(t) = \tau$, with $\tau \notin \Sigma$.

Given any element $x \in P \cup T$ of a Petri net $N = (P, T, F, \lambda)$, i.e. either a place or a transition, we write $\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$. Similarly we define $x \bullet = \{y \in P \cup T \mid (x, y) \in F\}$. For example, in Figure 2.5, we have $\bullet p_1 = \{t_1, t_6\}$, whereas $t_6 \bullet = \{p_1, p_2\}$.

We represent the state of a Petri net in terms of a *marking* M, which is a multiset of places, i.e. $M \in \mathcal{B}(P)$. For example, in Figure 2.5, place p_i is *marked* with a *token*, visualized by a black dot. Thus, the marking of the Petri net in Figure 2.5, as visualized, is $[p_i]$. Given a Petri net N and marking M, we refer to the pair (N, M) as a *marked net*.

The transitions of a Petri net allow us to manipulate its state, i.e. its marking. A transition $t \in T$ is *enabled* if all places p that have an outgoing arc to t contain a token. If a transition is enabled in marking M we write $(N, M)[t\rangle$. An enabled transition is able to *fire*. If we fire a transition t, it consumes a token from each place that has an outgoing arc to t. Subsequently, a token is produced in each place that has an incoming arc from t. For example, in Figure 2.5, t_1 is the only enabled transition in marking $[p_i]$, and, if it fires we obtain new marking $[p_1, p_2]$. In marking $[p_1, p_2]$ we are able to fire both t_2 and t_3, in any order.

Definition 2.2 (Enabled Transition, Firing Rule). *Let $N = (P, T, F, \lambda)$ be a Petri net with labelling function $\lambda \colon T \to \Sigma \cup \{\tau\}$, and let $M \in \mathcal{B}(P)$ be a marking of N. Transition $t \in T$ is enabled in M, written as $(N, M)[t\rangle$, if and only if $\forall p \in \bullet t \left(M(p) > 0\right)$. An enabled transition in marking M is able to fire, which results in marking $M' = (M - \bullet t) \uplus t\bullet$. We write such a transition firing as $(N, M) \xrightarrow{t} (N, M')$.*

Let $\sigma = \langle t_1, ..., t_n \rangle \in T^*$ be an arbitrary sequence of transitions. Sequence σ is a firing sequence of N in marking M, yielding marking M', if and only if, we are able to find markings $M_1, ..., M_n \in \mathcal{B}(P)$ s.t. $(N, M) \xrightarrow{t_1} (N, M_1) \xrightarrow{...} (N, M_{n-1}) \xrightarrow{t_n} (N, M')$. We overload notation and simply write $(N, M) \xrightarrow{\sigma} (N, M')$.

Reconsider Figure 2.5, in which we are able to generate multiple firing sequences from marking $[p_1]$ that yield marking $[p_3, p_4]$, e.g. $\langle t_1, t_2, t_3 \rangle$, $\langle t_1, t_3, t_2 \rangle$, $\langle t_1, t_3, t_4, t_6, t_2, t_3 \rangle$.

Given a Petri net $N = (P, T, F, \lambda)$ and marking $M \in \mathcal{B}(P)$, we define the language of Petri net N, given (initial) marking M, as:

$$\mathfrak{L}(N, M) = \left\{\sigma \in T^* \mid \exists M' \in \mathcal{B}(P) \left((N, M) \xrightarrow{\sigma} (N, M')\right)\right\} \tag{2.7}$$

Observe that the language of a Petri net, as defined in Equation 2.7 is *prefix-closed*. In some cases, we are interested to project the elements of the language of a Petri net to their corresponding labels in Σ. To this end, we additionally define:

$$\mathfrak{L}_{\Sigma}(N, M) = \left\{\sigma \in \Sigma^* \mid \exists \sigma' \in \mathfrak{L}(N, M) \left((\lambda_*(\sigma'))_{\downarrow_{\Sigma}} = \sigma\right)\right\} \tag{2.8}$$

Furthermore, given a Petri net $N = (P, T, F, \lambda)$ and marking $M \in \mathcal{B}(P)$, we are, in some cases, interested in the markings that are reachable, starting from marking M, i.e.

$$\mathfrak{R}(N, M) = \{M' \in \mathcal{B}(P) \mid \exists \sigma \in T^* \left((N, M) \xrightarrow{\sigma} (N, M')\right)\} \tag{2.9}$$

Observe that, $M \in \mathfrak{R}(N, M)$, i.e. by means of empty firing sequence ϵ. A marking $M \in \mathcal{B}(P)$ is referred to as a *deadlock* if and only if:

$$\nexists t \in T \left((N, M)[t\rangle\right) \tag{2.10}$$

Furthermore, a marking $M \in \mathcal{B}(P)$ is referred to as a *livelock* if and only if:

$$\forall M' \in \mathfrak{R}(N, M) \left(\mathfrak{R}(N, M') = \mathfrak{R}(N, M)\right) \tag{2.11}$$

Several classes of Petri nets exist, either based on graph-theoretical/structural properties based on behavioural properties. In the context of this thesis, we consider the structural notion of *workflow nets* and several (behavioural) notions of *soundness*. We primarily base our definitions on [2, 8].

A workflow net is any arbitrary Petri net that has an additional set of structural properties. Workflow nets were introduced as a modelling notation for process oriented information systems and typically describe the behaviour of an instance of such a process. Therefore, a workflow net needs to specify a clear start- and corresponding end point of the process it describes. Furthermore, it is not allowed to consist of multiple detached parts.

Definition 2.3 (Workflow net). *Let $N=(P,T,F,\lambda)$ be a Petri net with labelling function $\lambda\colon T\rightarrow\Sigma\cup\{\tau\}$. Let $p_i,p_o\in P$ with $p_i\neq p_o$. Petri net N is a workflow net (WF-net) if and only if:*

1. $\bullet p_i=\emptyset$
2. $p_o\bullet=\emptyset$
3. *Let $t\notin T$, the Petri net $N'=(P,T\cup\{t\},F\cup\{(t,p_i),(p_o,t)\},\lambda)$ is strongly connected, i.e. there is a directed path between each pair of nodes in N'.*

In some cases, we alternatively write a Petri net, of which we know it is a workflow net, as $N=(P,T,F,\lambda,p_i,p_o)$, i.e. to explicitly indicate the unique source- and sink place. Observe that the Petri net presented in Figure 2.5 in fact adheres to 2.3 and is thus a workflow net.

2.3 merely specifies structural/graph-theoretical properties of a Petri net, i.e. given a workflow net, we have limited capabilities to reason about its quality, e.g. whether or not it is free of deadlocks. To this end, we define several behavioural classes for Petri nets which allow us to reason about the behaviour described by the Petri net, as well as its quality and correctness as a process model. Prior to this, we present two behavioural properties which we subsequently utilize within the definitions of the behavioural classes.

Definition 2.4 (Liveness, Boundedness). *Let $N=(P,T,F,\lambda)$ be a Petri net with labelling function $\lambda\colon T\rightarrow\Sigma\cup\{\tau\}$ and let $M\in\mathscr{B}(P)$ be a marking of N.*

- *Marked net (N,M) is live if and only if:*

$$\forall M'\in\mathfrak{R}(N,M),t\in T\left(\exists M''\in\mathfrak{R}(N,M')\left((N,M'')[t\rangle\right)\right) \tag{2.12}$$

- *Marked net (N,M) is k-bounded ($k\in\mathbb{N}$) if and only if:*

$$\forall M'\in\mathfrak{R}(N,M),p\in P\left(M'(p)\leq k\right) \tag{2.13}$$

In case a Petri net is 1-bounded, we call such net *safe*. Note that any k-bounded Petri net has a strictly finite set of reachable markings, i.e. $\mathfrak{R}(N,M)$ is a finite set. As such, whenever $\mathfrak{R}(N,M)$ is finite, we know that (N,M) is *bounded*.

We subsequently define different increasingly strict variants of soundness of workflow nets.

Definition 2.5 (Easy Soundness). *Let $N = (P, T, F, \lambda, p_i, p_o)$ be a workflow net with labelling function $\lambda \colon T \to \Sigma \cup \{\tau\}$. N is* easy sound *if and only if:*

$$[p_o] \in \mathfrak{R}(N, [p_i]) \tag{2.14}$$

Observe that *easy soundness* guarantees that if we place a token in the initial place p_i of a workflow net, it is possible to end up in a marking that only marks the sink place p_o. As the name suggests, easy soundness is a rather weak correctness property of a workflow net, i.e. we are only guaranteed that we are able to at least once correctly mark the sink place p_o. A slightly stronger notion of behavioural correctness, which is of particular interest in the context of this thesis, is *relaxed soundness*. Relaxed soundness ensures us that we are able to fire each transition within the Petri net at least once, and that we are, in the end, able to reach marking $[p_o]$.

Definition 2.6 (Relaxed Soundness). *Let $N = (P, T, F, \lambda, p_i, p_o)$ be a workflow net with labelling function $\lambda \colon T \to \Sigma \cup \{\tau\}$. N is* relaxed sound *if and only if:*

$$\forall t \in T \left(\exists \sigma \in T^* \left((N, [p_i]) \xrightarrow{\sigma} (N, [p_o]) \land t \in_* \sigma \right) \right) \tag{2.15}$$

We finally introduce the notion of *soundness* which is the strongest correctness notion defined in terms of workflow nets that we consider in this thesis.

Definition 2.7 (Soundness). *Let $N = (P, T, F, \lambda, p_i, p_o)$ be a workflow net with labelling function $\lambda \colon T \to \Sigma \cup \{\tau\}$. N is* sound *if and only if:*

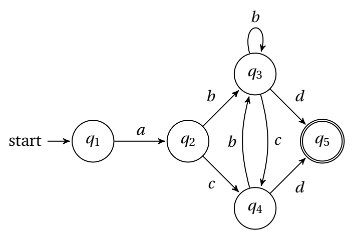
1. *(*safeness*) $(N, [p_i])$ is safe (i.e. $\forall M \in \mathfrak{R}(N, [p_i]), p \in P \left(M(p) \leq 1 \right)$).*
2. *(*option to complete*) $\forall M \in \mathfrak{R}(N, [p_i]) \left([p_o] \in \mathfrak{R}(N, M) \right)$.*
3. *(*absence of dead parts*) $\forall t \in T \left(\exists M \in \mathfrak{R}(N, [p_i]) \left((N, M)[t\rangle \right) \right)$*

Thus, a sound workflow net guarantees us that there is only one deadlock, i.e. $[p_o]$. Furthermore we are guaranteed that any place within the workflow net contains at most one token at any point in time. Finally, we are guaranteed that we are able to, from each reachable marking, eventually uniquely mark the sink place p_o. This requirement implies an additional property, i.e. *proper completion*, which is formalized as:

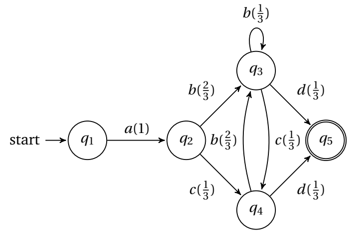
$$\forall M \in \mathfrak{R}(N, [p_i]) (p_o \in_+ M \implies M = [p_o]) \tag{2.16}$$

This requirement is often presented alongside the other requirements, however, since it is directly implied by the *option to complete* property, we omit it from 2.7.

Finally, observe that if a workflow net is sound, it is also relaxed sound. Moreover, any relaxed sound workflow net is in turn easy sound. Observe that the Petri net depicted in Figure 2.5 is a sound workflow net.



(a) Example deterministic automaton.



(b) Example probabilistic automaton.

Figure 2.6: Two example automata based on the underlying (deterministic) automaton $A_1 = (\{q_1,...,q_5\},\{a,b,c,d\},\{((q_1,a),q_2),...,((q_4,d),q_5)\},q_1,\{q_5\})$.

2.2.2 Automata

In essence, Petri nets, as presented in subsection 2.2.1 allow us to describe a language, i.e. a collection of sequences of letters (represented by transitions/labels). Here we present the notion of (probabilistic) automata, which essentially allow us to describe languages as well. The main difference with respect to Petri nets is the fact that within an automaton we do not have the notion of a marking, i.e. it only consists of states and transitions. Given a certain state, a number of transitions/actions are possible which in turn define the next state.

Definition 2.8 (Automaton). *A* non-deterministic automaton *(also non-deterministic finite state machine) is a* 5*-tuple* $A = (Q, \Sigma, \delta, q^0, F)$*, where*

- Q *is a finite set of states,*
- Σ *is a finite set of symbols,*
- $\delta\colon Q \times \Sigma \to \mathscr{P}(Q)$ *is a transition relation,*
- $q^0 \in Q$ *is the initial state,*
- $F \subseteq Q$ *is the set of accepting states,*
- $\forall q \in Q \setminus F (\exists q' \in Q, a \in \Sigma (q' \in \delta(q,a)))$*: non-accepting states have outgoing arc(s).*

In some cases we have $|\delta(q,a)| \leq 1, \forall q \in Q, a \in \Sigma$. In such case the automaton is *deterministic*, and, we alter the transition function to be simply $\delta\colon Q \times \Sigma \nrightarrow Q$.

Consider Figure 2.6a, in which we depict an example deterministic automaton. States of the automaton are visualized as circles, transitions are represented by arcs with the corresponding label depicted next to the arc. The initial state is depicted by means of a circle with an incoming arc labelled *start*. Final states are visualized by means of a circle with a double border. A sequence $\sigma \in \Sigma^*$ is in the automaton's language if $((q^0,\sigma(1)),q'),((q',\sigma(2)),q''),...,((q''',\sigma(|\sigma|)),q^f) \in \delta$ and $q^f \in F$. From a graphical perspective, any sequence corresponding to a path from the initial state to a accepting state is in the automaton's language. For example, in Figure 2.6a the sequences $\langle a, c, d\rangle$ and $\langle a, b, b, c, b, d\rangle$ are in the language of the automaton.

In some cases, it is more likely that, within a state, a certain label occurs than another label. To this end, we additionally define probabilistic automata, which extend the notion of non-deterministic automata by including an additional transition probability function.

Definition 2.9 (Probabilistic Automaton). *Given a* (non-)deterministic automaton $A = (Q, \Sigma, \delta, q^0, F)$*. A probabilistic automaton* (PA) *is a* 6*-tuple* $(Q, \Sigma, \delta, q_0, F, \gamma)$*, where,* $\gamma \colon Q \times \Sigma \times Q \to [0,1]$ *is the transition probability function.*

For the probability function we require:

1. *If an arc labelled a connects q to q', then the corresponding probability is non-zero:*

$$\forall q, q' \in Q, a \in \Sigma \left(q' \in \delta(q, a) \Leftrightarrow \gamma(q, a, q') > 0 \right) \tag{2.17}$$

2. *The sum of probabilities of outgoing arcs of a state* $q \in Q \setminus F$ *equals one:*

$$\forall q \in Q \setminus F \left(\exists a \in \Sigma \left(\delta(q, a) \neq \emptyset \right) \implies \left(\sum_{a \in \Sigma} \sum_{q' \in Q} \gamma(q, a, q') \right) = 1 \right) \tag{2.18}$$

3. *The sum of probabilities of outgoing arcs of an accepting state* $q \in F$ *is smaller than one:*

$$\forall q \in F \left(\exists a \in \Sigma \left(\delta(q, a) \neq \emptyset \right) \implies \left(\sum_{a \in \Sigma} \sum_{q' \in Q} \gamma(q, a, q') \right) < 1 \right) \tag{2.19}$$

For a given state $q \in Q$ *and label* $a \in \Sigma$*, we denote the conditional probability of observing label a, whilst being in state q, as* $P(a \mid q)$*, where:*

$$P(a \mid q) = \sum_{\{q' \in Q \mid q' \in \delta(q,a)\}} \gamma(q, a, q') \tag{2.20}$$

Consider Figure 2.6b in which we depict an example probabilistic automaton. The underlying structure of the automaton is equal to the automaton presented in Figure 2.6a. In this case however, when we are in state s_2, label b occurs with probability $\frac{2}{3}$, whereas label c occurs with probability $\frac{1}{3}$. Clearly, we are able to compute the probability of the occurrence of a certain sequence of the automaton's language as well, i.e. since the transitions are independent we just need to multiply the probability of occurrence of each transition related to the label present in the sequence. For example, the probability of the sequence $\langle a, c, d\rangle$ is $1 \times \frac{1}{3} \times \frac{1}{3} = \frac{1}{9}$.

Observe that an accepting state is allowed to have outgoing arcs. For example, consider Figure 2.7, in which accepting state q_5 describes that an activity with label e can be observed with probability $\frac{1}{2}$. We accordingly define the probability of not observing any label for such an accepting state q as $P(\varnothing \mid q) = 1 - \sum_{a \in \Sigma} P(a \mid q)$. Observe that, Equation 2.19 ensures that $P(\varnothing \mid q) > 0$, i.e. $P(\varnothing \mid q) = 0$ contradicts the fact that q is an accepting state.

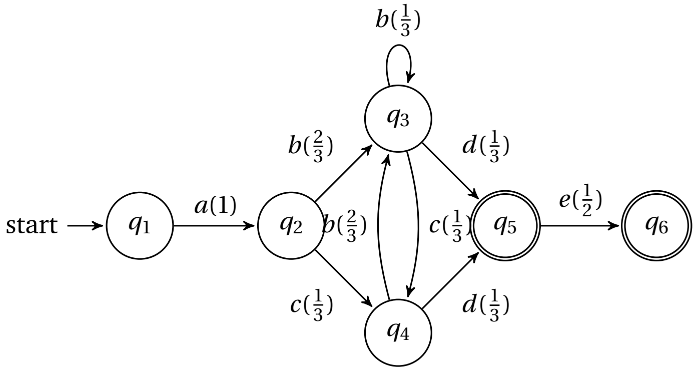


Figure 2.7: Probabilistic automaton in which the accepting state has an outgoing arc. The accepting state q_5 describes that an activity with label e can be observed with probability $\frac{1}{2}$.

2.3 Event Data

Within process mining, we explicitly study event data recorded during the execution of a certain (business) process. An *event* is considered any data record that is generated during the execution of a process at the lowest possible level of granularity. As such, we consider events to be the *atoms of process mining data*. In practice, the granularity of an event depends on the application domain as well as the way that the behaviour is recorded. In some cases, an event simply describes what activity of the process was executed at what point in time. In other cases, events describe different stages of the execution of a single activity, e.g. events relate to, amongst others, *scheduling*, *starting*, *suspending*, *resuming* and *completing* the activity. Moreover, an event typically records additional information such as the resource that executed the corresponding activity, a customer's account balance, a loan request amount, document ID's, etc.

2.3.1 Event Logs

As exemplified by the event log provided in Table 1.1 (in chapter 1, section 1.1 on page 4), an *event* is a value assignment of a set of (domain specific) attributes. For example, reconsider the first event depicted for Patient-id 1237. It assigns value 1533 to attribute *Event-id*, value 1237 to attribute *Patient-id*, value *ER Registration* to attribute *Activity* and so on. The number of available event attributes differs per application domain, i.e. data recorded in healthcare processes differs from the data recorded in a process originating from the financial domain. Moreover, within the same application domain, different data are logged across different processes, i.e. for different processes companies use different supporting information systems, possibly provided by different vendors. Finally, even within the same process some attribute values, are not available, or even defined, e.g. due to information system updates, when events are recorded. Therefore, when formalizing the notion of events, we do not explicitly characterize the associated attributes. We merely define an event as a tuple of data values that at least contains a *case identifier* which allows us to deduce to what *process instance* the event belongs. For example, in Table 1.1, the *Patient-id* column is a candidate to act as a case identifier, i.e. it allows us to deduce for what patient, an instance of the process is executed. Moreover, we assume that each event captured within the data relates to the execution of an *activity* within the process. Any other type of data is simply referred to as *additional payload*.

Definition 2.10 (Event, Event Projections)**.** *Let $\mathcal{E}$ denote the* universe of events. *Given an event $e \in \mathcal{E}$, we assume the existence of the following projection functions:*

- Case projection

  Let $\mathcal{C}$ denote the set of all possible case identifiers, i.e. a case identifier $c \in \mathcal{C}$ uniquely identifies the process instance to which the event belongs. We define the corresponding projection function $\pi_{c} \colon \mathcal{E} \rightarrow \mathcal{C}$.

- Activity projection

  Let $\mathcal{A}$ denote the set of all possible activities, i.e. an activity $a \in \mathcal{A}$ describes a well-defined activity within a process. We define the corresponding projection function $\pi_{a} \colon \mathcal{E} \rightarrow \mathcal{A}$.

- Arbitrary payload projection

  Let $\mathcal{D}$ denote the universe of data values for arbitrary event payload. Let A denote the universe of data attributes, and, let $d \in A$ denote such arbitrary data attribute. We define the corresponding projection function $\pi_{d} \colon \mathcal{E} \nrightarrow \mathcal{D}$. Observe that π_{d} is a partial function, as not every event is necessarily describing a value for the data attribute d.

Observe that, according to 2.10, the *case*- and *activity projection* are total functions, i.e. we are always able to access a corresponding value. Opposed to projection functions, we use an indicative symbol referring to the name of the data attribute to access it, rather than its index in the underlying Cartesian product. We primarily do so for clarity, i.e. from a technical point of view, the universe of events can be seen as a collection of tuples.

We use one specific attribute that represents the case identifier, i.e. c, which allows us to identify to what instance of the process the event belongs. However, note that, depending on the underlying process, multiple *case notions* exist. Consider that we store data related to students following several courses offered at a university, e.g. we record attendance of lectures, results for (practice) exams, etc. In such case, we are both able to use a student as a case identifier as well as an individual course. In case we use the student (represented by his/her student identifier) as a case identifier, we are considering the behaviour of students across several courses. However, if we consider the course as a case identifier, we study the behaviour of several students with respect to that course. For the purpose of this thesis, we simply assume that, in each possible process, for each event at least one primary, unique, case identifier exists. For any other attribute, except the activity attribute, the corresponding projection function is partial, i.e. for some events no value is available. Consider the first event depicted in Table 1.1, i.e. e_{1533}, for which we have, when taking the *Patient-id* as a case identifier: $\pi_{\texttt{c}}(e_{1533}) = 1237$, $\pi_{\texttt{a}}(e_{1533}) =$ ER Registration, $\pi_{\texttt{time stamp}}(e_{1533}) =$ 22-10-2014T11:15:41, $\pi_{\texttt{resource}}(e_{1533}) = 122A1$, $\pi_{\texttt{leucocytes}}(e_{1533}) = \emptyset$, etc.

During the execution of a process, events are recorded in the underlying information system, i.e. in *event logs*. Moreover, the case identifier typically ties a subset of events together, e.g. reconsider all events in Table 1.1 related to *Patient-id* 1237. Typically, even though not explicitly listed in 2.10, a *time-stamp* attribute is (omni)present as well. This allows us to order the events recorded into a sequence. In the context of this thesis, we do not assume that every event comprises of a timestamp, however, we do assume that there exists a strict partial order amongst the events. Observe that, such strict partial order can also be obtained by maintaining the order in which the events are returned by the data base query used to obtain them from the information system.

Definition 2.11 (Event Log). *Let $\mathscr{E}$ denote the universe of events and let $\prec$ be an associated strict partial order. An event log is a* strictly finite *sequence of events, i.e. $L \in \mathscr{E}^*$, as induced by the strict partial order $\prec$, such that:*

1. $\forall 1 \leq i < j \leq |L| \left(L(i) \prec L(j)\right)$*; The log respects the corresponding total order.*
2. $\forall i, j \in \{1, \ldots, |L|\} \left(L(i) = L(j) \implies i = j\right)$*; Each event occurs exactly once in the event log.*

Thus, as defined in 2.11, an event log is a finite sequence of events. A large majority of the existing process mining algorithms however considers an additional projection on the event log, i.e. the notion of a *trace*. A trace refers to a subsequence of events of the event log, all of which are related to the same case identifier, and thus, the same underlying process instance.

Definition 2.12 (Trace). *Let $L \in \mathscr{E}^*$ be an event log. A* trace*, related to a process instance as identified by case identifier $c \in \mathscr{C}$, is a sequence $\sigma \in \mathscr{E}^*$ for which:*

1. $\sigma \subseteq_* L$*; Traces are subsequences of event logs.*
2. $elem(\sigma) = \{e \in L \mid \pi_{\texttt{c}}(e) = c\}$*; The trace contains all events related to c*

Given an event log L, we let $traces(L) \in \mathscr{P}(\mathscr{E}^*)$ denote the collection of the traces described by the event log.

A *trace* is a sequence of events, related to the same case identifier, that respects the event log's strict partial order. Therefore, an alternative way to interpret an event log is as a *collection of traces*.

2.3.2 Control-Flow Perspective

The explicit notion of traces plays a central role in most process mining algorithms. Furthermore, several techniques ignore large parts of the additional payload present

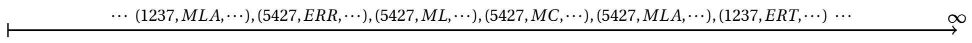

$\cdots$ (1237, MLA, $\cdots$), (5427, ERR, $\cdots$), (5427, ML, $\cdots$), (5427, MC, $\cdots$), (5427, MLA, $\cdots$), (1237, ERT, $\cdots$) $\cdots$
∞

Figure 2.8: Example event stream S.

in events and just focus on the activities performed for cases. This view is achieved by projecting each trace in an event log on its corresponding activities, i.e. given $\sigma \in L$ we apply $\pi_{\mathtt{a}_*}(\sigma)$. Thus, we transform each trace within the event log into a sequence of activities, which is known as the *control-flow perspective*. To this end, we define the notion of a *simple event log*, which explicitly adopts the aforementioned control-flow oriented projection.

Definition 2.13 (Event Log, Simple). *Let $L \in \mathscr{E}^*$ be an event log. A corresponding simple event log $\tilde{L} \in \mathscr{B}(\mathscr{A}^*)$ represents the control-flow perspective of event log L, i.e.*

$$\tilde{L} = \biguplus_{\sigma \in traces(L)} \pi_{\mathtt{a}_*}(\sigma) \tag{2.21}$$

Observe that a simple event log is a multiset of sequences of activities, as multiple traces of events are able to map to the same sequence of activities. As such, each member of a simple event log is referred to as a trace, yet each sequence $\sigma \in \mathscr{A}^*$ for which $\tilde{L}(\sigma) > 0$ is referred to as a *trace-variant*. Thus, in case we have $\tilde{L}(\sigma) = k$, there are k traces describing trace-variant σ, and, the cardinality of trace-variant σ is k.

In some cases, we are only interested in the set of activities $A \subseteq \mathscr{A}$ for which at least one event is present in the event log that describes such activities. We define such a set, given an event log $L \in \mathscr{E}^*$, as $A_L = \{a \in \mathscr{A} \mid \exists e \in_* L(\pi_{\mathtt{a}}(e) = a)\}$, or equivalently, in case of a simple event log $\tilde{L}$, we have $A_{\tilde{L}} = \{a \in \mathscr{A} \mid \exists \sigma \in \tilde{L}(a \in_* \sigma)\}$.

2.3.3 Event Streams

In this thesis, we assume the main type of data to be an *event stream*, rather than an event log. In essence, an event stream is a data stream (section 1.2) in which we assume the data packets to be events. We adopt the notion of online/real-time *event stream*-based process mining, in which the data is assumed to be an infinite sequence of events. Since in practice, several instances of a process run in parallel, we have no guarantees with respect to the arrival of events related to the same process instance. Thus, new events related to a certain process instance are likely to be emitted onto the stream in a dispersed manner. This implies, that our knowledge of the activities related to process instances changes over time. Consider Figure 2.8, in which we depict a few of the events (compact notation) that we also presented in the event log of Table 1.1. The first event visualized, i.e. represented as (1237, MLA, $\cdots$), refers to the event with event-id 1536 in Table 1.1, i.e. it relates to the execution of the *Measure Lactic Acid* activity executed for patient 1237. The four subsequent events relate to the treatment of the patient with patient-id 5427, after which the final event depicted in Figure 2.8 again relates to patient 1237.

We formally define an event stream, like an event log, as a sequence of unique events. However, we explicitly assume the stream to be of infinite size.

Definition 2.14 (Event Stream). *An event stream S is an* infinite sequence of unique events*, i.e. $S \in \mathscr{E}^*$ s.t. $dom(S) = \mathbb{N} \wedge \forall i, j \in \mathbb{N} \big(S(i) = S(j) \implies i = j\big)$.*

Observe that an event stream is infinite both in terms of the *past* and the *future*. This implies that, prior to the point in time at which we start observing an event stream, an infinite number of events was potentially already emitted onto the stream. Similarly, in the future, an infinite number of events will be emitted onto the event stream. Therefore, given an event stream $S \in \mathscr{E}^*$ and any $i \in \mathbb{N}$, in the remainder of this thesis, $S(i)$ relates to the i^{th} *event observed on the stream*.

Given an event stream S, we define $E_S = elem(S)$. In the context of this thesis we assume that the order of arrival of events defines a strict partial order on the events emitted, i.e. the stream characterizes strict partial order $(E_S, \prec)$. We furthermore assume that the order of arrival of events on the event stream is in line with the actual order of execution. We also assume that event arrival is an atomic operation and do not assume the existence of a multiple channel stream.

2.4 Process Mining

In this section, we present preliminary concepts, specific to the field of process mining, which are essential for this thesis. We provide a formal definition of process discovery algorithms, as well as commonly used data structures in process discovery. We furthermore present the notion of alignments, and finally, several quality metrics which are used for the evaluation of the different techniques presented in this thesis.

2.4.1 Process Discovery

In this section, we introduce preliminary concepts related to process discovery. In particular, we formalize process discovery as a function, mapping an event log into a process model. Secondly, we present common data abstractions, used by different process discovery algorithms.

As described in subsection 1.1.1, the main aim of offline process discovery is to discover a process model, given an event log. Given that an event log is defined as a sequence of events, we define process discovery as a function mapping a finite sequence of events to a process model of an a arbitrary formalism.

Definition 2.15 (Process Discovery Algorithm). *Let $\mathscr{E}$ denote the universe of events. A process discovery algorithm α is a function that, given a finite sequence of events, discovers a process model, i.e. $\alpha \colon \mathscr{E}^* \rightarrow \mathscr{M}$.*

Intermediate Representations

The majority of existing conventional process discovery algorithms share a common underlying algorithmic mechanism. As a first step, the event log is transformed into a data abstraction of the input event log, in this thesis alternatively referred to as an *intermediate representation*, on the basis of which they discover a process model.

Additionally, several process discovery algorithms actually use the same, or very similar, intermediate representations. In this section, we therefore formalize the *directly follows relation*, i.e. a commonly used intermediate representation. Furthermore, we briefly discuss alternative intermediate representations and refine process discovery in the context of intermediate representations.

The directly follows relation The *directly follows relation* can be considered as a cornerstone of process discovery algorithms, i.e. numerous discovery algorithms use it as a primary/supporting artefact, to discover a process model. As an illustrative example, consider 2.2, in which we illustrate the notion of the *directly follows abstraction*, i.e. as used by the *Alpha algorithm* [11].

Example 2.2 (The directly follows relation and the Alpha algorithm)**.** *Consider a* simple event log *describing only two types of traces, i.e.* $\tilde{L} = [\langle a,b,c,d\rangle, \langle a,c,b,d\rangle] \in \mathscr{B}(\mathscr{A}^*)$*. Given such an event log, the Alpha algorithm computes a* directly follows relation*, which counts the number of occurrences of direct precedence relations amongst the different activities present in the event log. Activity a is* directly followed *by activity b, written as* $a > b$*, if there exists a simple trace in the given event log of the form* $\sigma = \sigma' \cdot \langle a,b\rangle \cdot \sigma''$ *(here both* σ' *and* σ'' *are potentially empty, i.e.* ϵ*). Hence, in our example, we deduce* $a > b$*,* $a > c$*,* $b > c$*,* $b > d$*,* $c > b$*,* $c > d$*, which all occur once. Using these relations as a basis, the Alpha algorithm constructs a process model, in the form of a Petri net.*

As 2.2 shows, an event log is first translated into a *directly follows relation*, which is subsequently used to discover a process model. We formally define the directly follows relation in 2.16.

Definition 2.16 (Directly Follows Relation)**.** *Let* $L \in \mathscr{E}^*$ *be an event log. The directly follows relation* $>_L$*, is a multiset over* $\mathscr{A} \times \mathscr{A}$*, i.e.* $>_{\tilde{L}} \in \mathscr{B}(\mathscr{A} \times \mathscr{A})$*, for which, given* $a, b \in \mathscr{A}$*:*

$$>_L (a,b) = \sum_{\sigma \in_+ L} \left| \{ i \in \{1, \ldots, |\sigma - 1|\} \mid \pi_{\mathfrak{a}}(\sigma(i)) = a \wedge \pi_{\mathfrak{a}}(\sigma(i+1)) = b \} \right| \tag{2.22}$$

We write $a >_L b$ *if* $(a,b) \in_+ >_L$*, and* $a \not>_L b$ *if* $(a,b) \notin_+ >_L$*.*

Process discovery algorithms such as the Inductive Miner [78], the Heuristics Miner [121, 122], the Fuzzy Miner [66], and most of the commercial process mining tools use (amongst others) the directly follows relation as an intermediate structure.

Alternative representations Several process discovery algorithms have been proposed [22, 37, 106, 123, 131, 133, 137], that are inspired by language-based region theory [21]. Region theory and process discovery are conceptually close, i.e. the goal is to discover a process model from sequential data, yet typically differ on the requirements posed on the resulting process models. In these approaches, the event log is typically transformed into its *prefix-closure*. In some cases, the elements of the prefix-closure serve as a basis for an additional abstraction, e.g. constraints of an ILP. We present the exact characterization of these constraints in more detail in section 6.2.

Other work, translates the event log into an automaton [10].[4] Subsequently, on the basis of *state-based region theory*[21], such an automaton is transformed into a Petri net.

Process discovery with intermediate representations We refine conventional process discovery, i.e. a presented in 2.15, by splitting the discovery function α into two steps. In the first step, the event log is translated into the intermediate representation as used by the discovery algorithm. In the second step, the intermediate representation is translated into a process model. In the remainder we let $\mathbf{T_I}$ denote an *intermediate representation type*, whereas, $\mathcal{U}_{\mathbf{T_I}}$ denotes the set of all possible intermediate representations of type $\mathbf{T_I}$.

Definition 2.17 (Abstraction Function; Event Log). *Let* $\mathbf{T_I}$ *denote an arbitrary intermediate representation type. An abstraction function* $\lambda_{\mathbf{T_I}}$ *is a function mapping a multiset of sequences of activities to an intermediate representation of type* $\mathbf{T_I}$*.*

$$\lambda_{\mathbf{T_I}} : \mathscr{E}^* \to \mathcal{U}_{\mathbf{T_I}} \tag{2.23}$$

For example, we consider the concept of a *directly follows relation* as a specific type of intermediate representation. Any actual directly follows relation is thus part of the corresponding universe of directly follows relations.

Using 2.17, we define process discovery in terms of intermediate representations in 2.18.

Definition 2.18 (Process Discovery Algorithm - Intermediate Representation). *Let* $\mathbf{T_I}$ *denote an intermediate representation type. Let* $\mathscr{M}$ *denote the universe of process models. An intermediate representation based process discovery algorithm* $\alpha_{\mathbf{T_I}}$ *maps an intermediate representation of type* $\mathbf{T_I}$ *to a process model.*

$$\alpha_{\mathbf{T_I}} : \mathcal{U}_{\mathbf{T_I}} \to \mathscr{M} \tag{2.24}$$

Observe that the second step of the Alpha algorithm as described in 2.2, i.e. translating the directly follows abstraction into a Petri net, is an example instantiation of the abstraction function defined in 2.18. Every discovery algorithm that uses an intermediate representation internally can be expressed as a composition of the $\lambda_{\mathbf{T_I}}$ and $\alpha_{\mathbf{T_I}}$ functions. Thus, given an event log $L \in \mathscr{E}^*$ and an intermediate representation type $\mathbf{T_I}$, we obtain $\alpha(L) = \alpha_{\mathbf{T_I}}(\lambda_{\mathbf{T_I}}(L))$. For example, consider Figure 2.9 depicting the Alpha algorithm in terms of $\alpha_{\mathbf{T_I}}$ and $\lambda_{\mathbf{T_I}}$.

2.4.2 Alignments

Reconsider the the Petri net depicted in Figure 2.5. Furthermore, assume we are given a full trace of events, which, projected onto activities, looks as follows: $\langle a, b, c, d, e \rangle$. We observe that, by firing transitions t_1, t_2, t_3, t_5 and t_7, such sequence of activities is

[4]Also referred to as transition systems, yet typically, an initial state and accepting states are identified as well.

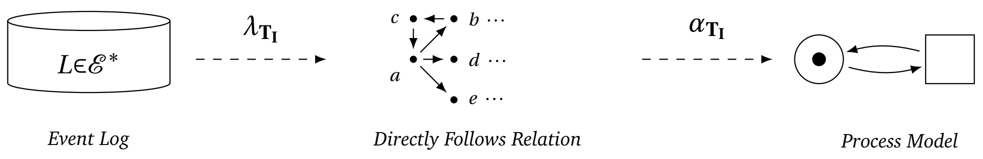


Figure 2.9: The Alpha algorithm in terms of its intermediate representation. Here, the intermediate representation is a *directly follows relation*.

γ_1:

a	b	c	d	e
t_1	t_2	t_3	t_5	t_7

γ_2:

x	a	$\gg$	d	e	z
$\gg$	t_1	t_3	t_4	t_7	$\gg$

Figure 2.10: Example alignments for $\langle a,b,c,d,e\rangle$ and $\langle x,a,d,e,z\rangle$ with the Petri net presented in Figure 2.5.

produced by the Petri net in Figure 2.5. Hence, to explain the trace of events in terms of the process model, it is rather easy to relate it to a sequence of transition firings which results in the same sequence of activity labels, after applying the Petri net's corresponding labelling function λ. Consider γ_1 in Figure 2.10, in which we present such a relation, i.e. an alignment of the simple trace $\langle a,b,c,d,e\rangle$ and the example net N_1 of Figure 2.5.

If we alternatively consider another example trace, e.g. $\langle x,a,d,e,z\rangle$, we observe several problems. For example, activities x and z are not labels of the Petri net, and thus never correspond to the execution of a transition within the Petri net. Furthermore, according to the Petri net, at least an activity c (represented by transition t_3) must be executed in-between activity a and d. *Alignments* allow us to identify and quantify the aforementioned problems, and moreover, allow us to express deviations in terms of the reference model. Conceptually, an alignment relates the execution of transitions in a Petri net and the activities observed in a simple trace $\sigma \in \mathscr{A}^*$ in a given (simple) event log. Observe Figure 2.10, in which we present alignments for the simple traces $\langle a,b,c,d,e\rangle$ and $\langle x,a,d,e,z\rangle$ with respect to the Petri net presented in Figure 2.5.

Alignments are sequences of pairs, e.g. $\gamma_1 = \langle (a,t_1),(b,t_2),...,(e,t_7)\rangle$. Each pair within an alignment is referred to as a *move*. The first element of a move refers to an activity of the trace, whereas the second element refers to a transition. The goal is to create pairs of the form (a,t) s.t. $\lambda(t) = a$, e.g. all moves in γ_1 are of this form. The sequence of activity labels in the alignment needs to equal the input trace (when ignoring the $\gg$-symbols). The sequence of transitions in the alignment needs to correspond to a $\sigma \in T^*$, such that, given the designated initial marking M_i and final marking M_f of the process model, we have $M_i \xrightarrow{\sigma} M_f$ (again ignoring the $\gg$-symbols). For the Petri net presented in Figure 2.5, we have $M_i = [p_i]$ and $M_f = [p_o]$. In some cases, we are not able to construct a move of the form (a,t) s.t. $\lambda(t) = a$. In case of trace $\langle x,a,d,e,z\rangle$, we are not able to relate x and z to any transition in the Petri net. Furthermore, we at least need to execute transition t_3, with activity label c (not present in the trace), in order to form a sequence of transitions that leads to marking M_f, starting from M_i. In such cases, we use the *skip-symbol* $\gg$ in either the activity- or the transition

$\gamma_2:$

x	a	$\gg$	d	e	z
$\gg$	t_1	t_3	t_4	t_7	$\gg$

$\gamma_3:$

x	a	$\gg$	$\gg$	d	e	z
$\gg$	t_1	t_2	t_3	t_5	t_7	$\gg$

Figure 2.11: Two possible alignments for $\langle x, a, d, e, z\rangle$ and N_1, as presented in Figure 2.5.

part of a move, in order to indicate that either an activity is observed that we are not able to relate to the execution of a transition, or, an activity was not observed that should have been observed, according to the process model. For example, consider γ_2 in Figure 2.10, which contains three skip-symbols. Verify that again, when ignoring skip-symbols, the sequence of activity labels equals the input trace, and, the sequence of transitions is valid firing sequence for M_i and M_f. If a move is of the form (a, t) we call this a *synchronous move*, $(a, \gg)$ is an *activity move* and $(\gg, t)$ is a *model move*.

Note that, in some cases, we need to construct activity moves, even though the label of the activity move is part of the Petri net. In the example, any simple trace of the form $\langle a, a, ...\rangle$, either has a corresponding alignment $\langle(a, t_1), (a, \gg), ...\rangle$ or $\langle(a, \gg), (a, t_1), ...\rangle$, i.e. an alignment of the form $\langle(a, t_1), (a, t_1), ...\rangle$ violates the fact that the transition part of the alignment corresponds to an element of the net's language, since such a trace cannot contain two executions of t_1. Furthermore, note that, some transitions have no observable activity label, i.e. transitions t with $\lambda(t) = \tau$. These transitions are often used for routing purposes, e.g. we use t_6 in the Petri net presented in Figure 2.5 to loop back in the process. Clearly, it is not possible to observe the execution of such a transition. Hence, we always construct moves of the form $(\gg, t)$, e.g. $(\gg, t_6)$ in the context of the Petri net presented in Figure 2.5. Even-though these moves are model moves, we often treat them as synchronous moves in the underlying algorithm that is used to compute alignments.

Definition 2.19 (Alignment [13]). *Let $\sigma \in \mathscr{A}^*$. Let $N = (P, T, F, \lambda)$ be a Petri net and let M_i, M_f denote N's initial and final marking. Let $\gg \notin \mathscr{A} \cup T \cup \Sigma \cup \{\tau\}$. A sequence $\gamma \in ((\mathscr{A} \cup \{\gg\}) \times (T \cup \{\gg\}))^*$ is an alignment if and only if:*

1. $(\pi_1(\gamma))_{\downarrow_{\mathscr{A}}} = \sigma$*; activity part (excluding $\gg$'s) equals σ.*
2. $(N, M_i) \xrightarrow{(\pi_2(\gamma))_{\downarrow_T}} (N, M_f)$*; transition part (excluding $\gg$'s) in Petri net language.*
3. $\forall (a, t) \in_* \gamma\, (a \neq \gg \vee t \neq \gg)$*; $(\gg, \gg)$ is not valid in an alignment.*

We let Γ denote the universe of alignments and let $\Gamma(N, \sigma, M_i, M_f)$ denote all alignments of N and σ given M_i and M_f.

Given the definition of alignments as presented in 2.19, several alignments, i.e. sequences of moves adhering to 2.19, exist for a given trace and Petri net. For example, consider alignment γ_3 depicted in Figure 2.11 which, according to 2.19 is an alignment of $\langle x, a, d, e, z\rangle$ and N_1 as well. The main difference between γ_2 and γ_3, i.e. both aligning trace $\langle x, a, d, e, z\rangle$ with N_1, is the fact that γ_2 binds the execution of t_4 to the observed activity d, whereas γ_3 binds the execution of t_5 to the observed activity d. Clearly, both explanations are possible, however, to be able to bind the executed activity d to t_5, alignment γ_3 requires the explicit execution of transition t_2 as well. Since activity b is not observed in the given trace, we observe the presence of move $(\gg, t_2)$ in γ_3, which is not needed in γ_2. Both alignments are feasible, however, we prefer alignment γ_2 over γ_3 as it minimizes non-synchronous moves, i.e. moves of the form $(a, \gg)$ or $(\gg, t)$.

As exemplified by alignments γ_2 and γ_3, we need means to be able to rank and compare alignments and somehow express our preference of certain alignments with respect to others. To this end we define a cost-function over the moves of an alignment. The cost of an alignment is simply the sum of the costs of its individual moves. Typically, synchronous moves are assigned a low, or even 0, cost. The costs of model- and activity moves are usually higher than the costs of synchronous moves. We formalize these notions in 2.20.

Definition 2.20 (Alignment cost function [13])**.** *Let $\mathscr{A}$ denote the universe of activities. Let $N = (P, T, F, \lambda)$ denote a labelled Petri net, where $\lambda\colon T \to \Sigma \cup \{\tau\}$ is the Petri net labelling function. Let $\gg \notin \mathscr{A} \cup \Sigma \cup \{\tau\}$. We define the move-cost-function as a function:*

$$c_m\colon (\mathscr{A} \cup \{\gg\}) \times (T \cup \{\gg\}) \to \mathbb{R}_{\geq 0} \tag{2.25}$$

Furthermore, given an instantiation of a cost-move-function c_m, we define alignment cost function

$$c_\Gamma\colon ((\mathscr{A} \cup \{\gg\}) \times (T \cup \{\gg\}))^* \to \mathbb{R}_{\geq 0} \tag{2.26}$$

We characterize $c_\Gamma^{c_m}$, for a given $\gamma \in ((\mathscr{A} \cup \{\gg\}) \times (T \cup \{\gg\}))^*$, as:

$$c_\Gamma^{c_m}(\gamma) = \sum_{i=1}^{|\gamma|} c_m(\gamma(i)) \tag{2.27}$$

Assume we assign cost 0 to synchronous moves and cost 1 to activity/model moves. In this case the cost of γ_1 is 0. The cost of alignment γ_2 is 3, whereas the cost of alignment γ_3 is 4. Hence, the cost of γ_2 is lower than the cost of γ_3 and we prefer it over γ_2. In general, we are able to use an arbitrary instantiation of c_m, however, in the context of this chapter we explicitly assume the usage of the *unit cost function*.

Definition 2.21 (Alignment unit cost function)**.** *The unit cost function of alignments is a function $c^{\mathbf{1}}_{move}\colon (\mathscr{A} \cup \{\gg\}) \times (T \cup \{\gg\}) \to \mathbb{R}_{\geq 0}$, where:*

1. *$c_m(a,t) = 0 \Leftrightarrow a \in \mathscr{A}, t \in T$ and $\lambda(t) = a$ or $a = \gg, t \in T$ and $\lambda(t) = \tau$,*
2. *$c_m(a,t) = \infty \Leftrightarrow a \in \mathscr{A}, t \in T$ and $\lambda(t) \neq a$,*
3. *$c_m(a,t) = 1$ otherwise.*

Since we assume unit-costs throughout this chapter, for simplicity, we omit the cost function sub- and superscript and simply write $c(\gamma)$, rather than $c_\Gamma^{c_m}(\gamma)$. We write $\gamma^\star$ to refer to *an optimal alignment*, i.e. $\gamma^\star = \mathbf{arg\,min}_{\gamma \in \Gamma(N, \sigma, M_i, M_f)} c(\gamma)$. Thus, an optimal alignment minimizes alignment costs. Consequently, computing an optimal

$$\overline{\gamma}_1 : \begin{array}{||c|c|c|} a & c & d \\ \hline t_1 & t_3 & t_4 \end{array} \qquad \overline{\gamma}_2 : \begin{array}{||c|c|c|c|} a & \gg & c & d \\ \hline t_1 & t_2 & t_3 & t_5 \end{array}$$

Figure 2.12: Two prefix-alignments for $\langle a,c,d\rangle$ and N_1.

alignment is simply defined as a minimization problem. It is important to note that optimality in alignments is not a unique property, i.e. multiple optimal alignments exist. Furthermore, in this thesis, we don't discuss actually finding optimal alignments in detail, i.e. we merely use the fact that an algorithm to find an optimal alignment exists and we use it as a black-box.

In some cases, an event log contains fragments of behaviour related to process instances that are not completed yet, i.e. processes that were still running when the event data was extracted. Furthermore, when considering the topic of this thesis, i.e. streams of events, it is likely that the behaviour we observe on the event stream largely relates to uncompleted process instances. To this end, prefix-alignments, i.e. a relaxed alternative to conventional alignments, can be used as an alternative. In essence, we relax requirement two of 2.19 in such a way that after executing the transition part of the alignment, the final marking M_f *can still be reached.*

Definition 2.22 (Prefix-Alignment [13]). *Let $\sigma \in \mathscr{A}^*$ be a (unfinished) sequence of activities. Let $N = (P,T,F,\lambda)$ be a Petri net with labelling function $\lambda\colon T \to \Sigma \cup \{\tau\}$ and let M_i, M_f denote N's initial and final marking. Let $\gg \notin \mathscr{A} \cup T \cup \Sigma \cup \{\gg\}$. A sequence $\overline{\gamma} \in ((\mathscr{A} \cup \{\gg\}) \times (T \cup \{\gg\}))^*$ is* a prefix-alignment *if and only if:*

1. $(\pi_1(\overline{\gamma}))_{\downarrow_{\mathscr{A}}} = \sigma$*; activity part (excluding $\gg$'s) equals σ.*
2. $\exists \sigma' \in T^* \left((N, M_i) \xrightarrow{(\pi_2(\overline{\gamma}))_{\downarrow_T} \cdot \sigma'} (N, M_f) \right)$*; the final marking M_f* can still be reached after firing the sequence of transitions described in the prefix-alignment in the given Petri net.
3. $\forall (a,t) \in_* \overline{\gamma} (a \neq \gg \vee t \neq \gg)$*;* $(\gg,\gg)$ *is not valid in a prefix-alignment.*

We let $\overline{\Gamma}$ denote the universe of prefix-alignments and let $\overline{\Gamma}(N,\sigma,M_i,M_f)$ denote all prefix-alignments of N and σ given M_i and M_f.

Consider Figure 2.12, in which we depict two example prefix-alignments of incomplete trace $\langle a,c,d\rangle$ and the running example Petri net of Figure 2.5. Observe that, for both alignments we need to either append $\langle t_7\rangle$ or $\langle t_8\rangle$ to obtain marking M_f, and thus, the *relaxed requirement 2* of 2.22 is satisfied. Similar to conventional alignments, several prefix-alignments exist that correctly align a prefix and a Petri net. Hence, we again need means to rank and compare prefix-alignments. For example, in Figure 2.12, we prefer $\overline{\gamma}_1$ over $\overline{\gamma}_2$, since it only contains synchronous moves whereas $\overline{\gamma}_2$ contains a (unnecessary) model move. Since a prefix-alignment, like a conventional alignment, is a sequence of moves, the cost of a prefix-alignment is defined in the exact same manner as the costs of conventional alignments, i.e. it is simply the sum of the costs of its individual moves. Observe that cost function c_Γ is, in 2.20, defined over a sequence of moves, and thus, given some prefix-alignment $\overline{\gamma}$, $c_{\Gamma}(\overline{\gamma})$ is readily defined. As a consequence, we again have the notion of *optimality*. For example, $\overline{\gamma}_1$ is an optimal prefix-alignment for $\langle a,c,d \rangle$ and N_1. In this case we denote an optimal prefix-alignment for a given $\sigma \in \mathscr{A}^*$ and Petri net N as $\overline{\gamma}^{\star}$.

2.4.3 Computing (Prefix-)Alignments

In [13] it is shown that computing an optimal, conventional, alignment is equivalent to solving a *shortest path problem on the state-space of the synchronous product net of N and σ*. The exact nature of such synchronous product net and an equivalence proof of the two problems is outside of the scope of this thesis, i.e. we merely use existing algorithms for the purpose of (prefix-)alignment computation as a black box. We therefore refer to [13] for these definitions and proofs. However, note that, to be able to compute optimal alignments, the process model, i.e. Petri net N, is required to be easy sound, cf. 2.5. Moreover, since within an easy sound Petri net, token generators potentially exist, in practice the costs of a synchronous move, and moves of the form $(\gg, t)$ where $\lambda(t) = \tau$ in particular, are set to a very small number $r \in \mathbb{R}_{>0}$, where $r << 1$ (much smaller than 1, close to 0).

Any shortest path algorithm to compute conventional alignments, is easily altered to compute prefix-alignments. In fact, in line with the relaxation of requirement two of 2.19, such alteration only consists of adding more states to the set of final states of the search problem. Hence, to compute optimal (prefix-)alignments we are able to use any algorithm designed for finding shortest paths in a graph. However, in [13, 127] the A^* algorithm [68] is proposed and evaluated. To compute conventional alignments, the states in the state-space that correspond to the final marking of the Petri net (together with explaining all behaviour of the trace) are used as the target states of the shortest-path-search. In case of prefix-alignments, any state that still allows for reaching a state that represents the final marking of the Petri net (again together with explaining all behaviour of the prefix) is considered a target state.

As a concrete implementation for the purpose of alignment calculation exists, we simply assume that we are able to utilize an oracle function $\overline{\omega}$, e.g. the algorithm proposed in [13]. Since we primarily focus on the notion of prefix-alignments in the remainder of this thesis, cf. chapter 7, we only define such an oracle for the computation of prefix-alignments.

Definition 2.23 (Prefix-alignment oracle). *Let $\mathscr{N}$ denote the universe of Petri nets, $\mathscr{A}$ the universe of activities and let $\mathscr{M}$ denote the universe of markings. A prefix-alignment oracle $\overline{\omega}$ is a function of the form:*

$$\overline{\omega} \colon \mathscr{N} \times \mathscr{A}^* \times \mathscr{M} \times \mathscr{M} \to \overline{\Gamma} \tag{2.28}$$

Where $\overline{\omega}(N,\sigma,M_i,M_f) \in \overline{\Gamma}(N,\sigma,M_i,M_f)$ and optimal for σ and N.

Observe that, the alignment oracle takes a Petri net, a trace and two markings, one representing the start- and one representing the target marking of the underlying search. We assume it is able to return an optimal prefix-alignment.

2.4.4 Measuring Quality in Process Mining

In this section, we describe the different means that are used to evaluate the results of the different algorithms proposed in this thesis. Some of these techniques and/or quality metrics originate from the more broad domain of data mining, whereas others are more process mining specific.

Evaluation of Binary Classification

A commonly studied problem in data mining, is *binary classification*. In binary classification, one aims to predict, for unlabelled data instances, a corresponding classification which is only allowed to be one of two values.

For example, assume that we are given certain data attributes related to the characteristics of a natural person, e.g. the type of sports he/she performs (if any), height, weight, etc. Based on this data, we construct an algorithm that predicts whether or not the person smokes. When the algorithm predicts a value `true`, this represents the prediction that the person is expected to smoke. When the algorithm predicts a value `false`, this represents the prediction that the person is not expected to smoke. Furthermore, in the context of this example, we assume that a predicted value `true` is alternatively referred to as a *positive prediction*.[5] If we now run the algorithm on a large body of data, of which we actually know whether the people described in the data smoke or not, we are able to categorize each prediction in the following way:

- *True Positive (TP)*

  The prediction is a positive label, and, the data element is actually of this type. In the context of our example, we predict a smoker to smoke.

- *False Positive (FP)*

  The prediction is a positive label, yet, the data element actually has a negative label. In the context of our example, we predict a non-smoker to smoke.

- *True Negative (TN)*

  The prediction is a positive label, and, the data element is actually of this type. In the context of our example, we predict a non-smoker not to smoke.

- *False Negative (FN)*

  The predict is a negative label, yet, the data element actually has a positive label. In the context of our example, we predict a smoker to be a non-smoker.

Clearly, a perfect predictor only predicts the classes TP and TN, i.e. it always predicts correctly. However, since such a perfect predictor often does not exist, there exist several quality metrics, derived from the label classification, that allow us to judge the quality of a predictor. In the remainder, let $|TP|$ denote the number of predictions of the TP class, let $|FP|$ denote the number of predictions of the FP class, etc. Here, we list the derived quality metrics that are of interest in the context of this thesis.

[5]Even-though this in no way implies that smoking needs to be perceived as being positive.

- $recall = \frac{|TP|}{|TP|+|FN|}$.

  Ratio of correctly predicted positive instances, relative to all positive instances in the data. Recall is a value in the range $[0,1]$ and is 0 if none of the instances is correctly predicted to be positive. It is 1, if all positive instances in the data are predicted as such.

- $precision = \frac{|TP|}{|TP|+|FP|}$.

  Ratio of correctly predicted positive instances, relative to all predicted positive instances in the data. Precision is a value in the range $[0,1]$ and is 0 if none of the instances is correctly predicted to be positive. It is 1, if no negative instance is falsely predicted to be positive.

- $F1\ Score = 2 \cdot \frac{precision \cdot recall}{precision+recall}$

  The *F1 Score* represents the harmonic mean of precision and recall. It allows us to measure the accuracy of a binary classifier.

To evaluate some experiments conducted in the context of this thesis, we use the aforementioned binary classification with the corresponding accuracy metrics. Clearly, the definition of TP, FP, etc., depends on the particular aim of the algorithm under study.

Process-Mining-Based Replay-Fitness and Precision

Apart from the metrics described in the previous section, i.e. originating from binary classification, the majority of the techniques presented in this thesis is evaluated on the basis of process mining quality metrics, i.e. as briefly presented in subsection 1.1.4. Here, we again highlight these quality metrics, and indicate, if applicable, their interaction with the different process mining artefacts, i.e. event data and/or process models.

- *Replay-Fitness*

  Quantifies to what degree a given process model describes the behaviour as captured in the event log. Let $L \in \mathscr{E}^*$ and let $\tilde{L} \in \mathscr{A}^*$ denote the corresponding simple view on the event log. Furthermore, let $M \in \mathscr{M}$ denote a model of arbitrary formalism over set of labels Σ, and let $\mathfrak{L}(M) \in \Sigma^*$ denote the model's language.[6] In case $\tilde{L} \subseteq \mathfrak{L}(M)$, the replay-fitness equals 1. In case $\tilde{L} \cap \mathfrak{L}(M) = \emptyset$, the replay-fitness equals 0. Furthermore, in case $\emptyset \subset \tilde{L} \setminus \mathfrak{L}(M) \subset \tilde{L}$, the replay-fitness value f is typically $0 < f < 1$, and gets closer to 1 for small values of $|\tilde{L} \setminus \mathfrak{L}(M)|$. The exact value fitness value in such a case typically depends on the underlying implementation of the metric.

- *Precision*

[6] Observe that the set of labels Σ used in a model and the set of possible activities $\mathscr{A}$ are not necessarily equal.

Quantifies to what degree a given process model describes behaviour that is not observed in the event log. Let $L \in \mathscr{E}^*$ and let $\tilde{L} \in \mathscr{A}^*$ denote the corresponding simple view on the event log. Furthermore, let $M \in \mathscr{M}$ denote a model of arbitrary formalism over set of labels Σ, and let $\mathfrak{L}(M) \in \Sigma^*$ denote the model's language. In case $\mathfrak{L}(M) \subseteq \tilde{L}$, the precision equals 1. Furthermore, in general, when $|\mathfrak{L}(M) \setminus \tilde{L}|$ increases, the precision value is expected to decrease. Again, the exact value depends on the underlying implementation of the metric.

- *Generalization*

  Quantifies to what degree a given process model generalizes beyond the behaviour observed in the event log. Let $L \in \mathscr{E}^*$ and let $\tilde{L} \in \mathscr{A}^*$ denote the corresponding simple view on the event log. Furthermore, let $M \in \mathscr{M}$ denote a model of arbitrary formalism over set of labels Σ, and let $\mathfrak{L}(M) \in \Sigma^*$ denote the model's language. We assume that M was constructed, in some way, on the basis of L. Finally, let $L' \in \mathscr{E}^*$ be an event log from the same underlying process, i.e. with respect to L, s.t. $L \cap L' = \emptyset$. In general, the larger $|\tilde{L}' \cap \mathfrak{L}(M)|$, the higher the generalizing ability of the model.

- *Simplicity*

  Quantifies to what degree a given process model is interpretable by a human analyst. Simplicity typically only considers the model, i.e. it ignores the underlying event log. There is no definitive simplicity measure defined, however, often metrics such as average number of transitions/places are used. Furthermore, graph complexity measures are used as well.

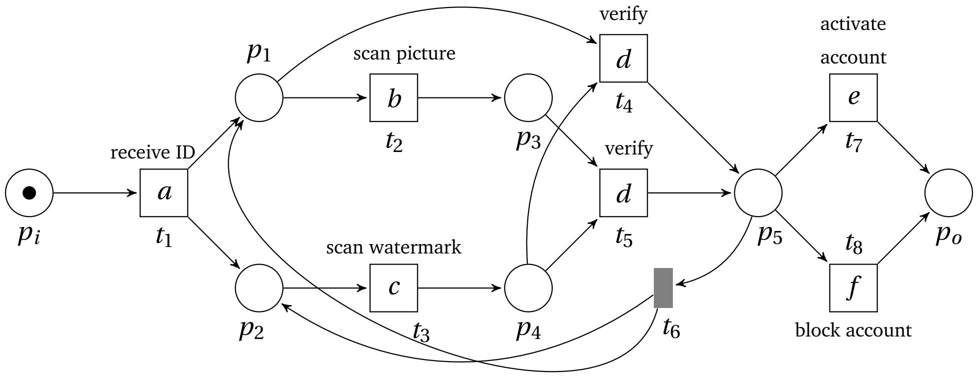
A more detailed discussion on the theoretical foundations and properties of the aforementioned metrics, in particular regarding replay-fitness, precision and generalization, we refer to [5, 109].

2.5 Running Example

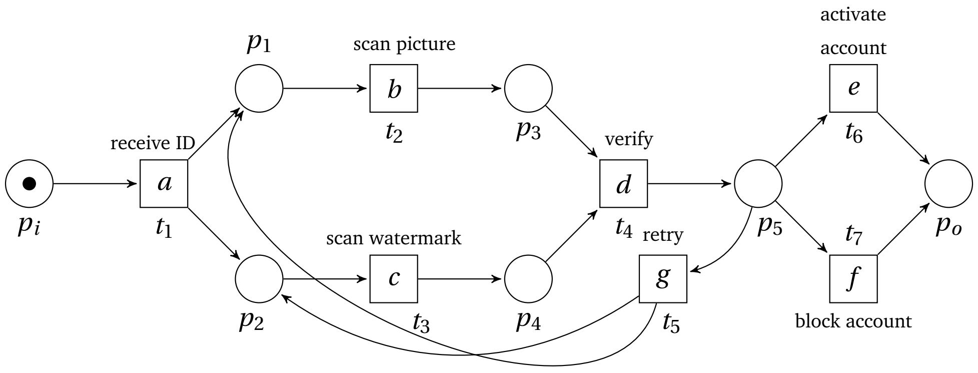
In this thesis, we use a simple running example to clarify, where needed, complex concepts. The running example is based on a simplified, fictive, process related to an online ID verification process, i.e. as already presented in Figure 2.5. However, we introduce two variants of the running example, as shown in Figure 2.13 on page 55.

The model in Figure 2.13a is the same as the model presented earlier, i.e. Figure 2.5. It describes that after *receiving the ID*, we are able to perform the *scan picture*- and *scan watermark* activity concurrently, of which *scanning the picture* is optional. After a *verification* step, we either *activate*- or *block the account*, or, we again *scan the picture and watermark*. However, the decision to rescan these artefacts is not explicitly captured within the information system.

The main difference between the models depicted in respectively Figure 2.13a and Figure 2.13b relates to the presence of duplicate transition labels and unobservable transitions. In Figure 2.13b we do not have an unobservable transition, i.e. such as t_6 in Figure 2.13a, nor duplicate transition labels, i.e. such as t_4 and t_5 in Figure 2.13a.



(a) Running example; A Petri net with unobservable transitions and duplicated transition labels.



(b) Simplified running example; A Petri net without unobservable transitions and only consisting of unique transition labels.

Figure 2.13: Running example processes related to an online ID verification process, modelled as Petri nets.

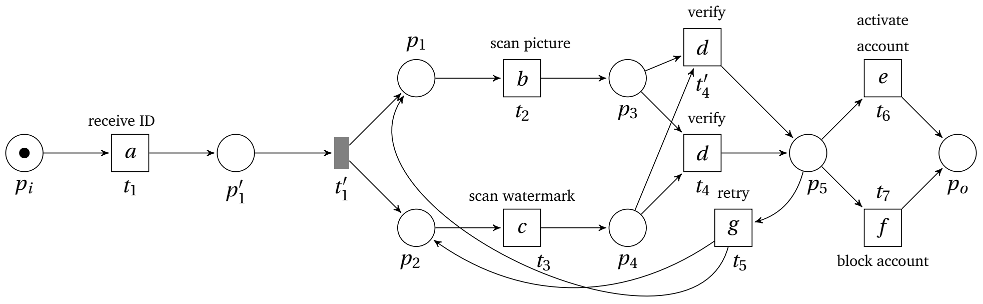


Figure 2.14: A Petri net with an unobservable transition and a duplicate transition label, describing the same behaviour as the Petri net depicted in Figure 2.13b.

As a consequence, in the Petri net of Figure 2.13b, we always need to execute both the picture and the watermark scan. Moreover, when rescanning the picture and watermark, we always observe a preceding *retry* activity.

Finally, note that, it is also possible to construct a Petri net with similar behaviour with respect to Figure 2.13b, that does have silent transitions and/or duplicate labels. For example, consider Figure 2.14, in which we present such a Petri net. In this case, we have a (non)-functional unobservable transition t'_1 in the beginning of the process, i.e. after transition t_1. Moreover, transition t'_4, like transition t_4, describes label d.

Chapter 3

Efficient Event Storage: A Primer for General Purpose Process Mining

Conventional process mining techniques were designed to use event logs as an input. As such, these techniques assume their source of data to be finite and static, rather than infinite and dynamic and/or evolving. As a result, a direct adoption of offline process mining techniques to the domain of event streams is not possible. In this chapter, we propose a generic formal model that allows us to transform the events emitted on an event stream into an event log. Such an event log can subsequently be used by any arbitrary process mining algorithm. Moreover, the vast majority of techniques presented in the subsequent chapters use and/or extend the formalism presented in this chapter. We present several instantiations of the formal model using existing data storage techniques ranging from different areas of data stream mining. We additionally present a memory efficient instantiation that explicitly exploits behavioural similarity among the different running process instances.

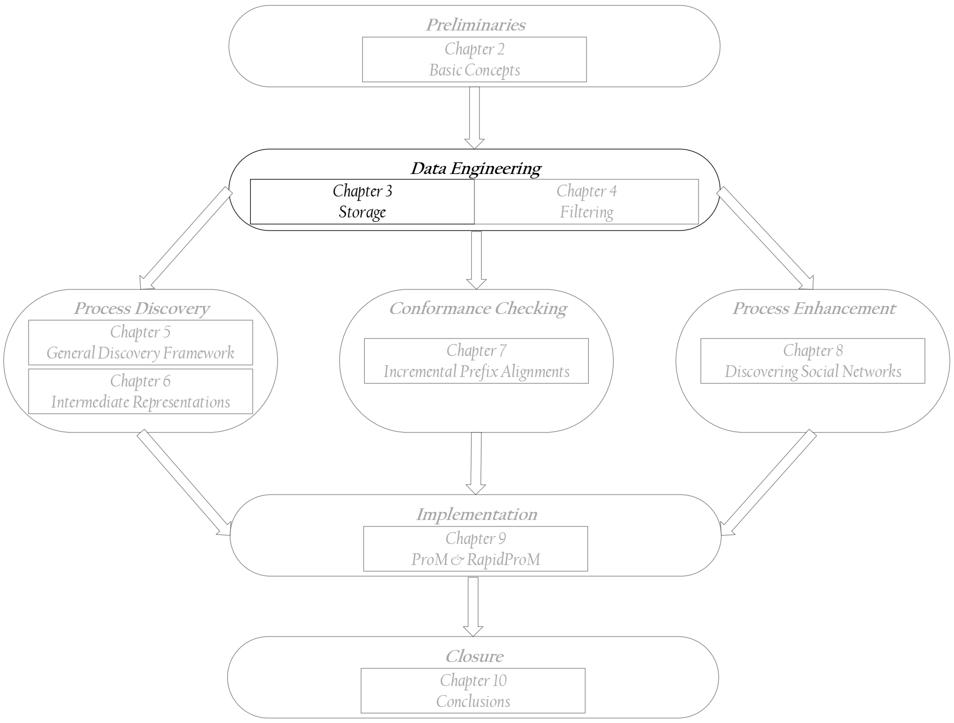


Figure 3.1: The contents of this chapter, i.e. efficient event data storage techniques, highlighted in the context of the general structure of this thesis.

3.1 Introduction

The vast majority of offline process mining techniques is defined in the context of event logs. Recall that an event log is defined as a finite sequence of events induced by an underlying *strict partial order*, cf. 2.11 on page 42. Similarly, an event stream is a sequence of events, yet it is of a strictly infinite fashion. Consequently, a natural approach to lift existing process mining techniques to the domain of online event stream based process mining, is to directly store the events emitted onto the event stream in memory.

Storing the stream as a whole allows offline process mining algorithms to be applied directly, as we store the same type of artefact, i.e. a sequence of events. Note however, that this potentially requires an infinite amount of memory, i.e. it violates the requirements as defined in the general streaming data model, cf. section 1.2.

To avoid the need for infinite memory, in this chapter, we present several approaches to utilize existing methods and algorithms, originating from the field of data stream mining/analysis, for the purpose of stream-based event storage. In particular, we formalize the notion of an *event store* and show that the methods discussed comply to the proposed formalization. An event store essentially describes a subsequence of all events observed on the event stream so far, and thus, represents a finite view on the stream under study. We additionally present new means to efficiently store event data originating from the execution of a process by exploiting control-flow based behavioural similarity among different running process instances.

We evaluate the proposed approach by means of several experiments. In particular, we focus on the impact of the newly proposed storage technique with respect to. the quality of the behavioural structure it allows us to deduce. Second, we investigate the impact with respect to. memory usage. Our experiments show that we are able to achieve comparable quality when using the newly proposed technique whilst needing a considerably smaller memory footprint.

The remainder of this chapter is organized as follows. In section 3.2, we formalize the notion of event stores, which represent subsequences of the events observed thus-far on the input event stream. In section 3.3, we present several data storage oriented techniques, originating from the data streaming domain, that fit the event store formalization. In section 3.4, we present a newly designed storage technique that exploits behavioural similarity amongst running process instances to decrease memory usage and enhance process mining results. In section 3.6, we discuss related work. Finally, section 3.7, concludes this chapter.

3.2 Event Stores

Throughout this chapter, we present several approaches that allow us to temporarily store events emitted on an event stream and translate the corresponding set of events into an event log. In this section, we present the notion of an *event store*, which formalizes and generically captures the techniques presented.

We consider an event stream $S \in \mathscr{E}^*$ on which we, over time, receive new events.

Table 3.1: Requirements and desiderata for instantiations/implementations of event stores.

Requirements	
Requirement	*Description*
RQ 1	Events in an event store respect the order as imposed by the stream.
Desiderata	
Desideratum	*Description*
DS 1	Events are not re-inserted after deletion.
DS 2	Removal of events respects the order as imposed by the underlying process instance.

We temporarily store these events in an *event store* Φ, which describes a time-evolving sequence of events, based on the events observed on the stream in the past. The sequence of events described by an event store is a finite subsequence of the input event stream. Ideally, the event store represents a sequence of events recently emitted on the event stream, however, this is not a necessity. We formalize the notion of an event store in 3.1.

Definition 3.1 (Event Store). *Let $\mathcal{E}$ denote the universe of events, let $i, N \in \mathbb{N}_0$ and let $S \in \mathcal{E}^*$ be an event stream. An* event store *of size N at time i, describes a subsequence of events, of maximal size N, of the first i events observed on S, i.e. $\Phi_N^i \colon \mathcal{E}^* \to \mathcal{E}^*$, s.t.:*

$$\Phi_N^i(S) \in \{\sigma \in \mathcal{E}^* \mid \sigma \subseteq_* S_{1\dots i} \wedge |\sigma| \leq N\} \tag{3.1}$$

An event store describes a subsequence of the event stream, bounded by a maximum size limit N. Observe that, even though we define an event stream as a function, we only characterize the set of eligible sequences for any possible event store Φ_N^i. As indicated in section 1.2, ideally N is polylogarithmic in the size of the stream, as in such case, memory consumption grows with a strictly slower rate as the stream's size. However, in practice, we are also able to base N on the amount of available memory. This ensures that, even though the input sequence of an event store is potentially infinite, an event store describes a strictly finite sequence of events.

Observe that the fact that an event store represents a subsequence of its input, implicitly poses a strict behavioural requirement on any instantiation/implementation of an event store, i.e. the original order of the events within the stream needs to be maintained. Apart from this strict behavioural requirement, we propose the following *desiderata* for any instantiation/implementation of an event store.

1. When we remove an event from the event store, it remains removed forever, i.e. it cannot be reinserted (e.g. from secondary storage).

2. Removal of events related to the same process instance is performed in-order, i.e. an event e related to case identifier c that is observed on the stream, prior to another event e' that is also related to case identifier c is removed before, or, at the same time as e'.

We list the requirements and desiderata for event stores in Table 3.1.

Even though an event store is allowed to describe an arbitrary subsequence of its input, we additionally define an *event store update function*. We utilize such function to define event stores incrementally, i.e. for input stream S we compute an event store at data point i, based on the previous event store at $i-1$ combined with (new) event $S(i)$.

Definition 3.2 (Event Store Update Function). *Let $\mathscr{E}$ denote the universe of events, and let $N \in \mathbb{N}_0$. An* event store update function $\overrightarrow{\Phi}_N$ *is a function $\overrightarrow{\Phi}_N : \mathscr{E}^* \times \mathscr{E} \rightarrow \mathscr{E}^*$, s.t. given a stored sequence $\sigma \in \mathscr{E}^*$ and new event $e \in \mathscr{E}$, $\overrightarrow{\Phi}_N(\sigma, e) \subseteq_* \sigma \cdot \langle e \rangle$, and $|\overrightarrow{\Phi}_N(\sigma, e)| \leq N$.*

Observe that the update function, even though defined for arbitrary input sequences, typically operates on the previously stored sequence in the underlying event store. Moreover, it yields, given a sequence of events and a new event, a new sequence of events. Furthermore, this new sequence is a subsequence of the input sequence concatenated with the new event. Given the notion of an event store and an event store update function, we aim to characterize the event store for element i in terms of an update of the event store at $i-1$, i.e.

$$\Phi^i_N(S) = \overrightarrow{\Phi}_N(\Phi^{i-1}_N(S), S(i)) \tag{3.2}$$

An event store update function needs to be able to compute $\Phi^i_N(S)$ on the basis of $\Phi^{i-1}_N(S)$ (potentially by updating the underlying data structure representing $\Phi^{i-1}_N(S)$), the newly received event $S(i)$, and, possibly the value of i. Given that we incrementally compute event stores for increasing $i \in \mathbb{N}_0$, we are able to quantify any *new member* $\Phi^{i+}_N \in \mathscr{P}(\mathscr{E})$ and/or *removed member(s)* $\Phi^{i-}_N \in \mathscr{E}^*$ of the event store for any of such i. Therefore, given $\Phi^i_N(S)$ which is, incrementally, based on $\Phi^{i-1}_N(S)$, we define:

$$\Phi^{i+}_N = elem(\Phi^i_N(S)) \setminus elem(\Phi^{i-1}_N(S)) \tag{3.3}$$

$$\Phi^{i-}_N = \sigma \in \mathscr{E}^* \ s.t. \ \sigma \subseteq_* \Phi^{i-1}_N \wedge \forall e \in_* \Phi^{i-1}_N \left(e \notin^* \Phi^i_N \implies e \in \sigma \right) \tag{3.4}$$

Observe that, when explicitly using/implementing the incremental update function, cf. 3.2, then by definition $0 \leq |\Phi^{i+}_N| \leq 1$. Hence, either $\Phi^{i+}_N = \emptyset$, or, it is a singleton set containing the i^{th} element, i.e. $\Phi^{i+}_N = \{S(i)\}$. The elements of Φ^{i-}_N are those elements that are no longer part of Φ^i_N, yet that were part of Φ^{i-1}_N. Moreover, it contains these events in order, i.e. adhering to their original position in the event stream (as implicitly maintained by the event store).

Observe that, as mentioned in subsection 2.3.1, most process mining algorithms operate on the notion of traces, rather than (partial) orders of events. In essence, such trace is just a projection of the (partial) order of events onto sequences of events sharing the same case identifier. Clearly, we are able to perform such projection on an event store as well. We, therefore, assume the existence of an additional function $\Phi^i_{\mathscr{C},N} : \mathscr{E}^* \times \mathscr{C} \rightarrow \mathscr{E}^*$ that allows us to fetch the sequence of events related to a given case identifier $c \in \mathscr{C}$. In particular, given $c \in \mathscr{C}$ and an event store Φ^i_N, we enforce:

1. $\Phi^i_{\mathscr{C},N}(S, c) \subseteq_* \Phi^i_N(S)$

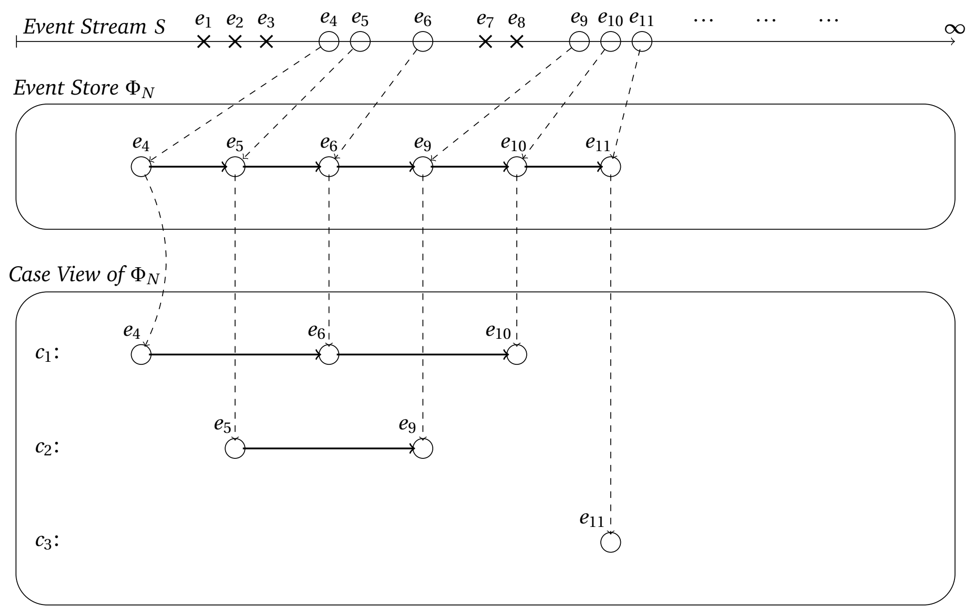

Event Stream S
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9 e_{10} e_{11} … … …
∞
Event Store Φ_N
e_4 e_5 e_6 e_9 e_{10} e_{11}
Case View of Φ_N
c_1: e_4 e_6 e_{10}
c_2: e_5 e_9
c_3: e_{11}

Figure 3.2: Schematic overview of the notion of an event store Φ^i_N and its corresponding case projection (with $i = 11$ and arbitrary size $N \geq 6$).

2. $\forall e \in_* \Phi^i_N(S)(\pi_{\mathsf{c}}(e) = c \implies e \in_* \Phi^i_{\mathscr{C},N}(S,c))$

3. $\forall e \in_* \Phi^i_{\mathscr{C},N}(S,c)(\pi_{\mathsf{c}}(e) = c)$

In the remainder, we simply write $\Phi^i_N(S,c)$ for $\Phi^i_{\mathscr{C},N}(S,c)$, i.e. we omit the $\mathscr{C}$ subscript, as it is clear from the input arguments. Moreover, if in some case the value of i and N is of minor or zero influence, we omit these values/variables as well, i.e. we simply write $\Phi(S)$ or $\Phi(S,c)$. Observe that, in some instantiations of $\Phi(S)$ it is more natural to let the corresponding internal structures directly store the events on a *case level*. As such, when iteratively querying the collection of cases maintained within the event store, we do not need to additionally project the event store on the requested case identifier, i.e. we are able to directly access it from memory.

Consider Figure 3.2, in which we present a schematic overview of an event store, as well as its projection on the universe of case identifiers. Observe that in the example, we visualize $\Phi^{11}_N(S)$. Within the figure, events labelled $\times$ are not part of the event store, e.g. $e_1 \notin^* \Phi^{11}_N(S)$, whereas events marked $\bigcirc$ are part of the event store. We observe that $\Phi^{11}_N(S) = \langle e_4, e_5, e_6, e_9, e_{10}, e_{11}\rangle$, and moreover, $\Phi^{11}_N(S,c_1) = \langle e_4, e_6, e_{10}\rangle$, $\Phi^{11}_N(S,c_2) = \langle e_5, e_9\rangle$ and $\Phi^{11}_N(S,c_3) = \langle e_{11}\rangle$.

In line with simple event logs, cf. subsection 2.3.2, we define a simple event store $\tilde{\Phi}^i_N(S) \in \mathscr{B}(\mathscr{A}^*)$, which represents the control-flow perspective based projection of the event store Φ^i_N at time i. Similarly, we assume to be able to fetch such simple trace for a given case identifier $c \in \mathscr{C}$, i.e. $\tilde{\Phi}^i_N(S, c) \in \mathscr{A}^*$. It should be clear that, in principle, both simple event stores $\tilde{\Phi}^i_N$, as well as the case level event stores $\Phi^i_{\mathscr{C},N}$ presented earlier are derived artefacts of any event store, i.e. we are able to obtain them by (iteratively) applying a number of projections. However, as already mentioned, in some cases it is more efficient to primarily store the events emitted on the event stream indexed on a per-case basis.

3.3 Conventional Storage Techniques

In this section, we present several instantiations of event stores and associated update functions. The underlying techniques originate from the streaming domain, intended for general, stream-based, data storage. Therefore, different techniques are applicable in different settings, i.e. depending on the intent of the process mining analysis performed. In some cases, we need to slightly alter and/or extend the original algorithms to make them effectively implement an event store. We discuss four different types of storage, i.e. sliding windows, reservoirs, frequent item-set based approaches and time decay based approaches.

3.3.1 Sliding Windows

Arguably the easiest way to store a recent subset of events emitted onto an event stream is by means of a *sliding window*. Within a sliding window, recent behaviour, i.e. behaviour that is observed within a recent period of time, is maintained in-memory. We formalize the notion of a sliding window, in terms of the notion of an event store update function, in 3.3.

Definition 3.3 (Event Store Update Function; Sliding Window). *Let $\mathscr{E}$ denote the universe of events. Let $N \in \mathbb{N}_0$, we define a sliding window-based event store update function $\overrightarrow{\Phi}_{N,sw} \colon \mathscr{E}^* \times \mathscr{E} \rightarrow \mathscr{E}^*$, where, given $\sigma \in \mathscr{E}^*$,*

$$\overrightarrow{\Phi}_{N,sw}(\sigma, e) = \begin{cases} \sigma \cdot \langle e \rangle & \text{if } |\sigma| < N \\ \langle \sigma(|\sigma| - N + 2), \ldots, \sigma(|\sigma|), e \rangle & \text{otherwise} \end{cases} \tag{3.5}$$

Observe that, by definition, the result of applying $\overrightarrow{\Phi}_{N,sw}$ is always a subsequence of its input sequence concatenated with the newly described event. Moreover, regardless of the length of the input sequence of the update function, the resulting sequence is always of length N (or smaller if the input sequence's length is strictly smaller than N). Consider algorithm 3.1 in which we provide a concise algorithmic description of an event store based on the concept of a sliding window.

The algorithm expects an event stream S and a maximum window size N as an input. The algorithm maintains a list of events, which is initially empty. Whenever we receive a new event e at index i of the stream, we append it to the tail of the list. However, if $i > N$ we subsequently remove the head of the list. Note that, within the algorithmic description, the list `w` itself represents $\Phi^i_N(S)$.

Algorithm 3.1: `Sliding Window-Based Event Store`

```
input: $S \subseteq \mathcal{E}^*$, $N \in \mathbb{N}_0$
begin
    $i \leftarrow 1$;
    w $\leftarrow$ empty list of events;
    while $true$ do
        if $|w| = N$ then
            remove w(1);
        append $S(i)$ to w;
        $i \leftarrow i + 1$;
```

The sliding window, as described previously, is also known as a *sequence-based* sliding window, i.e. the stream/window itself defines what elements are part of the window. An alternative approach is to use the event arrival-time as a criterion for membership of the window. In such case, we only maintain all events with an arrival time within a time interval Δ with respect to. the current time t, i.e. all events e with $t - \Delta \leq \pi_{\texttt{timestamp}}(e) \leq t$. Note that, in case the number of events arriving within the specified time window Δ exceeds the available memory budget N, we resort to saving Observe that both models are static since either the number of elements to consider, or, the time interval to consider, is fixed.

When formalizing a time-based sliding window, we additionally need access to a global clock that allows us to assess the *current time*. The criterion for event deletion now changes into checking whether the time-difference of the time of arrival of the oldest events of the window and the current time exceeds the given time interval. Observe that, as is the case with sequence-based sliding windows, a time-based sliding window is a subsequence of its input.

There also exist variants of sliding window approaches that are *dynamic*, i.e. the window size changes over time [25]. However, the emphasis of such techniques, is on finding an accurate representation of the underlying generating data distribution that is subject to concept drift. This means, that as soon as a concept drift within the underlying distribution is detected, i.e. either sudden or gradual, the older parts of the window describing the distribution prior to the detected drift are removed. Note that finding and/or characterizing drifts in the context of event streams is very challenging due to the fact that events related to the same process instance arrive dispersed over the stream. This does not align well with the general model used in data streams where data items are often assumed to be independently sampled from the underlying distribution. Despite this challenge, dealing with concept drift in the context of event streams is not covered (extensively) within this thesis.

The relatively simple nature of the sliding window is, at the same time, the largest limiting factor from a process mining perspective. As indicated in section 1.3, we expect events related to multiple different process instances to arrive dispersed on the stream. As such, we do not have any guarantee with respect to the completeness of the traces present in the sliding window. It is likely that, at some point, we drop fragments behaviour for a certain process instance that is still ongoing, i.e. for which other events are present in the sliding window as well. Observe that this has a potential negative impact with respect to. process mining algorithms, as most algorithms assume the traces to represent complete executions of the underlying process.

3.3.2 Reservoir Sampling

The concept of *reservoir sampling* is presented by numerous authors [76, Section 3.4.2] [88, 115]. It is a sampling method that allows us to pick a random sample of size N out of a set consisting of K elements (where $K > N$), in one-pass. In particular, the assumption is that the exact value of K is unknown, and, it is inefficient to determine it. The general idea of the approach is to maintain a reservoir, i.e. typically simply an array, of size N. Any algorithm maintaining a reservoir is a *reservoir sampling algorithm* if, after inspecting and processing the i^{th} element of the total of unknown K elements with $i > N$, the reservoir represents a *true random sample* of size N of the i items seen so far. Therefore, by induction, after receiving all K items, the reservoir is a true random sample of all K items, even though K is potentially infinite.

The concept of reservoir sampling, by its sheer definition, aligns well with the data stream model, i.e. we aim at one-pass handling of events whilst using finite, ideally bounded, memory. The standard approach of reservoir sampling is to place the first N elements of the stream in the reservoir. Subsequently, element $i+1$ with $i \geq N$ is added with probability $\frac{N}{i+1}$. Observe that, for increasing i, the probability of addition decreases, which ensures that after completion, in the finite scenario, the reservoir in fact represents a true random sample of size N of the items seen so far. If we decide to add an element to the reservoir, its index is determined randomly, using a uniform distribution over $\{1,...,N\}$, i.e. $\texttt{unif}\{0,N\}$. Consider 3.4, in which we formalize reservoir sampling in terms of an event store update function $\overrightarrow{\Phi}_N$.

Definition 3.4 (Event Store Update Function; Reservoir Sampling)**.** *Let $\mathscr{E}$ denote the universe of events, let $i,N \in \mathbb{N}_0$ and let $e \in \mathscr{E}$, $\sigma \in \mathscr{E}^*$. Furthermore, let $r \in \mathit{unif}\{1,i\}$ denote a random variable drawn from a uniform discrete distribution with ranges 1 and i and let $j \in \mathit{unif}\{1,|\sigma|\}$ denote a random variable drawn from a uniform discrete distribution with ranges 1 and $|\sigma|$. We define a reservoir sampling-based event store update function $\overrightarrow{\Phi}_{N,rs}$ as a function $\overrightarrow{\Phi}_{N,rs}\colon \mathscr{E}^* \times \mathscr{E} \to \mathscr{E}^*$, where*

$$\overrightarrow{\Phi}_{N,rs}(\sigma,e) = \begin{cases} \sigma \cdot \langle e \rangle & \text{if } |\sigma| < N \\ \sigma & \text{if } |\sigma| \geq N \wedge r > N \\ \langle \sigma(1),...,\sigma(j-1),\sigma(j+1),...,\sigma(|\sigma|)\rangle \cdot \langle e \rangle & \text{otherwise} \end{cases} \tag{3.6}$$

Observe that the update function requires random variable r, drawn from a discrete uniform distribution, i.e. $\texttt{unif}\{0,...,i\}$, where i is used to determine the probability of inclusion, i.e. $\frac{N}{i}$. Note that, when using the update function on top of an event store, i.e. $\Phi_N^i(S) = \overrightarrow{\Phi}_N(\Phi_N^{i-1}(S), S(i))$, the value of i used in the update function equals the index of the event store. In case the input sequence is of length strictly smaller than N

Algorithm 3.2: `Reservoir Sampling-Based Event Store`

```
input: $S \subseteq \mathscr{E}^*$, $N \in \mathbb{N}$
begin
    $i \leftarrow 0$;
    a $\leftarrow \epsilon$;                                        // reservoir
    w $\leftarrow \epsilon$;                          // internal sliding window
    while $true$ do
        $i \leftarrow i + 1$;
        $e \leftarrow S(i)$;
        if $|a| < N$ then
            append $e$ to both a and w;
        else
            $r \leftarrow$ discrete random value from unif$\{1, i\}$;
            if $r \leq N$ then
                $j \leftarrow$ discrete random value from unif$\{1, N\}$;
                remove a$(j)$ from w;
                a$(j) \leftarrow e$;
                append $e$ to w;
```

the update function returns the sequence concatenated with the new event. If this is not the case and the random variable r exceeds the value of N we simply return the input sequence. In any other case, i.e. the sequence size is at least N and $r \leq N$, we remove the element of the sequence at position j and include the new event as the last element of the returned sequence. Observe that, by iteratively applying the update function, we are guaranteed that the corresponding event store never exceeds size N.

Consider algorithm 3.2, in which we present an algorithmic description of a reservoir sampling based event store. The reservoir itself is represented by an array a, which we keep on filling until its size equals N. Subsequently we sample variable r from the uniform distribution over range $\{1, ..., i\}$, i.e. unif$\{1, i\}$. Whenever we obtain a value $r \leq N$, we sample variable j from the uniform distribution over range $\{1, ..., N\}$, i.e. unif$\{1, N\}$, and replace the element at position j in a. Due to the random nature of a reservoir, a is not a valid instantiation of an event store, i.e. it violates *RQ 1*, cf. Table 3.1. We, therefore, maintain an internal sliding window w that actually captures the event store. When we remove a certain event from the reservoir, we also remove it from w and append the new event to w.

The aforementioned approach indeed allows us to construct an event store, using reservoir sampling as an underlying basis. Observe that, contrary to the use of a sliding window, it is not necessarily meaningful to adopt such a reservoir in the context of process mining. Assume that we add an event at position $i > N$ related to some process instance identified by case identifier $c \in \mathscr{C}$. Also assume that we decide not to add the next event e' on the stream related to the same process instance. Furthermore, we do decide to add the subsequent event e'', again related to the same process instance. If we subsequently derive the trace related to the case identifier $c \in \mathscr{C}$, it is not containing e', whereas ideally, it should. In fact, from a certain point of view, this may even be regarded as faulty. It moreover violates *DS 2* of Table 3.1.

We therefore alternatively propose to perform the sampling based on the case-id's present in events, rather than for each event individually. We hence maintain a reservoir of case-id's, and, additionally incorporate an internal sliding window to represent Φ_N^i. When a new event arrives we check whether its case-id is present in the reservoir. If this is the case, we forward the event to the sliding window. If the case-id is not in the reservoir, yet we still have open spaces in the reservoir, we add the event's case-id to the reservoir and subsequently forward the event to the sliding window. However, if the maximum capacity of the reservoir is reached, we decide whether the case-id needs to be added to the reservoir or not, i.e. we add it with probability $\frac{N}{i+1}$. If we decide not to add it, nothing happens, i.e. the event is not forwarded to the internal sliding window. If we decide to add it, we replace, randomly, one of the existing case-id's in the reservoir with the new case-id. We subsequently traverse the sliding window and remove all events related to the removed case-id. Finally, we forward the newly received event to the internal sliding window.

The internal sliding window of the approach represents the event store maintained by the reservoir sampling-based approach. However, when replacing a case-id, we need to traverse the internal sliding window and remove each event that corresponds to the removed case-id. Moreover, when translating the reservoir based event-level store to a case level, we need an additional iteration through the sliding window.

We therefore alternatively propose to implement $\Phi_{\mathscr{C},N}^i$ directly, by grouping the events that are eligible for storage in the event store, i.e. according to the fact that their case-id is present in the reservoir, by case. The main idea is to maintain a separate list of events for each case-id in the reservoir. In this case, when a new event arrives and its case-id is present in the reservoir, we append the event to the list that is maintained for its case-id. If the event triggers removal of a case-id in the reservoir, the event is the new head element of a new list that is stored corresponding to its case-id. Whenever an event's case-id is not present in the reservoir and neither added to the reservoir, the event is ignored.

Consider algorithm 3.3 on page 68 in which we present an algorithmic description of applying reservoir sampling for the purpose of directly storing a case-level event store. The algorithm expects an event stream $S \in \mathscr{E}^*$ as an input and two additional parameters $k, N \in \mathbb{N}$, subject to $k < N$, and ideally $N \mod k = 0$. Here, k represents the number of positions in the reservoir, i.e. representing the number of process instances maintained, and N represents the maximum number of events present in the event store. For each incoming event, the algorithm checks whether it already maintains an entry related to the event's case-id. If this is the case, it simply appends the new event to the sequence of events already stored for that case. To ensure strictly using finite memory, we limit the length of such sequence to $\frac{N}{k}$. Thus, whenever the new event yields a sequence length exceeding $\lfloor \frac{N}{k} \rfloor$, we remove the head element of the list. As such, the algorithm needs $O(N)$ memory.

The algorithm as presented does not allow us to reconstruct the event store Φ_N^i,

Algorithm 3.3: `Reservoir Sampling-Based Event Store (Case Level)`

```
input: $S \subseteq \mathscr{E}^*$, $N, k \in \mathbb{N}$
begin
    $i \leftarrow 0$;
    $a_c \leftarrow \epsilon$;                                  // sequence of case-id's
    $a_\sigma \leftarrow \epsilon$;   // sequence of sequences of events (max. length $\lfloor \frac{N}{k} \rfloor$)
    while true do
        $i \leftarrow i + 1$;
        $e \leftarrow S(i)$;
        if $\pi_c(e) \in_* a_c$ then
            $j \leftarrow$ index of $c$ in $a_c$;
            $a_\sigma(j) \leftarrow a_\sigma(j) \cdot \langle e \rangle$;
            if $|a_\sigma(j)| > \lfloor \frac{N}{k} \rfloor$ then
                $a_\sigma(j) \leftarrow a_\sigma(j)_{2 \ldots |a_\sigma(j)|}$;
        else
            if $|a_c| < k$ then
                $a_c \leftarrow a_c \cdot \pi_c(e)$;
                $a_\sigma \leftarrow a_\sigma \cdot \langle \langle e \rangle \rangle$;
            else
                $r \leftarrow$ discrete random value from unif$\{1, i\}$;
                if $r \leq N$ then
                    $r \leftarrow$ discrete random value from unif$\{1, k\}$;
                    $a_c(r) \leftarrow \pi_c(e)$;
                    $a_\sigma(r) \leftarrow \langle e \rangle$;
```

i.e. such information is lost. We are only able to query the case view, $\Phi^i_{\mathscr{C},N}$, directly. Thus, in case we aim at storing Φ^i_N, we again need to resort to using an internal sliding window to represent Φ^i_N. Also note that, when sampling on a case level, the original property of reservoir sampling, i.e. being a true random sample is, from an event perspective, no longer guaranteed.

In [115] it is observed that after inserting an element at position i in the stream within the reservoir, with $i > N$, it is more efficient to compute the subsequent j elements to ignore, rather than doing a single random trial upon receiving each new event. As such, three alternative approaches are presented that allow us to generate such value $j \in \mathbb{N}$. In [14], it is argued that a uniform sample is in several application domains undesirable, since the stream under study is likely to evolve. Therefore, given the r^{th} item at time t (i.e. $r \leq t$), to prioritize recent items, the use of a bias function $f(r,t)$ that relates to the probability $p(r,t)$ of the r^{th} element being in the reservoir at time t is proposed. Clearly, these optimizations/alterations are applicable on top of the basic framework as presented here, i.e. in algorithm 3.2 and algorithm 3.3.

3.3.3 Maintaining Frequent Cases

By using reservoir sampling, we obtain a sample of the data emitted onto the stream. In our context, this implies either a sample of the events or the cases emitted on the stream. Since each event amounts to the occurrence of a case, cases that relate to process instances with the largest number of executed activities have the highest probability of being included within the reservoir. Such property is interesting in case we aim to track these types of cases, yet the result of the reservoir algorithm is non-deterministic. As an alternative, algorithms exist that are designed to approximate the most frequent items on an event stream [41], i.e. the `SpaceSaving` algorithm [91], the `Frequent` algorithm [45, 74] and the `Lossy Counting` [84] algorithm respectively. Here, we do not present a formal definition of an event store update function in terms of frequency approximation, yet we do provide an algorithmic sketch of each of the aforementioned algorithms.

Consider algorithm 3.4, in which we provide an algorithmic description of a case-level event store based on the `SpaceSaving` algorithm. The algorithm maintains a set of pairs X of the form $X \subseteq \mathscr{C} \times \mathbb{N}$. A pair $(c, j) \in X$ represents that the sequence of events observed, related to case identifier c is stored in array $\mathtt{a}_\sigma$ at index j. For each case identifier $c \in \mathscr{C}$, we define a counter $v(c)$, which is initialized at 0. These case identifier-based counters are used to determine what case identifier is eligible for replacement, when the set X reaches the maximal allowed size k. When a new event is received, we check whether the event's corresponding case identifier is already present in X, i.e. in cases there exists a pair of the form $(c, j) \in X$ (cf. line 8). If this is the case, we append the new event to the corresponding sequence of events as stored in $\mathtt{a}_\sigma(j)$. In case the addition leads to storing more than $\lfloor \frac{N}{k} \rfloor$ elements within $\mathtt{a}_\sigma$, we remove the head element of $\mathtt{a}_\sigma(j)$. If there is no pair in X that relates to $\pi_{\mathtt{c}}(e)$, we check if X contains less than k events. If this is the case, we add a new pair $(c, |X|)$ to X and create a new sequence representing c, by appending $\langle\langle e \rangle\rangle$ to $\mathtt{a}_\sigma$, i.e. line 13 - line 16. If the size of X equals k we search for the case identifier $c \in \mathscr{C}$ that has an entry in X and has the minimal $v(c)$ value, i.e. line 17. We replace the corresponding pair (c, j) in X, by $(\pi_{\mathtt{c}}(e), j)$, i.e. the case related to the newly received event is taking over the position of the removed case identifier c in $\mathtt{a}_\sigma$. Hence, we reset the value of $\mathtt{a}_\sigma(j)$ to $\langle e \rangle$, and moreover, we set the value of $v_{\pi_{\mathscr{C}}(e)}$ to $v(c) + 1$.

It is important to note the subtle difference of the impact of the $v(c)$-counter in the context of the process mining inspired version of the `Space Saving` algorithm with respect to. the conventional use of the algorithm in streaming applications. In the conventional setting, the stream is assumed to contain multiple equally valued data points, e.g. multiple data points relate to the same product available in an online shop. For each observable data point d, a counter $v(d)$ is maintained. The algorithm then guarantees that the actual data points stored, according to the corresponding $v(d)$-values, are the most frequent ones (subject to approximation). As in process mining we assume each event to be unique, the only multiple appearing omni-present data object is the case identifier. As such, we use the case identifier's value as the v-counter's range. The side-effect of this approach is that the algorithm has the tendency to store the longest traces, again subject to approximation error.

Algorithm 3.4: `Frequency-Based Event Store (Space Saving)`

```
input  : $S \subseteq \mathscr{E}^*$, $N, k \in \mathbb{N}$
begin
    $X \leftarrow \emptyset$;      // set of (case-id,index) pairs, i.e. $X \subseteq \mathscr{C} \times \{1,...,k\}$
    $\mathsf{a}_\sigma \leftarrow \epsilon$;   // sequence of sequences of events (max. length $\lfloor \frac{N}{k} \rfloor$)
    initialize $v(c) \leftarrow 0, \forall c \in \mathscr{C}$;
    $i \leftarrow 0$;
    while $true$ do
        $i \leftarrow i + 1$;
        $e \leftarrow S(i)$;
        if $\exists (c, j) \in X (c = \pi_{\mathsf{c}}(e))$ then
            $v(c) \leftarrow v(c) + 1$;
            $\mathsf{a}_\sigma(j) \leftarrow \mathsf{a}_\sigma(j) \cdot \langle e \rangle$;
            if $|a_\sigma(j)| > \lfloor \frac{N}{k} \rfloor$ then
                $\mathsf{a}_\sigma(j) \leftarrow \mathsf{a}_\sigma(j)_{2...|\mathsf{a}_\sigma(j)|}$;
        else if $|X| < k$ then
            $X \leftarrow X \cup (c, |X| + 1)$;
            $v(c) \leftarrow 1$;
            $\mathsf{a}_\sigma \leftarrow \mathsf{a}_\sigma \cdot \langle\langle e \rangle\rangle$;
        else
            $(c, j) \leftarrow \underset{(c,j) \in X}{\arg\min}(v(c))$;
            $\mathrm{v}(\pi_{\mathsf{c}}(e)) \leftarrow \mathrm{v}_c + 1$;
            $X \leftarrow (X \cup \{(\pi_{\mathscr{C}}(e), j)\}) \setminus \{(c, j)\}$;
            $\mathsf{a}_\sigma(j) \leftarrow \langle e \rangle$;
```

The Frequent algorithm, which we depict in algorithm 3.5 on page 72, operates in a similar fashion. It again maintains a set of pairs X of the form $X \subseteq \mathscr{C} \times \mathbb{N}$. A pair $(c, j) \in X$ represents that the sequence of events observed, related to case identifier c is stored in array $\mathtt{a}_\sigma$ at index j. It however additionally stores a set J, which represents those $j \in \{1, ..., k\}$, for which $\nexists (c, j) \in X$. As such, initially, when $X = \emptyset$, we have $J = \{1, ..., k\}$. When an event arrives related to a case identifier c that is already present in X, i.e. $(c, j) \in X$, exactly the same procedure is applied as in the case of the SpaceSaving algorithm, i.e. $v(c)$ is increased and the new event is simply appended to $\mathtt{a}_\sigma(j)$. In the case the new event relates to a case identifier that is not present in X, we simply add a pair (c, j) to X, where j is a "free" index, i.e. line 14 - line 19.

However, the main difference is in the use, and update strategy of, the $v(c)$-counters, cf. line 21 - line 26. Whenever the set X is of size k, and a new event arrives related to a case identifier that is not present in X, we decrease the counters of all case identifiers present in X by one. Whenever such counter gets 0, i.e. for some $c \in \mathscr{C}$, we have $v(c) = 0$, we remove its entry, i.e. (c, j) from X. We add the corresponding index j back to the set of free indices J, and set $\mathtt{a}_\sigma(j)$ to ϵ. Observe that, this difference in counter strategy leads to the fluctuation of the size of set X in case of the Frequent algorithm. On the contrary, in the SpaceSaving algorithm, once the size of set X reaches k, it remains of size k.

Finally, the Lossy Counting algorithm, cf. algorithm 3.6 on page 73, adopts a relatively different strategy, yet it again maintains a similar base set $X \subseteq \mathscr{C} \times \mathbb{N}$ and uses the notion of case identifier-specific counters, i.e. $v(c)$. When an event arrives related to a case identifier c that is already present in X, i.e. $(c, j) \in X$, $v(c)$ is increased and the new event is simply appended to $\mathtt{a}_\sigma(j)$. The length of any entry in $\mathtt{a}_\sigma$ is however bounded by user-specified value w rather than $\lfloor \frac{N}{k} \rfloor$.

The algorithm conceptually divides the stream into buckets of length k and keeps track to which bucket the *currently arriving event* belongs. Regardless of the actual size of X, whenever an event arrives on the event stream that relates to a case identifier c that is not present in X, an entry (c, j) is added to X. Note that in this context, j again refers to a "free index". The corresponding counter, i.e. $v(c)$ gets a value assigned $\Delta + 1$, i.e. where Δ represents the current bucket-id. As such, we potentially over-approximate the number of occurrences of the new case identifier c by at most Δ. When we observe that a newly arriving event enters a *new bucket*, i.e. given its index i, we have $i \mod k = 0$, the algorithm starts cleaning set X. Upon clean-up, in order to be retained within X, an element on average needs to occur in each previous bucket at least once, i.e. $v(c) \geq \Delta$ needs to hold in order to be maintained in set X. However, note that, for some case identifiers we actually overestimate this value, i.e. as we initialize the corresponding counter with Δ.

Note that the Lossy Counting algorithm, due to its clean-up strategy, has a different space complexity compared to the Frequent- and Space Saving algorithms. Both of the latter algorithms guarantee that we maintain at most k case identifiers. Since we cap the length of each of the traces stored in $\mathtt{a}_\sigma$ at $\lfloor \frac{N}{k} \rfloor$, we are guaranteed that, in terms of events, both algorithms have $O(N)$ space complexity. However, in case of the Lossy Counting algorithm, the worst-case space complexity of (conventional) Lossy Counting, i.e. the number of entries that is present in X, is $O(k \log(\frac{1}{k}|S|))$. In fact,

Algorithm 3.5: `Frequency-Based Event Store (Frequent)`

```
input  : $S \subseteq \mathscr{E}^*$, $N, k \in \mathbb{N}$
begin
    $X \leftarrow \emptyset$;      // set of (case-id,index) pairs, i.e. $X \subseteq \mathscr{C} \times \{1,...,k\}$
    $J \leftarrow \{1,2,...,k\}$;                                    // set of free indices
    $a_\sigma \leftarrow \epsilon$;   // sequence of sequences of events (max. length $\lfloor \frac{N}{k} \rfloor$)
    initialize $v(c) \leftarrow 0, \forall c \in \mathscr{C}$;
    $i \leftarrow 0$;
    while $true$ do
        $i \leftarrow i + 1$;
        $e \leftarrow S(i)$;
        if $\exists (c,j) \in X (c = \pi_c(e))$ then
            $v(c) \leftarrow v(c) + 1$;
            $a_\sigma(j) \leftarrow a_\sigma(j) \cdot \langle e \rangle$;
            if $|a_\sigma(j)| > \lfloor \frac{N}{k} \rfloor$ then
                $a_\sigma(j) \leftarrow a_\sigma(j)_{2...|a_\sigma(j)|}$;
        else if $|X| < k$ then
            $j \leftarrow$ some value present in $J$;
            $J \leftarrow J \setminus j$;
            $X \leftarrow X \cup (c,j)$;
            $v(c) \leftarrow 1$;
            $a_\sigma(j) \leftarrow \langle e \rangle$;
        else
            foreach $(c,j) \in X$ do
                $v(c) \leftarrow v(c) - 1$;
                if $v(c) = 0$ then
                    $X \leftarrow X \setminus (c,j)$;
                    $J \leftarrow J \cup \{j\}$;
                    $a_\sigma(j) \leftarrow \epsilon$;
```

Algorithm 3.6: `Frequency-Based Event Store (Lossy Counting)`

```
input  : $S \subseteq \mathscr{E}^*$, $k, w \in \mathbb{N}$
begin
    $X \leftarrow \emptyset$;        // set of (case-id,index) pairs, i.e. $X \subseteq \mathscr{C} \times \mathbb{N}$
    $J \leftarrow \mathbb{N}$;        // set of free indices
    $\Delta \leftarrow 0$;
    $\mathrm{a}_\sigma \leftarrow \epsilon$;        // sequence of sequences of events
    initialize $v(c) \leftarrow 0, \forall c \in \mathscr{C}$;
    $i \leftarrow 0$;
    while $true$ do
        $i \leftarrow i + 1$;
        $e \leftarrow S(i)$;
        if $\exists (c, j) \in X (c = \pi_{\mathfrak{c}}(e))$ then
            $v(c) \leftarrow v(c) + 1$;
            if $|a_\sigma(j)| = w$ then
                $\mathrm{a}_\sigma(j) \leftarrow \mathrm{a}_\sigma(j)_{2 \ldots |\mathrm{a}_\sigma(j)|}$;
            $\mathrm{a}_\sigma(j) \leftarrow \mathrm{a}_\sigma(j) \cdot \langle e \rangle$;
        else
            $j \leftarrow$ some value present in $J$;
            $J \leftarrow J \setminus j$;
            $X \leftarrow X \cup (c, j)$;
            $v(c) \leftarrow 1 + \Delta$;
            $\mathrm{a}_\sigma(j) \leftarrow \langle e \rangle$;
        if $\lfloor \frac{i}{k} \rfloor \neq \Delta$ then
            $\Delta \leftarrow \lfloor \frac{i}{k} \rfloor$;
            foreach $(c, j) \in X$ do
                if $v(c) < \Delta$ then
                    $X \leftarrow X \setminus (c, j)$;
                    $J \leftarrow J \cup \{j\}$;
                    $\mathrm{a}_\sigma(j) \leftarrow \epsilon$;
```

therefore, we use $J \leftarrow \mathbb{N}$, rather than $J \leftarrow \{1,...,k\}$ in algorithm 3.6, line 2. Often, given a user-specified maximal error value $\epsilon \in [0,1]$, a value of $k = \frac{1}{\epsilon}$ is used as the bucket size parameter. Using such value provides certain guarantees with respect to. the approximation quality of an element's frequency, i.e. $v(c)$ in the context of algorithm 3.6.[1] Moreover, when using $k = \frac{1}{\epsilon}$, we obtain $O(\frac{1}{\epsilon}\log(\epsilon|S|))$ as space complexity. Observe that, when we also use a maximum number of $w = \frac{1}{\epsilon}$ of events per case identifier in algorithm 3.6, we deduce its corresponding space complexity to be $O(\frac{1}{\epsilon^2}\log(\epsilon|S|))$.

Finally, observe that the practical applicability of the algorithms discussed here, and other similar algorithms that allow us to track/approximate the most frequent items on a stream, is potentially limited from a process mining perspective. As indicated, by design, they track the most frequent elements on stream. Hence, if we use the case identifiers as elements in this context, we are implicitly tracking those cases for which the most activities are performed. This is not necessarily unusable, i.e. these types of process instances typically relate to outliers and/or problematic cases, and they are therefore interesting for further investigation. However, the majority of the process mining algorithms is explicitly designed under the assumption that such type of cases are not part of the input event data.

3.3.4 Time Decay

The time-based sliding window, as briefly described in subsection 3.3.1, is an instantiation of a broader class of storage algorithms, known as decay functions. The essence of such functions/algorithms is to assign a weight to data items that arrive on the stream, based on their age. The *older* a data item is, the *higher* its weight is and, the *higher* an item's weight, the *less* importance the item gets. A function assigning weights to data items, based on their timestamp, is a *decay function*, if the function assigns a weight 1 at the moment of arrival. Furthermore, the function needs to be monotone non-increasing, for increasing time, i.e. if event e arrives later than e', the weight of e is smaller or equal to the weight of e'.

In some cases, the weight of the data items is adopted directly within the function intended to be computed on top of the data stream. For example, deriving some numeric value in which the influence of data items with respect to. that value is scaled using the decay function. However, in the context of process mining the weight of an event has no particular meaning. Nonetheless, the weight of an event can be used in order to determine whether or not an event, or set of events is eligible to be removed from the event store, i.e. to be part of Φ_N^{i-}. As shown in [42], by using the concept of landmarks, it is possible to compute such decay efficiently, i.e. we compute an intermediary weight value upon arrival and scale it to obtain the true decay value as time increases.

[1] Using such value $k = \frac{1}{\epsilon}$ in fact also provides such approximation quality guarantees for the `Frequency` and `Space Saving` algorithm.

Table 3.2: Guarantees of the different techniques presented here with respect to. the quality requirements for event stores as quantified in Table 3.1.

Technique	*RQ 1* (respect order)	*DS 1* (no re-insertion)	*DS 2* (removal case compliant)
Sliding Window (cf. subsection 3.3.1)	✓	✓	✓
Reservoir Event-Level (cf. subsection 3.3.2)	✓	✓	×
Reservoir Case-Level (cf. subsection 3.3.2)	✓	✓	✓
Frequency Approximation (cf. subsection 3.3.3)	✓	✓	✓
Time Decay (cf. subsection 3.3.4)	✓	✓	✓

3.3.5 Guarantees

In this section, we quantify to what degree the different types of techniques as presented in this section allow us to guarantee the requirements and desiderata as defined for event stores in section 3.2, in Table 3.1. We schematically depict this quantification in Table 3.2. We observe that all techniques allow us to guarantee all requirements/desiderata as defined in Table 3.1, except for reservoir sampling on event level. Clearly, in such case, we are able to ignore events related to certain cases for which we are already maintaining some events. Hence, removal of events from the event store, not compliant to the ordering imposed by the process instance (in this case instant removal due to ignoring the events), is possible.

3.4 Exploiting Behavioural Similarity

section 3.3 shows that we are able to instantiate both event-level and case-level event stores using existing data storage techniques originating from different areas of data stream processing. Note however that, so far, we have not touched upon the actual quality of the event store maintained. The sliding window approach, either from an event-level or case-level perspective, is likely to only maintain snippets of traces, rather than complete trace behaviour. Similarly, it is possible to delete the events related to a certain case-id from a reservoir sample and/or frequent item-based approach and, in a later phase, inserting an event related to the same case-id. For some process mining algorithms, this is not necessarily a problem, i.e. certain algorithms only look at local relations between at most two consecutive events. However, other algorithms do heavily rely on the notion of completeness of cases and/or the explicit knowledge of when a trace starts and/or ends.

We are able to partly solve the aforementioned quality problems related to conventional data stream storage techniques, by assuming that we are explicitly aware of a set of unique start- and end events of the process. For example, assume a new event $e \in \mathscr{E}$ arrives with $\pi_{\mathsf{c}}(e) = c$, and furthermore, c is not yet described by any event currently present in the event store. In such case, event e is only eligible for addition to the event store, if the corresponding activity is a unique start event. Even more so, in case we have explicit knowledge of end events we are able to use this to select candidates for removal from the event store. Note that, by additionally adapting such strategies, it is likely that the behaviour of the storage algorithms starts to deviate

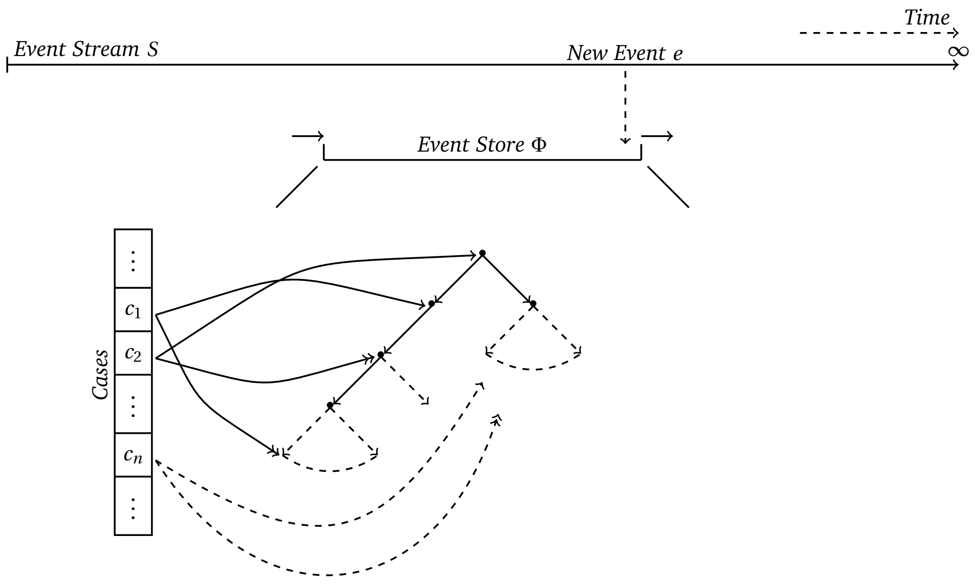


Figure 3.3: Schematic overview of the proposed prefix-tree based case-level event store. We maintain a prefix-tree that represents the process behaviour. For each case, we keep track of the oldest and newest received event, i.e. still present in the underlying event store. By creating paths from the root to leaves of the prefix-tree, we are able to create behaviour that exceeds the behaviour stored in the underlying event store.

significantly from their original design and/or intent, and, as a consequence, certain underlying theoretical properties and/or guarantees are invalidated. However, regardless of the assumption of the explicit availability of unique start- and/or end-activities, size parameter N does only allow us to store a bounded share of recent events for the corresponding case-id. Even when applying a dynamic scheme that allows us to exceed this value, if space permits, the inherent requirement to maintain the event data in finite memory at some point requires us to drop fragments of cases, or uncompleted cases as a whole.

Within real-life processes, multiple instances of the process exhibit similar behaviour, e.g. first a certain set of activities needs to be performed in order to perform a different subsequent set of activities. In this section, we, therefore, propose an alternative storage strategy that aims to exploit inter-process instance similarity, in order to increase the overall quality and completeness of the data stored related to the currently running process instances. Consider Figure 3.3, in which we depict a schematic overview of the proposed approach. We store the events arriving on the event stream in an event store, i.e. any event store as described in section 3.3, which adheres to *DS 2*. However, we replicate the behaviour using a *prefix-tree*, cf.

subsection 2.1.5, as an internal representation of the underlying process. We propose to use the *control-flow perspective* as a main driver for defining the edges and nodes of the prefix-tree, however, theoretically any form of omni-present payload is usable. Whenever we remove an event e from the event store and the prefix-tree, we aim to identify whether there are other process instances describing some event e', which is still in the event store, that actually describes the same behaviour. If this is the case, we use such event, i.e. its presence in the stored prefix-tree, as place-holder for the removed event e. Thus, even though event e is no longer present in the underlying event store, when we fetch the trace corresponding to the case identifier of e, we let event e' act as a replacement for event e. This allows us to obtain, from the start of traces, trace completeness for the active active process instances.

3.4.1 Maintaining Control-Flow Oriented Prefix-Trees

As indicated and illustrated in Figure 3.3, we maintain a prefix-tree that represents the process behaviour. Moreover, we propose to use the *control-flow perspective* as a main driver for defining the edges and nodes of the prefix-tree. To determine what behaviour needs to be represented by the prefix-tree, we additionally maintain a regular event store, e.g. a sliding window. The main idea of the approach is that, each process instance captured within the underlying event store, represented by its case identifier, describes a path in the prefix-tree. Furthermore, in case multiple process instances describe the same sequence of activities in the beginning of the process, these instances share (a part of) a path in the prefix-tree. When at some point certain events are removed from the underlying event store, we are still able to, given the fragment of current behaviour, walk back to the root of the prefix-tree and thus assess the removed history of the corresponding process instance. The enhanced completeness, i.e. from the start of traces, in turn has a potential beneficial impact on the process mining algorithm used.

The approach roughly works as follows. We maintain an internal event store that adheres to *DS 2*, e.g. a sliding window, and a prefix-tree in which each arc represents the *execution of an activity*. For each case-id c that is active, i.e. there is some event present in the internal event store Φ^i_N that is related to c, we maintain two pointers that point to a node in the tree. One pointer, i.e. $\vec{p}_r$, points to the vertex that represents the *most recently received event* observed on the event stream for case c. The other pointer, i.e. $\vec{p}_o$, relates to the case's *oldest event* related to case c that is still present in the underlying event store. Whenever we receive a new event, we assess the vertex in the tree pointed at by $\vec{p}_r$, i.e. related to the most recent event for the case. Subsequently, we check whether or not that corresponding vertex has an outgoing edge that describes the same activity as the newly received event. If so, we shift pointer $\vec{p}_r$ to that vertex, if not, we create an outgoing arc decorated with the activity of the newly received event from the current vertex to the new vertex. After creating the vertex, we update pointer $\vec{p}_r$ to the new vertex. In case an event is dropped from the event store, we shift the oldest-event-pointer to the vertex that is reached by the activity described by the deleted event.

Consider Figure 3.4 in which exemplify maintaining a prefix-tree based event store.

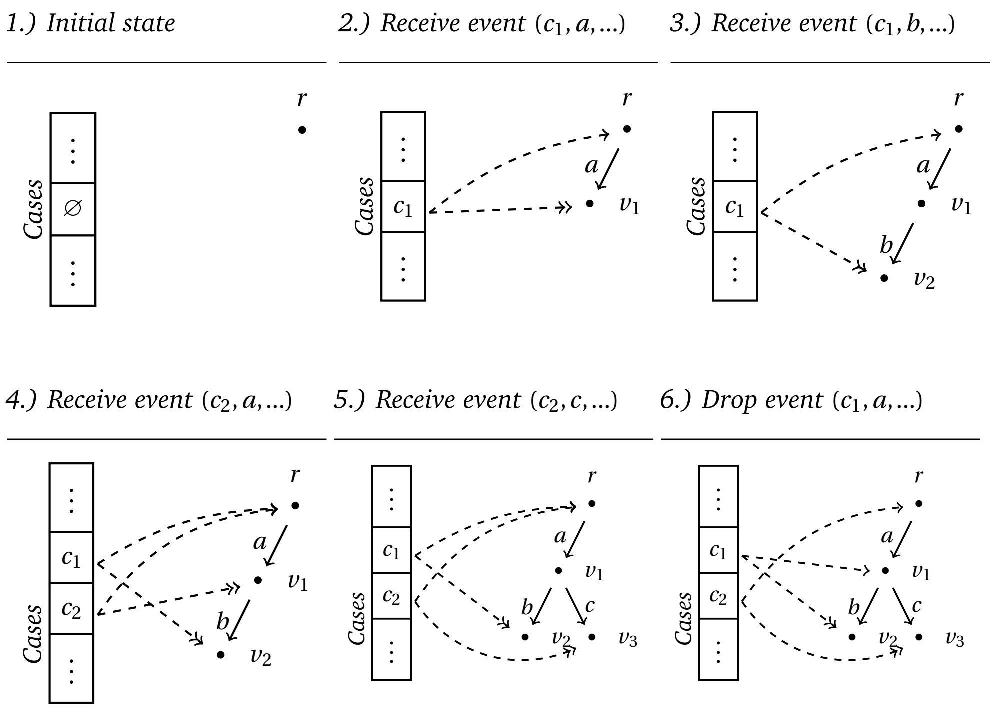

1.) Initial state
2.) Receive event $(c_1, a, ...)$
3.) Receive event $(c_1, b, ...)$
4.) Receive event $(c_2, a, ...)$
5.) Receive event $(c_2, c, ...)$
6.) Drop event $(c_1, a, ...)$
Cases
∅
c_1
c_2
r
a
b
c
v_1
v_2
v_3

Figure 3.4: Example of maintaining a prefix-tree based event store. Note that, we assume all arriving events to be stored in the (initially empty) underlying event store. Single-headed arrows indicate *start*-pointers, i.e. pointers $\vec{p}_o$ representing the oldest event of the case identifier in the corresponding event store, double headed arrows indicate *end*-pointers, i.e. pointers $\vec{p}_r$ representing the most recently received event.

In Figure 3.4, we represent events as a (case identifier, activity)-pair (depicted as $(c, a, ...)$), since these are the event attributes effectively used in construction of the prefix-tree. When we tap into the stream, i.e. situation *1.) Initial state*, we have not yet received any events and hence the prefix-tree only contains a root node. Moreover, we do not maintain any pointers to the tree.

In situation *2.)*, we receive a new event $(c_1, a, ...)$ which we assume to be temporarily stored in the backing event-level event store. We first add an entry for case c_1 in the *Cases*-array. Since there is no behaviour recorded yet for c_1, we know the new event is the first event (at least at this point in time) related to case-id c_1. We create a new vertex and connect it to the root by means of an incoming edge with label a. We add two pointers for case c_1, the recent pointer points to the newly added vertex. The other pointer, i.e. referring to the oldest possible behaviour seen for the case points to the root vertex.

In situation *3.)* we receive a new event related to case identifier c_1, i.e. $(c_1, b, ...)$. We observe that c_1 is already present within the *Cases*-array. We traverse the recent behaviour pointer of case c_1, i.e. pointing to vertex v_1. Since v_1 does not have any outgoing arcs, we create a new vertex, i.e. v_2 with an incoming arc labelled b from v_1.

In situation *4.)* we receive an event related to case identifier c_2, i.e. $(c_2, a, ...)$. Since there is no entry for c_2 in the *Cases*-array, we create a new entry for c_2. Like c_1, the first event for c_2 describes the execution of activity a. Therefore, there is no need to modify the prefix-tree. We only create two pointers for c_2, one pointing to the root r of the tree, and one pointing to v_1.

In situation *5.)* we again receive an event for case c_2, i.e. $(c_2, c, ...)$. In this case, we create a new vertex v_3 with an incoming arc labelled with c from v_1. Furthermore, we assume that the arrival of the new event causes the event-level store to drop the first received event, i.e. $(c_1, a, ...)$. To account for the removal, we shift the oldest behaviour pointer of case c_1 to vertex v_1 (which we visualize in *6.)*).

Observe that, even though event $(c_1, a, ...)$ is removed from the backing event-level event store, we are still able to reconstruct the trace $\langle a, b \rangle$ for case-id c_1. We are able to do this by traversing back to the root r from v_2, i.e. the vertex pointed at by c_1. This is the main advantage of the described approach, i.e. without additional memory load, we are able to reconstruct traces, even though some corresponding events are removed from the underlying event store.

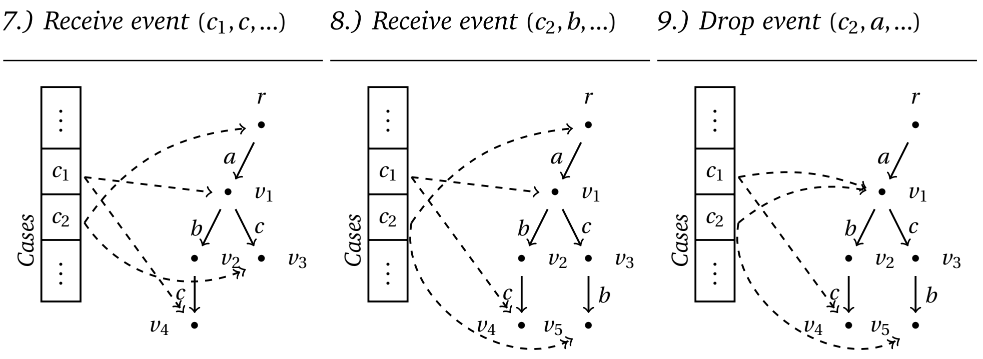
3.4.2 Accounting for Event Removal

In the example in Figure 3.4, receiving event $(c_2, c, ...)$ yields the removal of event $(c_1, a, ...)$ in the internal event store. As such, we shift the start pointer of case c_1 to vertex v_1. In the example, the edge leading into vertex v_1 is still covered by a different case, i.e. also case c_2 describes that activity at that position. However, this does not always happen. Hence, we propose a mechanism that allows us to account for removal that results in parts of the tree that are no longer covered by any case identifier in the underlying event store. In one of the approaches, we simply drop any possible subtree that is no longer covered. In the other approach, we shift the subtree underneath the root of the prefix-tree.

Consider Figure 3.5, in which we continue the example presented in Figure 3.5. In *7.)*, we receive a new event related to case identifier c_1, describing the execution of an activity labelled c. We thus create a new vertex, i.e. v_4, and connect vertex v_2 to vertex v_4 by means of an arc labelled c. We also update the terminal pointer of case identifier c_1. We subsequently, in *8.)*, receive an event related to case c_2, for which we update the prefix-tree and the pointers accordingly. Observe that, in *9.)*, event $(c_2, a, ...)$ is dropped. Hence, we shift the corresponding start pointer of c_2.

In this new situation the edge from vertex r to v_1, labelled with a is no longer covered by any case identifier, i.e. both cases c_1 and c_2 start at v_1. In general, it is possible that the removal of a certain event leads to the presence of parts of the prefix-tree that are no longer covered by any case. We propose two strategies to account for such situation.

1. *Drop*; In the *drop strategy*, we completely drop the sub-tree formed by the uncovered component. in the context of the example, this implies that we decide



7.) Receive event $(c_1, c, ...)$
8.) Receive event $(c_2, b, ...)$
9.) Drop event $(c_2, a, ...)$
Cases
c_1
c_2
r
a
v_1
b
c
v_2
v_3
v_4
v_5

Figure 3.5: Example of removal a prefix-tree based event store, causing parts of the prefix-tree to be uncovered.

to drop all events related to cases c_1 and c_2, yielding an empty tree, i.e. we just retain the root.

2. *Shift*; In the *shift strategy*, we position the sub-tree that is actually covered by one or more cases directly below the root, i.e. we start shifting the behaviour within the tree such that the events present in the internal event store are still present in the prefix-tree. in the context of the example, this implies that vertex v_1 is removed from the prefix-tree. Moreover, two new edges connecting vertex r to vertices v_1 and v_2, labelled b and c respectively, are added.

Observe that, when adopting the *drop* strategy, we potentially remove events from the prefix-tree based event store that are present within the internal event store. This implies that we potentially describe fewer events than are actually present in the underlying event store, however, those traces that we describe are in fact *complete from the start of the cases*. Similarly, when adopting the shift strategy we always describe everything present in the internal event store, at the potential cost of incompleteness at the start of cases.

Note that, as indicated, it is only possible to adopt the aforementioned strategies, if we assume that removal of events in the underlying event store is case compliant, i.e. *DS 2* of Table 3.1. As such, the event-level reservoir sampling based event store is not eligible to be used as an underlying event store, as it does not guarantee this. All other event stores described in section 3.2 are in principle usable as an underlying store.

3.4.3 Reducing the Overall Memory Footprint

Thus far, we have illustrated how to maintain a prefix-tree representing process behaviour, on the basis of an underlying event store. We have proposed to only represent the events in the prefix-tree as activities, and to leave the events, as-is, in the event store. This does however only allow us to reconstruct the control-flow perspective of events that are described by the prefix-tree, but not present in the event store any-more. Therefore, in this section, we briefly illustrate how to integrate the additional payload directly into the prefix-tree. We furthermore illustrate how to avoid event duplication in prefix-tree based event storage.

Within the example, cf. Figure 3.4 and Figure 3.5, we, for the ease of simplicity, only store events as a (case identifier, activity)-pair. As presented in 2.10, an event is considered to be a tuple of arbitrary size, with arbitrary payload, which mainly depends on the process under study. Hence we aim to store the event payload within the tree as well, as it allows us to fully reconstruct the events. As such, we simply store the payload of each event within the vertices of the prefix-tree. For example, we are able to store, in each vertex, for each case-id a set of key-value pairs representing the additional event payload.

When events start to be removed from the underlying event store, i.e. pointers are being shifted, if space permits, we retain the event payload stored in the vertices until the corresponding case identifier is completely removed from storage. Note however, that if we do not do this, we are effectively cheating as we are not really removing the events from memory. If memory does not permit, we aim at removing the corresponding payload from the vertex, upon a pointer shift. The edges of the prefix-tree itself however only describe the control-flow, and not the additional data perspectives. As such, when reconstructing traces, we are unaware what payload relates to the events that we derive on the basis the edges of the prefix-tree. Hence, whenever we reconstruct the case-level event log, we need to apply sampling for those vertices that no longer contain payload information for a certain case.

To use the prefix-tree based storage, we do not need to duplicate all events and their payload, i.e. as conventionally stored in the internal event store. Consider the example in which we use a sliding window as an event store. To effectively apply the prefix-tree based storage on top of the sliding window, we in fact only need to store case identifiers within the sliding window, i.e. $\mathscr{C}^*$. When an event arrives, we append the corresponding case identifier to the sliding window and store all the payload within vertices in the prefix-tree. In this way, we only duplicate the case identifiers. Observe that, since we use a pointer structure from the case identifiers to the prefix-tree, whenever a case identifier is dropped, we are able to shift the pointer(s) accordingly.

3.5 Evaluation

In this section, we evaluate the performance of different storage approaches in terms of data quality and memory usage. In particular, we assess the effect of using prefix-tree based storage on top of a sliding window.

3.5.1 Data Quality

In this section, we evaluate the impact of several different storage techniques on data quality. In particular, we focus on how well the different storage techniques are able to accurately describe the event data, as complete as possible. As we expect the event data to describe incomplete process instances in several cases, we do not measure process mining specific quality measures, such as replay-fitness and/or precision of some derived model, based on the event store. Rather, we use the *directly follows relation*, as described in section 2.4.1, as a proxy to *measure trace completeness*. Recall that the directly follows relation describes what activities are able to directly follow what other activities. For example, given a simple trace in a simple event store $\tilde{\Phi}$, e.g. $\langle ..., a, b, ... \rangle$, we observe that a is directly followed by b, and hence $a >_{\tilde{\Phi}} b$ holds. Hence, within the experiments performed, we assess to what degree the storage techniques under study allow us to reconstruct the directly follows relation of the underlying event log that is used to generate the stream.

Within the experiments performed, we first compute the complete directly follows relation, based on the underlying event logs used, i.e. the *ground truth*. We subsequently generate a stream, ordered on timestamps present in the event data. This ensures that multiple cases run in parallel within the event stream. For each of the storage techniques used, i.e. sliding window and prefix-tree based storage (using a sliding window internally) with sizes 500, 1000, 2500, we construct a corresponding event log after each received event. Based on such event log we again compute a directly follows relation, which we compare against the directly follows relation computed on the event log as a whole. We track the first 5000 events of the stream for each technique.

We compute precision and recall, cf. section 2.4.4, i.e. $recall = \frac{|TP|}{|TP|+|FN|}$ and $precision = \frac{|TP|}{|TP|+|FP|}$, on the basis of the discovered directly follows relation and the directly follows relation based on the whole event log. We moreover explicitly keep track of start- and end activities, i.e. the set of activities that occur at least once at the start/end of a trace within the event log. We use the following classification to evaluate the approach:

- *True Positive (TP)*

  We observe a relation of the form $a > b$ in the discovered relation that is also observed in the ground truth, and/or, we observe a start/end activity in the event log based on the event stream that is also a start activity in the ground truth.

- *False Positive (FP)*

  We observe a relation of the form $a > b$ in the discovered relation that is not observed in the ground truth, and/or, we observe a start/end activity in the event log based on the event stream that is not in the ground truth.

- *True Negative (TN)*

  We do not observe a relation of the form $a > b$, which is also not described in the ground truth, and/or, we do not observe a start/end activity in the event log based on the event stream, which is also not described in the ground truth.

- *False Negative (FN)*

  We do not observe a relation of the form $a > b$, which is observed in the ground truth, and/or, we do not observe a start/end activity in the event log based on the event stream, which is in fact observed in the ground truth.

Observe that we expect the false positives to only be related to wrongly discovered start/end activities. Due to trace incompleteness, we are able to observe an activity a the last position of a trace, currently stored in the event store, that in the underlying event log never occurs as a last activity. We are unable to observe directly follows relations in the data that are not in the underlying event log. Observe that we expect the recall of the different techniques to be rather low. It is likely that the event store only stores a fraction of the behaviour present in the original event log, and thus the ground truth. As such, we expect the event stores to describe a large portion of *false negatives*, thus lowering the overall recall value throughout the experiment. On the other hand, we expect the precision to be rather high. In particular, we never observe any directly follows relation that is not part of the ground truth, as it reflects the same portion of data. However, we do expect the event stores to describe different start/end activities due to trace-incompleteness of the data. Moreover, we expect to see the prefix-tree to have higher precision values than the standard sliding window, as it aims to keep traces within its store a bit longer by exploiting shared prefixes.

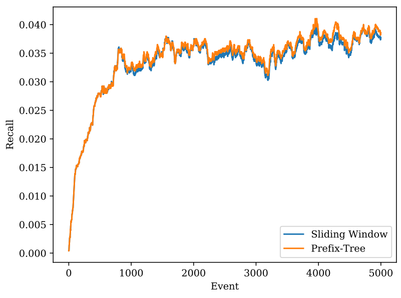
In Figure 3.6 and Figure 3.7, on page 84 and page 85, we present quality experiments related to the sliding window based event store and prefix-tree based storage, respectively. All experiments involve real data, i.e. extracted from different information systems. For all sizes investigated, i.e. 500, 1000 and 2500, we observe that overall, precision values of the prefix-tree based storage to be slightly higher than the sliding window based event store, yet the difference is not significant. Also for recall values, the prefix-tree based event store outperforms the sliding window based event store, however, the difference in quality is again negligible. For both data sets used, we observe that the prefix-tree based storage allows us to obtain slightly higher recall / precision values, yet the difference is negligible and non-significant.

In Table 3.3, we show the average results in terms of recall, precision and f1-score (harmonic mean of recall and precision). The results include the event logs used in Figure 3.6 and Figure 3.7, together with additional event logs.

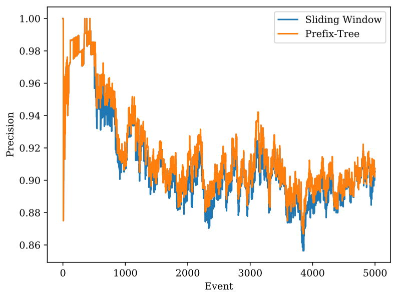
3.5.2 Memory Usage

Aside from the experiments related to quality in terms of the directly follows relation, we assess the memory usage of both techniques, i.e. sliding window versus prefix-based storage. Clearly, the sliding window based approach has a constant memory footprint. However, the prefix-tree based event store needs potentially fewer memory entries compared to the sliding window based store, since some process instances cover a similar prefix within the tree. In Figure 3.8 and Figure 3.9, on page 86 and page 87, we present experiments related to the memory usage of the sliding window based event store and prefix-tree based storage, respectively. We use the same event logs as the ones used in Figure 3.6 and Figure 3.7. In particular, we assess the number of events described for both techniques as well as the number of memory entries needed to do so. In case of the sliding window, the number of memory entries needed equals the window size. In case of the prefix-tree based storage, we measure the number of edges present in the tree.

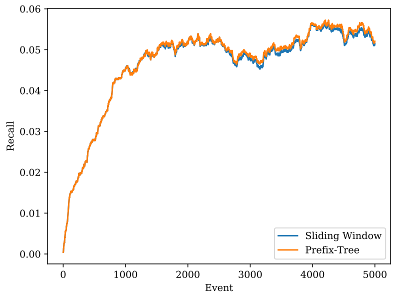
We observe that in all cases, the prefix-tree based storage allows us to describe a



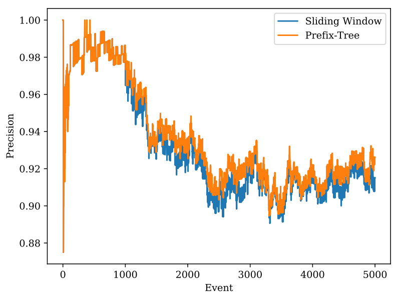
(a) Recall, size = 500.



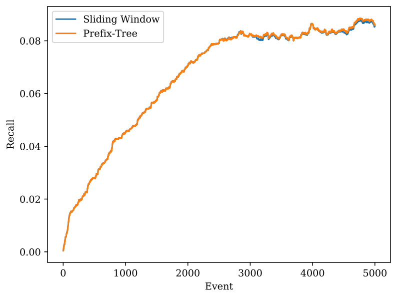
(b) Precision, size = 500.



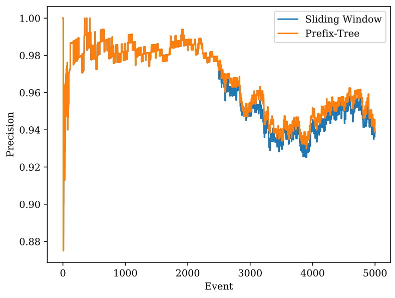
(c) Recall, size = 1000.



(d) Precision, size = 1000.

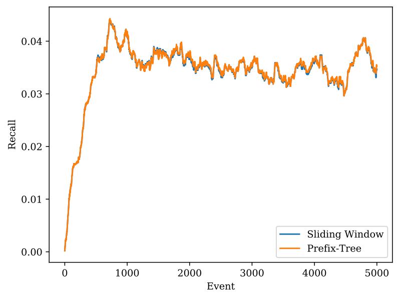


(e) Recall, size = 2500.

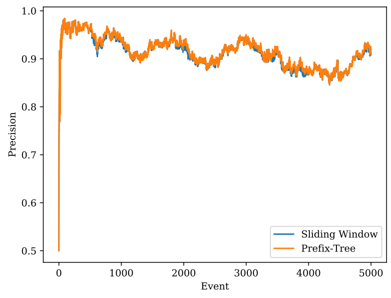


(f) Precision, size = 2500.

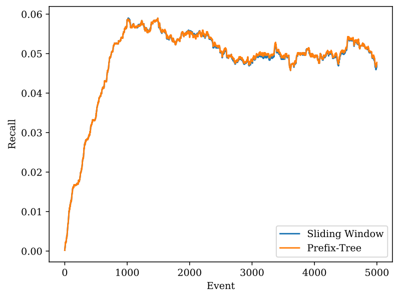
Figure 3.6: Precision and recall of sliding window versus prefix-tree based storage using window sizes of 500, 1000 and 2500, based on the BPI Challenge 2015 *Municipality 1* event log [48].



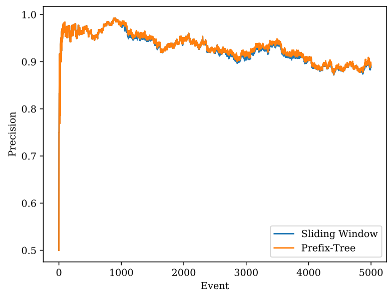
(a) Recall, size = 500.



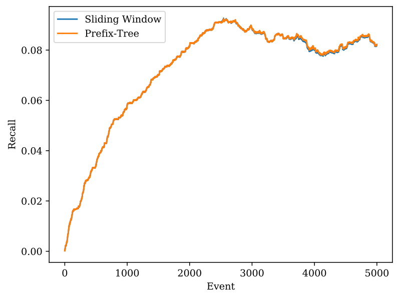
(b) Precision, size = 500.



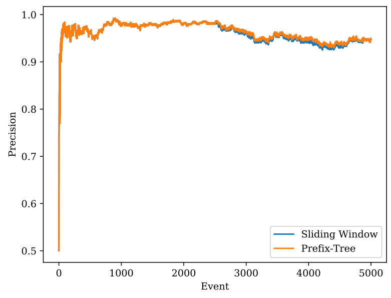
(c) Recall, size = 1000.



(d) Precision, size = 1000.

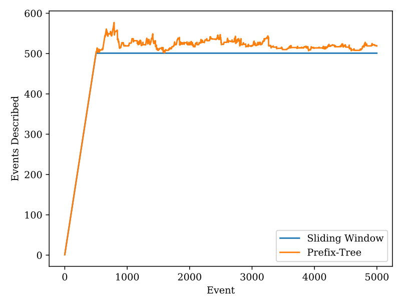


(e) Recall, size = 2500.

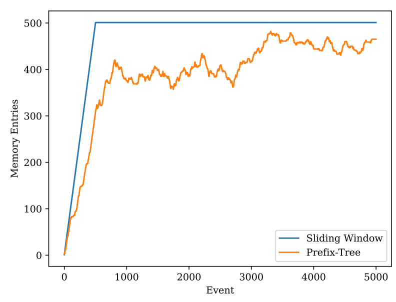


(f) Precision, size = 2500.

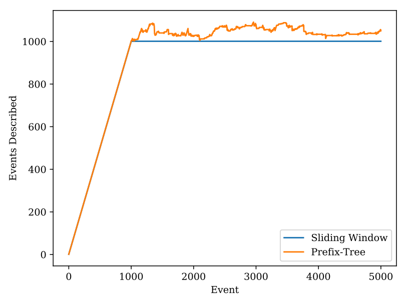
Figure 3.7: Precision and recall of sliding window versus prefix-tree based storage using window sizes of 500, 1000 and 2500, based on the BPI Challenge 2015 *Municipality 2* event log [48].



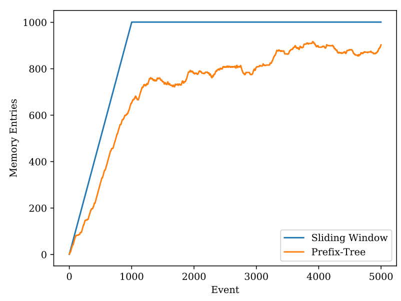
(a) Number of events described, size = 500.



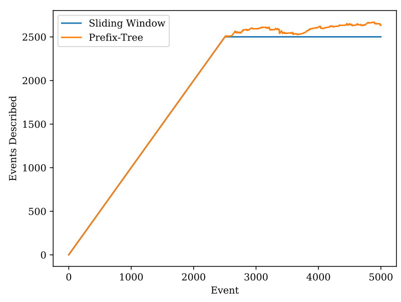
(b) Number of memory entries, size = 500.



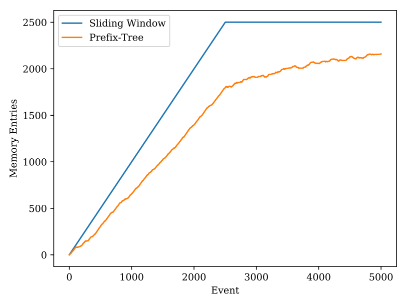
(c) Number of events described, size = 1000.



(d) Number of memory entries, size = 1000.

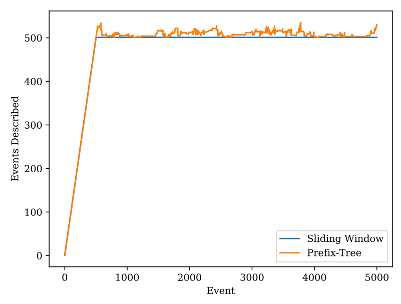


(e) Number of events described, size = 2500.

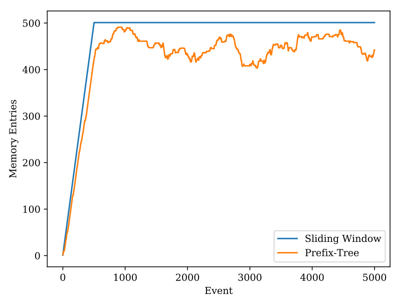


(f) Number of memory entries, size = 2500.

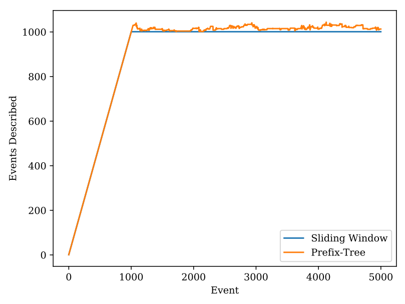
Figure 3.8: Memory usage of sliding window versus prefix-tree based storage using window sizes of 500, 1000 and 2500, based on the BPI Challenge 2015 *Municipality 1* event log [48].



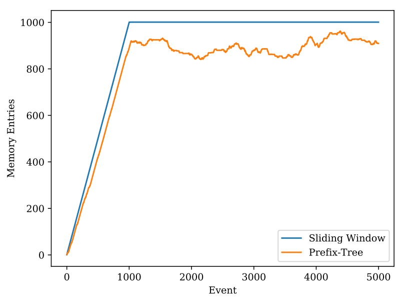
(a) Number of events described, size = 500.



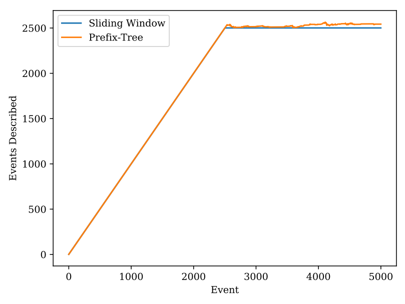
(b) Number of memory entries, size = 500.



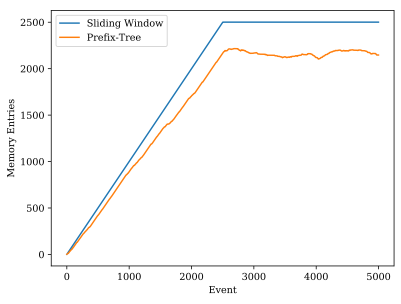
(c) Number of events described, size = 1000.



(d) Number of memory entries, size = 1000.



(e) Number of events described, size = 2500.



(f) Number of memory entries, size = 2500.

Figure 3.9: Memory usage of sliding window versus Prefix-tree based storage using window sizes of 500, 1000 and 2500, based on the BPI Challenge 2015 *Municipality 2* event log [48].

Table 3.3: Average Quality results for different event logs studied, computed over the first 5000 events of each generated event stream.

	Technique	*Size*	*Recall (Avg.)*	*Precision (Avg.)*	*F1-Score (Avg.)*
BPI 2015 Municipality 1 [48]	Sliding Window	500	0.03	0.91	0.06
	Sliding Window	1000	0.05	0.93	0.09
	Sliding Window	2500	0.07	0.96	0.12
	Prefix-Tree	500	0.03	0.92	0.07
	Prefix-Tree	1000	0.05	0.94	0.09
	Prefix-Tree	2500	0.07	0.97	0.12
BPI 2015 Municipality 2 [48]	Sliding Window	500	0.03	0.91	0.07
	Sliding Window	1000	0.05	0.93	0.09
	Sliding Window	2500	0.07	0.96	0.13
	Prefix-Tree	500	0.03	0.91	0.07
	Prefix-Tree	1000	0.05	0.93	0.09
	Prefix-Tree	2500	0.07	0.96	0.13
BPI 2015 Municipality 3 [48]	Sliding Window	500	0.03	0.92	0.06
	Sliding Window	1000	0.04	0.93	0.07
	Sliding Window	2500	0.06	0.96	0.10
	Prefix-Tree	500	0.03	0.92	0.06
	Prefix-Tree	1000	0.04	0.94	0.07
	Prefix-Tree	2500	0.06	0.96	0.10

small fraction of additional behaviour. This is in line with the results in Figure 3.8 and Figure 3.9, i.e. the slightly higher precision values. Note that we do not observe a clear relation between the size of the (underlying) sliding window and the amount of additional described behaviour. We furthermore observe that in all cases, the prefix-tree based event store needs fewer memory entries to describe behaviour. This implies that in the event data used, there is indeed shared behaviour in terms of trace prefixes, which we are able to exploit within the prefix-tree.

3.6 Related Work

Little work has been done in the context of explicit temporal storage of streaming data originating from the execution of processes. In [34, 35] the Lossy Counting algorithm is used to store an algorithmic-specific abstraction for the purpose of online process discovery. Furthermore, in [69] prefix-trees are used as well, however, not with the intention to store full trace history. Rather, after receiving a few events related to the same case a clean-up action is performed. In [69] it is merely shown that using such prefix-tree oriented structure is more efficient for the corresponding process discovery algorithm, compared to [35].

In offline process mining, little work has been published in the context of (efficient) storage of event data. Most academic and commercial process mining tools support the `IEEE eXtensible Event Stream (XES)` [1] standard, which specifies an `XML`-based storage standard for event data. Several open-source `java`-libraries have been written and documented with the specific aim to implement lightweight efficient implementations of the standard [86]. For example, the `XESLite - Database (XL-DB)` implementation allows us to use secondary storage on a single node computer.

As such, the implementation allows us to analyse event data that exceeds internal memory, yet it does not allow us to distribute it over multiple nodes.

Additionally, some work has been performed with respect to large-scale storage, i.e. particularly in the light of big event logs, i.e. event logs that no longer fit the memory of a single computer. In [54, 70] the authors assess process discovery in the context of data intensive environments. In particular, both works focus on the applicability of using the Hadoop[2] distributed data storage framework, in order to apply the flexible heuristic miner [122]. As such, the main focus of these works is towards computing the underlying data abstractions rather than actual storage. However, a connector to any Hadoop ecosystem has been implemented and published in the light of the aforementioned work [71]

In light of large scale storage, recently some studies have investigated the applicability of relational databases in the context of event log storage [50, 108]. In [50] a first relational database based event log storage design was proposed, i.e. `RXES`. It adheres to the XES standard, yet allows for storage of the event data in a relational database. In `RXES`, an event is allowed to belong to different process instances, i.e. which is often the case in practice. The work in [108] also presents a storage design on the bases of relational databases, i.e. `DBXES`. Moreover, the authors motivate that a lot of discovery algorithms use intermediate data structures, and thus propose to pre-compute these structures at event insertion time, to reduce overall computational complexity of several discovery techniques. The aforementioned rationale is also applied in this thesis, i.e. in chapter 5, where we propose a similar, stream-based architecture. Additionally, work has been performed on translation of conventional relational databases into `XES`-based event logs [114]. Note however, that this work in principle does not aim to solve big data problems. The primary focus of the work is to extract event data from complex databases, not tailored towards process mining. Similarly, work has been done in the design of process/event-aware versions of OLAP cubes [27, 116, 117]. In these works, the emphasis is more on efficient event selection for the purpose of multi-perspective process mining, instead of the actual size of the underlying event data.

3.7 Conclusion

In recent years, several different process mining techniques have been proposed. All of these techniques operate on the notion of an event log, i.e. a collection of executed business process events, describing traces of process behaviour. Such an event log is a static, historical source of information. Being based on such event logs, offline process mining techniques cannot be applied on event streams, without any modification. Therefore, new techniques are required that allow us to (temporarily) store events emitted onto an event stream, and subsequently apply any process mining technique on top of such storage.

[2] `htttp://hadoop.apache.org/`

3.7.1 Contributions

In this chapter, we have presented several ways to store stream-based event data originating from live execution of (business) processes. We have constructed a formal definition, i.e. *event stores*, and defined a corresponding incremental update scheme. Moreover, we have shown several possible instantiations of such an event store, using different existing data storage techniques originating from different areas of data stream processing. We have identified what are, from a process mining perspective, the main problems of using these techniques "out of the box". Based on these observations, combined with the potential presence of similar behaviour in process oriented event data, we have proposed a new data storage technique that overcomes these shortcomings. Using real event data we have conducted experiments with the proposed technique. We observe that we are able to slightly improve the quality of the process mining analyses applied on top of such storage, yet the increase is not significant. However, the use of prefix-tree based storage does allow us to use less memory, as the real data used for the experiments indeed shows some shared behaviour among different process instances.

3.7.2 Limitations

One of the main challenges in process mining is adequately discovering and visualizing the inherent parallelism present in processes. Such parallelism, together with loop behaviour, i.e. repeated executions of certain parts of the process, form challenges for the applicability of the proposed techniques. Parallelism yields a large variety in terms of behaviour. As a consequence, in the context of prefix-tree based storage, a large amount of *unique paths*, i.e. sequences of edges from the root to a leaf of the process trees, are constructed. Hence, we are not able to effectively exploit behavioural similarity.

The aforementioned problem is partly solved by representing the vertices as the Parikh representation of the sequence they represent. For example, if traversing the edges leading to a specific vertex, starting from the root, yields sequence $\langle a, b, c, b, d \rangle$, we transform it to multiset $[a, b^2, c, d]$. All possible permutations of $\langle a, b, c, b, d \rangle$ now lead to the vertex representing $[a, b^2, c, d]$. Using such abstraction on top of the state represented by vertices in the prefix-tree yields an acyclic graph instead of a tree. This allows us to compress the prefix-tree and handle parallelism. However, reconstructing cases is possibly more computationally complex since vertices potentially have multiple incoming arcs.

Storage of additional payload within the prefix-tree also poses problems with respect to removal of events. As indicated in subsection 3.4.3, we either keep payload in memory for an extended period of time, or remove it and resort to sampling. Clearly, when we store the payload for an additional period of time, this potentially causes us to exceed available memory. At the same time, when removing it from vertices no longer covered by the corresponding case identifier, sampling potentially leads to misleading results.

3.7.3 Open Challenges & Future Work

The techniques covered in this chapter solely focus on temporal storage of events. In particular, we assess the applicability of existing techniques to do so, and, compare it with prefix-tree based storage, which exploits the fact that an event stream originates from a running process instance. None of the techniques however explicitly focusses on intelligently removing traces of event behaviour. The prefix-tree based storage allows us to approximate the past behaviour of a process instance that is already removed from the underlying event store. However, whenever such event store completely removes everything related to a certain process instance, it is removed from the prefix-tree as well. In that sense the prefix-tree based storage completely depends on the quality of the underlying event store, with respect to. its own behaviour.

In light of the aforementioned, it remains a challenge to estimate:

1. For events related to a *completely new case identifier*, whether or not it is likely that this is indeed new behaviour, or it relates to ongoing, already stored, behaviour.
2. For traces of behaviour that are already present in the event store, how likely it is that new events are to be expected related to the case identifier.

When we are able to estimate the two main focal points, i.e. related to process instance initialization and termination, we are able to build a new type of event store. Such an event store needs to be able to selectively ignore certain events from insertion if these are likely to jeopardize data quality, i.e. not due to noise yet due to the high likelihood that data was missed. Moreover, it needs to be able to assess what parts of the behaviour are safe to remove, i.e. which process instances are most likely terminated and thus are not likely to generate new events. Hence, we envision tailor-made techniques for the purpose of event storage, i.e. more advanced means of temporal event storage.

Chapter 4

Filtering Infrequent Behaviour From Event Streams

The techniques introduced in chapter 3, allow us to temporarily store the events emitted on the event stream. As a consequence, the techniques allow us to lift process mining to the domain of streaming data. However, in practice, noise, infrequent behaviour and/or other types of anomalies are present within streaming event data. Therefore, in this chapter, we present an online filtering technique that allows us to identify and remove events that relate to infrequent behaviour. The technique acts as an event processor, i.e. we are able to plug the filtering technique on top of an input stream and generate an output event stream out of which infrequent behaviour is removed.

The contents presented in this chapter are based on the following publication:

S.J. van Zelst, M. Fani Sani, A. Ostovar, R. Conforti, and M. La Rosa. Filtering Spurious Events from Event Streams of Business Processes. *Advanced Information Systems Engineering, 30th International Conference, CAiSE 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings,* volume 10816 of *Lecture Notes in Computer Science,* pages 35-52. Springer, 2018;

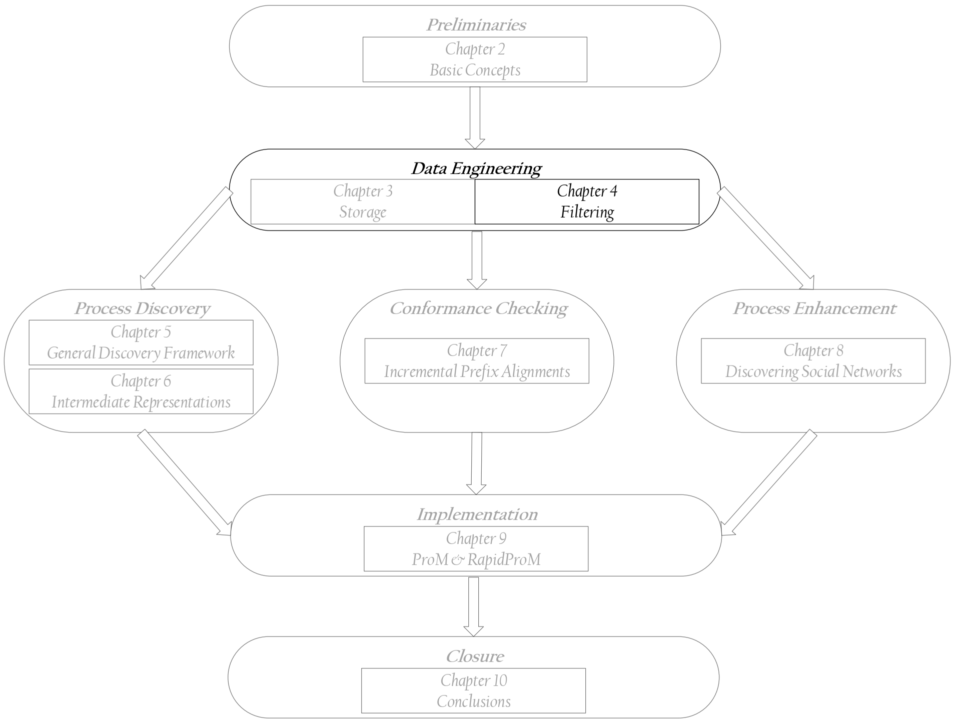


Figure 4.1: The contents of this chapter, i.e. techniques for the purpose of event stream filtering, highlighted in the context of the general structure of this thesis.

4.1 Introduction

In the previous chapter, we presented several techniques that allow us to temporarily store events emitted onto an event stream, originating from the underlying process. Thus far, we have assumed an event stream to be free of noise and anomalous behaviour. However, in reality, several factors cause this assumption to be wrong, e.g. the supporting information system may trigger the execution of an inappropriate activity that does not belong to the process, or the system may be overloaded resulting in logging errors. The existence of these anomalies in event streams, and event data in general, easily leads to unreliable results.

For example, reconsider the running example process as presented in Figure 2.13a, on page 55. When we apply a state-of-the art process discovery algorithm, i.e. the Inductive Miner [78], on an event log containing *noise-free behaviour*, i.e. only traces that are in the language of the example process, we obtain the model depicted in Figure 4.2a.

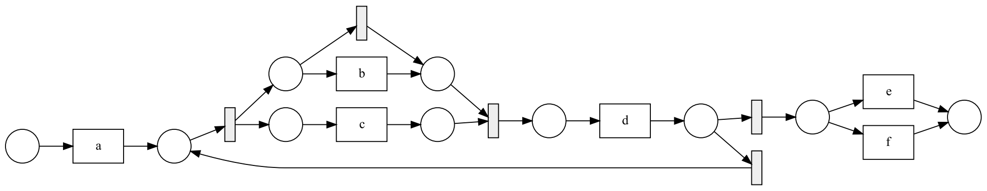
Observe that the Petri net depicted in Figure 4.2a is slightly different compared to the model as presented in Figure 2.13a. This is due to the fact that the algorithm used to discover the model does not allow us to discover duplicate labels, i.e. in Figure 2.13a both transition t_4 and t_5 have a label d. However, the language described by both models is the same. The model depicted in Figure 4.2b is discovered using the same algorithm, and almost the same data. However, we added a duplicate c- and e activity to one of the traces in the input data. As a result, the model now describes that we are able to always repeat both activity c and e infinitely often, even though in the data, both activities were duplicated only once. Hence, only a slight fraction of infrequent noisy behaviour in the event data easily leads to process models that are severely under-fitting with respect to the input data.

In the case of the Inductive Miner, the resulting process model is still a sound workflow net (cf. 2.5). This is due to the algorithm itself, i.e. the Inductive Miner by definition returns sound workflow nets. Other algorithms do not provide such guarantees and are therefore even more affected by the presence of noisy behaviour. For example, in case of the Alpha algorithm [11], the duplication of the c-activity does not allow us to find any Petri net place connected to a transition with the corresponding label, i.e. a transition labelled c.[1] As such, that transition is enabled at any point in time and hence such model is even more imprecise than the model as discovered by the Inductive Miner (cf. Figure 4.2b).

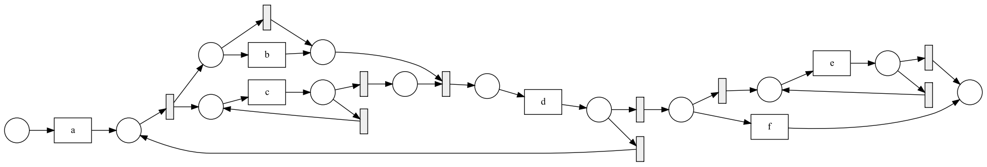
To tackle the aforementioned problem in the context of event streams, we present a general-purpose event stream filter designed to detect and remove *infrequent behaviour* from event streams. The presented approach relies on a time-evolving subset of the behaviour of the total event stream, out of which we infer an incrementally-updated model that represents this behaviour. In particular, we build a collection of dynamically updated *probabilistic automata* (cf. 2.9) that represent the subset of behaviour and are used to filter out infrequent behaviour. The filter we propose primarily focuses on the control-flow of the process (i.e. the sequential ordering of activities). As such we aim to determine whether an activity executed in the context of a certain process instance is plausible, given the recent history of executed activities for that same process instance.

Using a corresponding implementation of the approach, we evaluate the accuracy and performance of the filter by means of multiple quantitative experiments. To this end, we evaluate the proposed filter using a set of streams generated from a collection of synthetic process models inspired by real-life business processes. Moreover, to illustrate the applicability

[1] As the Alpha algorithm has difficulty to (re)discover the running example model, i.e. on the basis of noise-free behaviour, we do not explicitly show its result here, we merely indicate the additional problems caused by the presence of noise.



(a) Discovered based on data without infrequent noisy behaviour.



(b) Discovered based on data containing infrequent noisy behaviour.

Figure 4.2: Result of applying the Inductive Miner [78] on data obtained from the running example process (cf. Figure 2.13a), with (cf. Figure 4.2a) and without (cf. Figure 4.2b) the presence of infrequent noisy behaviour.

of the approach with respect to existing online process mining techniques, we asses the benefits of the proposed filter when applied prior to the application of online drift detection.

The remainder of this chapter is organized as follows. First, in section 4.2, we present a general taxonomy of behaviour, on the basis of control-flow, and indicate, which classes of behaviour are potentially identified and removed by the proposed filter. In section 4.3 we present the general architecture of the proposed filtering method. In section 4.4, we present the main, automaton based, approach, which we evaluate in section 4.5. We present related work in the domain of event-based filtering in section 4.6. We conclude this chapter and discuss directions for future work in section 4.7.

4.2 A Control-Flow-Oriented Taxonomy of Behaviour

Prior to presenting the general architecture of the proposed filter, we present a control-flow-oriented behavioural taxonomy, specifically tailored towards process mining data. We furthermore highlight which of the identified classes of the taxonomy, are covered by the proposed filter.

In general, we identify three major data characteristics, along the lines of which we are able to classify process mining data.

- *Trustworthiness*
  Indicates to what degree the recorded behaviour corresponds to reality, i.e. what actually happened during the execution of the process. In the case that the behaviour, e.g. emitted on an event stream, correctly reflects reality, e.g. no erroneous duplication of events, we consider the behaviour as trustworthy.
- *Compliance*
  Indicates to what degree the recorded behaviour is in accordance with predefined rules and/or expectations of the process. In some cases, rules and/or legislations dictate that explicit forms of behaviour are required. In other cases, service level agreements dictate

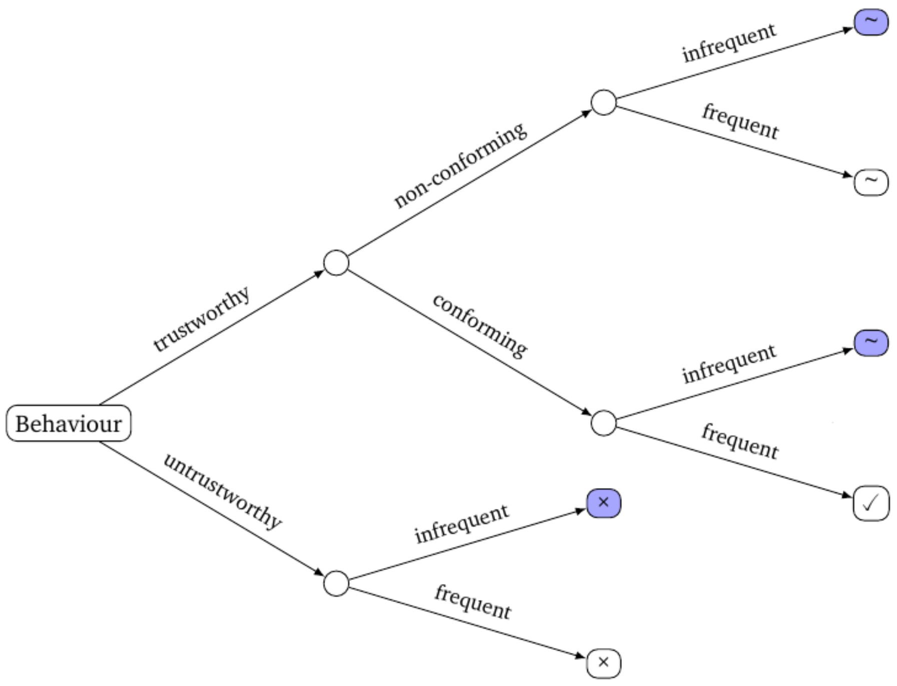


Figure 4.3: Control-flow-oriented taxonomy of behaviour. The types of behaviour that, ideally, are used in process mining are marked with ✓. Behaviour that is ideally removed is marked with ×. Behaviour of which the type of process mining analysis determines inclusion, is marked with ∼. The types of behaviour that are identified by the proposed filter, i.e. infrequent behaviour, are highlighted in blue.

an idealized and/or expected execution of behaviour. In the case that the behaviour correctly corresponds to such rules/expectations, we consider the behaviour as compliant.

- *Frequency*

  Indicates the relative frequency of the behaviour, i.e. compared to other observed executions of the same process.

In Figure 4.3, we present a graphical overview of the different characteristics, and their relation. It depends on the type of process mining task one performs, to what degree we aim to include certain types of behaviour in the analysis. However, observe that, when behaviour is untrustworthy, in general, we are not interested in including it, i.e. we are not able to trust the behaviour, and thus draw any significant meaningful conclusions from it. Hence, even in the case of conformance-checking-oriented process mining studies, we aim to remove the untrustworthy behaviour. In fact, we aim to omit any form of untrustworthy behaviour. However, note that, even though behaviour is untrustworthy, systematic errors may cause it to occur frequently. Therefore, in Figure 4.3, we do distinguish between frequent and infrequent forms. In the case that behaviour is trustworthy, frequent and compliant, we always aim to include it. When we perform process discovery, it is most likely that we aim to only include such frequent compliant

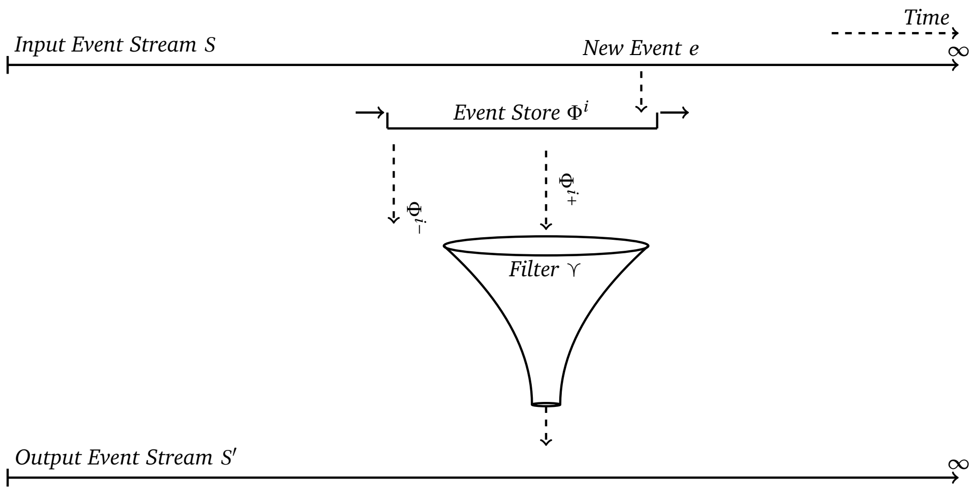


Figure 4.4: Schematic overview of the proposed filtering architecture.

behaviour and leave out any other type of trustworthy behaviour. When we apply conformance checking, it is more likely that all trustworthy behaviour is required to be included. Observe that in Figure 4.3, we explicitly highlight the types of behaviour, i.e. infrequent behaviour, that the presented filter is able to identify and remove.

4.3 Architecture

In this section, we present the basic architecture of the proposed event filter. The filter uses the behaviour stored in an underlying event store to identify and remove infrequent behaviour. Moreover, it is intended to be updated incrementally when new events are added to the event store. Based on the behaviour stored in the event store, we construct multiple *probabilistic automata* (cf. subsection 2.2.2, 2.9), which describe the behaviour captured within the event store. A state within a probabilistic automaton represents *a view* on recently observed behaviour of a specific process instance as represented by its corresponding case identifier. The outgoing transition probabilities of a state are based on observed behaviour for that state, as temporarily described by the event store. If, for a certain process instance, a new event arrives, we check whether that event is probable, based on the recorded probability distributions. If it is probable, we forward the event to the output stream, if not, we refrain from forwarding it.

In Figure 4.4, we depict a high-level overview of the architecture of the proposed filter. We assume that the input event stream S contains both proper- and spurious events. We maintain an event store Φ as defined and presented in chapter 3. In particular, within the filter, we use the case view of such store, i.e. $\Phi_{\mathcal{C}}$. A new event e is, in case it is stored within the event store Φ, forwarded to event filter $\curlyvee$. From an architectural point of view, we do not pose any strict requirements on the dynamics of the filter. We do however aim to let filter $\curlyvee$ reflect the behaviour captured within the event store Φ. Hence, the filter typically needs to process a new event within its internal representation, prior to applying the actual filtering. Furthermore, when events are removed from the underlying event store, i.e. caused by the addition of the newly received event e, we aim to process such removal within the filter as well. For the newly received event, the filter Υ either decides to emit the event onto output stream S', or, to discard it. In any case, it is always incorporated in the internal representation of the filter. We mainly do so because of the fact that typically, concept drift initially seems to be outlier behaviour, i.e. only over time, the concept drift becomes clear.

4.4 Automaton Based Filtering

Given the general architecture as presented in Figure 4.4, in this section we propose an instantiation of filter Υ. We first present the conceptual idea of the use of collections of probabilistic automata for the purpose of spurious event filtering, after which we describe how to incrementally maintain the collection of automata, and, how to effectively filter.

4.4.1 Prefix-Based Automata

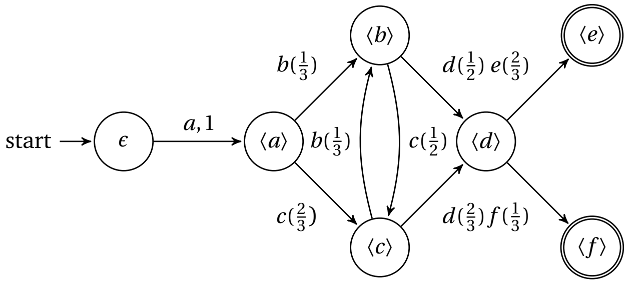
Within the proposed filter instantiation, a collection of probabilistic automata represents recent, *control-flow oriented*, behaviour observed on the event stream. These automata are used to determine whether new events are, according to their probability distributions, likely to be spurious or not. Each state within an automaton refers to a view on the (recent) historical behaviour of the process instances described by recently received events on the event stream. Such a state, for example, represents the three most recent activities performed for a certain process instance. The automata considered here can be regarded as extended/decorated variants of the transition systems described in [10].

The probabilities of the outgoing arcs of a state are based on the behaviour exhibited by process instances that have been in that state before and subsequently moved on to a new state by means of a new event. Upon receiving a new event, we assess the state of the process instance described by the event and check, based on the distribution as defined by that state's outgoing arcs, whether the new event is likely to be spurious or not. As an example, consider Figure 4.5a on page 100. Within the example automaton, for each process instance, the most recent event represents its state, e.g. if we have observed $\langle a, b, c\rangle$ for some process instance, the corresponding state is $\langle c\rangle$. Observe that for that state within the example automaton, based on historical behaviour, i.e. previously observed process instances, we recorded that in $\frac{2}{3}$ of these instances activity d was observed and in $\frac{1}{3}$ of these instances we observed activity b. Thus, if we observe, for the process instance with behaviour $\langle a, b, c\rangle$, a new event describing c, we deem it rather likely that the new event is spurious.

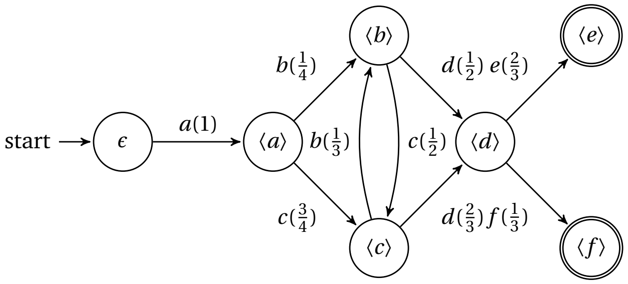
The probabilistic automata that we construct contain states that represent recent control-flow oriented behaviour for the process instances currently captured within the event store. As such, each state refers to a (partial) prefix of the process instance's most recent behaviour, and hence, we deem these automata *prefix-based automata*. Therefore, in prefix-based automata, a state q represents a (abstract view on a) *prefix of executed activities*, whereas outgoing arcs represent those activities $a \in \mathscr{A}$ that are likely to follow the prefix represented by q, and their associated probability of occurrence. We define two types of parameters, that allow us to deduce the exact state in the corresponding prefix automaton based on a prefix, i.e.

1. *Maximal Abstraction Window Size*

   Represents the size of the window to take into account when constructing states in the automaton. For example, if we use a maximal window size of 5, we only take into account



(a) Automaton based on three full (simple) traces of the running example, cf. Figure 2.13, i.e. $\langle a,b,c,d,e\rangle$, $\langle a,c,b,d,f\rangle$ and $\langle a,c,d,e\rangle$ and one partial (i.e. ongoing) trace $\langle a\rangle$.



(b) Similar automaton to Figure 4.5a, after receiving a new event c for incomplete trace $\langle a\rangle$ (i.e. yielding trace $\langle a,c\rangle$). As a consequence, the outgoing arcs of $\langle a\rangle$ get different probabilities.

Figure 4.5: Example of maintaining a prefix-based automaton, using a window size of 1 (the same for all abstractions).

the five most recent events present in the event store for the process instance under consideration as being recent behaviour.

2. *Abstraction*

   Represents the abstraction that we apply on top of the derived window of recent historical behaviour, i.e. subject to the *maximal window size* parameter, in order to define a state. We identify the following abstractions:

   - *Identity*

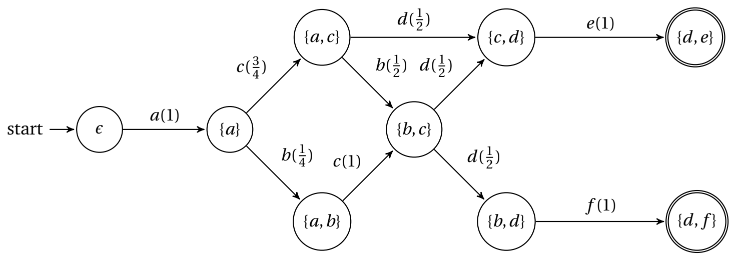
     Given window size $w \in \mathbb{N}$ and a trace $\sigma \in \mathscr{A}^*$, present in $\tilde{\Phi}$, the identity abstraction id^w yields the prefix as a state, i.e. $id^w\colon \mathscr{A}^* \to \mathscr{A}^*$, where $id(\sigma) = \sigma_{|\sigma|-(w+1)\ldots|\sigma|}$.

   - *Set*

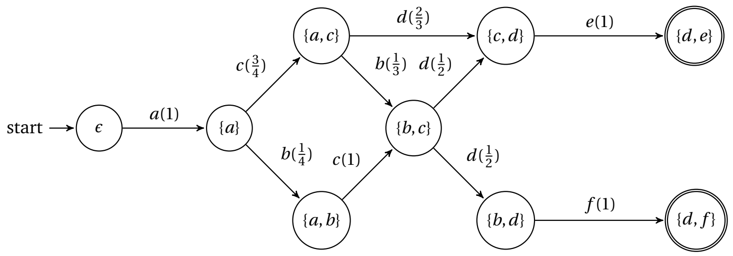
     Given window size $w \in \mathbb{N}$ and a trace $\sigma \in \mathscr{A}^*$, present in $\tilde{\Phi}$, the set abstraction indicates the presence of $a \in \mathscr{A}$ in the last w elements of σ, i.e. we apply $elem(\sigma_{|\sigma|-(w+1)\ldots|\sigma|})$.

   - *Parikh*

     Given window size $w \in \mathbb{N}$ and a trace $\sigma \in \mathscr{A}^*$, present in $\tilde{\Phi}$, the Parikh abstraction $parikh$ yields a multiset describing the number of occurrences of $a \in \mathscr{A}$ within σ, i.e. we apply $\sigma_{|\sigma|-(\vec{w}+1)\ldots|\sigma|}$.



(a) Automaton based on three full (simple) traces of the running example, cf. Figure 2.13, i.e. $\langle a,b,c,d,e\rangle$, $\langle a,c,b,d,f\rangle$ and $\langle a,c,d,e\rangle$ and one partial (i.e. ongoing) trace $\langle a,c\rangle$.



(b) Similar automaton to Figure 4.6a, after receiving a new event d for incomplete trace $\langle a,c\rangle$ (i.e. yielding trace $\langle a,c,d\rangle$). As a consequence, the outgoing arcs of $\{a,c\}$ get different probabilities.

Figure 4.6: Example of maintaining a prefix-based automaton, using a window size of 2 with a set abstraction.

Consider Figure 4.5 and Figure 4.6, in which we present different examples of prefix-automata. The automata in Figure 4.5 are based on four traces of behaviour, which are based on the running example depicted in Figure 2.13a. Three of these traces, i.e. $\langle a,b,c,d,e\rangle$, $\langle a,c,b,d,f\rangle$ and $\langle a,c,d,e\rangle$, are based on *full process behaviour*, i.e. these traces related to termination of the process instance. Finally, we also assume we maintain behaviour to a running instance of the process, for which only the (first) activity a is observed. In the automaton in Figure 4.5b, the effect of receiving a new event (an event describing activity c for trace $\langle a\rangle$), with respect to the transition probabilities is visualized, which is detailed on in the subsequent section, i.e. subsection 4.4.2. In Figure 4.6, we visualize the effect of subsequently receiving a d activity for the incomplete process instance $\langle a,c\rangle$. In Figure 4.5, we use a maximal window size of 1, together with the *identity* abstraction. Note that, due to this window size, each of the abstractions mentioned, i.e. *identity*, *set* and *Parikh*, yields the same automaton. In Figure 4.6, we use a maximal window size of 2, together with a set abstraction. For the traces of behaviour used in the two examples, a window size of 2 combined with a *Parikh* or *set* view retains the same automata. However, combining it with the identity abstraction yields a different automaton, e.g. using the *set/Parikh* abstraction yields $\{b,c\}$ or $[b,c]$ based on both $\langle b,c\rangle$ or $\langle c,b\rangle$. Furthermore, it is important to note that, when using a window size of k, we only start effectively using the automaton in filtering when the received events for a certain process instance describe a sequence of length k. Hence, in Figure 4.6, only the states describing two events, e.g. $\{a,c\}$, $\{a,b\}$ etc. are used when filtering.

As exemplified by the automata depicted in Figure 4.5 and Figure 4.6, the window size influences the degree of generalization of the automaton. For example, the automata in Figure 4.5 allow for an infinite repetition of b and c activities in states $\langle c \rangle$ and $\langle b \rangle$, respectively. The automata in Figure 4.6 do not allow this, i.e. they are more precise with respect to their training data. Observe that, increasing the maximal window size is likely to generate automata of larger size, i.e. we are able to distinguish a wider variety of states, and is thus likely to be more memory intensive. Hence, we have a trade-off between precision of the automata with respect to training data and memory complexity.

4.4.2 Incrementally Maintaining Collections of Automata

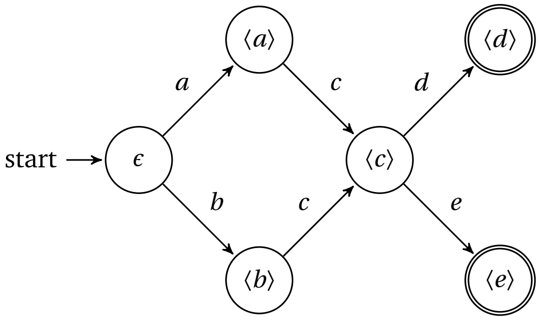
In this section, we indicate how we aim to maintain a collection of automata which we use to filter. Prior to this, we motivate the need for using multiple automata within filtering.

Consider that we observe an event stream, running on the basis of just two simple traces, i.e. $\langle a, c, d \rangle$ and $\langle b, c, e \rangle$. Furthermore, observe that within the data, there is a *long-term dependency* between, on the one hand, a and d, and, b and e, on the other hand. Consider Figure 4.7, in which we present two automata constructed on the basis of the two simple traces. In both automata we use an identity abstraction, yet in Figure 4.7a, we use window size of 1, whereas in Figure 4.7b, we use a window size of 2. For simplicity, we have omitted the probabilities of the edges of the automata. Note that, when only using the automaton depicted in Figure 4.7a, we no longer observe the long-term dependency. As a result, whenever an event describes the occurrence of activity e after earlier observed prefix $\langle a, c \rangle$, we are not able to identify this as being infrequent behaviour, i.e. both $\langle a, c \rangle$ and $\langle b, c \rangle$ are translated into state $\langle c \rangle$. In the automaton in Figure 4.7b, this is however possible. Hence, we aim to use automata using different window sizes, which allows us to generalize on the one hand (smaller window sizes), yet, also allows us to detect certain long-distance patterns (larger window sizes).

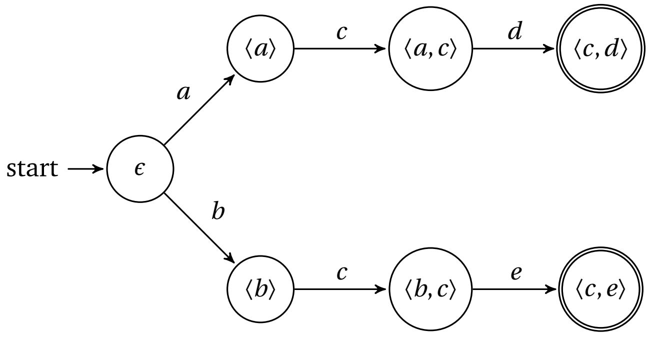
As new events are emitted on the stream, we aim to keep the automata up-to-date in such way that they reflect the behaviour present in the event store at time i, i.e. $\Phi^i(S)$. Let $k > 0$ represent the maximal abstraction window size we want to take into account when building automata. We maintain k prefix-automata, where for $1 \leq j \leq k$, automaton $PA_j = (Q_j, \Sigma_j, \delta_j, q_j^0, F_j, \gamma_j)$ uses maximal abstraction window size j to define its state set Q_j. Upon receiving a new event, we incrementally update the k maintained automata. Consider receiving the i^{th} event $S(i) = e$, with $\pi_{\mathsf{c}}(e) = c$ and $\pi_{\mathsf{a}}(e) = a$. Moreover, assume that the event is added to the event store Φ^i, i.e. $\Phi^{i+} = \{e\}$, and hence we aim at processing it within the collection of automata. We additionally let $\sigma = \sigma' \cdot \langle a \rangle = \pi_{\mathsf{a}}^*(\Phi^i(S, c))$, i.e. σ represents the current trace (control-flow perspective) known for case c whereas σ' represents the complete prefix of the current trace stored for case c, excluding the activity described by the newly received event e.

To update automaton PA_j we apply the abstraction of choice on the prefix of length j of the newly received event in σ', i.e. $\langle \sigma'(|\sigma'| - j + 1), ..., \sigma'(|\sigma'|) \rangle$, to deduce the corresponding state $q_{\sigma'} \in Q_j$. The newly received event influences the probability distribution as defined by the outgoing arcs of $q_{\sigma'}$, i.e. it describes that $q_{\sigma'}$ can be followed by activity a. Therefore, instead of storing the probabilities of each γ_j, we store weighted outdegree of each state $q_j \in Q_j$, i.e. $\deg_j^+(q_j)$. Moreover, we store the individual contribution of each $a \in \mathscr{A}$ to the outdegree of q_j, i.e. $\deg_j^+(q_j, a)$ with $\deg_j^+(q_j, a) = 0 \Leftrightarrow \delta(q_j, a) = \emptyset$. Observe that $\deg_j^+(q_j) = \sum_{a \in \mathscr{A}} \deg_j^+(q_j, a)$, and, that deducing the empirical probability of activity a in state q_j is trivial, i.e. $P(a \mid q_j) = \frac{\deg_j^+(q_j, a)}{\deg_j^+(q_j)}$.

Reconsider the example automaton in Figure 4.5, and consider that we receive an event



(a) Automaton (probabilities omitted) describing the behaviour of the process, i.e. traces $\langle a,c,d \rangle$ and $\langle b,c,e \rangle$, using a window size of 1.



(b) Automaton (probabilities omitted) describing the behaviour of the process, i.e. traces $\langle a,c,d \rangle$ and $\langle b,c,e \rangle$, using a window size of 2.

Figure 4.7: Two automata, using different window lengths, i.e. length 1 and 2, describing the behaviour of the process, i.e. traces $\langle a,c,d \rangle$ and $\langle b,c,e \rangle$. Only using a window length of 2 allows us to observe the *long-term dependencies* as described by the data.

related to activity c, which in turn belongs to the same case as the simple trace $\langle a \rangle$. Hence, we obtain a new simple trace $\langle a,c \rangle$ for the corresponding case identifier. As we use a window size of 1, we deduce that the corresponding abstraction, and thus the new state in the automaton related to that case is $\langle c \rangle$. Clearly, the previous state is $\langle a \rangle$. We observe a total of 4 traces that describe an action out of state $\langle a \rangle$, three of which describe activity c, i.e. $\langle a,c,b,d,f \rangle$, $\langle a,c,d,e \rangle$ and $\langle a,c \rangle$. Only one of the simple traces describes activity b after state a, i.e. $\langle a,b,c,d,e \rangle$. Hence, we deduce empirical probability $\frac{3}{4}$ for activity c and $\frac{1}{4}$ for activity b.

Updating the automata based on events that are removed from the event store, i.e. based on the elements of Φ^{i-}, is performed as follows. Again, assume that we receive a new event e at time $i > 0$ related to a process instance identified by some case identifier c. For all $c \in \mathscr{C}$, we let $\sigma'_c = \Phi^{i-1}(S,c)$, $\sigma_c = \Phi^{i}(S,c)$ and moreover, we let $\Delta_c(i) = |\sigma'_c| - |\sigma_c|$. Observe that for any case identifier $c \in \mathscr{C}$, that *does not relate to the newly received event*, we have:

$$\Delta_c(i) \geq 0 \tag{4.1}$$

Observe that this is the case since events are potentially dropped for such case, yet no new events are received, hence $|\sigma_c| \leq |\sigma'_c|$. In a similar fashion, for the process instance identified by case c that relates to the new event e, we have:

$$\Delta_c(i) \geq -1 \tag{4.2}$$

Observe that this is the case since either $|\sigma_c| = |\sigma'_c| + 1$, or, $|\sigma_c| \leq |\sigma'_c|$.

Thus, to keep the automata in line with the events stored in the event window, in the former case we need to update the automata if $\Delta_c i > 0$, i.e. at least one event is removed for the corresponding case identifier, whereas in the latter case we need to update the automata if $\Delta_c i \geq 0$. Therefore, we define $\Delta'_c(i) = \Delta_c(i)$ for the former case and $\Delta'_c(i) = \Delta_c(i) + 1$ in the latter case. Henceforth, if for any $c \in \mathscr{C}$, we need to update the maintained automata to account for removed events in case:

$$\Delta'_c(i) > 0 \tag{4.3}$$

To update the collection of k maintained automata, for each $1 \leq j \leq \Delta'_c(i)$ we generate sequences $\langle \sigma'(j) \rangle$, $\langle \sigma'(j), \sigma'(j) + 1 \rangle$, ..., $\langle \sigma'(j), ..., \sigma'(j+k) \rangle$ (subject to $|\sigma'| > j + k$). For each generated sequence, we apply the abstraction of choice to determine corresponding state q, and subsequently reduce the value of $\deg^+(q)$ by 1. Moreover, assume that the state q corresponds to sequence $\langle \sigma'(j), \sigma'(j+1), ..., \sigma'(j+l) \rangle$ with $1 \leq j \leq \Delta'_c(i)$ and $1 \leq l < k$, we additionally reduce $\deg^+(q, a)$ by 1, where $a = \sigma'(j + l + 1)$.

4.4.3 Filtering Events

After receiving an event and subsequently updating the collection of automata, we determine whether the new event is spurious or not. To determine whether the newly arrived event is spurious, we assess to what degree the empirical probability of occurrence of the activity described by the new event is an outlier with respect to the probabilities of other outgoing activities of the current state. Given the set of k automata, for automaton $PA_j = (Q_j, \Sigma_j, \delta_j, q^0_j, F_j, \gamma_j)$ with prefix-length j $(1 \leq j \leq k)$, we characterize an automaton specific filter as $\curlyvee_j \colon Q_j \times \Sigma_j \to \mathbb{B}$.[2] Note that an instantiation of a filter $\curlyvee_j$ often needs additional input, e.g. a threshold value or range. The exact characterization of $\curlyvee_j$ is a parameter of the approach, however, we propose and evaluate the following instantiations:

- *Fractional*;

  Considers whether the probability obtained is higher than a given threshold, i.e. $\curlyvee^F_j \colon Q_j \times \Sigma_j \times [0,1] \to \mathbb{B}$, where:

  $$\curlyvee^F_j(q_j, a, \kappa) = 1 \text{ if } P(a \mid q_j) < \kappa \tag{4.4}$$

- *Heavy Hitter*;

  Considers whether the probability obtained is higher than a fraction of the maximum outgoing probability, i.e. $\curlyvee^H_j \colon Q_j \times \Sigma_j \times [0,1] \to \mathbb{B}$, where:

  $$\curlyvee^H_j(q_j, a, \kappa) = 1 \text{ if } P(a \mid q_j) < \kappa \cdot \max_{a' \in \mathscr{A}} P(a' \mid q_j) \tag{4.5}$$

[2] It is also possible to have $rng(\curlyvee_i) = [0,1]$, i.e. indicating the probability of an event being spurious, however, the filters we propose here all map to boolean values.

- *Smoothed Heavy Hitter*;

  Considers whether the probability obtained is higher than a fraction of the maximum outgoing probability subtracted with the non-zero average probability. Let $NZ = \{a \in \Sigma_j \mid P(a \mid q_j) > 0\}$, we define $\Upsilon_j^{SH} \colon Q_j \times \Sigma_j \times [0,1] \rightarrow \mathbb{B}$, where:

$$\Upsilon_j^{SH}(q_j, a, \kappa) = 1 \text{ if } P(a \mid q_j) < \kappa \cdot \left(\max_{a' \in \mathscr{A}} P(a' \mid q_j) - \frac{\sum\limits_{a' \in NZ} P(a' \mid q_j)}{|NZ|} \right) \tag{4.6}$$

For a newly received event, each automaton, combined with a filter of choice yields a boolean result indicating whether or not the new event is spurious. In the remainder, we assume that we apply the same filter on each automaton and we assume that when any of the k maintained automata signals an event to be spurious, the event itself is spurious. However, observe that this is not a strict necessity, i.e. different filters can be applied and alternative noise classifications schemes are eligible as well, e.g. majority vote. Finally, note that maintaining/filtering the automata can be performed in parallel, e.g. we maintain an automaton on each node within a cluster.

4.5 Evaluation

In this section, we evaluate the proposed event filter in two ways. First, we assess filtering accuracy and time performance on randomly generated event streams, based on *synthetic* process models, i.e. a collection of process models that resemble business processes often present in organizations. Second, we assess the applicability of our filter in combination with an existing class of online process mining techniques, i.e. *concept drift detection techniques*. In the latter experiment, we consider both synthetic and real-life datasets.

4.5.1 Filtering Accuracy and Time Performance

For this first set of experiments, we generated several event streams using 21 variations of the loan application process model presented in [51]. These variations are inspired by the change patterns as presented in [119]. Out of 21 stable models, we generated 5 different random event streams, each describing 5000 process instances, with a varying amount of events. For each generated stream, we randomly inserted spurious events with insertion probabilities ranging from 0.025 to 0.15 in steps of 0.025. In these experiments, we use a simple sliding window with fixed size as an implementation for Φ. We internally maintain a projection of the form $\mathscr{C} \rightarrow \mathscr{A}^*$ to accommodate for the case-view of Φ, i.e. $\Phi_{\mathscr{C}}$. Given a sliding window of maximal-size N, the first N events are used to construct an initial set of automata and are not considered within the evaluation. Moreover, each event arriving after the first N events that relates to any process instance that was observed within the first N events is ignored as well. As such we only consider new process instances within the filter.

Accuracy

We assess the impact of a wide variety of parameters on filtering accuracy. These are the maximal abstraction window size, the particular abstraction of use, the filtering technique and the filter threshold. The values of these parameters, used within the experiments, are presented

Table 4.1: Parameters of *Data Generation* and *Experiments* with *Synthetic Data*

Data Generation

Artefact/Parameter	*Value*
Number of Models	21
Number of Event Logs, generated per model	5
Probability of spurious event injection, per event log	$\{0.025, 0.05, ..., 0.15\}$

Experiments

Sliding Window Size	$\{2500, 5000\}$
Maximal Abstraction Window Size	$\{1, 3, 5\}$
Abstraction	{Identity (id), Parikh ($parikh$), Set ($elem$)}
Filter	{Fractional (Υ^F), Heavy Hitter (Υ^H), Smoothed Heavy Hitter (Υ^{SH}}
Filter Threshold (κ)	$\{0.05, 0.1, ..., 0.5\}$

in Table 4.1. Here, we mainly focus on the degree in which maximal abstraction size, abstraction, filtering method and window size influence the filtering quality. The results for each of these parameters are presented in Figures 4.8 – 4.11 on pages 107 – 110. Note that, to reduce the amount of data points and ease interpretability of the figures, we show results for noise levels 0.025, 0.05, 0.1 and 0.15, and threshold levels $0.05 - 0.25$.

For the maximal abstraction window size (cf. Figure 4.8), we observe that a prefix-size of 1 tends to outperform prefix-sizes of 3 and 5. This is an interesting observation as it shows that, for this collection of models and associated streams, ignoring large parts of a trace's history improves the results. Note that, for maximal prefix length k, we use k automata, and signal an event to be spurious whenever one of these signals that this is the case. Using a larger maximal prefix-length potentially identifies more spurious events, yielding higher recall values. However, a side effect is potentially lower precision values. Upon inspection, this indeed turns out to be the case, i.e. the differences in F1 score are explained by higher recall values for increased maximal prefix lengths, however, at the cost of lower precision.

As for the abstraction used (cf. Figure 4.9), we observe that the *Identity*- outperforms both the *Parikh*- and the *Set* abstraction (for these results a maximal window size of 1 is ignored, as all of the abstractions yield the same automaton). The results are explained by the fact that within the collection of models used, the amount of parallelism is rather limited, which does not allow us to make full use of the generalizing power of both the *Parikh*- and *Set* abstraction. At the same time, loops of short length exist in which order indeed plays an important role, which is ignored by the two aforementioned abstractions. Upon inspection, the recall values of all three abstractions are relatively equal, however, precision is significantly lower for both the *Parikh*- and *Set* abstraction. This can be explained by the aforementioned generalizing power of these abstractions, and, in turn, explains the difference in F1 score.

For the filter method used (cf. Figure 4.10), we observe that the *Smoothed Heavy Hitter* and *Heavy Hitter* outperform the *Fractional* filter for increasing threshold values. This is explained by the fact that the fractional filter poses a rigorous requirement on events to be considered non-spurious, e.g. a threshold value of $\frac{1}{4}$ requires an activity to occur at least in 25% of all behaviour in a certain state. The other two filters solve this by using the maximal observed value, i.e. if a lot of behaviour is possible, the maximum value is lower and hence the requirement to be labelled non-spurious is lower.

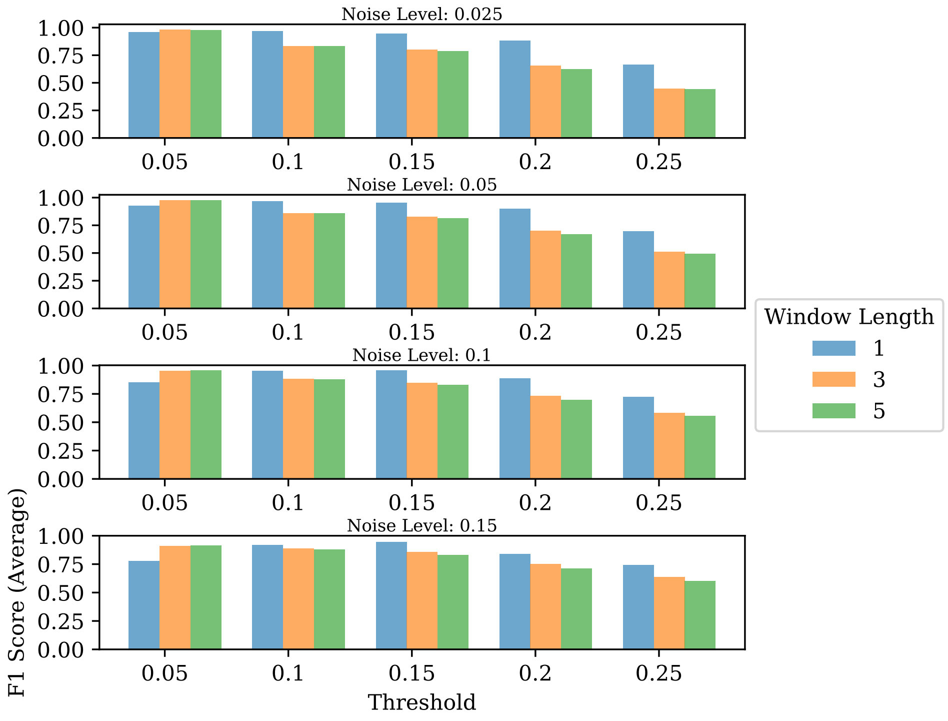


Figure 4.8: Average F1 score for different abstraction window sizes.

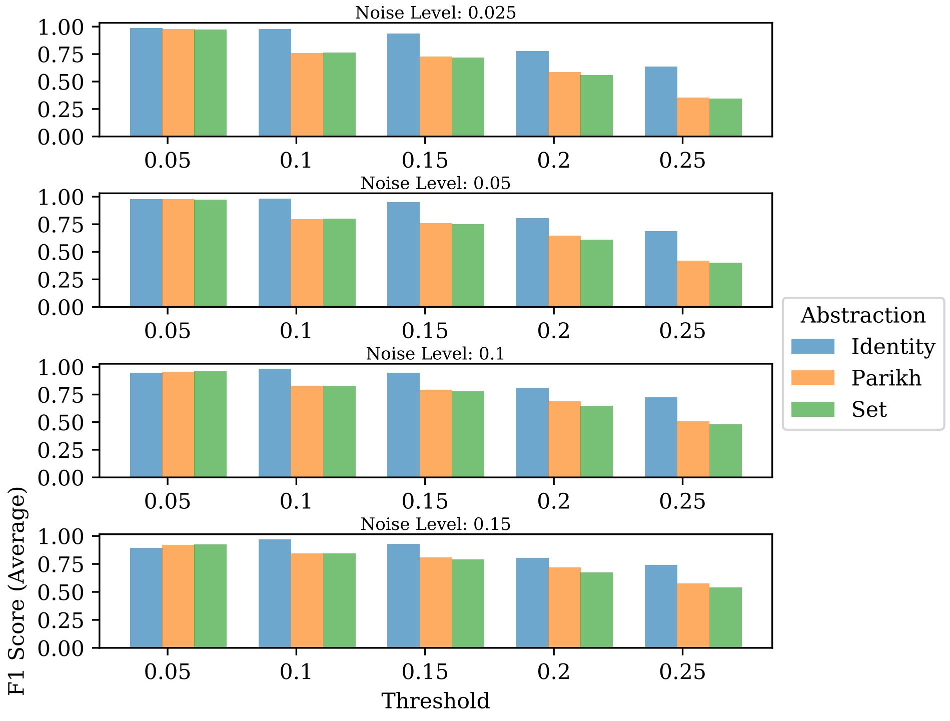


Figure 4.9: Average F1 score for different abstractions.

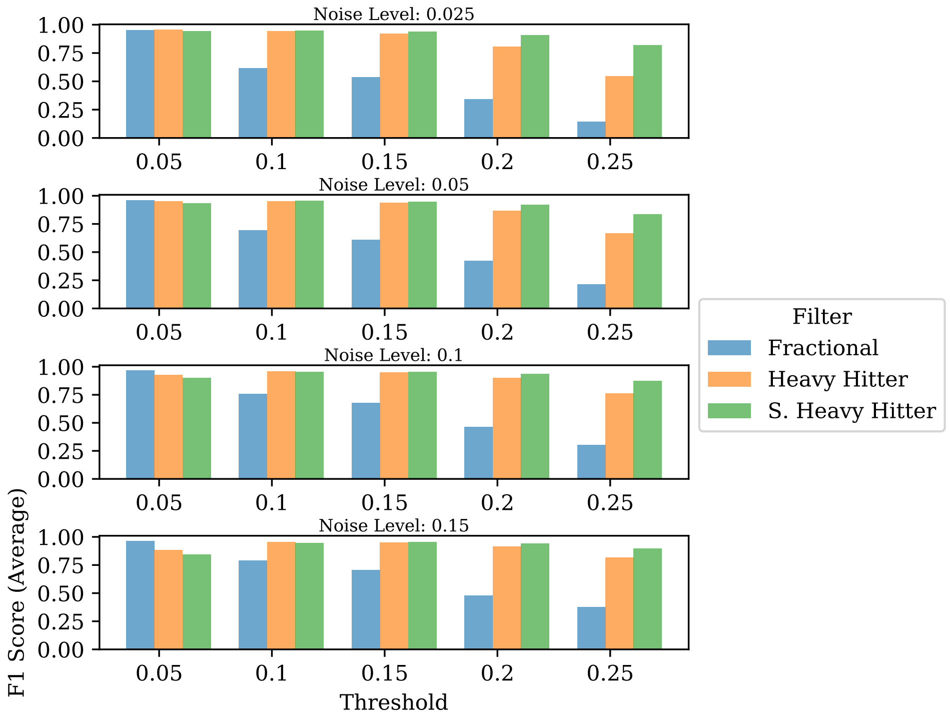


Figure 4.10: Average F1 score for different filter types.

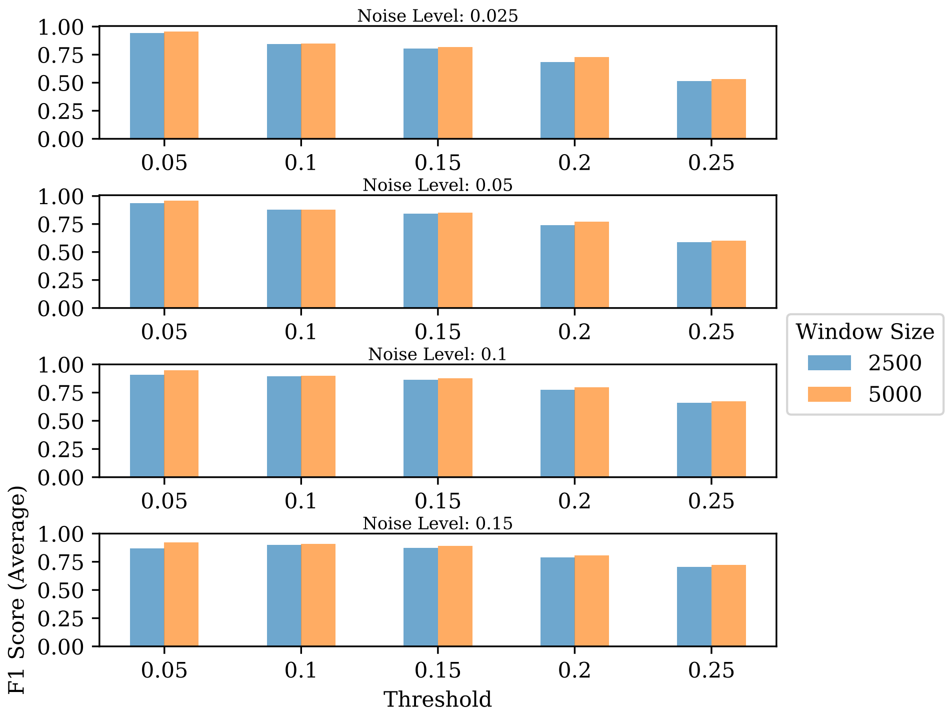


Figure 4.11: Average F1 score for different sliding window sizes.

Finally, we observe that an increased sliding window size does not affect the filter results significantly (cf. Figure 4.11). Since the process is stable, i.e. there is no concept-drift within the generated streams, this indicates that both window sizes used are large enough to deduce automata that allow us to accurately filter the event stream.

Figure 4.12, on page 112, shows how the average F1 score varies based on the percentage of noise and the threshold level. We observe that the F1 score slightly converges for the different threshold levels as noise increases (cf. Figure 4.12a). Interestingly, in Figure 4.12b, we observe that for relatively low threshold values, the range of F1 score values for various noise levels is very narrow, i.e. the filtering accuracy is less sensitive to changes in the noise level. This effect diminishes as the threshold increases, leading to more scattered yet lower F1 score values. Observe that, these observations coincide with the *Kendall rank correlation coefficient values* [12, 75] of $0,1792$ (Figure 4.12a) and $-0,8492$ (Figure 4.12b) respectively. We conclude that, for the dataset used, the threshold level seems to be the most dominant factor in terms of the F1 score.

Time Performance

The sliding window maintains a finite representation of the stream, thus, memory consumption of the proposed filter is finite as well. Hence, we focus on time performance, using one stream per base model with 15% noise, and several different parameter values. The experiments were performed on an Intel Xeon CPU (6 cores) 3.47GHz system with 24GB memory. Average event handling time was ~ 0.017 ms, leading to handling ~ 58.8 events per ms. These results confirm that automaton-based filtering is suitable to work in real-time/event stream based settings.

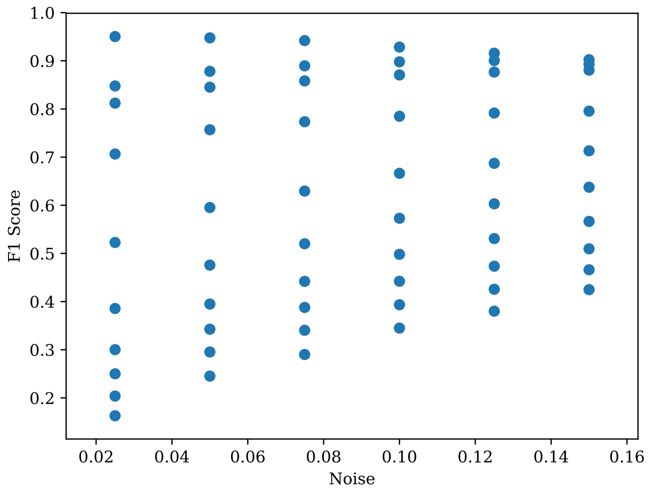
4.5.2 Drift Detection Accuracy

In a second set of experiments, we evaluate the impact of our filter on the accuracy of process drift detection. For this, we use a state-of-the-art technique for drift detection that works on event streams [97]. We apply our filter to the event streams generated from a variety of synthetic and real-life logs, with different levels of noise, and compare drift detection accuracy with and without the use of the proposed filter. We first discuss the experimental setup, after which we compare drift detection results obtained with and without the use of our filter.

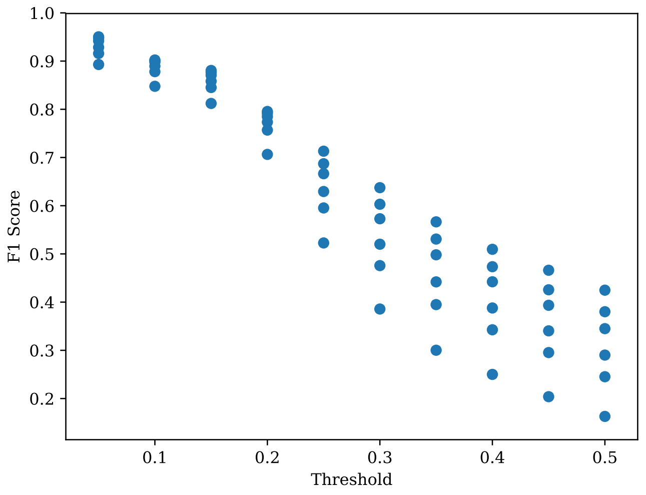
Experimental Setup

For these experiments, we used the 18 event logs proposed in [97] as a basis. The event data are generated by simulating a model featuring 28 different activities (combined with different intertwined structural patterns). Additionally, each event log contains nine drifts obtained by injecting control-flow changes into the model. Each event log features one of the twelve *simple change patterns* [119] or a combination of them. Simple change patterns may be combined through the insertion ("I"), resequentialization ("R") and optionalization ("O") of a pattern. This produces a total of six possible *nested change patterns*, i.e. "IOR", "IRO", "OIR", "ORI", "RIO", and "ROI". For a detailed description of each change pattern, we refer to [97].

Starting from these 18 event logs, we generated 36 additional event logs i.e. two for each original event log. One of the two generated event log contains 2.5% noise and the other contains 5% of noise. Noise is generated by means of inserting random events into traces of each log. Hence, the final corpus of data consists of 54 event logs, i.e. 12 simple patterns and 6 composite patterns with 0%, 2.5%, and 5% noise, each containing 9 drifts and approximately $250,000$ events.



(a) Average F1 score per noise level.



(b) Average F1 score per filter threshold level.

Figure 4.12: Average F1 score per noise- and threshold level.

Results on Synthetic Data

In this experiment, we evaluate the impact of the proposed filter on the accuracy of the drift detection technique proposed in [97]. We use the previously described corpus of data for the experiments. Figure 4.13 on page 114 illustrates the F1 score and mean delay of the drift detection, before and after the application of our filter over each change pattern.

The filter, on average, successfully removes 95% of the injected noise, maintaining and even improving the accuracy of the drift detection (with F1 score of above 0.9 in all but two change patterns). This is achieved whilst delaying the detection of a drift by less than 720 events on average (approximately 28 traces).

When considering noise-free event streams (cf. Figure 4.13a), the filter preserves the accuracy of the drift detection. For some change patterns ("rp", "cd", "IOR", and "OIR"), our filter improves the accuracy of the detection by increasing its precision. This is due to the removal of sporadic event relations, that cause stochastic oscillations in the statistical test used for drift detection. Figure 4.13b and Figure 4.13c show that noise negatively affects drift detection, causing the F1 score to drop, on average, to 0.61 and 0.55 for event streams with 2.5% and 5% of noise, respectively. This is not the case when our filter is applied, where an F1 score of 0.9 on average is achieved.

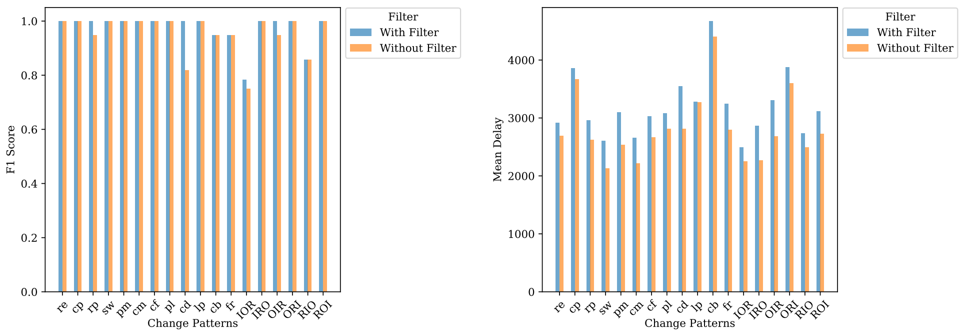
Finally, in terms of detection delay, the filter on average increases the delay by 370, 695, and 1087 events (15, 28, and 43 traces) for the logs with 0%, 2.5%, and 5% noise, respectively. This is the case since changes in process behaviour immediately following a drift are treated as noise.

Results on Real-Life Data

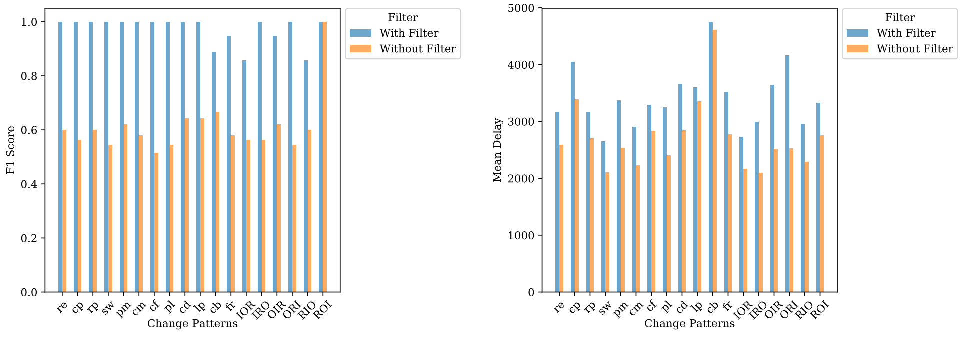
In this experiment, we assess whether the positive effects of our filter on drift detection, observed on synthetic data, translate to real-life data. For this, we used an event log containing cases of Sepsis (a life-threatening complication of an infection) from the ERP system of a hospital [85], i.e. as presented earlier in Table 1.1. The event log contains 1,050 cases with a total of 15,214 events belonging to 16 different activities.

For this experiment, we attempt to detect concept drift over the last 5,214 events, as the first 10,000 events are used to train the filter. Figure 4.14 plots the significance probability *p-value* curves of the statistical tests used for drift detection, both without (Figure 4.14a) and with (Figure 4.14b) the use of our filter. In order to detect a drift, the p-value of the drift detection technique needs to be below a user-specified significance probability threshold, commonly set to 0.05. Moreover,the p-value needs to be lower than the threshold for a given window of ϕ events. In the unfiltered case, cf. Figure 4.14a, we see two clear regions of p-values below the threshold, i.e. after the 2067$^{\text{th}}$ event and after the 4373$^{\text{rd}}$ event. In the case when applying the filter, cf. Figure 4.14b, we observe that there is much more oscillation in the p-value and we do not detect a clear drift.

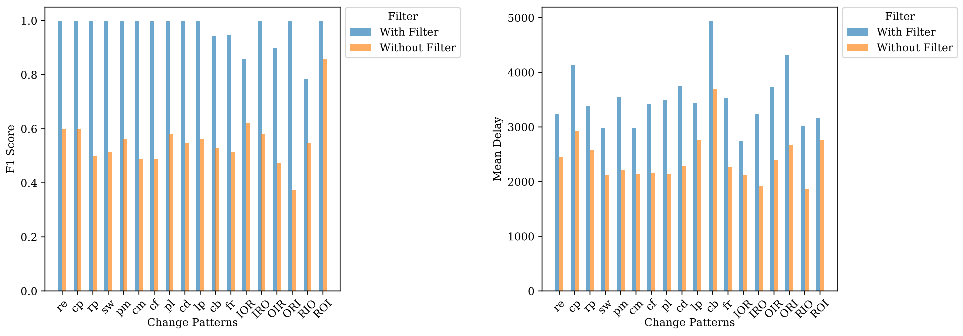
In the experiments with synthetic logs, we observed that the filter reduced the number of false positives (drift detected when it did actually not occur). To verify if this is also the case for the real-life event log, we profiled the direct-follows dependencies occurring before and after the drifts. The profiling indicates that while direct-follows dependencies "IV Antibiotics $\longrightarrow$ Admission NC" and "ER Sepsis Triage $\longrightarrow$ IV Liquid" are observed several times across the entire event stream, the infrequent direct-follows dependencies "Admission NC $\longrightarrow$ IV Antibiotics" and "IV Liquid $\longrightarrow$ ER Sepsis Triage" appear only in the proximity of the two drifts. These two infrequent dependencies cause a change in the underlying $\alpha+$ relations between the activities, which we use to detect the drifts (in this case changing from causal to concurrent). This change in the relation results in the detection of the drifts. These infrequent dependencies are removed



(a) Noise ratio = 0%.



(b) Noise ratio = 2.5%.



(c) Noise ratio = 5%.

Figure 4.13: Drift detection F1 score and mean delay (in number of events) per change pattern, obtained from the drift detection technique in [97] over filtered versus unfiltered event streams.

when applying the filter, which in turn does not lead to a clear concept drift. In light of these insights, we can argue that the two drifts detected over the unfiltered event stream are indeed false positives, confirming what we already observed on the experiments with synthetic logs, i.e. that our filter has a positive effect on the accuracy of drift detection.

4.6 Related Work

With respect to noise filtering in the context of conventional process mining, i.e. using event logs, several approaches are described in literature [40, 58, 118]. The approach proposed by Wang et al. [118] relies on a reference process model to repair a log whose events are affected by labels that do not match the expected behaviour of the reference model. The approach proposed by Conforti et al. [40] removes events that cannot be reproduced by an automaton constructed using frequent process behaviour recorded in the log. Fani Sani et al. [58] propose an approach that uses conditional probabilities between sequences of activities to remove events that are unlikely to occur in a given sequence. Finally, in [61] Fani Sani et al. propose to repair fragments of traces containing infrequent behaviour by means of replacing these fragments with more dominantly observed behaviour.

The problem of detecting spurious events from event streams of business processes shares similarities with the problem of outlier detection in temporal data, e.g. reading sensor data. In this context, we observe three types of techniques:

1. Techniques to detect if *entire sequences of events are anomalous*.
2. Techniques to detect if a *single data point within a sequence is an outlier*.
3. Techniques to detect *anomalous patterns within a sequence*.

For a detailed discussion regarding techniques for outlier detection in temporal data, we refer to the works by Gupta et al. [67] for events with continuous values, and by Chandola et al. [38] for events with discrete values.

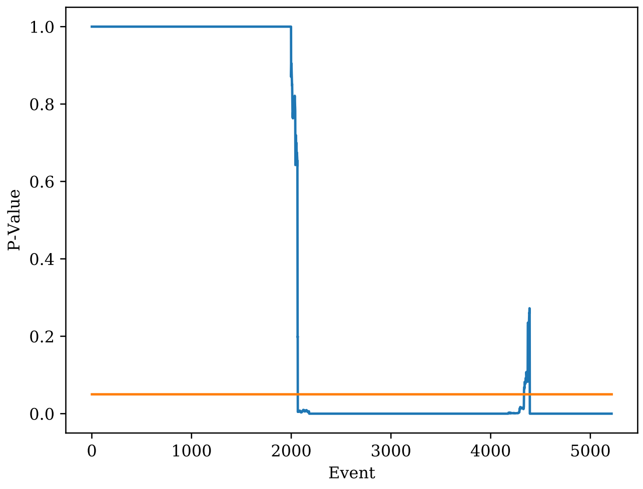
4.7 Conclusion

The existence of noise in event data typically causes the results of process mining algorithms to be inaccurate. Just the sheer existence of a fraction of noisy behaviour in a single trace, potentially significantly reduces the accuracy of process mining artefacts such as discovered process models. As such, the reliability of process mining results, based on event streams containing noise, is affected considerably.

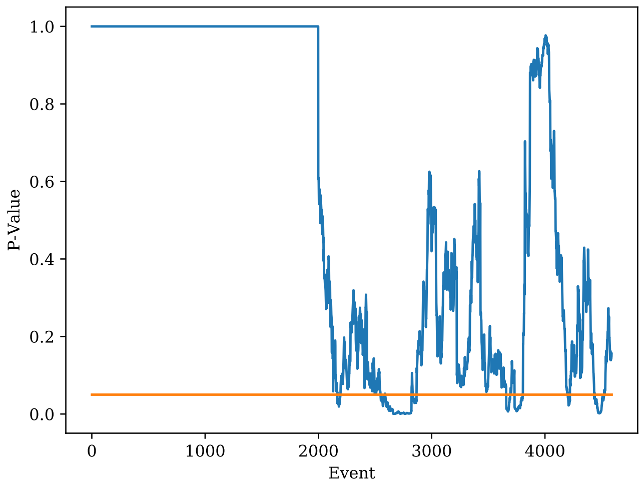
4.7.1 Contributions

We proposed an event stream based filter for online process mining, based on probabilistic non-deterministic automata which are updated dynamically as the event stream evolves. A state in one of these automata represents a potentially abstract view on the recent history of process instances observed on the stream. The empirical probability distribution defined by the outgoing arcs of a state is used to classify new behaviour as being spurious or not.

The time measurements of the corresponding implementation indicate that our filter is suitable to work in real-time settings. Moreover, our experiments on accuracy show that, on a set of stable event streams, we achieve high filtering accuracy for different instantiations of



(a) P-values obtained without applying filtering.



(b) P-values obtained with applying filtering.

Figure 4.14: P-values without filtering and with the proposed filter, for the Sepsis event log [85].

the filter. Finally, we show that our filter significantly increases the accuracy of state-of-the-art online drift detection techniques.

4.7.2 Limitations

In this section, we discuss the filtering technique presented in this chapter. In particular, we highlight the limitations and boundaries of the applicability of the approach and provide insights in potential solutions to overcome these limitations. Furthermore, we discuss the threats to validity of the experiments conducted in the context of this chapter.

Approach

The technique presented in this chapter is particularly designed to detect *spurious events*, i.e. events that are observed, yet their occurrence seems unlikely. As-is, the technique does not incorporate domain-specific knowledge, i.e. it solely uses data abstractions combined with occurrence frequencies to determine whether events are spurious or not. As such, rare events, i.e. related to infrequent process executions and/or ad-hoc solutions to eminent problems are likely to be filtered out of the resulting event stream. In some cases this is not a problem, i.e. in case we aim to obtain an overall view on the process containing only mainstream behaviour. However, note, that in some cases, e.g. in process monitoring, such deviant cases are of particular interest. Hence, we always need to carefully assess the exact intent of the process mining analysis, prior to adopting the paper as presented here as a stream processor.

The filter incorporates all behaviour observed within the collection of maintained automata. Again, this helps to accommodate for concept drift, i.e. after observing changed behaviour for a while, the behaviour becomes mainstream and proper events are no longer falsely filtered. However, this also poses challenges in filtering. It is likely that a spurious event either generates a new state in one of the automata, or, the next event after the spurious event is deemed to be noisy as we did not often observe such event following the spurious event. In the case we generate a new state in an automaton, any subsequent event is trivially frequent and is not filtered. Thus, this potentially causes the filter to no longer identify spurious events. We are able to accommodate for this problem, for example by tagging the case identifier related to the spurious event to be spurious. If we subsequently block all events related to the case identifier, we overcome the aforementioned problem, yet we are likely to generate incomplete traces of behaviour in the output stream. In case a spurious event does map into an existing state in the automaton, it is likely that any following behaviour is infrequent. Thus, in such case, we falsely label proper events as being spurious.

Finally, note that spurious events only cover a relatively small part of the full spectrum of noise within (streaming) process mining. Hence, in case an event was, for some unknown reason, not observed for a particular process instance, it is likely that any subsequent proper event is labelled spurious. This is due to the fact that we miss information for a specific process instance, and thus are effectively in an incorrect state for that given process instance within the automata. Such a problem is potentially solved, by trying to observe whether a spurious event is easily explainable in terms of relatively close neighbour states within the automata. In such case, generating an artificial event, prior to the current event even restores the behaviour as described by the stream.

Experiments

Here, we primarily focus on potential threats to validity of the experiments performed in the context of this thesis chapter.

The collection of models used for the synthetic experiments related to filtering accuracy (see subsection 4.5.1) represents a set of closely related process models. As such these results are only representative for data that originates from processes that exhibit similar types and relative amounts of control-flow constructs compared to the process models used.

Similarly, within these experiments, the events are streamed trace by trace, rather than using event-level time stamps. Note that, since the process is stable we expect the automata to be based on a sufficient amount of behaviour, similar to streaming parallel cases.

Finally note that, we do observe that our filter can be applied on real-life data, i.e. in order to enhance concept drift detection accuracy. However, due to the absence of a ground-truth, it is hard to determine whether the results obtained are valid and/or improve with respect to the unfiltered case.

4.7.3 Open Challenges & Future Work

In the approach presented, filtering is immediately applied when an event arrives, taking into account only the recent history for that event. As shown in our experiments, the filter already enhances concept drift detection. However, it is very likely that events originating from a newly occurred drift are labelled as noise. A potential solution to this problem, is to apply a (dynamic) filtering delay. Using such a delay, an event is immediately processed within the maintained collection of automata. However, the actual filtering of such event is delayed. Note that such delay, only partially solves the problem, i.e. if the delay is chosen wrongly, events related to the last execution of the "old" version of the process are likely to be filtered out.

Currently, we use one specific abstraction for each maintained automaton. Moreover, whenever we observe a spurious event in either one of these automata, we signal the event to be spurious. It is interesting to assess whether it is possible to further improve filtering accuracy by using an ensemble of automata where different abstractions are used for each maximal abstraction window size. Moreover, it is interesting to assess whether certain dynamic voting schemes are applicable in such settings, i.e. to consider the event to be spurious if the majority of automata agrees on this.

Another interesting direction for future work is towards large-scale experiments, using a larger collection of models and associated event streams. In such a way, we are able to quantify to what degree the filter is applicable if the underlying process depicts certain levels of control-flow behaviour. In particular, as automata are in principle not designed to model parallelism in a compact manner, it is expected that the filtering accuracy decreases upon processes exhibiting more parallel behaviour.

Finally, in line with the limitations, it is interesting to study techniques that allow us to specify, in some way, a degree of belief with respect to an event being spurious. A potential solution could be the following approach. When we observe that an event is spurious, we create two pointers from a case identifier to the automata. One represents the state prior to the observed spurious event and one corresponds to the state after receiving the spurious event. If the new event, according to one of the two states is non-spurious, we proceed from that specific state and forward the new event to the output stream.

Chapter 5

Avoiding Data Redundancy by Learning Intermediate Representations

Event stores, as defined in chapter 3, allow us to apply any type of process mining algorithm on the basis of an event stream. However, to effectively apply these algorithms, we need to iteratively forward the event store to the process mining algorithm of choice. Even if we aim to apply the algorithm in a batch fashion, i.e. we discover a model after receiving a batch of k events, it is likely that we reuse events that were already analysed in the previous execution of the algorithm of choice. Therefore, in this chapter, we define conventional process discovery as a two-step approach consisting of a translation of the input event data into an intermediate data structure, i.e. *intermediate representations*, and a subsequent translation into a process model. Such a characterization thus transposes the challenge of event stream-based process discovery into learning intermediate representations in an online fashion. We present a generic architecture that allows us to adopt several classes of existing process discovery techniques in the context of event streams, on the basis of learning intermediate representations. We furthermore show that, indeed, the architecture covers a wide class of process discovery algorithms.

The contents presented in this chapter are based on the following publication:

S.J. van Zelst, B.F. van Dongen, W.M.P. van der Aalst. Event Stream-Based Process Discovery using Abstract Representations. *Knowl. Inf. Syst.*, 54(2):407-435, 2018

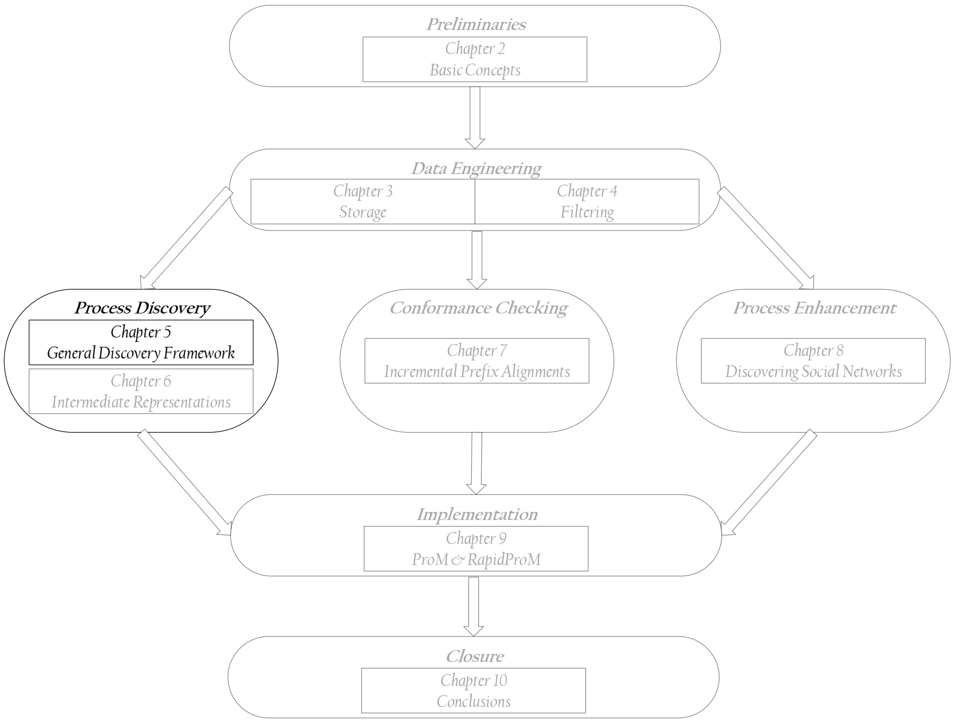


Figure 5.1: The contents of this chapter, i.e. event stream-based process discovery on the basis of intermediate representations, highlighted in the context of the general structure of this thesis.

5.1 Introduction

A multitude of process discovery algorithms have been developed in the context of conventional, offline process mining [10, 11, 78, 122, 123]. These algorithms all use an event log as an input, i.e. a static collection of events with an associated strict partial order (cf. subsection 2.3.1).[1] The techniques presented in chapter 3 allow us to (temporarily) store events emitted onto the stream. We are hence able to apply the aforementioned existing process discovery algorithms directly on top of such a subsequence of the event stream. However, whilst doing so, it is likely that we introduce unnecessary rework. Consider the case in which we maintain a sliding window based event store of size N, and are interested in discovering a process model after receiving each batch of $\frac{N}{4}$ events. This implies that each event within the stream is handled exactly 4 times by the process discovery algorithm of choice.

In this chapter, we address the aforementioned problem by defining process discovery as a two-step approach, i.e. translation of the event data into the algorithm-specific *intermediary representation*, cf. subsection 2.4.1, which is subsequently translated into a process model. As a consequence, we shift the focus of online process discovery towards designing data structures and associated update mechanisms that solely store the bare minimum amount of data, i.e. the intermediary representation itself, to be able to perform the discovery task. We show that several classes of process discovery algorithms apply the aforementioned two-step computational scheme. For example, both the Alpha algorithm [11] and the Inductive Miner [78] discover Petri nets by means of analysing (direct) precedence relations of the activities captured within the event log. Other approaches, like the Heuristic Miner [121, 122], the Fuzzy Miner [66], and most of the commercial process mining tools use (amongst others) the same precedence relations as an intermediate structure. Based on this generic two-phase computational model, we focus on efficient storage of the intermediate data structures used by these algorithms, rather than the event data as a whole.

To adopt algorithms that employ such computational scheme in a streaming context, it suffices to derive and/or approximate the intermediate representation based on the event stream. Apart from lowering the overall memory footprint, using intermediate representations as a basis for process discovery has several advantages:

1. *Reusability*;
   We *reuse* existing process discovery techniques as much as possible by predominantly focusing on learning the intermediate representations from event streams.
2. *Extensibility*;
   Once we design and implement a method for approximating a certain intermediate representation, any (future) algorithm using the same intermediate representation is automatically ported to event streams.
3. *Anonymity*;
   In some cases, laws and regulations dictate that we are not allowed to store all event data. Some intermediate representations ignore large parts of the data, effectively storing a summary of the actual event data, and therefore comply with *anonymity* regulations.

The remainder of this chapter is organized as follows. In section 5.2, we present the general architecture of intermediate representation-based process discovery. In section 5.3, we provide several instantiations of the architecture. In section 5.4, we present an empirical evaluation of several instantiations of the architecture. In section 5.5, we present related work. Finally, section 5.6 concludes this chapter.

[1]Note that most algorithms use simple traces, i.e. sequences of executed activities, for discovery.

5.2 Architecture

In this section, we present the general architecture of online process discovery on the basis of intermediate representations. The proposed architecture captures the use of intermediate representations in the context of event streams in a generic manner. In the remainder, given an arbitrary *data structure type* $\mathbf{T_D}$, we let $\mathscr{U}_{\mathbf{T_D}}$ denote the universe of data structures of type $\mathbf{T_D}$. A data type $\mathbf{T_D}$ might refer to an array or a (collection of) hash table(s), yet it might also refer to some implementation of a stream-based frequent-item approximation algorithm such as Lossy Counting [84], i.e. as presented in subsection 3.3.3. We require any instance $\mathtt{i}_{\mathbf{T_D}} \in \mathscr{U}_{\mathbf{T_D}}$ of such typed data structure to use *finite memory*.

Definition 5.1 (Abstraction Function; Event Stream). *Let $\mathscr{E}$ denote the universe of events, let $\mathbf{T_D}$ denote an arbitrary data structure type, let $\mathbf{T_I}$ denote an intermediate representation type, let $i \in \mathbb{N}_0$ and let $S \in \mathscr{E}^*$ be an event stream. An event stream-based abstraction function ψ^i is a function mapping the first i observed events on an event stream onto an instance of $\mathbf{T_D}$ and an intermediate representation of type $\mathbf{T_I}$, i.e.*

$$\psi^i : \mathscr{E}^* \rightarrow \mathscr{U}_{\mathbf{T_D}} \times \mathscr{U}_{\mathbf{T_I}} \tag{5.1}$$

Given the abstraction function at time i, i.e. ψ^i as described in 5.1, we are able to discover a corresponding process model. As such, we quantify α^i as:

$$\alpha^i = \alpha_{\mathbf{T_I}}(\pi_2(\psi^i(S))) \tag{5.2}$$

Observe that, we are able to apply the aforementioned discovery function on *any instantiation of* ψ^i, given that the discovery algorithm and the abstraction function use the same intermediate representation. Furthermore, note that 5.1 resembles the definition of an event store (as presented in 3.1). However, in the case of an event stream-based abstraction function, we have more freedom, i.e. we are required to map the first i events onto some data structure and corresponding intermediate representation. Such a data structure is potentially a subsequence of the input stream, as described by event stores, yet this is not necessarily the case. On the contrary, we aim at designing these data structures in such a way that we store only the bare minimum information to be able to construct the corresponding intermediate representation.

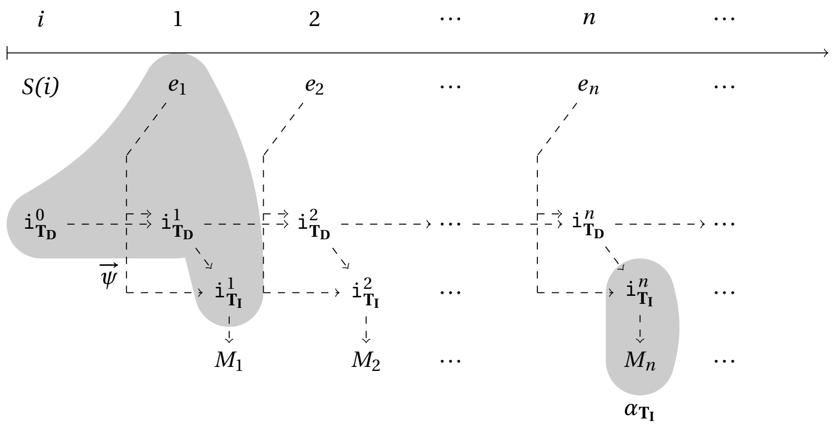
In line with event stores, we define an additional update function which allows us to update a given data structure and corresponding intermediate abstraction on the basis of an individual event.

Definition 5.2 (Abstraction Update Function; Event Stream). *Let $\mathscr{E}$ denote the universe of events, let $\mathbf{T_D}$ denote an arbitrary data structure type and let $\mathbf{T_I}$ denote an intermediate representation type. An event stream-based abstraction update function $\overrightarrow{\psi}$ is a function that updates a given data structure and intermediate representation on the basis of an individual event, i.e.*

$$\overrightarrow{\psi} : \mathscr{U}_{\mathbf{T_D}} \times \mathscr{U}_{\mathbf{T_I}} \times \mathscr{E} \rightarrow \mathscr{U}_{\mathbf{T_D}} \times \mathscr{U}_{\mathbf{T_I}} \tag{5.3}$$

Given the notion of an event stream-based abstraction function and an event stream-based abstraction update function, we incrementally characterize the abstraction function for element i in terms of an update of the underlying data structure and intermediate representation derived at element $i-1$, i.e.

$$\psi^i(S) = \overrightarrow{\psi}(\psi^{i-1}(S), S(i)) \tag{5.4}$$



i
1
2
$\cdots$
n
$\cdots$
$S(i)$
e_1
e_2
e_n
$\mathtt{i}^0_{\mathbf{T_D}}$
$\mathtt{i}^1_{\mathbf{T_D}}$
$\mathtt{i}^2_{\mathbf{T_D}}$
$\mathtt{i}^n_{\mathbf{T_D}}$
$\vec{\psi}$
$\mathtt{i}^1_{\mathbf{T_I}}$
$\mathtt{i}^2_{\mathbf{T_I}}$
$\mathtt{i}^n_{\mathbf{T_I}}$
M_1
M_2
M_n
$\alpha_{\mathbf{T_I}}$

Figure 5.2: Detailed overview of the proposed architecture.

As indicated, we are able to quantify online process discovery on the basis of event stream-based abstraction functions, using both event stores and conventional process discovery algorithms as a basis. In such case, the data structure type is simply a sequence of events, i.e. represented by $\mathscr{E}^*$. As such, the event stream-based abstraction function for element i, is refined to $\psi^i : \mathscr{E}^* \to \mathscr{E}^* \times \mathscr{U}_{\mathbf{T_I}}$, with corresponding characterization $\psi^i(S) = (\Phi^i(S), \lambda_{\mathbf{T_I}}(\Phi^i(S)))$.

Consider Figure 5.2, in which we present a schematic overview of the proposed architecture. The first incremental update component $\vec{\psi}$, highlighted in grey on the left-hand side of Figure 5.2, incrementally updates a data structure (of type $\mathbf{T_D}$) and the corresponding intermediate representation (of type $\mathbf{T_I}$) when new events arrive on the event stream. Using conventional process discovery on the basis of intermediate representations, i.e. $\alpha_{\mathbf{T_I}}$, we translate the maintained intermediate representation into a process model.

5.3 Instantiating Intermediate Representation Based Discovery

When using an event store based instantiation of the architecture, as briefly described in section 5.2, the corresponding intermediate representation is always in direct correspondence with the events present within the event store. This is the case, because we (re)compute the abstraction on the basis of the contents of the event store, i.e. $\psi^i(S) = (\Phi^i(S), \lambda_{\mathbf{T_I}}(\Phi^i(S)))$. Alternatively, we are able to incrementally design the corresponding update function in such way, that it explicitly utilizes the previous intermediate representation, together with the newly received event. Nonetheless, we still store all events as emitted to the event stream.

It is however often the case that, when we aim at incrementally maintaining intermediate representations on the basis of event streams, we need a significantly smaller representation of the event stream to actually do so. For example, to update an existing directly follows relation for some new event $e = (c, a, ...)$, it suffices to check whether there exists some event of the form $e' = (c, a', ...)$, that was the most recent event received for case identifier c. If so, we deduce that (a', a) needs to be part of the directly follows abstraction. As a side-effect of such design however, we need to employ a separate ageing mechanism on the intermediate representation that we maintain, i.e. we no longer keep track of the full history related to case identifier c. As such, the exact relation of the instantiation of $\mathscr{U}_{\mathbf{T_I}}$ and the intermediate representation becomes less clear.

In the remainder of this section, we show the applicability of the proposed architecture by presenting several instantiations for different existing process discovery algorithms. Since a large class of conventional process discovery algorithms is based on the directly follows abstraction, or some derivative thereof, we first present how to compute it. Subsequently we highlight, for each algorithm using the directly follows abstraction as a basis, the main changes and/or extensions that need to be applied with respect to the basic scheme. To illustrate the generality of the architecture, we also show a completely different class of discovery approaches, i.e. region-based techniques [10, 123]. These techniques work fundamentally different compared to the aforementioned class of algorithms and use different intermediate representations.

5.3.1 The Directly Follows Abstraction

Recall that the *directly follows relation* describes pairs of activities (a, a'), with $a, a' \in \mathscr{A}$, written as $a > a'$, if there exists some trace σ in an event log of the form $\sigma = \sigma' \cdot \langle e, e' \rangle \cdot \sigma''$ where $\pi_{\mathsf{a}}(e) = a$ and $\pi_{\mathsf{a}}(e') = a'$.

In an event stream-based setting, assume that we maintain an event store Φ as presented in chapter 3 as an instantiation for $\mathscr{U}_{\mathbf{T_D}}$. As an instantiation of the directly follows relation, i.e. $\mathscr{U}_{\mathbf{T_I}}$, we maintain a multiset of activity-pairs, i.e. of the form $\mathscr{B}(\mathscr{A} \times \mathscr{A})$. Given some multiset $B \in \mathscr{B}(\mathscr{A} \times \mathscr{A})$, $B(a, a')$ indicates how often the $a > a'$ occurs within the behaviour as captured by the underlying event store Φ. When a new event $e \in \mathscr{E}$ with $\pi_{\mathsf{c}}(e) = c$ arrives on the stream at index i, we are able to derive a possible new activity pair by inspecting event store $\Phi^i(c)$. We simply deduce such pair by inspecting the last two elements of $\Phi^i(c)$ (under the assumption that e is added to the event store). Similarly, the elements of Φ^{i-} allow us to keep the directly follows relation up to date with the contents of the event store.

As indicated, the main problem with using an event store as an instantiation for $\mathscr{U}_{\mathbf{T_D}}$ is memory redundancy. For each case identifier $c \in \mathscr{C}$, we store multiple events whereas we effectively only need the last event related to each case identifier in order to derive a new directly follows relation. We therefore alternatively instantiate ψ^i for the directly follows relation as follows:

$$\psi^i_{dfr} : \mathscr{E}^* \to \mathscr{P}(\mathscr{C} \times \mathscr{A}) \times \mathscr{B}(\mathscr{A} \times \mathscr{A}) \tag{5.5}$$

Here, the first argument of the range of ψ^i_{dfr}, i.e. $\mathscr{P}(\mathscr{C} \times \mathscr{A})$, represents a set of pairs of the form $(c, a) \in \mathscr{C} \times \mathscr{A}$, that represent the last activity a seen for the process instance identified by case identifier c. As such, we have at most one entry for each case identifier within the set. We are able to use several storage techniques as presented in chapter 3 to effectively implement $\mathscr{P}(\mathscr{C} \times \mathscr{A})$. For example, consider algorithm 5.1, in which we construct a sliding window which consists of elements that are pairs of the form $(c, a) \in \mathscr{C} \times \mathscr{A}$. When we receive a new event, related to case identifier c, we check whether a tuple of the form (c, a) is present in the window. If this is the case, we are able to deduce a new directly follows relation of the form $(a, \pi_{\mathsf{a}}(e))$. The newly deduced directly follows relation needs to be subsequently forwarded to the component that implements the second component of ψ^i_{dfr}, i.e. $\mathscr{B}(\mathscr{A} \times \mathscr{A})$. We subsequently remove (c, a) from the sliding window. We always add the new event, i.e. $(\pi_{\mathsf{c}}(e), \pi_{\mathsf{a}}(e))$ at the end of the sliding window, and we remove the first element of the sliding window if the size of the window exceeds N.

Algorithm 5.1: `Sliding Window Based Instantiation of` $\mathscr{P}(\mathscr{C} \times \mathscr{A})$

```
input: $S \subseteq \mathscr{E}^*$, $N \in \mathbb{N}_0$
begin
    $i \leftarrow 1$;
    $w \leftarrow \epsilon$ ;                          // empty list of tuples in $\mathscr{C} \times \mathscr{A}$
    while $true$ do
        $e \leftarrow S(i)$;
        if $|w| \geq N$ then
            remove w(1);
        if $\exists (c,a) \in_* w\,(c = \pi_c(e))$ then
            deduce new relation $(a, \pi_a(e))$;
            remove corresponding tuple $(c,a)$ from w;
        append $(\pi_c(e), \pi_a(e))$ to w;
        $i \leftarrow i+1$;
```

Observe that, algorithm 5.1 is inspired by the notion of a sliding window, yet it does not implement a sliding window directly, i.e. some elements are removed earlier than in a conventional setting. Additionally, we have $O(N)$ time complexity (where N represents the maximum sliding window size) to actually find a pair (c,a) if a new event comes in. In fact, observe that the mechanism in algorithm 5.1 effectively removes the entry related to the oldest/recently most inactive process instance.

We are also able to instantiate the set of (case identifier, activity)-pairs by means of other storage-oriented streaming algorithms such as reservoir sampling, decay based techniques and frequency approximation algorithms. For example, consider algorithm 5.2, in which we show a corresponding implementation on the basis of the *Space Saving* algorithm. Recall that the *Space Saving* algorithm counts the frequency of case identifiers by means of the v_c-counters, cf. subsection 3.3.3, which are used as a criterion for insertion/deletion in the underlying storage component. In this case, we store the elements of $\mathscr{C} \times \mathscr{A}$ in the internal set X, yet for each case identifier $c \in \mathscr{C}$ we maintain an associated counter v_c. Upon receiving a new event, we check whether there already exists a (case identifier, activity)-pair that relates to the case identifier of the new event. If this is the case, we update the corresponding counter, deduce a new directly follows relation and replace the existing (case identifier, activity)-pair on the basis of the new event, cf. lines 8-10. In any other case, we are not able to deduce any new directly follows relation, yet we update set X, depending on its size. Note that when set X reaches its maximum allowed size, the pair (c,a) for which the corresponding $v(c)$ is minimal amongst all case identifiers present in X is removed. Furthermore, the counter of the newly arrived case identifier is equal to the counter of the removed case identifier, increased by one.

As both algorithm 5.1 and algorithm 5.2 signify, we are able to maintain a collection of (case identifier, activity)-pairs, using a multitude of existing data storage algorithms. From time to time, within these algorithms, we are able to deduce a new directly follows relation. As such, the two algorithms, in essence, generate a *stream of directly follows relations*. As a consequence, to maintain the actual multiset of pairs $(a,a') \in \mathscr{A} \times \mathscr{A}$, we are again able to utilize any existing stream-based storage algorithm. A concrete implementation of the update function $\overrightarrow{\psi}$, as defined in Equation 5.3, is therefore achieved by appending the deduced directly follows

Algorithm 5.2: `Space Saving Based Instantiation of` $\mathscr{P}(\mathscr{C} \times \mathscr{A})$

```
input  : $S \in \mathscr{E}^*$, $N \in \mathbb{N}_0$
begin
1   $X \leftarrow \emptyset$;                // empty set of tuples, i.e. $X \in \mathscr{P}(\mathscr{C} \times \mathscr{A})$
2   $i \leftarrow 0$;
3   $v_c \leftarrow 0, \forall c \in \mathscr{C}$;
4   while $true$ do
5       $i \leftarrow i + 1$;
6       $e \leftarrow S(i)$;
7       if $\exists_{(c,a) \in X}(c = \pi_{\mathtt{c}}(e))$ then
8           $v_c \leftarrow v_c + 1$;
9           deduce new relation $(a, \pi_{\mathtt{a}}(e))$;
10          $X \leftarrow (X \setminus \{(c,a)\}) \cup \{(c, \pi_{\mathtt{a}}(e))\}$;
11      else if $|X| < k$ then
12          $X \leftarrow X \cup \{(\pi_{\mathtt{c}}(e), \pi_{\mathtt{a}}(e))\}$;
13          $v_{\pi_{\mathtt{c}}}(e) \leftarrow 1$;
14      else
15          $(c,a) \leftarrow \underset{(c,a) \in X}{\arg\min}(v_c)$;
16          $v_{\pi_{\mathtt{c}}(e)} \leftarrow v_c + 1$;
17          $X \leftarrow (X \setminus \{(c,a)\}) \cup \{(\pi_{\mathtt{c}}(e), \pi_{\mathtt{a}}(e))\}$;
```

relations to the corresponding output stream. We are thus able to instantiate the update function using an arbitrary combination of stream-based storage methods. For example, we are able to store pairs of the form $(c,a) \in \mathscr{C} \times \mathscr{A}$ using a sliding window, and the actual directly follows relation by means of a reservoir sample. The exact choice of such combination of algorithms mainly depends on the aim of the process discovery, i.e. discovery of recent behaviour versus discovery of predominant behaviour.

In the remainder of this section, we describe existing process discovery algorithms that function on the basis of the directly follows relation. These algorithms differ in the way they use/interpret the relation, and in some cases need auxiliary input to be able to discover a process model. We, therefore, highlight, for each of these algorithms, the main changes and/or extensions that need to be applied with respect to the basic scheme presented here.

The Alpha algorithm

The Alpha algorithm [11] transforms the directly follows abstraction into a Petri net. When adopting the Alpha algorithm to an event stream context, we directly adopt the scheme described in the previous section. However, the algorithm explicitly needs a set of *start-* and *end activities*.

Approximating the start activities seems rather simple, i.e. whenever we receive an event related to a new case identifier, the corresponding activity represents a start activity. However, given that we at some point remove (case, activity)-pairs from the underlying data structure, we might designate some activities falsely as start activities, i.e. a new case identifier may in fact refer to a previously removed case identifier. Similarly, approximating the end activities is

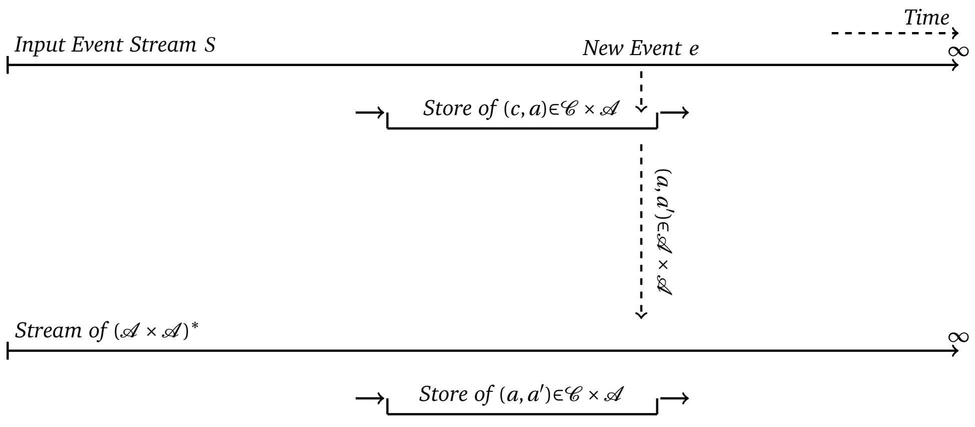

Time
Input Event Stream S
New Event e
∞
Store of $(c,a)\in\mathscr{C}\times\mathscr{A}$
$(a,a')\in\mathscr{A}\times\mathscr{A}$
Stream of $(\mathscr{A}\times\mathscr{A})^*$
∞
Store of $(a,a')\in\mathscr{C}\times\mathscr{A}$

Figure 5.3: Instantiation of online intermediate representation based discovery for the directly follows relation. We store pairs of the form $(c,a)\in\mathscr{C}\times\mathscr{A}$ using existing data stream storage techniques. Using such store, we deduce new relations of the form $(a,a')\in\mathscr{A}\times\mathscr{A}$, which we forward to a stream of the form $(\mathscr{A},\mathscr{A})^*$.

rather complex, as we are often not aware when a case terminates. A potential solution is to apply a *warm-up* period in which we try to observe cases that seem to be terminated, e.g. by identifying cases that have long periods of inactivity or by assuming that (case, activity)-pairs that are dropped out of the underlying data structure are terminated. However, this also largely depends on the actual implementation that we use for this. In the general sense, since we approximate case termination, this approach potentially leads to falsely select certain activities as end activities.

We can also deduce start- and end activities from the directly follows abstraction. A start activity is an activity $a\in\mathscr{A}$ with $\nexists a'\in\mathscr{A}(a'\neq a\wedge a'>a)$ and an end activity is an activity $a\in\mathscr{A}$ with $\nexists a'\in\mathscr{A}(a'\neq a\wedge a>a')$. This works if these activities are only executed once at the beginning, respectively the end, of the process. In case of loops or multiple executions of start/end activities within the process, we potentially falsely neglect certain activities as being either start and/or end activities. In section 5.6.2, we discuss this problem in more depth.

The Heuristics Miner

The Heuristics Miner [121, 122] is designed to cope with noise in event logs. To do this, it effectively counts the number of occurrences of activities, as well as the >-relation. Based on the directly follows abstraction it computes a derived metric $a\Rightarrow b=\frac{|a>b|-|b>a|}{|a>b|+|b>a|+1}$ that describes the relative causality between two activities a and b ($|a>b|$ denotes the number of occurrences of $a>b$). The basic scheme presented in subsection 5.3.1 suffices for computing $a\Rightarrow b$, as long as we explicitly track, or, approximate, the frequencies of (activity, activity)-pairs.

The Inductive Miner

The Inductive Miner [78], like the Alpha algorithm, uses the directly follows abstraction together with start- and end activities. It tries to find patterns within the directly follows abstraction that indicate certain behaviour, e.g. parallelism. Using these patterns, the algorithm splits an event log into several smaller event logs and repeats the procedure. Due to its iterative nature, the Inductive Miner guarantees to find *sound workflow nets*, cf. 2.7. The Inductive Miner has also been extended to handle noise and/or infrequent behaviour [79]. This requires, like the Heuristics Miner, to explicitly count the >-relation.

Observe that, the standard Inductive Miner algorithm, only works when storing event stores as defined in chapter 3, i.e. due to the recursive use of the event data. However, in [77], a version of the Inductive Miner is presented in which the inductive steps are directly performed on the directly follows abstraction. In the context of event streams this is the most adequate version to use as we only need to maintain a frequency-aware directly follows abstraction.

5.3.2 Region Theory

Several process discovery algorithms [10, 22, 37, 123, 137] are based on *region theory* which is a solution method for the *Petri net synthesis problem* [20]. In Petri net synthesis, given a behavioural description of a system, i.e. either a transition system or a language, one aims to synthesize a Petri net that exactly describes the same behaviour as described by the input behavioural description. Hence, classical region theory techniques ensure strict formal properties with respect to the resulting process models. Process discovery algorithms based on region theory aim to relax these properties, in order to improve the resulting models from a process mining perspective, e.g. avoiding overfitting of the model with respect to the input data. We identify two different region theory approaches, i.e. *language-based* and *state-based* region theory, which use different forms of intermediate representations.

Language-Based Approaches

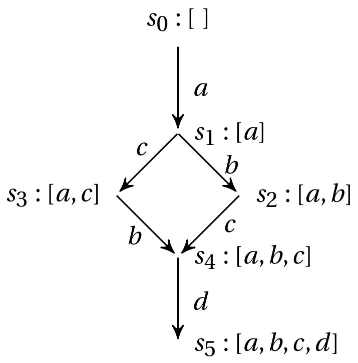
Algorithms based on *language-based* region theory [22, 123, 137] rely on a control-flow based *prefix-closure*, cf. Equation 2.2, of the input event log, i.e. the set of all prefixes of all traces. Clearly, we are able to (incrementally) construct such set of prefixes by simply using an event store as a backing storage component. However, in the case of ILP-Based process discovery [123, 137], which we discuss in greater detail in chapter 6, such a collection of prefixes is further abstracted. In particular, given a sequence of activities $\sigma \in \mathscr{A}^*$, the ILP-based process discovery algorithm abstracts it into a *constraint*, which we are able to represent by a pair $(\vec{\sigma}_{1 \dots |\sigma|-1}, \sigma(|\sigma|))$, i.e. the Parikh representation of the *prefix of σ*, combined with the last activity in the sequence σ. As such, instantiating the architecture for this algorithm is relatively similar to the directly follows relation instantiation, i.e.

$$\psi^i_{ilp} \colon \mathscr{E}^* \to \mathscr{P}(\mathscr{C} \times \mathscr{B}(\mathscr{A}) \times \mathscr{A}) \times \mathscr{B}(\mathscr{B}(\mathscr{A}) \times \mathscr{A}) \tag{5.6}$$

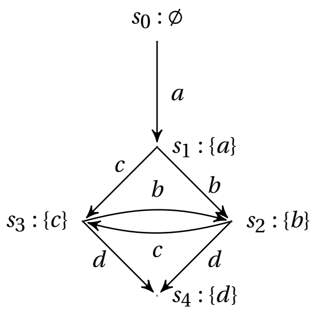
Observe that, we store per case identifier the *latest* constraint, i.e. as represented by $\mathscr{P}(\mathscr{C} \times \mathscr{B}(\mathscr{A}) \times \mathscr{A})$. The second component we store is the actual (frequency aware) intermediate abstraction of the algorithm, i.e. the constraints of the form $(\vec{\sigma}_{1 \dots |\sigma|-1}, \sigma(|\sigma|))$, represented by $\mathscr{B}(\mathscr{B}(\mathscr{A}) \times \mathscr{A})$. Note that we are again able to use any existing stream-based storage technique to instantiate ψ^i_{ilp} and the corresponding update function $\overrightarrow{\psi}$.

State-Based Approaches

Within process discovery based on state-based regions [10], a transition system is constructed based on *a specific view* of each trace present within the input data. Examples of a view are the



(a) Multiset abstraction (unbounded).



(b) Set abstraction (max. set size 1).

Figure 5.4: Example transition systems based on simple event store $\tilde{\Phi} = [\langle a, b, c, d\rangle, \langle a, c, b, d\rangle]$.

complete prefix of the trace, the multiset projection of the prefix of the trace, etc. The *future of a trace* can be used as well, i.e. given an event within a trace, the future of the event are all events happening after the event. However, future-based views are not applicable in an event stream setting, as the future behaviour of process instances is unknown.

As an example, based on a simple event store $\tilde{\Phi} = [\langle a, b, c, d\rangle, \langle a, c, b, d\rangle]$, consider Figure 5.4. In Figure 5.4a, states are represented by a multiset view of the prefixes of the traces, i.e. the state is determined by the multiset of activities seen before. Activities make up the transitions within the system, i.e. the first activity in both traces is a, thus the empty multiset is connected to multiset $[a]$ by means of a transition labelled a. In Figure 5.4a we do not limit the maximum size of the multisets. Figure 5.4b shows a set view of the traces with a maximum set size of 1. Again the empty set is connected with set $\{a\}$ by means of a transition labelled a. For trace $\langle a, b, c, d\rangle$ for example, the second activity is a b and thus state $\{a\}$ has an outgoing transition labelled b to state $\{b\}$. This is the case, i.e. a connection to state $\{b\}$ rather than $\{a, b\}$, due to the size restriction of size 1.[2]

In case we aim at instantiating the architecture for the purpose of state-based region theory, we store the most recent abstraction derived for a case identifier. As such, when a new event occurs for a given case identifier, we are able to derive a new abstraction. Again, we are able to generate a stream of derived artefacts, in this case being trace abstractions. Hence, as a second component, we employ some arbitrary data storage technique that keeps a collection of such abstractions in memory.

Note that, the set of abstractions itself does not allow us to derive the underlying transition system, i.e. once a new trace abstraction is created for a case identifier, the previous abstraction is removed. As such, we are not able to derive the underlying relations between these states. We, therefore, need to maintain the constructed transition system in memory. Whenever we derive a new abstraction on the basis of a new event, we know from which abstraction the new abstraction is derived. Moreover, the event describes an activity, hence the new event allows us to connect the old abstraction to the new abstraction by means of an arc labelled with the activity that is described by the new event. Ageing of the transition system is in line with the contents of the storage of the abstractions, i.e. whenever an abstraction gets removed,

[2]Observe the similarity of transition systems and the probabilistic automata used in chapter 4.

the corresponding state and all its associated arcs need to be removed from the transition system. Note that, within this instantiation, depending on the underlying storage technique of choice, the transition system potentially contains disconnected components, which likely leads to complex process models. Moreover, the exact semantics of such disconnected transition system is unclear.

5.4 Evaluation

In this section, we present an evaluation of several instantiations of the architecture. We also consider performance aspects of the implementation.

5.4.1 Structural Analysis

As a first visual experiment we investigate the steady-state behaviour of the *Inductive Miner* [78]. Within this experiment we use the Lossy Counting algorithm [84] as an implementation for both the collection of pairs of the form $(c, a) \in \mathscr{C} \times \mathscr{A}$ as well as for storage of the directly follows relation itself. To create an event stream, we created a timed Coloured Petri Net [72] in CPN-Tools [126] which simulates a synthetic process model describing a process related to a loan application process [51]. The event stream, and all other event streams used for experiments, are free of noise. The CPN model used is able to simulate multiple cases being executed simultaneously.

In Figure 5.5, we show the behaviour of the Inductive Miner over time based on a random simulation of the CPN model. We configured the algorithm with $|X|_{\mathscr{C} \times \mathscr{A}} \leq 75$ and $|X|_{\mathscr{A} \times \mathscr{A}} \leq 75$, i.e. we have at most 75 elements in the underlying sets used in the Lossy Counting algorithm. Initially (Model 1) the Inductive Miner only observes a few directly follows relations, all executed in sequence. After a while (Model 2) the Inductive Miner observes that there is a choice between *Prepare acceptance pack* and *Reject Application*. In Model 3, the first signs of parallel behaviour of activities *Appraise property*, *Check credit history* and *Assess loan risk* become apparent. However, an insufficient amount of behaviour is emitted onto the stream to effectively observe the parallel behaviour yet. In Model 4, we identify a large block of activities within a choice construct. Moreover, an invisible transition loops back into this block. The Inductive Miner tends to show this type of behaviour given an incomplete directly follows abstraction. Finally, after sufficient behaviour is emitted onto the stream, Model 5 shows a Petri net version that in fact describes the same behaviour as the process model generating the event stream.

Figure 5.5 shows that the Inductive Miner is able to find the original model based on the event stream. We now focus on comparing the Inductive Miner with other algorithms described in the paper. All discovery techniques discover a Petri net or some alternative process model that is *translatable to* a Petri net. The techniques, however, differ in terms of guarantees with respect to the resulting process model. The Inductive Miner guarantees that the resulting Petri nets are sound workflow nets, whereas the ILP Miner and the Transition System Miner do not necessarily yield sound process models. To perform a proper behavioural comparative analysis, the soundness property is often a prerequisite. Hence, we perform a structural analysis of all the algorithms by measuring structural properties of the resulting Petri nets.

Using the off-line variant of each algorithm we first compute a reference Petri net. We generated an event log which contains a *sufficient amount of behaviour* such that the discovered Petri nets describe all behaviour of the model used within the experiment reported on in Figure 5.5 Based on the reference Petri net we create a structure matrix in which each row/column corresponds to a transition in the reference model. If, in the Petri net, two labelled transitions

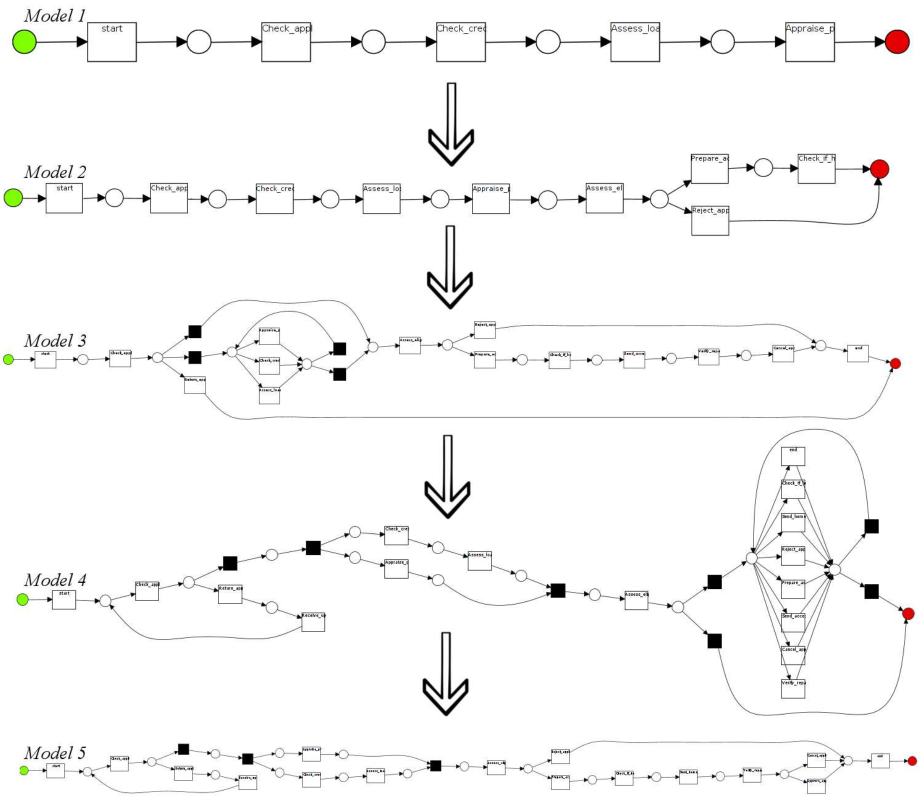


Figure 5.5: Visual results of applying the Inductive Miner on a stream.

are connected by means of a place, the corresponding cells in the matrix get value 1. For example, given the first Petri net of Figure 5.5, the transitions labelled *start* and *Check_application_completeness* (in the figure this is “Check_appl”) are connected by means of a place. Hence, the distance between the two labels is set to 1 in the corresponding matrix. If two transitions are not connected, the corresponding value is set to 0.

Using an event stream-based on the CPN-Model, after each newly received event, we use each algorithm to discover a Petri net. For each Petri net, we again construct the structure matrix. We apply the same procedure as applied on the reference model. However, if in a discovered Petri net a certain label is not present, we set all cells in the corresponding row/column to -1, e.g. in model 1 of Figure 5.5 there is no transition labelled *end*, thus the corresponding row and column consist of -1 values. Given a matrix M based on the streaming variant of an algorithm, we compute the distance to the reference matrix M_R as:

$$d_{M,M_R} = \sqrt{\sum_{i,j \in \{1,2,...,15\}} ((M(i,j) - M_R(i,j))^2}$$

For all algorithms, the internal data structures used were based on Lossy Counting, with size

100.

Since the Inductive Miner and the Alpha algorithm are completely based on the same abstraction, we expect them to behave similarly. Hence, we plot their corresponding results together in Figure 5.6a. Interestingly, the distance metric follows the same pattern for both algorithms. Initially, there is a steep decline in the distance metric after which it becomes zero. This means that the reference matrix equals the matrix based on the discovered Petri net. The distance shows some peaks in the area between 400 until 1000 received events. Analysing the resulting Petri nets at these points in time showed that some activities were not present in the resulting Petri nets at those points. The results for the Transition Systems Miner (TS), the ILP Miner and the Heuristics Miner are depicted in Figure 5.6b. We observe that the algorithms behave similarly to the Alpha- and Inductive Miner, which intuitively makes sense as the algorithms all have the same data structure capacity. However, the peeks in the distance metric occur at different locations. For the Heuristics Miner, this is explained by the fact that it takes frequency into account and thus uses the directly follows abstraction differently. The Transition System Miner and the ILP Miner use different intermediate representations, and have a different update mechanism than the directly follows abstraction, i.e. they always update their abstraction whereas the directly follows abstraction only updates if, for a given case, we already received a preceding activity.

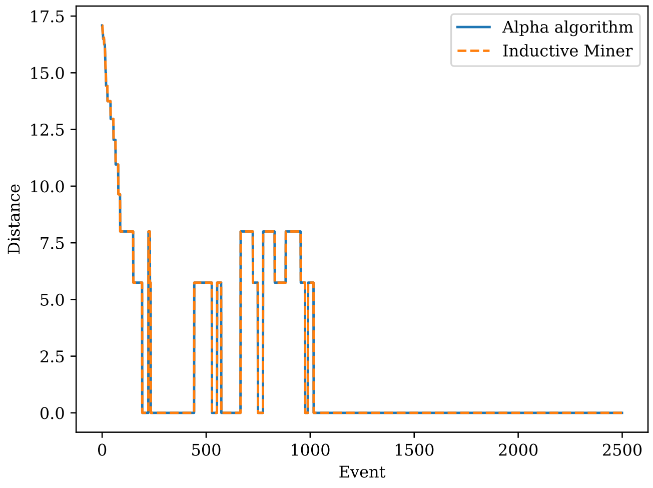
5.4.2 Behavioural Analysis

Although the previous experiments provide interesting insights with respect to the functioning of the algorithms in a streaming setting, they only consider structural model quality. A distance value of 0 in Figure 5.6 indicates that the resulting model is very similar to the reference model. It does not guarantee that the model is in fact equal, or, entails the same behaviour as the reference model. Hence, in this section, we focus on measuring quantifiable similarity in terms of *behaviour*. We use the Inductive Miner as it provides formal guarantees with respect to initialization and termination of the resulting process models. This, in particular, is a requirement to measure behavioural similarity in a reliable manner. We adapt the Inductive Miner to a streaming setting by instantiating the architecture as presented in this chapter, using the scheme described in subsection 5.3.1, combined with its algorithm-specific modifications. For finding start- and end activities we inspect the directly follows relation and select those activities that have no predecessor, or, successor, respectively. We again use Lossy Counting [84] to implement both underlying data structures, i.e. the elements $(c,a) \in \mathscr{C} \times \mathscr{A}$, and the directly follows relation itself, i.e. pairs $(a,a') \in \mathscr{A} \times \mathscr{A}$.

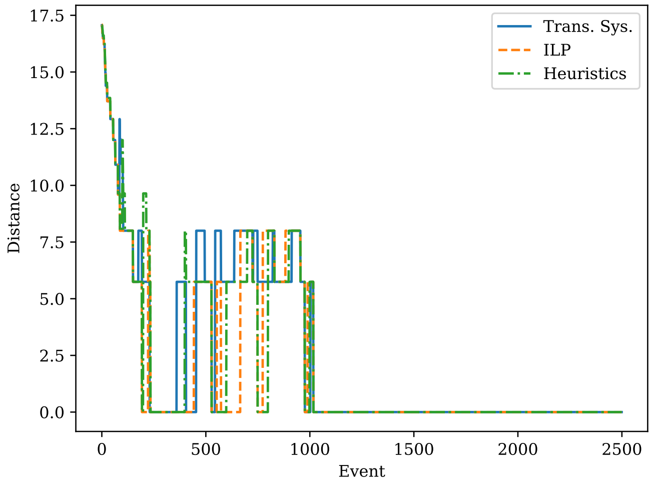
We assess under what conditions the Inductive Miner instantiation is able to discover a process model with the same behaviour as the model generating the stream. In the experiment, after each received event, we query the miner for its current result and compute replay-fitness and precision measures based on a complete corresponding event log describing the behaviour of the underlying process. Recall that replay-fitness, cf. subsection 1.1.4, quantifies the amount of behaviour present in an event log that is also described by the process model. A replay-fitness value of 1 indicates that all behaviour in the event log is described by the process model, a value of 0 indicates that none of the behaviour is described. Precision on the other hand refers to the amount of behaviour described by the process model that is also present in the event log. A precision value of 1 indicates that all behaviour described by the model is present in the event log. The lower the precision value, the more the model allows for additional behaviour.

In Figure 5.7 and Figure 5.8 (pages 134–135), the results are presented for varying capacity sizes of the underlying data structures.

For the smallest data structure sizes, i.e. Figure 5.7a, we identify that the replay-fitness

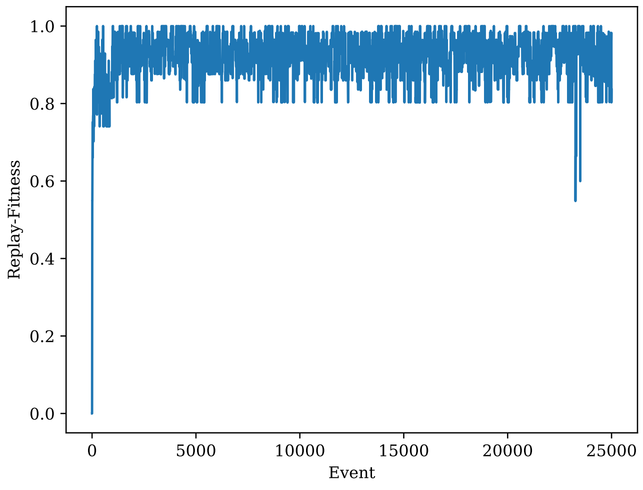


(a) Distances for the Alpha algorithm and the Inductive Miner.

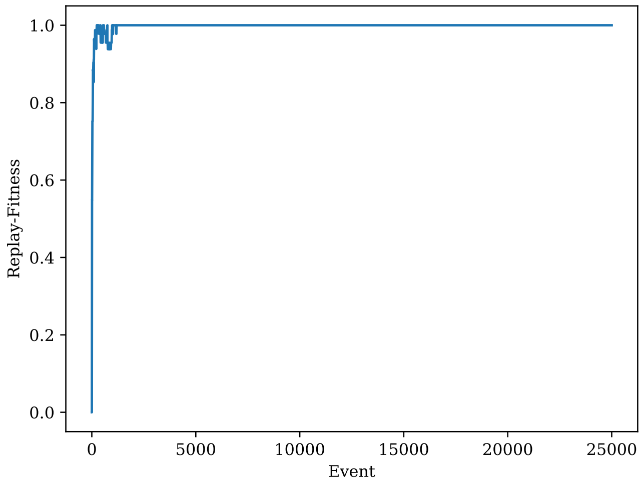


(b) Distances for State/Language Based Region Theory (*Trans. Sys./ILP*) and the Heuristics Miner (*Heuristics*).

Figure 5.6: Distance measurements for the Alpha algorithm, Inductive Miner (IM), ILP Miner (ILP), Transition Systems Miner (TS) and Heuristics Miner.

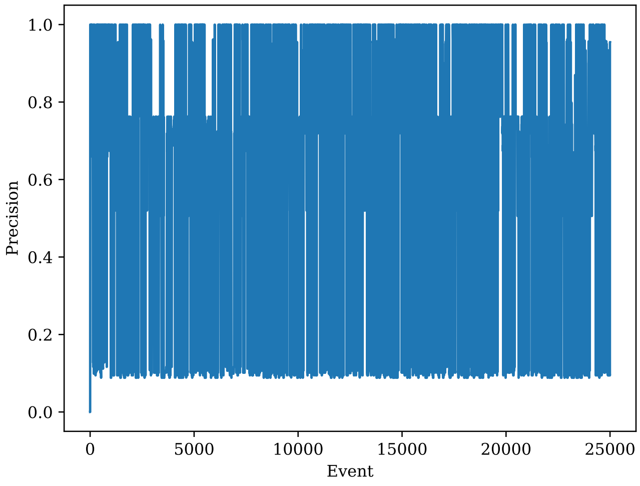


(a) Size constraints $|X|_{\mathscr{C}\times\mathscr{A}} = 25$ and $|X|_{\mathscr{A}\times\mathscr{A}} = 25$.

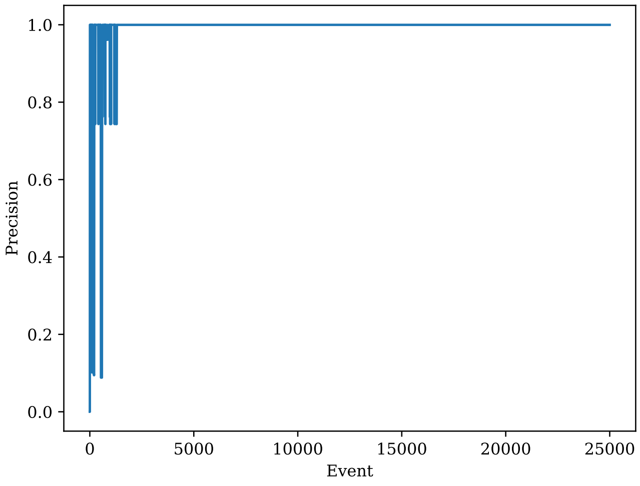


(b) Size constraints $|X|_{\mathscr{C}\times\mathscr{A}} = 75$ and $|X|_{\mathscr{A}\times\mathscr{A}} = 75$.

Figure 5.7: Replay-fitness measures based on applying the Stream Inductive Miner: Increasing memory helps to improve fitness.



(a) Size constraints 25/25.



(b) Precision results with size constraints 75/75.

Figure 5.8: Precision measures based on applying the Stream Inductive Miner: Increasing memory helps to improve precision.

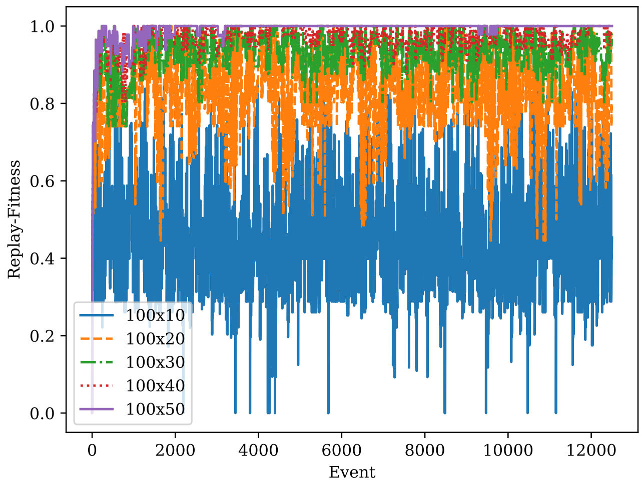
does not stabilize. When the data structure size increases, i.e. Figure 5.7b, we identify the replay-fitness to reach a value of 1 rapidly. The high variability in the precision measurements present in Figure 5.8a suggests that the algorithm is not capable of storing the complete directly follows abstraction. As a result, the Inductive Miner tends to create flower-like patterns, thus greatly under-fitting the actual process. This is confirmed by the relatively high values for replay-fitness in Figure 5.7a. The stable pattern present in Figure 5.8b, suggests that the sizes used within the experiment are sufficient to store the complete directly follows abstraction. Given that the generating process model is within the class of *re-discoverable process models* of the Inductive Miner, both a replay-fitness and a precision value of 1 indicate that the model is completely discovered by the algorithm.

In the previous experimental setting, we chose to use the same capacity for both supporting data structures. We additionally study the influence of the individual sizes of the collection of pairs of the form $(c,a) \in \mathscr{C} \times \mathscr{A}$ versus the directly follows relation itself. We denote these sizes as $|X|_{\mathscr{C} \times \mathscr{A}}$ and $|X|_{\mathscr{A} \times \mathscr{A}}$ respectively. In Figure 5.9 we depict the results of two different experiments in which we fixed the size of one of the two data structures and varied the size of the other data structure. Figure 5.9a depicts the results for a fixed value $|X|_{\mathscr{C} \times \mathscr{A}} \leq 100$ and varying sizes of the directly follows abstraction $|X|_{\mathscr{A} \times \mathscr{A}} \leq 10, 20, ..., 50$. Figure 5.9b depicts the results for a fixed value $|X|_{\mathscr{A} \times \mathscr{A}} \leq 100$ and varying sizes $|X|_{\mathscr{C} \times \mathscr{A}} \leq 10, 20, ..., 50$. As the results show, the lack of conversion to a replay-fitness value of 1 mostly depends on the size of directly follows relation and is relatively independent of the supporting data structure that allows us to construct it. Intuitively this makes sense as we only need one entry $(c,a) \in \mathscr{C} \times \mathscr{A}$ to deduce $a > a'$, given that the newly received event is (c, a'). Even if some case identifier c is dropped at some point in time, and reinserted later, still information regarding the directly follows abstraction can be deduced. However, if insufficient space is reserved for storing the directly follows relation, then the data structure is incapable of storing the complete directly follows abstraction, which negatively impacts the replay-fitness results.

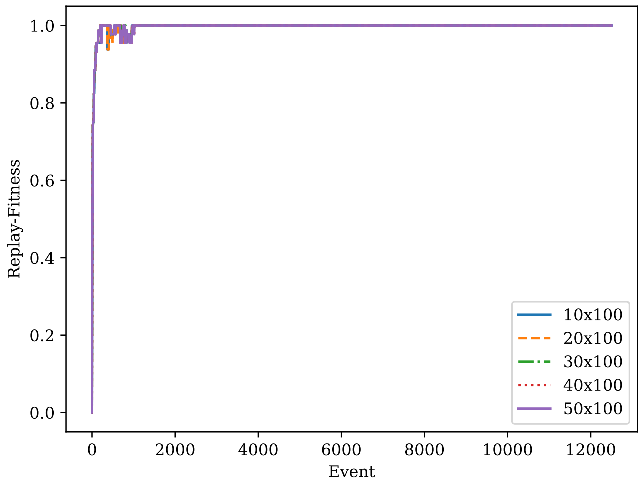
5.4.3 Concept Drift

In the previous experiments, we focused on a process model that describes observed *steady state* behaviour, i.e. the process model from which events are sampled does not change during the experiments. In this section, we assess to what extent the Inductive Miner based instantiation of the framework is able to handle concept drift [29, 104]. We focus on a *gradual drift*, i.e. the behaviour of the process model changes at some point in time, though the change is only applicable for *new instances of the process*, i.e. already active cases follow the old behaviour. In order to obtain a gradual drift, we manipulated the CPN simulation model that is used to generate the stream. In particular, we change a choice construct within the model into a parallel construct, and vice-versa. The first 500 process instances that are simulated follow the original model. All later cases are routed to the altered model are simulated accordingly.

Figure 5.10 depicts the results of applying the Inductive Miner on the described gradual drift. In Figure 5.10a, we depict the results using data structure sizes $|X|_{\mathscr{C} \times \mathscr{A}} \leq 100$ and $|X|_{\mathscr{A} \times \mathscr{A}} \leq 50$ (using Lossy Counting for both data structures). The blue solid line depicts the replay-fitness with respect to an event log containing behaviour *prior* to the drift, the red dashed line represents replay-fitness with respect to an event log containing behaviour *after* the drift. We observe that the algorithm again needs some time to stabilize in terms of behaviour with respect to the pre-drift model. Interestingly, at the moment that the algorithm seems to be stabilized with respect to the pre-drift model, the replay-fitness with respect to the post-drift model fluctuates. This indicates that the algorithm is not able to fully rediscover the pre-drift model, yet it produces a generalizing model which includes more behaviour, i.e. even behaviour that is part of the

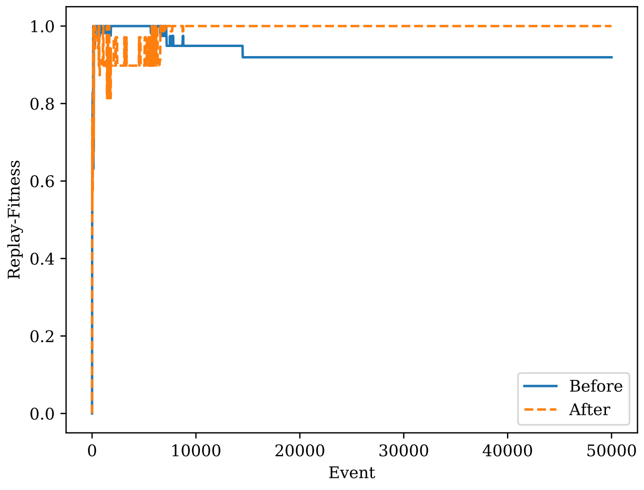


(a) $|X|_{\mathscr{C}\times\mathscr{A}} = 100$, $|X|_{\mathscr{A}\times\mathscr{A}} = 10, 20, \ldots, 50$.

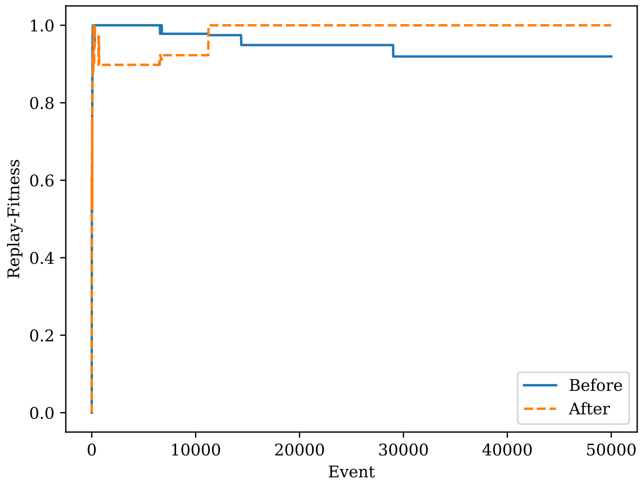


(b) $|X|_{\mathscr{C}\times\mathscr{A}} = 10, 20, \ldots, 50$, $|X|_{\mathscr{A}\times\mathscr{A}} = 100$.

Figure 5.9: Replay-fitness measures for the Stream Inductive Miner with varying sizes of the internal data structures used.



(a) $|X|_{\mathscr{C}\times\mathscr{A}} = 100$, $|X|_{\mathscr{A}\times\mathscr{A}} = 50$.



(b) $|X|_{\mathscr{C}\times\mathscr{A}} = 100$, $|X|_{\mathscr{A}\times\mathscr{A}} = 100$.

Figure 5.10: Replay-fitness measures for the Stream Inductive Miner, given an event stream containing concept drift.

Table 5.1: Aggregate performance measures for the Stream Inductive Miner.

	25x25	*50x50*	*75x75*
Avg. processing time (ns.):	4.7167,77	3.866,45	3.519,22
Stdev. processing time (ns.):	3.245,80	2.588,76	2.690,54
Avg. memory usage (byte):	75.391,75	81.013,60	84.695,86
Stdev. memory usage (byte):	762,55	1.229,60	1.724,98

post-drift model. The first event in the stream related to the new execution of the process, is the 6.415^{th} event. Indeed, replay-fitness with respect to the pre-drift model starts to drop around this point in Figure 5.10a. Likewise, the replay-fitness with respect to the post-drift model rapidly increases to a value of 1.0. Finally, around event 15.000 the replay-fitness with respect to the pre-drift model stabilizes completely, indicating that the prior knowledge related to the pre-drift model is completely erased from the underlying data structure. In Figure 5.10b, we depict results for the Inductive miner using sizes $|X|_{\mathscr{C}\times\mathscr{A}} = 100$ and $|X|_{\mathscr{A}\times\mathscr{A}} = 100$. In this case, we observe more stable behaviour, i.e. both the pre- and post-model behaviour stabilizes quickly. Interestingly, due to the use of a bigger size capacity of the Lossy Counting Algorithm, the drift is reflected longer in the replay-fitness values. Only after roughly the 30.000^{th} event the replay-fitness with respect to the pre-drift model stabilizes.

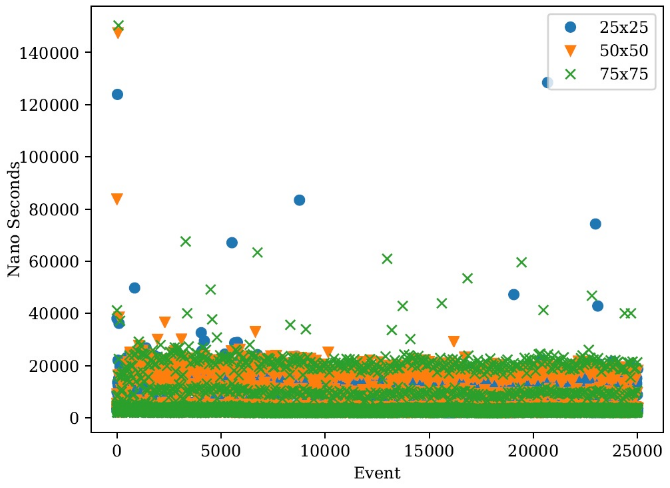
5.4.4 Performance Analysis

The main goal of the performance evaluation is to assess whether memory usage and processing times of the implementations are acceptable. As the implementations are of a prototypical fashion, we focus on *trends* in processing time and memory usage, rather than absolute performance measures. For both processing time and memory usage, we expect stabilizing behaviour, i.e. over time we expect to observe some non-increasing asymptote. If the processing time/memory usage keeps increasing over time, this implies that we are potentially unable to handle data on the stream or need infinite memory.

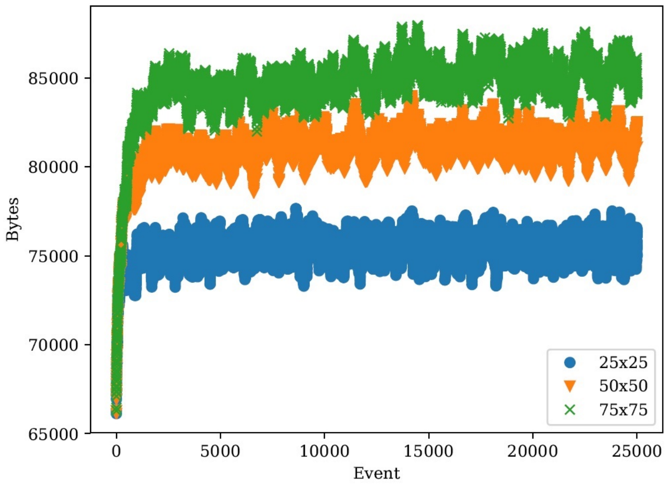
In this experiment, we measured the processing time and memory usage for handling the first 25.000 events emitted onto the stream. We again use the Inductive Miner with Lossy Counting and varying window sizes (parameter k): $|X|_{\mathscr{C}\times\mathscr{A}} = 25$ and $|X|_{\mathscr{A}\times\mathscr{A}} = 25$, $|X|_{\mathscr{C}\times\mathscr{A}} = 50$ and $|X|_{\mathscr{A}\times\mathscr{A}} = 50$ and $|X|_{\mathscr{C}\times\mathscr{A}} = 75$, $|X|_{\mathscr{A}\times\mathscr{A}} = 75$ (represented in the Figures as 25x25, 50x50 and 75x75 respectively). We measured the time the algorithm needs to update both data structures. The measured memory is the combined size of both data structures in bytes. The results of the experiments are depicted in Figure 5.11. Both figures depict the total number of events received on the x-axis. In Figure 5.11a, the processing time in nanoseconds is shown on the y-axis, whereas in Figure 5.11b, the memory usage in bytes is depicted. The aggregates of the experiments are depicted in Table 5.1.

As Figure 5.11a shows, there is no observable increase in processing times as more events have been processed. The average processing time seems to slightly decrease when the window size of the Lossy Counting data structure increases (see Table 5.1). Intuitively this makes sense as a bigger window size of the Lossy Counting algorithm implies less frequent clean-up operations.

Like processing time, memory usage of the Lossy Counting data structures does not show an increasing trend (Figure 5.11b). In this case however, memory usage seems to increase when the window size of the Lossy Counting algorithm is bigger. Again this makes sense, as fewer clean-up operations imply more active elements within the data structures, and hence, a higher



(a) Processing times in nanoseconds.



(b) Memory usage in bytes.

Figure 5.11: Performance measurements based on the Stream Inductive Miner.

memory usage.

5.5 Related Work

Several researchers have worked on the topic of stream-based process discovery. Hence, in that regard, the architecture presented in this chapter may be regarded as a generalization and standardization effort of some of the related work mentioned in this section.

In [35] an event stream-based variant of the Heuristics Miner is presented. The algorithm uses three internal data structures using both Lossy Counting [84] and Lossy Counting with Budget [103]. The authors use these structures to approximate a causal graph based on an event stream. The authors additionally present a sliding window based approach, which is comparable in nature to using a sliding window-based event store. However, the authors do not consider building the directly follows relation on top of the window, i.e. it is mainly considering the use of a sliding window as described in chapter 3. Built on top of the aforementioned work, an alternative data structure has been proposed based on prefix-trees [69]. In this work the authors deduce the directly follows abstraction directly from a prefix tree which is maintained in memory. The main advantage of using the prefix-trees is the reduced processing time and usage of memory. The prefix tree is however cleaned up at regular intervals and as such significantly differs from the prefix-based storage as described in this thesis, cf. section 3.4. Although the aforementioned method is designed to specifically construct a directly follows abstraction, it is straightforward to extend it to support length-k distance relations. Furthermore, in [55], Evermann et al. present an adaptation of the Flexible Heuristics Miner [122] in the context of large-scale distributed event stream settings. They provide an associated implementation using a distributed cloud-service based web service (Amazon Kinesis) and assert the associated performance.

In [100], Redlich et al. design an event stream-based variant of the CCM algorithm [99]. The authors identify the need to compute dynamic footprint information based on the event stream, which can be seen as the intermediate representation used by CCM. The dynamic footprint is translated to a process model using a translation step called footprint interpretation. The footprint interpretation step mainly reduces the overall complexity of the conventional CCM divide and conquer steps to allow the algorithm to be adapted in a streaming setting. The authors additionally apply an ageing factor to the collected trace information to fade out the behaviour extracted from older traces. Although the authors define event streams similarly to the notion adopted within this thesis, the evaluation relies heavily on the concept of *completed traces*. Similarly, in [31] Boushaba et al. developed an incremental extension of their block structured miner [30], which aims at finding patterns in a matrix representation of the directly follows relation. Within this work the authors assume the stream to be a sequence of *event logs containing completed traces*, rather than a stream of events.

In [34] Burattin et al. propose an event stream-based process discovery algorithm to discover *declarative process models*, i.e. models that constrain behaviour rather than specifying behaviour exactly. The structure described to maintain events and their relation to cases is comparable with the one used in [35]. The authors present several declarative constraints that can be updated on the basis of newly arriving events instead of an event log consisting of full traces.

5.6 Conclusion

In this chapter, we presented a generic architecture that allows for adopting existing process discovery algorithms in an event stream setting. The architecture is based on the observation that many existing process discovery algorithms translate a given event log into an intermediate representation and subsequently translate such intermediate representation into a process model. Thus, in an event stream-based setting, it suffices to approximate an intermediate representation on the basis of the event stream, in order to apply existing process discovery algorithms to streams of events. The exact behaviour present in the resulting process model greatly depends on the instantiation of the underlying techniques that approximate the intermediate representation.

5.6.1 Contributions

The architecture as proposed within this chapter can be regarded as both a generalization and standardization effort of existing and/or future event stream-based process discovery algorithms. As such, it describes a computational mechanism that is applicable to a large class of process discovery algorithms. Moreover, several instantiations of the architecture have been proposed and implemented in the process mining tool-kits ProM and RapidProM.

Within the experiments performed in the context of this chapter, we primarily focused on intermediate representation approximations using underlying algorithms designed for the purpose of frequent item mining on data streams. However, as presented, other types of data storage techniques can be used as well. We structurally evaluated and compared five different instantiations of the framework. From a behavioural perspective we focused on the Inductive Miner as it guarantees to produce sound workflow nets. The experiments show that the instantiation is able to capture process behaviour originating from a steady state-based process. Moreover, convergence of replay-fitness to a stable value depends on parametrization of the internal data structure. In case of concept drift, the size of the internal data structure of use impacts both model quality and the drift detection point. We additionally studied the performance of the Inductive Miner-based instantiation of the architecture. The experiments show that both processing time of new events and memory usage are non-increasing as more data is received.

5.6.2 Limitations

In this section, we discuss interesting phenomena observed during experimentation which should be taken into account when adopting the architecture presented in this paper, and, in event-stream-based process discovery in general. We discuss limitations with respect to the complexity of intermediate representation computation and discuss the impact of the absence of trace initialization and termination information.

Complexity of Abstract Representation Computation

In terms of minimizing the memory footprint needed to compute intermediate representations, there are limitations with respect to the algorithms we are able to adopt. This is mainly related to the computation of the intermediate representation within the conventional algorithm.

As an example, consider the Alpha+ algorithm [89] which extends the original Alpha algorithm such that it is able to handle self-loops and length-1-loops. For handling self-loops, the Alpha+ algorithm traverses the event log and identifies activities that are within a self-loop. Subsequently, it removes these from the log and after that calculates the directly follows

$$\begin{tikzcd} a \arrow[r] \arrow[dr] & c & \\ & e \arrow[r] & d \end{tikzcd}$$

(a) Event log based.

$$\begin{tikzcd} a \arrow[r] \arrow[dr] \arrow[drr, dashed] & c & \\ & e \arrow[r] \arrow[u, dashed] & d \end{tikzcd}$$

(b) Directly follows relation based.

Figure 5.12: Two directly follows relations derived by the (online) Alpha+ algorithm based on simple traces $\langle a, b, b, c\rangle$ and $\langle a, e, b, d\rangle$. One is purely event log based (Figure 5.12a) the other one is deduced on the basis of the directly follows relation itself (Figure 5.12b).

abstraction. For example, if we are given a simple event log (events projected onto activities) $\tilde{L} = [\langle a, b, c\rangle, \langle a, b, b, c\rangle]$, the algorithm will construct $\tilde{L}' = [\langle a, c\rangle]$ and compute directly follows metrics based on $\tilde{L}'$. Based on the full trace $\langle a, b, b, c\rangle$, the algorithm deduces that $b > b$ holds and thus it is removed completely from the directly follows relation.

In a streaming setting, we are able to handle this as follows. Whenever we observe some activity a to be in a self-loop and want to generate the directly follows abstraction, then for every pair $(a', a) \in \mathscr{A} \times \mathscr{A}$ and $(a, a'') \in \mathscr{A} \times \mathscr{A}$, s.t. $a \neq a'$ and $a \neq a''$, we deduce that (a', a'') is part of the directly follows abstraction whereas (a, a), (a', a) and (a, a'') are not. Although this procedure approximates the directly follows relation on the event stream, a simple example reveals that the relation is not always equal.

Imagine that we study an event stream, originating from a process consisting of just two simple trace-variants, i.e. $\langle a, b, b, c\rangle$ and $\langle a, e, b, d\rangle$. Any noise-free event log over this process is just a multiset over the two given traces. In case of the conventional Alpha+ algorithm, removing the b-activity leads to the two simple trace-variants $\langle a, c\rangle$ and $\langle a, e, d\rangle$. Consider the corresponding directly follows abstraction, depicted in Figure 5.12a. Observe that all possible directly follows pairs that we are able to observe on any stream generated by the process are: $(a, b), (a, e), (b, b), (b, c), (b, d), (e, b)$. Applying the described procedure yields the directly follows relation depicted in Figure 5.12b. Due to the information that is lost by only maintaining directly follows pairs, we deduce non-existing relations (a, d) and (e, c).

In general, it is preferable to adopt process discovery algorithms that constructs the intermediate representation in *one pass* over the event log. This is due to the fact that algorithms using more passes typically use some aggregate information derived from the previous pass into the next pass. Such information typically involves the event log as a whole. Hence, for this type of algorithms, it is more suitable to instantiate the framework using event stores, i.e. by maintaining an event log based on the stream.

Initialization and Termination

For the definitions presented in this paper, we abstract from trace initialization and/or termination, i.e. we do not assume the existence of explicit start/end events. Apart from the technical challenges related to finding these events, i.e. as described earlier regarding start/end activity sets used by the Alpha algorithm and Inductive Miner, this can have a severe impact on computing the intermediate representation as well.

If we assume the existence and knowledge of unique start and end activities, adopting any algorithm to cope with this type of knowledge is trivial. We only consider cases of which we identify a start event and we only remove knowledge related to cases of which we have seen the end event. The only challenge is to cope with the need to remove an unfinished case due to memory issues, i.e. how to incorporate this deletion into the data structure/intermediate representation that is approximated.

If we do not assume and/or know of the existence of start/end activities, whenever we encounter a case for which our data structure indicates that we have not seen it before, this case is identified as being a “new case”. Similarly, whenever we decide to drop a case from a data structure, we implicitly assume that this case has terminated. Clearly, when there is a long period of inactivity, a case might be falsely assumed to be terminated. If the case becomes active again, it is treated as a new case again. The experiments reported on in Figure 5.9 show that in case of the directly follows abstraction, this type of behaviour has limited impact on the results. However, in a more general sense, e.g. when approximating a prefix-closure on an event stream, this type of behaviour might be of greater influence with respect to resulting model. The ILP Miner likely suffers from such errors and, as a result, produces models of inferior quality.

In fact, for the ILP Miner the concept of termination is of particular importance. To guarantee a single final state of a process model, the ILP Miner needs to be aware of *completed traces*. This corresponds to explicit knowledge of when a case is terminated in an event stream setting. Like in the case of initialization, the resulting models of the ILP miner are greatly influenced by a faulty assumption on case termination.

5.6.3 Open Challenges & Future Work

The architecture presented in this chapter focuses on approximating intermediate representations and exploiting existing algorithms to discover a process model. As such, we strive to minimize the overall memory footprint used in order to perform the discovery algorithms. However, the bulk of the work might still be performed multiple times, i.e. several new events emitted to the stream might not change the abstract representation.

An interesting avenue for future work is therefore related to a completely incremental instantiation of the architecture, i.e. are we able to immediately identify whether new changes to the abstraction have an impact with respect to resulting model. For example, in case of the Alpha miner, if we deduce a directly follows relation that is already observed in the past, we are guaranteed that the resulting model is not changing with respect to any previously found model. However, deriving a new directly follows relation is more likely to generate either a new place in the corresponding Petri net, or, invalidate some of the previously found places. In case of the Inductive Miner, which iteratively partitions the directly follows abstraction, when an existing directly follows relation is observed, such partition is not violated. Moreover, if a new relation is observed, or a relation is removed, the impact of such modification is potentially local, i.e. the model has an associated hierarchy and a modification potentially only affects a local part of the model. Similarly, the constraints that we derive for ILP based process discovery allow us to validate Petri net places. Thus, given that we have already discovered a set of Petri net places, upon deducing a new constraint, we need $O(|P|)$ to at least verify the correctness of the previous model.

Chapter 6

Improving Process Discovery Results by Exploiting Intermediate Representations

In the previous chapter, we have presented an architectural framework that allows us to reduce and bound the memory footprint of online event stream based process discovery algorithms. In this chapter, we show that, for the class of process discovery approaches based on language-based region theory, we are able to significantly improve process discovery results by exploiting the intermediate representations used by the algorithm. We show that we are able to, under certain conditions, guarantee both structural and behavioural properties of the discovered process model. Moreover, we show that the intermediate representation related to language-based region theory allows us to effectively filter out outlier behaviour from event data.

The contents presented in this chapter are based on the following publications:

S.J. van Zelst, B.F. van Dongen, W.M.P. van der Aalst, and H.M.W. Verbeek. Discovering Workflow Nets using Integer Linear Programming. *Computing*, 100(5):529-556, 2018; `https://doi.org/10.1007/s00607-017-0582-5`

S.J. van Zelst, B.F. van Dongen, W.M.P. van der Aalst. Avoiding Over-Fitting in ILP-Based Process Discovery. *Business Process Management - 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 - September 3, 2015, Proceedings*, volume 9253 of *Lecture Notes in Computer Science*, pages 163–171. Springer, 2015; `http://doi.org/10.1007/978-3-319-23063-4_10`

S.J. van Zelst, B.F. van Dongen, and W.M.P. van der Aalst. ILP-Based Process Discovery Using Hybrid Regions. *ATAED 2015*, volume 1371 of *CEUR Workshop Proceedings*, pages 47-61. CEUR-WS.org, 2015; `http://ceur-ws.org/Vol-1371/paper04.pdf`

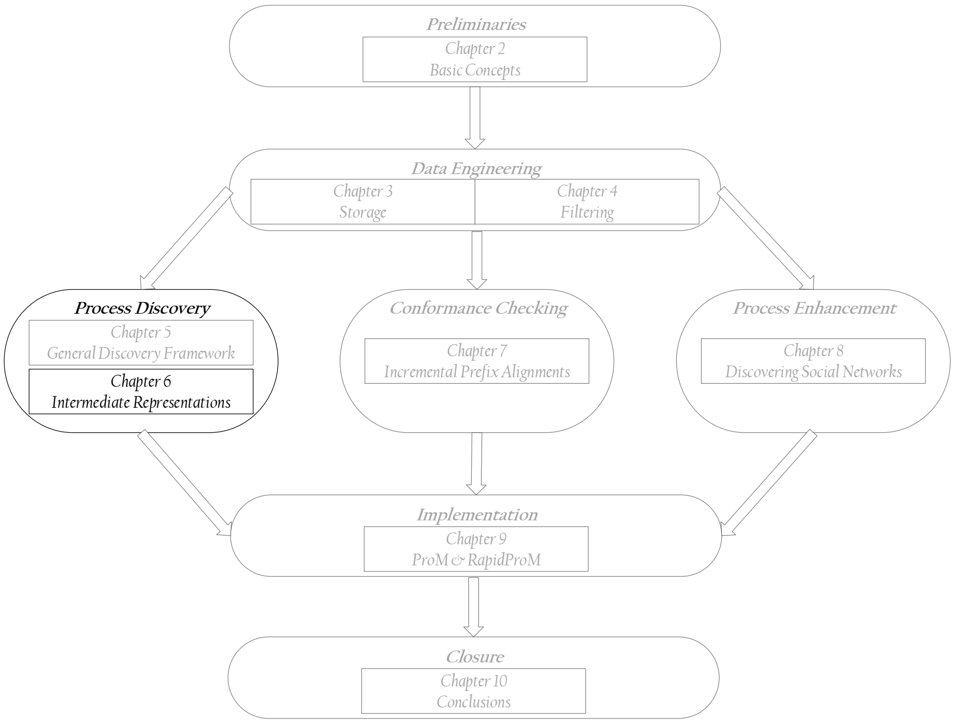


Figure 6.1: The contents of this chapter (highlighted in grey), i.e. exploiting intermediate representations in the context of ILP-based process discovery, in the context of the general structure of this thesis.

6.1 Introduction

The previous chapter highlights the importance of intermediate representations within process discovery. Thus far, we have considered intermediate representations from a fairly high level, i.e. we provided a basic description of the directly follows relation and language/state-based region theory and their subsequent use in process discovery. In this chapter, we go beyond such high-level consideration and dive deeper into a specific intermediate representation. In particular, we focus on process discovery based on language-based region theory [19, 43].

Language-based region theory, and in a broader context region theory (which also covers state-based region theory), represents a solution method to the classical *Petri net synthesis problem* [20]. Here the problem is to, given a *behavioural system description*, decide whether there exists a Petri net that allows for *all behaviour of the given system description*. Moreover, the resulting Petri net needs to exactly describe the behavioural system description, i.e. it must not allow for additional behaviour. If however, such Petri net does not exist, the resulting Petri net needs to minimize additional behaviour. Applying classical region theory using an event store as a system description results in Petri nets with *maximal* replay-fitness, i.e. a value of 1.0.[1] Moreover, precision is *maximized*. This is likely to result in models with *poor generalization* and *poor simplicity*. Using these techniques directly on real event data, therefore, results in process models that are not an adequate representation of the event data and do not allow us to reach the global goal of process mining, i.e. turning data into actionable knowledge.

In [123] a process discovery algorithm is proposed on top of language-based region theory. In particular, the concept of language-based region theory is translated to an Integer Linear Programming (ILP) [105] formulation. Such ILP formulation represents a mathematical optimisation problem and allows us to find an optimal solution, in terms of a set of variables, subject to a given set of constraints. The main contribution of the aforementioned work is a relaxation of the precision maximization property of language-based region theory. The algorithm still guarantees that the resulting process model is able to replay all behaviour in the event store. Opposed to state-of-the-art process discovery algorithms, the algorithm provides limited guarantees with respect to structural and behavioural properties of the resulting process models. Moreover, the algorithm only works well under the assumption that the event store only holds frequent behaviour that fits nicely into some underlying process model.

As motivated in chapter 4, real event data typically include low-frequent exceptional behaviour, e.g. caused by people deviating from the normative process, cases that require special treatment, employees/resources solving unexpected issues in an ad-hoc fashion etc. Considering all these irregularities together with "normal behaviour" yields incomprehensible models, both in classical region-based synthesis and region-based process discovery techniques. Therefore, in this chapter, we propose to tackle these problems by further extending and improving existing, region theory-based process discovery algorithms [123, 131, 133]. As such, the contents and contributions of this chapter are summarized into two main branches, i.e.

1. *Providing guarantees with respect to structural and behavioural properties of discovered models*;

   We prove that we are, under certain assumptions, able to discover *Workflow nets* (structural, cf. 2.3) which we prove to be *relaxed sound* (behavioural, cf. 2.6).

2. *Improving the quality of discovered process models in terms of process mining quality dimensions*;

   We present an integrated filter, that explicitly exploits the nature of the intermediate

[1]Recall the process mining quality dimensions as described in subsection 1.1.4.

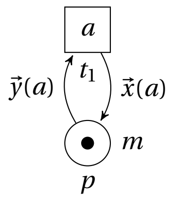


Figure 6.2: Visualization of the decision variables and their relation to Petri net places in ILP-based process discovery.

representation used in ILP-based process discovery. As such, we improve the quality of the discovered models in terms of process mining quality dimensions, cf. subsection 1.1.4

The techniques we present in this chapter are based on the *conventional event log assumption*, i.e. we assume the input event data to be a static collection of traces. Note that, as we elaborately described in chapter 3, there are several ways to obtain a static view on the basis of an event stream, i.e. by means of event stores. Therefore, when integrated with these event stores, the contents of this paper are implicitly applicable in a streaming setting.

The remainder of this chapter is organized as follows. In section 6.2, we describe the basic mechanics of process discovery using integer linear programming. In section 6.3, we prove that, constraint to certain assumptions, we are able to guarantee that ILP-based process discovery is able to produce *relaxed-sound workflow nets*. In section 6.4, we show that we are able to improve ILP-based process discovery results by exploiting the underlying intermediate representation used. In section 6.5, we evaluate the impact of the proposed filter on the corresponding process discovery results. In section 6.6, we present related work, primarily focussing on Petri net synthesis and its application to process discovery. Finally, in section 6.7, we conclude this chapter.

6.2 ILP-Based Process Discovery

In this section, we describe the basics of ILP-based process discovery. We first explain integer linear programming and the concept of regions, originating from classical region theory, and discuss their relation. Subsequently we show that we are able to construct a corresponding generalized objective function that is guaranteed to find minimal regions.

6.2.1 Regions and Integer Linear Programming

In [123] an Integer Linear Programming (ILP)-formulation [105] is presented which allows us to discover the places of a Petri net, based on event data stored in a (simple) event store. In case of ILP-based process discovery, the (optimal) solution to an ILP-formulation corresponds to a place in the resulting Petri net, i.e. the process model that we construct. The variables of the corresponding ILP-formulation are expressed in terms of the initial marking of the place and the arcs connected from/to the place. Consider Figure 6.2, in which we schematically introduce the variables used in ILP-based process discovery. For each activity label $a \in \mathscr{A}$, which is also present within the (simple) event store used within discovery, we have two boolean variables. Variable $\vec{x}(a)$ is a boolean variable representing the presence of an *incoming arc* from a transition with label a to the place corresponding to the solution of the ILP-formulation, whereas variable $\vec{y}(a)$

represents a corresponding *outgoing arc*. Variable m is a boolean variable representing the initial marking of the place.[2] Observe that, due to the coupling between variables and activity labels $a \in \mathscr{A}$, we have at most one transition within the resulting model that describes such label, i.e. no label duplication. Moreover, we are not able to discover invisible transitions, i.e. transitions $t \in T$ with $\lambda(t) = \tau$.

As indicated in section 6.1, *language-based region theory* forms a fundamental basis of ILP-based process discovery. In line with the concept of an ILP-formulation, we present the notion of regions in terms of the aforementioned set of binary decision variables. Subsequently we show how to utilize regions in the context of an ILP-formulation. The premise of a region is the fact that its corresponding place, given the prefix-closure of the current event store, does not block the execution of any sequence within the prefix-closure.

Definition 6.1 (Region)**.** *Let $\tilde{\Phi} \in \mathscr{B}(\mathscr{A}^*)$ be a simple event store. Furthermore, let $m \in \mathbb{B}$ and let $\vec{x}, \vec{y} \in \mathbb{B}^{|A_{\tilde{\Phi}}|}$. A triple $r = (m, \vec{x}, \vec{y})$ is a* region*, if and only if:*

$$\forall \sigma = \sigma' \cdot \langle a \rangle \in \overline{\tilde{\Phi}_+} \left(m + \vec{\sigma'}^{\intercal} \vec{x} - \vec{\sigma}^{\intercal} \vec{y} \geq 0 \right) \tag{6.1}$$

We let $\mathscr{F}(\tilde{\Phi})$ denote the set of all variable assignments adhering to Equation 6.1, i.e. the set of feasible regions, given $\tilde{\Phi}$.

Observe that Equation 6.1 is defined in terms of the Parikh representation of a non-empty activity sequence in the prefix-closure of the simple event store. In particular, it states that the prefix of such sequence (σ') needs to be able to account for firing the sequence as a whole (σ). Note that this is the case because the $\vec{x}$ vector represents incoming arcs, and thus if $\vec{x}(a) = 1$ and $\vec{\sigma'}(a) = 2$, in total, two tokens are produced in the corresponding place. Similarly, if $\vec{y}(b) = 1$ and $\vec{\sigma}(b) = 1$, we know that a transition labelled with b consumes one token. Observe that the Parikh vectors abstract from the ordering of activities within the activity sequences. However, since we generate an inequality for each element of the prefix-closure of the simple event store, i.e. $\overline{\tilde{\Phi}_+}$, the ordering of activities as exhibited by the traces is implicitly accounted for.

A region r, i.e. a triple of variable assignments that adheres to Equation 6.1, is translated to a Petri net place p as follows. Given some Petri net $N = (P, T, F, \lambda)$, in which we represent the unique transition with label a as $t_a \in T$. If $\vec{x}(a) = 1$, we add t_a to $\bullet p$, i.e. $F \leftarrow F \cup \{(t_a, p)\}$. Symmetrically, if $\vec{y}(a) = 1$, we add t_a to $p\bullet$, i.e. $F \leftarrow F \cup \{(p, t_a)\}$. Finally, if $m = 1$, place p is initially marked. Since translating a region to a place is deterministic, we are also able to translate a place to a region, e.g. a place p with $t \in \bullet p$ and $\lambda(t) = a$ corresponds to a region in which $\vec{x}(a) = 1$.

Prior to presenting the basic ILP-formulation for finding regions, we formulate regions in terms of matrices, which we use in the ILP-formulation.

Definition 6.2 (Region (Matrix Form))**.** *Let $\tilde{\Phi} \in \mathscr{B}(\mathscr{A}^*)$ be a simple event store. Let* $\mathbf{M}$ *and* $\mathbf{M}'$ *be two $|\overline{\tilde{\Phi}_+} \setminus \{\epsilon\}| \times |A_{\tilde{\Phi}}|$ matrices with $\mathbf{M}(\sigma, a) = \vec{\sigma}(a)$ and $\mathbf{M}'(\sigma, a) = \vec{\sigma'}(a)$ (where $\sigma = \sigma' \cdot \langle a' \rangle \in \overline{\tilde{\Phi}_+}$). Tuple $r = (m, \vec{x}, \vec{y})$ is a region if and only if:*

$$m\vec{1} + \mathbf{M}'\vec{x} - \mathbf{M}\vec{y} \geq \vec{0} \tag{6.2}$$

We additionally define matrix $\mathbf{M}_{\tilde{\Phi}}$ which is an $|\tilde{\Phi}_+| \times |A_{\tilde{\Phi}}|$ matrix with $\mathbf{M}_{\tilde{\Phi}}(\sigma, a) = \vec{\sigma}(a)$ for $\sigma \in_+ \tilde{\Phi}$, i.e. $\mathbf{M}_{\tilde{\Phi}}$ is the equivalent of $\mathbf{M}$ for all complete traces in the event store. We define

[2] In the context of this thesis we restrict the variables to be boolean, i.e. $\mathbb{B}$, however, when adopting the notion of arc weights and multiple tokens within the initial marking, we are able to use the set of natural numbers as well, i.e. $\mathbb{N}_0$.

a general process discovery ILP-formulation that guarantees to find a non-trivial region, i.e. regions not equal to $(0,\vec{0},\vec{0})$ or $(1,\vec{1},\vec{1})$, with the property that its corresponding place is always empty after replaying each trace within the event store.

Definition 6.3 (Process Discovery ILP-formulation [123]). *Let $\tilde{\Phi}\in\mathscr{B}(\mathscr{A}^*)$ be a simple event store and let* $\mathbf{M}$*,* $\mathbf{M}'$ *and* $\mathbf{M}_{\tilde{\Phi}}$ *be the matrices defined in 6.2. Let $c_m\in\mathbb{R}$ and $\vec{c_x},\vec{c_y}\in\mathbb{R}^{|A_{\tilde{\Phi}}|}$. The process discovery ILP-formulation, $ILP_{\tilde{\Phi}}$, is defined as:*

$$\begin{array}{rrl}
\textit{minimize} & z = c_m m + \vec{c_x}^\intercal \vec{x} + \vec{c_y}^\intercal \vec{y} & \textit{objective function}\\
\textit{such that} & m\vec{1} + \mathbf{M}'\vec{x} - \mathbf{M}\vec{y} \geq \vec{0} & \textit{theory of regions}\\
\textbf{and} & m\vec{1} + \mathbf{M}_{\tilde{\Phi}}(\vec{x}-\vec{y}) = \vec{0} & \textit{corresp. place is empty after each trace}\\
3 & \vec{1}^\intercal\vec{x} + \vec{1}^\intercal\vec{y} \geq 1 & \textit{at least one arc connected}\\
 & \vec{0} \leq \vec{x} \leq \vec{1} & \textit{i.e. } \vec{x}\in\{0,1\}^{|A_{\tilde{\Phi}}|}\\
 & \vec{0} \leq \vec{y} \leq \vec{1} & \textit{i.e. } \vec{y}\in\{0,1\}^{|A_{\tilde{\Phi}}|}\\
 & 0 \leq m \leq 1 & \textit{i.e. } m\in\{0,1\}
\end{array}$$

We let $\mathscr{F}_{ILP}(\tilde{\Phi})$ denote the set of all variable assignments (i.e. corresponding to regions) adhering to the constraints of the ILP-formulation as presented in 6.3, i.e. the *feasible regions of the ILP*. Thus, note that $(0,\vec{0},\vec{0}),(1,\vec{1},\vec{1}) \notin \mathscr{F}_{ILP}(\tilde{\Phi})$, i.e. it only contains non-trivial regions. Furthermore, we let $r^*\in\mathscr{F}_{ILP}(\tilde{\Phi})$ denote the region that minimizes the objective function z as defined in 6.3, i.e. the *optimal region*[4]. In the upcoming section, we show that we are able to characterize a generalized objective function, which allows us to discover places with certain characteristics, beneficial for process discovery, i.e. *minimal regions*

6.2.2 Guarantees on Discovering Minimal Regions

6.3 allows us to find a region that minimizes objective function $z = c_m m + \vec{c_x}^\intercal \vec{x} + \vec{c_y}^\intercal \vec{y}$. Multiple instantiations of z, i.e. in terms of *objective coefficients* c_m, $\vec{c_x}$ and $\vec{c_y}$, are possible. In [123], an objective function is proposed that minimizes the number of 1-valued entries in $\vec{x}$ and maximizes 1-valued entries in $\vec{y}$, i.e. in the region's corresponding place the number of incoming arcs is minimized whereas the number of outgoing arcs is maximized. In this section, we show that we are able to instantiate the aforementioned function with an arbitrary positive scalar and are guaranteed to find *minimal regions*. Prior to this, we define the notion of minimal regions.[5]

Definition 6.4 (Minimal region). *Let $\tilde{\Phi}\in\mathscr{B}(\mathscr{A}^*)$ be a simple event store and let $r = (m,\vec{x},\vec{y})\in\mathscr{F}(\tilde{\Phi})$. Region r is minimal if and only if it is not a linear combination of two other regions, i.e.*

$$\nexists r' = (m',\vec{x}',\vec{y}'), r'' = (m'',\vec{x}'',\vec{y}'') \in \mathscr{F}(\tilde{\Phi})\left(m = m' + m'' \wedge \vec{x} = \vec{x}' + \vec{x}'' \wedge \vec{y} = \vec{y}' + \vec{y}''\right)$$

To exemplify the notion of minimal regions, and our preference in finding these types of regions, consider Figure 6.3. Observe that for region r_2, corresponding to place p_2, we have $\vec{x_2}(a) = \vec{y_2}(b) = 1$ and for region r_3, corresponding to place p_3, we have $\vec{x_3}(b) = \vec{y_3}(c) = 1$. Furthermore, observe that the region $r_{2,3}$, representing place $p_{2,3}$, is a non-negative linear combination of the former two regions, i.e. $\vec{x_{2,3}} = \vec{x_2} + \vec{x_3}$ and $\vec{y_{2,3}} = \vec{y_2} + \vec{y_3}$. From a process discovery perspective, we prefer to find the places p_2 and p_3 as they, when viewed in isolation,

[4]We deem this region the *optimal region* to avoid confusion with respect to the well understood concept of minimal regions, cf. 6.4.

[5]Observe that the contents presented here origin from [133], in which we prove the theorems provided here for the class of *hybrid variables*.

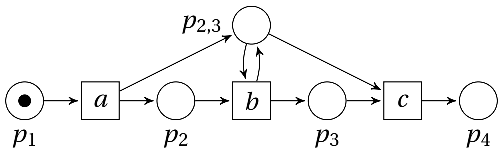


Figure 6.3: A labelled Petri net, where place $p_{2,3}$ represents a non-negative linear combination of places p_2 and p_3.

describe less behaviour and are thus more restrictive. Moreover, after adding places p_2 and p_3, place $p_{2,3}$ does not further constrain the behaviour of the Petri net.

We, therefore, propose a generalized *prefix-closure-based* objective function that incorporates a strictly positive trace-level scaling function β with signature $\beta \colon \overline{\tilde{\Phi}_+} \to \mathbb{R}_{>0}$, which guarantees to find such minimal regions. The scaling function β is required to map all sequences in the prefix-closure of the event store to some positive real value. The actual implementation is up to the user, although we present an instantiation of β that works well for process discovery. We show that the proposed generalized weighted prefix-closure-based objective function favours minimal regions, given any scaling function β.

Definition 6.5 (Generalized weighted prefix-closure-based objective function)**.** *Let $\tilde{\Phi} \in \mathscr{B}(\mathscr{A}^*)$ be a simple event store. Let $r = (m, \vec{x}, \vec{y}) \in \mathscr{F}(\tilde{\Phi})$ be a region and let $\beta \colon \overline{\tilde{\Phi}_+} \to \mathbb{R}_{>0}$ be a scaling function over $\overline{\tilde{\Phi}_+}$. We instantiate the generalized weighted prefix-closure-based objective function as $c_m = \sum_{\sigma \in \overline{\tilde{\Phi}}} \beta(\sigma)$, $\vec{c_x} = \sum_{\sigma \in \overline{\tilde{\Phi}}} \beta(\sigma)\vec{\sigma}$ and $\vec{c_y} = -\vec{c_x}$, i.e.*

$$z_\beta(r) = \sum_{\sigma \in \overline{\tilde{\Phi}_+}} \beta(\sigma)(m + \vec{\sigma}^\intercal(\vec{x} - \vec{y})) \tag{6.3}$$

Note that, if we let $\beta(\sigma) = 1$ for all $\sigma \in \overline{\tilde{\Phi}_+}$, which we denote as z_1, we instantiate the generalized objective function as the objective function proposed in [123].

To relate the behaviour in an event store to the objective function defined in 6.5, we instantiate the scaling function β making use of the frequencies of the traces present in the event store, i.e. we let $\beta(\sigma) = \overline{\tilde{\Phi}}(\sigma)$ leading to:

$$z_{\overline{\tilde{\Phi}}}(r) = \sum_{\sigma \in \overline{\tilde{\Phi}_+}} \overline{\tilde{\Phi}}(\sigma)(m + \vec{\sigma}^\intercal(\vec{x} - \vec{y})) \tag{6.4}$$

To assess the difference between z_1 and the alternative objective function $z_{\overline{\tilde{\Phi}}}$, consider the Petri net depicted in Figure 6.4. Assume we are given a simple event store $\tilde{\Phi} = [\langle a,b,d \rangle^5, \langle a,c,d \rangle^3]$. Let r_1 denote the region corresponding to place p_1, let r_2 correspond to p_2 and let r_3 correspond to p_3. In this case we have $z_1(r_1) = 1$, $z_1(r_2) = 1$ and $z_1(r_3) = 2$. For the alternative objective function instantiation, we have $z_{\overline{\tilde{\Phi}}}(r_1) = z_{\overline{\tilde{\Phi}}}(r_2) = z_{\overline{\tilde{\Phi}}}(r_3) = 8$. Thus, using $z_{\overline{\tilde{\Phi}}}$ leads to more intuitive objective values compared to using z_1 as $z_{\overline{\tilde{\Phi}}}$ evaluates to the absolute number of discrete time-steps a token remains in the corresponding place when replaying the event store $\tilde{\Phi}$ with respect to the place.

In [123] it is shown that objective function z_1 favours minimal regions. However, it does not provide a means to show that any arbitrary instantiation of z_β favours minimal regions. We, therefore, show that any instantiation of the generalized weighted prefix-closure-based objective

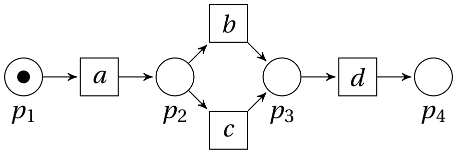


Figure 6.4: A simple labelled Petri net N, with $\mathfrak{L}_\Sigma(N,[p_1]) = \{\epsilon, \langle a\rangle, \langle a,b\rangle, \langle a,c\rangle \langle a,b,d\rangle, \langle a,c,d\rangle\}$.

function with an arbitrary scaling function $\beta\colon \overline{\tilde{\Phi}_+} \to \mathbb{R}_{>0}$ favours minimal regions. We first show that the objective value of a non-minimal region equals the sum of the two minimal regions defining it, after which we show that the given objective function maps each region to some positive real value, i.e. $rng(z_\beta) \subseteq \mathbb{R}_{>0}$.

Lemma 6.1 (Objective value composition of non-minimal regions). *Let $\tilde{\Phi} \in \mathscr{B}(\mathscr{A}^*)$ be a simple event store and let $r_1 = (m_1, \vec{x}_1, \vec{y}_1)$, $r_2 = (m_2, \vec{x}_1, \vec{y}_2)$ and $r_3 = (m_1 + m_2, \vec{x}_1 + \vec{x}_2, \vec{y}_1 + \vec{y}_2)$ with $r_1, r_2, r_3 \in \mathscr{F}(\tilde{\Phi})$. Let $z_\beta\colon \mathscr{F}(\tilde{\Phi}) \to \mathbb{R}$ where z_β is an instantiation of the generalized weighted objective function as defined in 6.5, then $z_\beta(r_3) = z_\beta(r_1) + z_\beta(r_2)$.*
Proof *(By definition of z_β)*
Let us denote $z_\beta(r_3)$:

$$\sum_{\sigma \in \overline{\tilde{\Phi}_+}} \beta(\sigma)\left((m_1 + m_2) + \vec{\sigma}^\intercal((\vec{x}_1 + \vec{x}_2) - (\vec{y}_1 + \vec{y}_2))\right)$$

$$\sum_{\sigma \in \overline{\tilde{\Phi}_+}} \beta(\sigma)\left(m_1 + \vec{\sigma}^\intercal(\vec{x}_1 - \vec{y}_1) + m_2 + \vec{\sigma}^\intercal(\vec{x}_2 - \vec{y}_2)\right)$$

$$\sum_{\sigma \in \overline{\tilde{\Phi}_+}} \beta(\sigma)\left(m_1 + \vec{\sigma}^\intercal(\vec{x}_1 - \vec{y}_1)\right) + \sum_{\sigma \in \overline{\tilde{\Phi}_+}} \beta(\sigma)\left(m_2 + \vec{\sigma}^\intercal(\vec{x}_2 - \vec{y}_2)\right)$$

Hence, $z_\beta(r_3) = z_\beta(r_1) + z_\beta(r_2)$. □

6.1 shows that the value of z_β for a non-minimal region equals the sum of the z_β values of the two regions it is composed of. If we additionally show that z_β can only evaluate to positive values, we show that z_β favours minimal regions.

Lemma 6.2 (Range of z_β is strictly positive). *Let $\tilde{\Phi} \in \mathscr{B}(\mathscr{A}^*)$ be a simple event store and let $r = (m, \vec{x}, \vec{y}) \in \mathscr{F}(\tilde{\Phi})$ be a non-trivial region. If z_β is an instantiation of the generalized weighted objective function as defined in 6.5, then it is characterized by $z_\beta : \mathscr{F}(\tilde{\Phi}) \to \mathbb{R}_{>0}$.*
Proof *(By definition of regions combined with z_β)*
Let $r = (m, \vec{x}, \vec{y})$ be a non-trivial region, i.e. $r \in \mathscr{F}(\tilde{\Phi})$. As r is a region, we have:

$$\forall \sigma = \sigma' \cdot \langle a \rangle \in \overline{\tilde{\Phi}} \left(m + \vec{\sigma'}^\intercal \vec{x} - \vec{\sigma}^\intercal \vec{y} \geq 0 \right) \tag{6.5}$$

Furthermore, observe that, given $\sigma = \sigma' \cdot \langle a \rangle$, we have $\vec{\sigma}(a') \geq \vec{\sigma'}(a')$, $\forall a' \in \mathscr{A}$, i.e. the Parikh value of any arbitrary activity in a sequence exceeds/is equal to the corresponding value in the prefix of the sequence. Thus, given $\tilde{\Phi}$, we have:

$$\forall \sigma = \sigma' \cdot \langle a \rangle \in \overline{\tilde{\Phi}}, a' \in \mathscr{A} \left(\vec{\sigma}(a') \geq \vec{\sigma'}(a') \right) \tag{6.6}$$

Using Equation 6.6, we substitute $\vec{\sigma'}^{\intercal}\vec{x}$ with $\vec{\sigma}^{\intercal}\vec{x}$ in Equation 6.5:

$$\forall \sigma = \sigma' \cdot \langle a \rangle \in \overline{\tilde{\Phi}} \left(m + \vec{\sigma}^{\intercal}(\vec{x} - \vec{y}) \geq 0 \right) \tag{6.7}$$

Combining $rng(\beta) \subseteq \mathbb{R}_{>0}$, $\vec{p}(\epsilon) = \vec{0}$ and $m \in \mathbb{N}$ with Equation 6.7, we find $z_{\beta}(r) \geq 0$:

$$\sum_{\sigma \in \overline{\tilde{\Phi}}} \beta(\sigma) \left(m + \vec{\sigma}^{\intercal}(\vec{x} - \vec{y}) \right) \geq 0 \tag{6.8}$$

If $m > 0$ then $\beta(\epsilon)(m + \vec{p}(\epsilon)^{\intercal}(\vec{x} - \vec{y})) = \beta(\epsilon)m$. As $\beta(\epsilon)m > 0$, this, together with Equation 6.7 and Equation 6.8 leads to $z_{\beta}(r) > 0$.

Observe that if $m = 0$ then for r to be a non-trivial *region, i.e. $r \in \mathscr{F}(\tilde{\Phi})$, then $\vec{x} \neq \vec{0}$, i.e. $\exists a \in \mathscr{A}$ s.t. $\vec{x}(a) > 0$. Thus, given $m = 0$, let $a \in \mathscr{A}$ s.t. $\vec{x}(a) > 0$. We know $\exists \sigma = \sigma' \cdot \langle a \rangle \in \overline{\tilde{\Phi}}$ and thus we have $\vec{\sigma}^{\intercal}\vec{x} = \vec{\sigma'}^{\intercal}\vec{x} + \vec{x}(a)$ and consequently $\vec{\sigma'}^{\intercal}\vec{x} = \vec{\sigma}^{\intercal}\vec{x} - \vec{x}(a)$. From Equation 6.5, we deduce:*

$$m + \vec{\sigma'}^{\intercal}\vec{x} - \vec{\sigma}^{\intercal}\vec{y} \geq 0$$

$$m + \vec{\sigma}^{\intercal}\vec{x} - \vec{x}(a) - \vec{\sigma}^{\intercal}\vec{y} \geq 0$$

$$m + \vec{\sigma}^{\intercal}\vec{x} - \vec{\sigma}^{\intercal}\vec{y} \geq \vec{x}(a)$$

$$m + \vec{\sigma}^{\intercal}(\vec{x} - \vec{y}) > 0$$

This, together with Equation 6.7 and Equation 6.8 again leads to $z_{\beta}(r) > 0$. □

By combining 6.1 and 6.2, we easily show that any instantiation of z_{β} favours minimal regions.

Theorem 6.1 (Any instantiation of z_{β} favours minimal regions). *Let $\tilde{\Phi} \in \mathscr{B}(\mathscr{A}^*)$ be a simple event store and let $r_1 = (m_1, \vec{x}_1, \vec{y}_1)$, $r_2 = (m_2, \vec{x}_1, \vec{y}_2)$ and $r_3 = (m_1 + m_2, \vec{x}_1 + \vec{x}_2, \vec{y}_1 + \vec{y}_2)$ with $r_1, r_2, r_3 \in \mathscr{F}(\tilde{\Phi})$. For any $z_{\beta} : \mathscr{F}(\tilde{\Phi}) \to \mathbb{R}$, being an instantiation of the generalized weighted objective function as defined in 6.5, we have $z_{\beta}(r_3) > z_{\beta}(r_1)$ and $z_{\beta}(r_3) > z_{\beta}(r_2)$.*
Proof *(By composition of 6.1 and 6.2)*
By 6.1 we know $z_{\beta}(r_3) = z_{\beta}(r_1) + z_{\beta}(r_2)$. By 6.2 we know that $z_{\beta}(r_1) > 0$, $z_{\beta}(r_2) > 0$ and $z_{\beta}(r_3) > 0$. Thus we deduce $z_{\beta}(r_3) > z_{\beta}(r_1)$ and $z_{\beta}(r_3) > z_{\beta}(r_2)$. Consequently, any instantiation of the objective function as defined in 6.5 favours minimal regions. □

Both objective functions presented, i.e. z_1 and $z_{\overline{\tilde{\Phi}}}$, are expressible in terms of the more general objective function z_{β} as presented in 6.5. As we have shown, the two objective functions potentially favour different, yet minimal, regions. Combining an objective function with the ILP-formulation presented in 6.3 establishes means to find Petri net places. However, solving one ILP only yields one solution and hence we need means to find a set of places, which together form a Petri net that accurately describes the input event store $\tilde{\Phi}$. In the next section, we therefore present means to find multiple regions, which we prove to together form a *relaxed sound workflow net*.

6.3 Discovering Relaxed Sound Workflow Nets

Using the basic ILP-formulation with an instantiation of the generalized weighted objective function only yields one, optimal, result. However, we are interested in finding multiple places that together form a workflow net. Therefore, in this section, we describe to effectively find multiple places that together constitute to a workflow net. We start by illustrating the causal-relation-based discovery strategy, after which we present the conditions necessary to discovery workflow nets. We furthermore characterize the behavioural guarantees with respect to the discovered workflow nets.

6.3.1 Discovering Multiple Places Based on Causal Relations

In [123] multiple approaches are presented to find multiple, different Petri net places. Here we adopt and generalize, the *causal approach*. One of the most suitable techniques to find multiple regions in a controlled and structured manner, is by exploiting causal relations present within an event store. A causal relation between activities a and b implies that activity a causes b, i.e. b is likely to follow (somewhere) after activity a. Several approaches exist to compute causalities [49]. For example, in case of the Alpha miner [11], the directly follows relation, cf. 2.16, is used to derive a causal relation, i.e. a causal relation from activity a to activity b, $a \rightarrow b$ holds, if $a > b$ and not $b > a$. As indicated, within the heuristic miner [121, 122], this relation was further developed to take frequencies into account as well. Given these multiple definitions, we assume the existence of a *causal relation oracle* which, given an event store, produces a set of pairs (a,b) indicating that activity a may have a causal relation with (to) activity b.

Definition 6.6 (Causal relation oracle). *A causal oracle ϱ maps a simple event store to a set of activity pairs, i.e. $\varrho : \mathscr{B}(\mathscr{A}^*) \rightarrow \mathscr{P}(\mathscr{A} \times \mathscr{A})$.*

A causal oracle only considers activities present in an event store, i.e. $\varrho(\tilde{\Phi}) \in \mathscr{P}(A_{\tilde{\Phi}} \times A_{\tilde{\Phi}})$. It defines a directed graph with $A_{\tilde{\Phi}}$ as vertices and each pair $(a,b) \in \varrho(\tilde{\Phi})$ as an arc between a and b. Later we exploit the graph-based view, for now, we refer to $\varrho(\tilde{\Phi})$ as a collection of pairs. When adopting a causal ILP process discovery strategy, we try to find net places that represent a causality found in the event store. Given an event store $\tilde{\Phi}$, for each pair $(a,b) \in \varrho(\tilde{\Phi})$ we enrich the constraint body of the process discovery ILP-formulation, cf. 6.3, with three constraints [123]:

1. $m = 0$;

   The place representing the causal relation is not initially marked.
2. $\vec{x}(a) = 1$;

   The place representing the causal relation always has an incoming arc from the transition with label a.
3. $\vec{y}(b) = 1$;

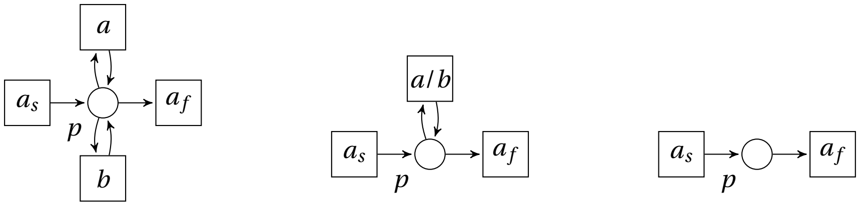
   The place representing the causal relation always has an outgoing arc to the transition with label b.

The three constraints ensure that if we find a solution to the ILP, it corresponds to a place which is not marked and connects transition a to transition b. Given pair $(a,b) \in \varrho(\tilde{\Phi})$ we denote the corresponding extended causality based ILP-formulation as $ILP_{(\tilde{\Phi}, a \rightarrow b)}$.

After solving $ILP_{(\tilde{\Phi}, a \rightarrow b)}$ for each $(a,b) \in \varrho(\tilde{\Phi})$, we end up with a set of regions that we are able to transform into places in a resulting Petri net. Since we enforce $m = 0$ for each causality, none of these places is initially marked. Moreover, due to constraints based on $m\vec{1} + \mathbf{M}_{\tilde{\Phi}}(\vec{x} - \vec{y}) = \vec{0}$, the resulting place is empty after replaying each trace in the input event store within the resulting Petri net. Since we additionally enforce $\vec{x}(a) = 1$ and $\vec{y}(b) = 1$, if we find a solution to the ILP, the corresponding place has both input and output arcs and is not eligible for being a source/sink place. Hence, the approach as-is does not allow us to find workflow nets. In the next section, we show that a simple pre-processing step performed on the event store, together with specific instances of $\varrho(\tilde{\Phi})$, allows us to discover workflow nets which are relaxed sound.

6.3.2 Discovering Workflow Nets

Recall that for a Petri net to be a workflow net, cf. 2.3, it has to have a unique source/sink place, and, each element within the Petri net needs to be on a path from the start place to the sink



(a) Solution in case $a \neq a_s$ and $b \neq a_f$. (b) Solution in case $a = a_s$ and $b \neq a_f$ or $a \neq a_s$ and $b = a_f$. (c) Solution in case $a = a_s$ and $b = a_f$.

Figure 6.5: Visualizations of trivial solutions to $ILP_{(f_{use}(\tilde{\Phi}),a \rightarrow b)}$ in terms of Petri net places.

place. Given an arbitrary simple event store, it is rather hard to determine how to construct such sink/source place, i.e. multiple activities usually happen at the start/end of a trace. Moreover, such activities are potentially repeated throughout a trace, which doesn't allow us to enable them only once at the beginning of a case. However, in case there exists a well-defined unique start activity a_s and end-activity a_e, we are guaranteed that each instance of the process is, when projected onto the control-flow perspective, is of the form $\langle a_s, ..., a_e \rangle$.

Clearly, not every (simple) event store describes event data of the form as described earlier, i.e. not every process instance defines a unique start/end activity. We therefore first define a class of event stores which consists of a unique start/end activity, cf. 6.7, after which we describe a (trivial) projection of arbitrary event stores into such event store.

Definition 6.7 (Unique start/end event store). *Let $\tilde{\Phi} \in \mathscr{B}(\mathscr{A}^*)$ be a simple event store with a corresponding set of activities $A_{\tilde{\Phi}}$. $\tilde{\Phi}$ is a **U**nique **S**tart/**E**nd event store (USE-store) if there exist $a_s, a_f \in A_{\tilde{\Phi}}$ s.t. $a_s \neq a_f$ and:*

$$\forall \sigma \in \tilde{\Phi}\, (\sigma(1) = a_s \wedge \forall i \in \{2, 3, ..., |\sigma|\} (\sigma(i) \neq a_s)) \tag{6.9}$$

$$\forall \sigma \in \tilde{\Phi} \Big(\sigma(|\sigma|) = a_f \wedge \forall i \in \{1, 2, ..., |\sigma| - 1\} (\sigma(i) \neq a_f) \Big) \tag{6.10}$$

Since the set of activities $A_{\tilde{\Phi}}$ is finite, it is trivial to transform any event store to a USE-store. Assume we have an event store $\tilde{\Phi}$ over $A_{\tilde{\Phi}}$ that is not a USE-store. We generate two "fresh" activities $a_s, a_f \in \mathscr{A}$ s.t. $a_s, a_f \notin A_{\tilde{\Phi}}$ and create a new event store $\tilde{\Phi}' \in \mathscr{B}(\mathscr{A}^*)$ over $A_{\tilde{\Phi}} \cup \{a_s, a_f\}$, by adding $\langle a_s \rangle \cdot \sigma \cdot \langle a_f \rangle$ to $\tilde{\Phi}'$ for each $\sigma \in \tilde{\Phi}$, i.e.

$$\tilde{\Phi}' = \Big[\sigma^k \in \mathscr{A}^* \mid \exists \sigma' \in_+ \tilde{\Phi} \Big(\sigma = \langle a_s \rangle \cdot \sigma' \cdot \langle a_f \rangle \wedge \tilde{\Phi}(\sigma') = k \Big) \Big] \tag{6.11}$$

We let $f_{use} : \mathscr{B}(\mathscr{A}^*) \rightarrow \mathscr{B}(\mathscr{A}^*)$ denote such USE-transformation. We omit a_s and a_f from the domain of f_{use} and assume that given some USE-transformation the two symbols are known.

Clearly, after applying a USE-transformation, finding a unique source- and sink place is trivial. It also provides an additional advantage considering the ability to find workflow nets. In fact, an instance $ILP_{(\tilde{\Phi},a \rightarrow b)}$ always has a solution if $\tilde{\Phi}$ is a USE-store. We provide the proof of this property in 6.3, after which we present an algorithm that, given specific instantiations of ϱ, is guaranteed to discover a workflow net.

Lemma 6.3 (A USE-store based causality has a solution). *Let $\tilde{\Phi} \in \mathscr{B}(\mathscr{A}^*)$ be a USE-store, cf. 6.7, with corresponding set of activities $A_{\tilde{\Phi}}$, unique start activity a_s and unique end activity a_f. For*

every $(a,b) \in \varrho(\tilde{\Phi})$ with $a \neq a_f$ and $b \neq a_s$, $ILP_{(\tilde{\Phi}, a \to b)}$ has a solution.
***Proof** (By construction)*
We consider the case $a \neq a_s$ and $b \neq a_f$. We show that variable assignment $\vec{x}(a_s) = \vec{x}(a) = \vec{x}(b) = \vec{y}(a) = \vec{y}(b) = \vec{y}(a_f) = 1$, all other variables 0 (Figure 6.5a), adheres to all constraints of $ILP_{(\tilde{\Phi}, a \to b)}$.

First of all, observe that the constraints of the form: $\vec{1}^\intercal \vec{x} + \vec{1}^\intercal \vec{y} \geq 1$, $\vec{0} \leq \vec{x} \leq \vec{1}$, $\vec{0} \leq \vec{y} \leq \vec{1}$ and $0 \leq m \leq 1$ are trivially satisfied. Moreover, the variable assignment indeed represents the causal relation between a and b, i.e. it adheres to $m = 0$, $\vec{x}(a) = 1$ and $\vec{y}(b) = 1$.

Consider constraints related to the notion of a region, i.e. constraints of the form

$$\forall \sigma = \sigma' \cdot \langle a' \rangle \in \overline{\tilde{\Phi}_+} (m + \vec{p}(\sigma')^\intercal \vec{x} - \vec{p}(\sigma)^\intercal \vec{y} \geq 0)$$

(corresponding to $m\vec{1} + \mathbf{M}'\vec{x} - \mathbf{M}\vec{y} \geq \vec{0}$) and let $\sigma = \sigma' \cdot \langle a' \rangle \in \overline{\tilde{\Phi}_+}$.

Case I: *$a' \neq a, a' \neq b, a' \neq a_f$. Since $a' \neq a_f$ we know $\vec{\sigma}(a_f) = 0$. Moreover, since $a' \neq a, a' \neq b$, we know that $\vec{\sigma'}(a) = \vec{\sigma}(a)$ and $\vec{\sigma'}(b) = \vec{\sigma}(b)$, and hence $\vec{\sigma'}(a)\vec{x}(a) - \vec{\sigma}(a)\vec{y}(a) = 0$ and $\vec{\sigma'}(b)\vec{x}(b) - \vec{\sigma}(b)\vec{y}(b) = 0$. Since $\vec{x}(a_s) = 1$ and a_s occurs uniquely at the start of each trace, if $\sigma' = \epsilon$ such constraint equals 0, and, 1 otherwise.*

Case II: *$a' = a$. We know $\vec{\sigma}(a_f) = 0$ and $\vec{\sigma'}(b) = \vec{\sigma}(b)$. Now $\vec{\sigma'}(a) = \vec{\sigma}(a) - 1$ and thus $\vec{\sigma'}(a)\vec{x}(a) - \vec{\sigma}(a)\vec{y}(a) = -1$. Since $a_s \in \sigma'$ we have $\vec{\sigma'}(a_s)\vec{x}(a_s) = 1$, and thus the constraint equals 0.*

Case III: *$a' = b$ Similar to* Case II.

Case IV: *$a' = a_f$. We again have $\vec{\sigma'}(a) = \vec{\sigma}(a)$ and $\vec{\sigma'}(b) = \vec{\sigma}(b)$. Since $\vec{\sigma}(a_f)\vec{y}(a_f) = \vec{\sigma'}(a_s)\vec{x}(a_s) = 1$, each constraint equals 0.*

For all constraints of the form $\forall \sigma \in f_{use}(\tilde{\Phi})(m + \vec{\sigma}^\intercal(\vec{x} - \vec{y}) = 0)$ (corresponding to $m\vec{1} + \mathbf{M}_{\tilde{\Phi}}(\vec{x} - \vec{y}) = \vec{0}$), we observe that these all correspond to Case IV. As we have seen in Case IV, all constraints of the form $m + \vec{p}(\sigma')^\intercal \vec{x} - \vec{p}(\sigma)^\intercal \vec{y}$ are equal to 0. In this case, the constraints are of the form $m + \vec{\sigma}^\intercal(\vec{x} - \vec{y})$, however, since $\vec{x}(a_f) = 0$, we again deduce that indeed for the given variable assignment these constraints equal 0.

In case we have $a = a_s$ and $b \neq a_f$ the region $\vec{x}(a_s) = \vec{x}(b) = \vec{y}(b) = \vec{y}(a_f) = 1$, all other variables 0 (Figure 6.5b), is a solution. The proof is similar to the proof of the previous case.

In case we have $a \neq a_s$ and $b = a_f$ the region $\vec{x}(a_s) = \vec{x}(a) = \vec{y}(a) = \vec{y}(a_f) = 1$, all other variables 0 (Figure 6.5b), is a solution. Again the proof is similar to the proof in the first case.

Finally in case we have $a = a_s$ and $b = a_f$ the region $\vec{x}(a_s) = \vec{y}(a_f) = 1$, all other variables 0 (Figure 6.5c), is a solution. Again the proof is similar to the proof in the first case. □

In algorithm 6.1, on page 157, we present an ILP-Based process discovery approach that uses a USE-store internally in order to find multiple Petri net places. For every $(a,b) \in \varrho(f_{use}(\tilde{\Phi}))$ with $a \neq a_f$ and $b \neq a_s$ it solves $ILP_{(f_{use}(\tilde{\Phi}), a \to b)}$. Moreover, it finds a unique source and sink place.

The algorithm constructs an initially empty labelled Petri net $N = (P, T, F, \lambda)$. Subsequently for each $a \in A_{\tilde{\Phi}} \cup \{a_s, a_f\}$ a transition t_a is added to T, with $\lambda(t_a) = a$. For each causal pair in the *USE-variant of input event store* $\tilde{\Phi}$, a place $p_{(a,b)}$ is discovered by solving $ILP_{(f_{use}(\tilde{\Phi}), a \to b)}$ after which P and F are updated accordingly. The algorithm adds an initial place p_i and connects it to t_{a_s} and similarly creates sink place p_o which is connected to t_{a_f}. For transition t_a related to $a \in A_{\tilde{\Phi}}$, we have $\lambda(t_a) = a$, whereas $\lambda(t_{a_s}) = \lambda(t_{a_f}) = \tau$.

The algorithm is guaranteed to always find a solution to $ILP_{(f_{use}(\tilde{\Phi}), a \to b)}$, hence for each causal relation a place is found. Additionally, a unique source and sink place are constructed. However, the algorithm does not guarantee that we find a connected component, i.e. requirement 3 of 2.3. In fact, the nature of ϱ determines whether or not we discover a workflow net. In Theorem 6.2 we characterize this nature and prove, by exploiting 6.3, that we are able to discover workflow nets.

Algorithm 6.1: `ILP-Based Process Discovery`

```
input  : $\tilde{\Phi} \in \mathscr{B}(\mathscr{A}^*)$, $\varrho\colon \mathscr{B}(\mathscr{A}^*) \rightarrow \mathscr{P}(\mathscr{A} \times \mathscr{A})$
output : $N = (P, T, F, \lambda)$
begin
    $P, T, F \leftarrow \emptyset$;
    let $a_s, a_f \notin A_{\tilde{\Phi}}$;
    $T \leftarrow \{t_a \mid a \in A_{\tilde{\Phi}} \cup \{a_s, a_f\}\}$;
    foreach $(a, b) \in \varrho(f_{use}(\tilde{\Phi}))$ do
        if $a \neq a_f \wedge b \neq a_s$ then
            $(m, \vec{x}, \vec{y}) \leftarrow$ solution to $ILP_{(f_{use}(\tilde{\Phi}), a \rightarrow b)}$;
            let $p_{(a,b)} \notin P$;
            $P \leftarrow P \cup p_{(a,b)}$;
            foreach $a' \in A_{\tilde{\Phi}} \cup \{a_s, a_f\}$ do
                if $\vec{x}(a') = 1$ then
                    $F \leftarrow F \cup \{(t_{a'}, p_{(a,b)})\}$;
                if $\vec{y}(a') = 1$ then
                    $F \leftarrow F \cup \{(p_{(a,b)}, t_{a'})\}$;
    let $p_i, p_o \notin P$;
    $P \leftarrow P \cup \{p_i, p_o\}$;
    $F \leftarrow F \cup \{(p_i, t_{a_s})\}$;
    $F \leftarrow F \cup \{(t_{a_f}, p_o)\}$;
    let $\lambda\colon T \rightarrow \mathscr{A} \cup \{\tau\}$;
    foreach $a \in A_{\tilde{\Phi}}$ do
        $\lambda(t_a) \leftarrow a$;
    $\lambda(t_{a_s}), \lambda(t_{a_f}) \leftarrow \tau$;
    return $(P, T, F, \lambda)$;
```

Theorem 6.2 (There exist sufficient conditions for finding workflow nets). *Let $\tilde{\Phi} \in \mathscr{B}(\mathscr{A}^*)$ be a simple event store with a corresponding set of activities $A_{\tilde{\Phi}}$. Let $f_{use}\colon \mathscr{B}(\mathscr{A}^*) \rightarrow \mathscr{B}(\mathscr{A}^*)$ denote a USE-transformation function. Let $a_s, a_f \in \mathscr{A}$ denote the unique start and end activity implicitly defined by $f_{use}(\tilde{\Phi})$. Let $\varrho\colon \mathscr{B}(\mathscr{A}^*) \rightarrow \mathscr{P}(\mathscr{A} \times \mathscr{A})$ be a causal oracle and consider $\varrho(f_{use}(\tilde{\Phi}))$ as a* directed graph. *If each $a \in A_{\tilde{\Phi}}$ is on a path from a_s to a_f in $\varrho(f_{use}(\tilde{\Phi}))$, and there is no path from a_s to itself, nor a path from a_f to itself, then `ILP-Based Process Discovery`$(\tilde{\Phi}, \varrho)$ returns a workflow net.*

Proof *(On the structure of the causal relation)*
By the nature of $\varrho(f_{use}(\tilde{\Phi}))$, i.e. each $a \in A_{\tilde{\Phi}}$ is on a path from a_s to a_f, combined with 6.3, we know that for each $(a, b) \in \varrho(f_{use}(\tilde{\Phi}))$ a corresponding place is found that has a transition labelled with a as an input and a transition labelled b as an output. Hence, every path in $\varrho(f_{use}(\tilde{\Phi}))$ corresponds to a path in the resulting net and as a consequence, every transition is on a path from a_s to a_f. As every place that is added has input transition ($\vec{x}(a) = 1$) and an output transition ($\vec{y}(b) = 1$), every place is also on a path from a_s to a_f. By construction this then also holds from p_i to p_o. □

Theorem 6.2 proves that if we use a causal structure that, when interpreting it as a graph, has the property that each $a \in A_{\tilde{\Phi}}$ is on a path from a_s to a_f, the result of algorithm 6.1 is a workflow net. The aforementioned property seems rather strict. However, there exists a causal graph definition that, in fact, guarantees this property [122]. Hence, we are able to use this definition as an instantiation for ϱ. As a consequence, we are always able to discover workflow nets, given an arbitrary event log/store.

Theorem 6.2 does not provide any behavioural guarantees, i.e. a workflow net is a purely graph-theoretical property. Recall that the premise of a region is that it does not block the execution of any sequence within the prefix-closure of an event store. Intuitively we deduce that we are therefore able to fire each transition in the workflow net at least once. Moreover, since we know that a_f is the final transition of each sequence in $f_{use}(\tilde{\Phi})$, and after firing the transition each place based on any $ILP_{(f_{use}(\tilde{\Phi}), a \to b)}$ is empty, we know that we are able to mark p_o, and, p_o is the only place containing a token at that point in time. These two observations hint on the fact that the workflow net is *relaxed sound*, which we prove in Theorem 6.3

Theorem 6.3. *Let $\tilde{\Phi} \in \mathscr{B}(\mathscr{A}^*)$ be a simple event store with a corresponding set of activities $A_{\tilde{\Phi}}$. Let $f_{use}\colon \mathscr{B}(\mathscr{A}^*) \to \mathscr{B}(\mathscr{A}^*)$ denote a USE-transformation function and let a_s, a_f denote the unique start and end activity implicitly defined by $f_{use}(\tilde{\Phi})$. Let $\varrho\colon \mathscr{B}(\mathscr{A}^*) \to \mathscr{P}(\mathscr{A} \times \mathscr{A})$ be a causal oracle. Let $N = (P, T, F, \lambda) =$ `ILP-Based Process Discovery`$(\tilde{\Phi}, \varrho)$. If N is a workflow net, then N is relaxed sound.*

Proof *(By construction of traces in the (simple) event store)*
Recall that a workflow net N is relaxed sound if and only if (2.6):

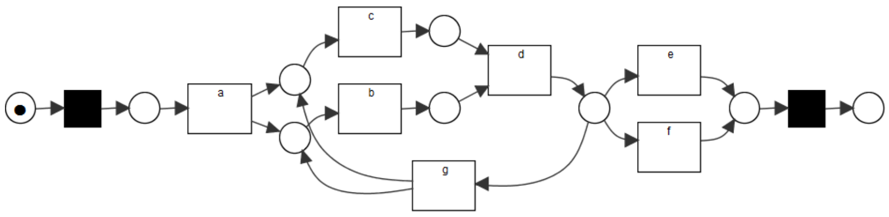
$$\forall t \in T \left(\exists M, M' \in \mathfrak{R}(N, [p_i]) \left((N, M)[t\rangle(N, M') \wedge [p_o] \in \mathfrak{R}(N, M')\right)\right)$$

Observe that t_{a_s} is trivially enabled in initial marking $[p_i]$ since $\bullet t_{a_s} = \{p_i\}$.

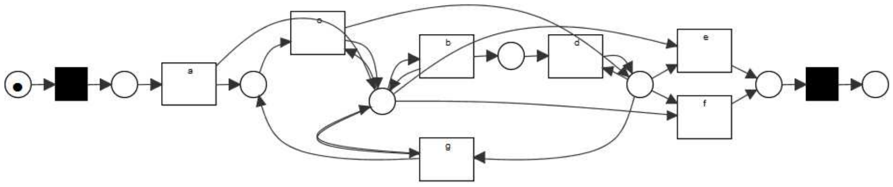
Consider an arbitrary $t \in T \setminus \{t_{a_s}, t_{a_f}\}$. We know $\exists \sigma \in_+ f_{use}(\tilde{\Phi}) \left(\sigma = \langle a_s\rangle \cdot \sigma' \cdot \langle \lambda(t)\rangle \cdot \sigma'' \cdot \langle a_f\rangle\right)$. Let $\langle t'_1, t'_2, ..., t'_n\rangle$ s.t. $\langle \lambda(t'_1), \lambda(t'_2), ..., \lambda(t'_n)\rangle = \sigma'$. The fact that each place $p \in P \setminus \{p_i, p_o\}$ corresponds to a region yields that we may deduce $[p_i] \xrightarrow{t_{a_s}} M'_1, M'_1 \xrightarrow{t'_1} M'_2, ..., M'_n \xrightarrow{t'_n} M'$ s.t. $M' \supseteq \bullet t$ (if there exists $p \in \bullet\, t$ s.t. $M'(p) = 0$, then p does not correspond to a region). Hence for any $t \in T \setminus \{t_{a_s}, t_{a_f}\}$ there exists a marking reachable from $[p_i]$ that enables t.

Now let $\langle t''_1, t''_2, ..., t''_n\rangle$ s.t. $\langle \lambda(t''_1), \lambda(t''_2), ..., \lambda(t''_n)\rangle = \sigma''$. Note that also, again by the fact that each place $p \in P \setminus \{p_i, p_o\}$ corresponds to a region, we may deduce $M' \xrightarrow{t''_1} M''_1$, $M''_1 \xrightarrow{t''_2} M''_2, ..., M''_{n-1} \xrightarrow{t''_n} M''_n$. Clearly, we have $M''_n \xrightarrow{t_{a_f}} M_f$ with $M_f(p_o) = 1$ since $t_{a_f}\bullet = \{p_o\}$, and this is the first time we fire t_{a_f}, i.e. $a_f \notin^* \langle a_s\rangle \cdot \sigma' \cdot \langle \lambda(t)\rangle \cdot \sigma''$. Clearly $M_f(p_i) = 0$ and because of constraints of the form $m\vec{1} + \mathbf{M}_{\tilde{\Phi}}(\vec{x} - \vec{y}) = \vec{0}$ we have $\forall p \in P \setminus \{p_i, p_o\}(M_f(p) = 0)$. Hence $M_f = [p_o]$ and thus after firing t there exists a firing sequence that leads to marking $[p_o]$ which proves that N is relaxed sound. □

We have shown that with a few pre- and post-processing steps and a specific class of causal structures we are able to guarantee to find workflow nets that are relaxed sound. These results are interesting since several process mining techniques require workflow nets as an input. The ILP problems solved still require their solutions to allow for all possible behaviour in the event store. As a result, the algorithm incorporates all infrequent exceptional behaviour and still results in over-fitting complex workflow nets. Hence, in the upcoming section we show how to efficiently prune the ILP constraint body to identify and eliminate infrequent exceptional behaviour.



(a) Result based on event store $\tilde{\Phi}_1$.



(b) Result based on event store $\tilde{\Phi}'_1$.

Figure 6.6: Results of applying algorithm 6.1 based on $\tilde{\Phi}_1$ and $\tilde{\Phi}'_1$.

6.4 Dealing with Infrequent Behaviour

In this section, we present an efficient pruning technique that identifies and eliminates constraints related to infrequent exceptional behaviour. We first present the impact of infrequent exceptional behaviour after which we present the pruning technique.

6.4.1 The Impact of Infrequent Exceptional Behaviour

In this section, we highlight the main cause of ILP-based discovery's inability to handle infrequent behaviour and we devise a filtering mechanism that exploits the nature of the underlying body of constraints. Consider for example the simple event store $\tilde{\Phi}_1$, which only contains traces that are part of the behaviour described by the running example net presented in Figure 2.13b:

$$\tilde{\Phi}_1 = \begin{bmatrix} \langle a,b,c,d,e\rangle^{10}, \\ \langle a,c,b,d,f\rangle^{9}, \\ \langle a,b,c,d,g,c,b,d,f\rangle^{12}, \\ \langle a,c,b,d,g,b,c,d,e\rangle^{11}, \\ \langle a,b,c,d,g,b,c,d,f\rangle^{13} \end{bmatrix}$$

When we apply the ILP-based process discovery algorithm as described in the previous sections, with a suitable causal relation, we obtain the *sound workflow net* depicted in Figure 6.6a. Observe that the model we obtain is equal, in terms of behaviour, to the simplified running example model of Figure 2.13b. In this case, however, we have an additional invisible start and end transition, due to the addition of a_s and a_f to each trace in the event store. If we create a simple event store $\tilde{\Phi}'_1$ by simply adding one instance of the trace $\langle a,b,b,c,d,d,e\rangle$, i.e. by duplicating the b and d activities, we obtain the Petri net depicted in Figure 6.6b.

Because of these two duplicated activities, some of the places found in Figure 6.6a are no longer a feasible solution to the ILP-formulation on the basis of $\tilde{\Phi}'_1$. Consider Table 6.1, in which

Table 6.1: Constraints based on trace $\langle a,b,b,c,d,d,e\rangle$ in event store $\tilde{\Phi}'_1$ (observe that we show the constraints based on $f_{use}(\tilde{\Phi}'_1)$).

(1) $m - \vec{y}(a_s) \geq 0$
(2) $m + \vec{x}(a_s) - \vec{y}(a_s) - \vec{y}(a) \geq 0$
(3) $m + \vec{x}(a_s) + \vec{x}(a) - \vec{y}(a_s) - \vec{y}(a) - \vec{y}(b) \geq 0$
(4) $m + \vec{x}(a_s) + \vec{x}(a) + \vec{x}(b) - \vec{y}(a_s) - \vec{y}(a) - 2\vec{y}(b) \geq 0$
(5) $m + \vec{x}(a_s) + \vec{x}(a) + 2\vec{x}(b) - \vec{y}(a_s) - \vec{y}(a) - 2\vec{y}(b) - \vec{y}(c) \geq 0$
(6) $m + \vec{x}(a_s) + \vec{x}(a) + 2\vec{x}(b) + \vec{x}(c) - \vec{y}(a_s) - \vec{y}(a) - 2\vec{y}(b) - \vec{y}(c) - \vec{y}(d) \geq 0$
(7) $m + \vec{x}(a_s) + \vec{x}(a) + 2\vec{x}(b) + \vec{x}(c) + \vec{x}(d) - \vec{y}(a_s) - \vec{y}(a) - 2\vec{y}(b) - \vec{y}(c) - 2\vec{y}(d) \geq 0$
(8) $m + \vec{x}(a_s) + \vec{x}(a) + 2\vec{x}(b) + \vec{x}(c) + 2\vec{x}(d) - \vec{y}(a_s) - \vec{y}(a) - 2\vec{y}(b) - \vec{y}(c) - 2\vec{y}(d) - \vec{y}(e) \geq 0$
(9) $m + \vec{x}(a_s) + \vec{x}(a) + 2\vec{x}(b) + \vec{x}(c) + 2\vec{x}(d) + \vec{x}(e) - \vec{y}(a_s) - \vec{y}(a) - 2\vec{y}(b) - \vec{y}(c) - 2\vec{y}(d) - \vec{y}(e) - \vec{y}(a_f) \geq 0$
(10) $m + \vec{x}(a_s) + \vec{x}(a) + 2\vec{x}(b) + \vec{x}(c) + 2\vec{x}(d) + \vec{x}(e) + \vec{x}(a_f) - \vec{y}(a_s) - \vec{y}(a) - 2\vec{y}(b) - \vec{y}(c) - 2\vec{y}(d) - \vec{y}(e) - \vec{y}(a_f) = 0$

we depict all constraints related to the simple trace $\langle a,b,b,c,d,d,e\rangle$. Moreover, consider the place $p_{(\{a,g\},\{b\})}$ with $\bullet p_{(\{a,g\},\{b\})} = \{a,g\}$ and $p_{(\{a,g\},\{b\})}\bullet = \{b\}$ in Figure 6.6a. We observe that constraints (4) to (10) evaluate to -1 for the variable assignment related to place $p_{(\{a,g\},\{b\})}$, i.e. $\vec{x}(a) = 1$, $\vec{x}(g) = 1$ and $\vec{y}(b) = 1$. Hence, we are not able to find the place after addition of simple trace $\langle a,b,b,c,d,d,e\rangle$. Moreover, note that the Petri net in Figure 6.6b contains a *token generator*, i.e. the place that is a selfloop of the transition with label b allowing for the production of any number of tokens on the input place of the transition with label d. Note that, similar problems occur for the place connecting the transition with label c to the transition with label d in Figure 6.6a.

The example shows that the addition of the simple trace $\langle a,b,b,c,d,d,e\rangle$ to $\tilde{\Phi}_1$, yields constraints that invalidate several places that are found in the workflow net in Figure 6.6a. As a result, the workflow net based on event store $\tilde{\Phi}'_1$ contains places with self-loops on the transitions labelled with b, c, d and g. In particular the self-loop on b is problematic, as firing the corresponding transition is unconstrained, as long as the token remains in the connected place, i.e. as long as we do not fire the transitions labelled with e or f. As indicated, this results in a token generator construct. In fact, the model in Figure 6.6a is a sound workflow net, whereas the model in Figure 6.6b, due to the presence of the token generator, is not, i.e. it is relaxed sound.

Due to the relative infrequency of trace $\langle a,b,b,c,d,d,e\rangle$ it is arguably acceptable to trade-off the perfect replay-fitness guarantee of ILP-based process discovery and return the workflow net of Figure 6.6a, given $\tilde{\Phi}'_1$. Hence, we need filtering techniques and/or trace clustering techniques in order to remove exceptional behaviour. However, apart from simple pre-processing, i.e. as presented in chapter 4 we aim at adapting the ILP-based process discovery approach itself to be able to cope with infrequent behaviour.

By manipulating the constraint body such that it no longer allows for all behaviour present in the input event store, we are able to deal with infrequent behaviour within event stores. Given the problems that arise because of the presence of exceptional traces, a natural next step is to leave out the constraints related to the problematic traces. An advantage of filtering the constraint body directly, i.e. rather than applying general-purpose filtering techniques such as the filtering technique we presented in chapter 4, is the fact that the constraints are based on the prefix-closure of the event store. Thus, even if all traces are unique yet they do share prefixes, we are able to filter. Moreover, the internal representation of the ILP-based process discovery algorithm, i.e. constraints, largely *ignores ordering relations*. Thus, even in the case of parallelism, some constraints at later phases of traces map onto the same constraints. Additionally, leaving out constraints decreases the size of the ILP's constraint body, which has a potential positive effect on the time needed to solve an ILP. We devise a graph-based filtering technique, i.e. *sequence encoding filtering*, that allows us to prune constraints based on trace frequency information. In this context, a sequence encoding is a formalization of the information present in traces required to construct region theory based constraints.

6.4.2 Sequence Encoding Graphs

As a first step towards sequence encoding filtering we define the relationship between sequences and constraints. We do this in terms of *sequence encodings*. A sequence encoding is a vector-based representation of a sequence in terms of region theory, i.e. representing the sequence's corresponding constraint.

Definition 6.8 (Sequence encoding). *Let $\tilde{\Phi} \in \mathcal{B}(\mathscr{A}^*)$ be a simple event store and let $A_{\tilde{\Phi}} \subseteq \mathscr{A}$ denote the corresponding set of activities that occur in the event store. $\vec{\phi}$: $A^*_{\tilde{\Phi}} \to \mathbb{N}_0^{2|A_{\tilde{\Phi}}|+1}$ denotes the sequence encoding function mapping every $\sigma \in A^*_{\tilde{\Phi}}$ to a $2 \cdot |A_{\tilde{\Phi}}|+1$-sized vector. We define $\vec{\phi}$ as:*

$$\vec{\phi}(\sigma' \cdot \langle a \rangle) = \begin{pmatrix} 1 \\ \vec{\sigma}' \\ -\vec{1}^\intercal \vec{\sigma' \cdot \langle a \rangle} \end{pmatrix} \quad \vec{\phi}(\epsilon) = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{6.12}$$

As an example of a sequence encoding vector consider sequence $\langle a_s, a, b \rangle$ originating from $\overline{f_{use}(\tilde{\Phi}'_1)}$, for which we have $\vec{\phi}(\langle a_s, a, b \rangle)^\intercal = (1,1,1,0,0,0,0,0,0,0,-1,-1,-1,0,0,0,0,0,0)$. Sequence encoding vectors directly correspond to region theory based constraints, e.g. if we are given $m \in \{0,1\}$ and $\vec{x}, \vec{y} \in \{0,1\}^{|A_{\tilde{\Phi}}|}$ and create a vector $\vec{r}$ where $\vec{r}(1) = m$, $\vec{r}(2) = \vec{x}(a_s)$, $\vec{r}(3) = \vec{x}(a)$, ..., $\vec{r}(9) = \vec{x}(g)$, $\vec{r}(10) = \vec{x}(a_f)$, $\vec{r}(11) = \vec{y}(a_s)$, ..., $\vec{r}(19) = \vec{y}(a_f)$, then $\vec{\phi}(\langle a_s, a, b \rangle)^\intercal \vec{r} = m + \vec{x}(a_s) + \vec{x}(a) - \vec{y}(a_s) - \vec{y}(a) - \vec{y}(b)$. As a compact notation for $\sigma = \sigma' \cdot \langle a \rangle$ we write $\vec{\phi}(\sigma)$ as a pair of the bag representation of the Parikh vector of σ' and a, i.e. $\vec{\phi}(\langle a_s, a, b \rangle)$ is written as $([a_s, a], b)$ whereas $\vec{\phi}(\langle a_s, a, b, c \rangle)$ is written as $([a_s, a, b], c)$. We do so, because the $\vec{x}$ and $\vec{y}$ vectors have the same value for all elements of the prefix of a sequence σ. Furthermore, for $\vec{\phi}(\epsilon)$ we write $([], \perp)$.

Consider the prefix-closure of $f_{use}(\tilde{\Phi}'_1)$ which generates the linear inequalities presented in Table 6.2. The table shows each sequence present in $\overline{f_{use}(\tilde{\Phi}'_1)}$ accompanied by its $\vec{\phi}$-value and the number of occurrences of the sequence in $\overline{f_{use}(\tilde{\Phi}'_1)}$, e.g. $\overline{f_{use}(\tilde{\Phi}'_1)}(\langle a_s, a \rangle) = 56$. Observe that there is a relationship between the occurrence of a sequence and its corresponding postfixes, i.e. after the 56 times that sequence $\langle a_s, a \rangle$ occurred, $\langle a_s, a, b \rangle$ occurred 36 times and $\langle a_s, a, c \rangle$ occurred 20 times (note: $56 = 36 + 20$). Furthermore, observe that multiple sequences actually map to the same constraint. For example, both $\langle a_s, a, b, c, d \rangle$ and $\langle a_s, a, c, b, d \rangle$ map to sequence encoding $([a, b, c], d)$. Due to coupling of sequences to constraints, i.e. by means of sequence encoding, we are able to count the number of occurrences of each constraint. Moreover, we are able to relate constraints related to a certain sequence to their predecessor and successor constraints, i.e. by means of coupling them to their underlying traces. The frequencies in $\overline{f_{use}(\tilde{\Phi}'_1)}$ thus allow us to decide whether the presence of a certain constraint is in line with predominant behaviour in the event store. For example, in Table 6.2, $\vec{\phi}(\langle a_s, a, b, b \rangle)$ relates to *infrequent behaviour* as it appears only once, whereas the other constraints related to prefixes of the same length are more frequent.

To apply filtering, we construct a weighted directed graph in which each sequence encoding acts as a vertex. We connect two vertices by means of an arc if the source constraint corresponds

Table 6.2: Schematic overview of sequence encodings based on $\overline{f_{use}(\tilde{\Phi}'_1)}$.

$\sigma \in \overline{f_{use}(\tilde{\Phi}'_1)}$	$\vec{\phi}(\sigma)^\intercal$, i.e. $(m, \vec{x}(a_s), \vec{x}(a), ..., \vec{y}(g), \vec{y}(a_f))$	$\vec{\phi}(\sigma)$ (shorthand)	$\overline{f_{use}(\tilde{\Phi}'_1)}(\sigma)$
ϵ	$(1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$	$([], \perp)$	56
$\langle a_s \rangle$	$(1,0,0,0,0,0,0,0,0,0,-1,0,0,0,0,0,0,0,0)$	$([], a_s)$	56
$\langle a_s, a \rangle$	$(1,1,0,0,0,0,0,0,0,0,-1,-1,0,0,0,0,0,0,0)$	$([a_s], a)$	56
$\langle a_s, a, b \rangle$	$(1,1,1,0,0,0,0,0,0,0,-1,-1,-1,0,0,0,0,0,0)$	$([a_s, a], b)$	36
$\langle a_s, a, c \rangle$	$(1,1,1,0,0,0,0,0,0,0,-1,-1,0,-1,0,0,0,0,0)$	$([a_s, a], c)$	20
$\langle a_s, a, b, b \rangle$	$(1,1,1,1,0,0,0,0,0,0,-1,-1,-2,0,0,0,0,0,0)$	$([a_s, a, b], b)$	1
$\langle a_s, a, b, c \rangle$	$(1,1,1,1,0,0,0,0,0,0,-1,-1,-1,-1,0,0,0,0,0)$	$([a_s, a, b], c)$	35
$\langle a_s, a, c, b \rangle$	$(1,1,1,0,1,0,0,0,0,0,-1,-1,-1,-1,0,0,0,0,0)$	$([a_s, a, c], b)$	20
$\langle a_s, a, b, b, c \rangle$	$(1,1,1,2,0,0,0,0,0,0,-1,-1,-2,-1,0,0,0,0,0)$	$([a_s, a, b^2], c)$	1
$\langle a_s, a, b, c, d \rangle$	$(1,1,1,1,1,0,0,0,0,0,-1,-1,-1,-1,-1,0,0,0,0)$	$([a_s, a, b, c], d)$	35
$\langle a_s, a, c, b, d \rangle$	$(1,1,1,1,1,0,0,0,0,0,-1,-1,-1,-1,-1,0,0,0,0)$	$([a_s, a, b, c], d)$	20
$\langle a_s, a, b, b, c, d \rangle$	$(1,1,1,2,1,0,0,0,0,0,-1,-1,-2,-1,-1,0,0,0,0)$	$([a_s, a, b^2, c], d)$	1
$\langle a_s, a, b, c, d, e \rangle$	$(1,1,1,1,1,1,0,0,0,0,-1,-1,-1,-1,-1,-1,0,0,0)$	$([a_s, a, b, c, d], e)$	10
$\langle a_s, a, b, c, d, g \rangle$	$(1,1,1,1,1,1,0,0,0,0,-1,-1,-1,-1,-1,0,0,-1,0)$	$([a_s, a, b, c, d], g)$	25
$\langle a_s, a, c, b, d, f \rangle$	$(1,1,1,1,1,1,0,0,0,0,-1,-1,-1,-1,-1,0,-1,0,0)$	$([a_s, a, b, c, d], f)$	9
$\langle a_s, a, c, b, d, g \rangle$	$(1,1,1,1,1,1,0,0,0,0,-1,-1,-1,-1,-1,0,0,-1,0)$	$([a_s, a, b, c, d], g)$	11
$\langle a_s, a, b, b, c, d, d \rangle$	$(1,1,1,2,1,1,0,0,0,0,-1,-1,-2,-1,-2,0,0,0,0)$	$([a_s, a, b^2, c, d], d)$	1
$\langle a_s, a, b, c, d, e, a_f \rangle$	$(1,1,1,1,1,1,1,0,0,0,-1,-1,-1,-1,-1,-1,0,0,-1)$	$([a_s, a, b, c, d, e], a_f)$	10
$\langle a_s, a, b, c, d, g, b \rangle$	$(1,1,1,1,1,1,0,0,1,0,-1,-1,-2,-1,-1,0,0,-1,0)$	$([a_s, a, b, c, d, g], b)$	13
$\langle a_s, a, b, c, d, g, c \rangle$	$(1,1,1,1,1,1,0,0,1,0,-1,-1,-1,-2,-1,0,0,-1,0)$	$([a_s, a, b, c, d, g], c)$	12
$\langle a_s, a, c, b, d, f, a_f \rangle$	$(1,1,1,1,1,1,0,1,0,0,-1,-1,-1,-1,-1,0,-1,0,-1)$	$([a_s, a, b, c, d, f], a_f)$	9
$\langle a_s, a, c, b, d, g, b \rangle$	$(1,1,1,1,1,1,0,0,1,0,-1,-1,-2,-1,-1,0,0,-1,0)$	$([a_s, a, b, c, d, g], b)$	11
$\langle a_s, a, b, b, c, d, d, e \rangle$	$(1,1,1,2,1,2,0,0,0,0,-1,-1,-2,-1,-2,-1,0,0,0)$	$([a_s, a, b^2, c, d^2], e)$	1
$\langle a_s, a, b, c, d, g, b, c \rangle$	$(1,1,1,2,1,1,0,0,1,0,-1,-1,-2,-2,-1,0,0,-1,0)$	$([a_s, a, b^2, c, d, g], c)$	13
$\langle a_s, a, b, c, d, g, c, b \rangle$	$(1,1,1,1,2,1,0,0,1,0,-1,-1,-2,-2,-1,0,0,-1,0)$	$([a_s, a, b, c^2, d, g], b)$	12
$\langle a_s, a, c, b, d, g, b, c \rangle$	$(1,1,1,2,1,1,0,0,1,0,-1,-1,-2,-2,-1,0,0,-1,0)$	$([a_s, a, b^2, c, d, g], c)$	11
$\langle a_s, a, b, b, c, d, d, e, a_f \rangle$	$(1,1,1,2,1,2,1,0,0,0,-1,-1,-2,-1,-2,-1,0,0,-1)$	$([a_s, a, b^2, c, d^2, e], a_f)$	1
$\langle a_s, a, b, c, d, g, b, c, d \rangle$	$(1,1,1,2,2,1,0,0,1,0,-1,-1,-2,-2,-2,0,0,-1,0)$	$([a_s, a, b^2, c^2, d, g], d)$	13
$\langle a_s, a, b, c, d, g, c, b, d \rangle$	$(1,1,1,2,2,1,0,0,1,0,-1,-1,-2,-2,-2,0,0,-1,0)$	$([a_s, a, b^2, c^2, d, g], d)$	12
$\langle a_s, a, c, b, d, g, b, c, d \rangle$	$(1,1,1,2,2,1,0,0,1,0,-1,-1,-2,-2,-2,0,0,-1,0)$	$([a_s, a, b^2, c^2, d, g], d)$	11
$\langle a_s, a, b, c, d, g, b, c, d, f \rangle$	$(1,1,1,2,2,2,0,0,1,0,-1,-1,-2,-2,-2,0,-1,-1,0)$	$([a_s, a, b^2, c^2, d^2, g], f)$	13
$\langle a_s, a, b, c, d, g, c, b, d, f \rangle$	$(1,1,1,2,2,2,0,0,1,0,-1,-1,-2,-2,-2,0,-1,-1,0)$	$([a_s, a, b^2, c^2, d^2, g], f)$	12
$\langle a_s, a, c, b, d, g, b, c, d, e \rangle$	$(1,1,1,2,2,2,0,0,1,0,-1,-1,-2,-2,-2,-1,0,-1,0)$	$([a_s, a, b^2, c^2, d^2, g], e)$	11
$\langle a_s, a, b, c, d, g, b, c, d, f, a_f \rangle$	$(1,1,1,2,2,2,0,1,1,0,-1,-1,-2,-2,-2,0,-1,-1,-1)$	$([a_s, a, b^2, c^2, d^2, f, g], a_f)$	13
$\langle a_s, a, b, c, d, g, c, b, d, f, a_f \rangle$	$(1,1,1,2,2,2,0,1,1,0,-1,-1,-2,-2,-2,0,-1,-1,-1)$	$([a_s, a, b^2, c^2, d^2, f, g], a_f)$	12
$\langle a_s, a, c, b, d, g, b, c, d, e, a_f \rangle$	$(1,1,1,2,2,2,1,0,1,0,-1,-1,-2,-2,-2,-1,0,-1,-1)$	$([a_s, a, b^2, c^2, d^2, e, g], a_f)$	11

to a sequence that is a prefix of a sequence corresponding to the target constraint, i.e. we connect $\vec{\phi}(\langle a_s, a \rangle)$ to $\vec{\phi}(\langle a_s, a, b \rangle)$ as $\langle a_s, a \rangle$ is a prefix of $\langle a_s, a, b \rangle$. The arc weight corresponds to trace frequency in the input event store.

Definition 6.9 (Sequence encoding graph). *Let $\tilde{\Phi} \in \mathcal{B}(\mathcal{A}^*)$ be a simple event store and let $A_{\tilde{\Phi}} \subseteq \mathcal{A}$ denote the corresponding set of described activities. A sequence encoding graph is a directed graph $G = (V, E, \psi)$ where $V = \{\vec{\phi}(\sigma) \mid \sigma \in \overline{\tilde{\Phi}}\}$, $E \subseteq V \times V$ s.t. $(\vec{\phi}(\sigma'), \vec{\phi}(\sigma)) \in E \Leftrightarrow \exists a \in A_{\tilde{\Phi}} \left(\sigma' \cdot \langle a \rangle = \sigma\right)$ and $\psi \colon E \to \mathbb{N}$ where, for $(v_1, v_2) \in E$:*

$$\psi(v_1, v_2) = \sum_{\substack{\sigma \in_+ \overline{\tilde{\Phi}} \\ \vec{\phi}(\sigma) = v_2}} \overline{\tilde{\Phi}}(\sigma) - \sum_{\substack{\sigma' \in_+ \overline{\tilde{\Phi}} \\ \sigma' \cdot \langle a \rangle \in_+ \overline{\tilde{\Phi}} \\ \vec{\phi}(\sigma' \cdot \langle a \rangle) = v_2 \\ \vec{\phi}(\sigma') \neq v_1}} \overline{\tilde{\Phi}}(\sigma') \tag{6.13}$$

Consider the sequence encoding graph in Figure 6.7 on page 163, based on $f_{use}(\tilde{\Phi}'_1)$, as an example. By definition, $([], \perp)$ is the root node of the graph and connects to all one-sized

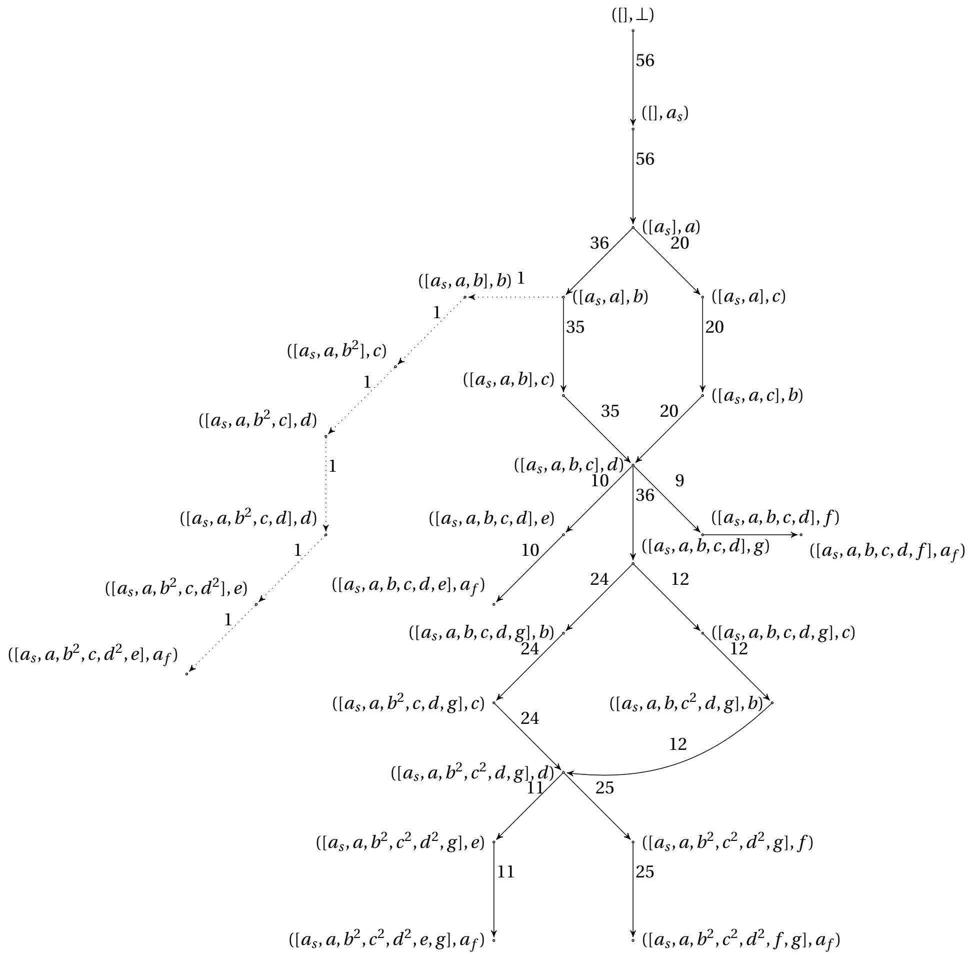

([], ⊥)
56
([], a_s)
56
([a_s], a)
36
20
([a_s, a, b], b)
1
([a_s, a], b)
([a_s, a], c)
1
35
20
([a_s, a, b^2], c)
1
([a_s, a, b], c)
([a_s, a, c], b)
35
20
([a_s, a, b^2, c], d)
1
([a_s, a, b, c], d)
10
9
36
([a_s, a, b^2, c, d], d)
([a_s, a, b, c, d], e)
([a_s, a, b, c, d], f)
1
10
([a_s, a, b, c, d], g)
([a_s, a, b, c, d, f], a_f)
([a_s, a, b^2, c, d^2], e)
([a_s, a, b, c, d, e], a_f)
24
12
1
([a_s, a, b, c, d, g], b)
([a_s, a, b, c, d, g], c)
([a_s, a, b^2, c, d^2, e], a_f)
24
12
([a_s, a, b^2, c, d, g], c)
([a_s, a, b, c^2, d, g], b)
24
12
([a_s, a, b^2, c^2, d, g], d)
11
25
([a_s, a, b^2, c^2, d^2, g], e)
([a_s, a, b^2, c^2, d^2, g], f)
11
25
([a_s, a, b^2, c^2, d^2, e, g], a_f)
([a_s, a, b^2, c^2, d^2, f, g], a_f)

Figure 6.7: An example sequence encoding graph G_1', based on example event store $\tilde{\Phi}_1'$.

Algorithm 6.2: SEF-BFS

```
input  : $G = (V, E, \psi)$, $\kappa : V \to \mathbb{P}(V)$
output : $C \subseteq V$
begin
    $C \leftarrow \emptyset$
    Let $Q$ be a FIFO queue
    $Q.enqueue(([], \perp))$
    while $Q \neq \emptyset$ do
        $v \leftarrow Q.dequeue()$
        for $v' \in \kappa(v)$ do
            $C \leftarrow C \cup \{v'\}$
            $Q.enqueue(v')$
    return $C$
```

sequences. Within the graph we observe the relation among different constraints, combined with their absolute frequencies based on $\tilde{\Phi}'_1$ Furthermore, observe that in multiple cases, certain behaviour maps on the same constraint, then diverges, and later again merges into the same constraint. Because of this, in case certain parts of behaviour are noisy, at later phases, such behaviour still contributes to the filtering distribution.

6.4.3 Filtering

Given a sequence encoding graph, we are able to filter out constraints. In algorithm 6.2 we devise a simple breadth-first traversal algorithm, i.e. `Sequence Encoding Filtering - Breadth-First Search` (`SEF-BFS`), that traverses the sequence encoding graph and concurrently constructs a set of ILP constraints. The algorithm needs a function as an input that is able to determine, given a vertex in the sequence encoding graph, what portion of adjacent vertices remains in the graph and which are removed.

Definition 6.10 (Sequence encoding filter)**.** *Given simple event store $\tilde{\Phi}$ over set of activities $A_{\tilde{\Phi}}$ and a corresponding sequence encoding graph $G = (V, E, \psi)$. A sequence encoding filter is a function $\kappa : V \to \mathscr{P}(V)$.*

Note that κ is an abstract function and might be parametrized. Observe that in principle, we are able to apply the same filters on a vertex in the sequence encoding graph as described in chapter 4, i.e. *Fractional*, *Heavy Hitter* and *Smoothened Heavy Hitter*. Recall that in fractional filtering, we ignore any outgoing arc that has a value smaller than a certain, user specified, fraction level of the sum of all outgoing arcs. In case of (smoothened) heavy hitter filtering, we take such fraction of the maximum outgoing value (possibly corrected). In general, it is desirable that $\kappa(v) \subseteq \{v' \mid (v, v') \in E\}$, i.e. it only considers vertices reached by v by means of an arc. Given any instantiation of κ, it is straightforward to construct a filtering algorithm based on breadth-first graph traversal, i.e. `SEF-BFS`.

The algorithm inherits its worst-case complexity of breadth-first search, multiplied by the worst-case complexity of κ. Thus, in case κ's worst-case complexity is $O(1)$ then we have $O(|V| + |E|)$ for the `SEF-BFS`-algorithm. It is trivial to prove, by means of induction on the length of a sequence encoding's corresponding sequence, that a sequence encoding graph is acyclic. Hence, termination is guaranteed.

As an example of executing the SEF-BFS algorithm, reconsider Figure 6.7. Assume we use heavy hitter filtering with a threshold value of 0.35. Vertex $([], \perp)$ is initially present in Q and will be analysed. Since $([], a_s)$ is the only child of $([], \perp)$, it is added to Q. Vertex $([], \perp)$ is removed from the queue and is never inserted in the queue again due to the acyclic property of the graph. Similarly, since $([a_s], a)$ is the only child of $([], a_s)$ it is added to Q. Both children of $([a_s], a)$, i.e. $([a_s, a], b)$ and $([a_s, a], c)$ are added to the queue since the maximum corresponding arc value is 36, and, $0.35 * 36 = 12.6$, which is smaller than the lowest arc value 20. When analysing $([a_s, a], b)$ we observe a maximum outgoing arc with value 35 to vertex $([a_s, a, b], c)$ which is enqueued in Q. Since $0.35 * 35 = 8.75$, the algorithm does not enqueue $([a_s, a, b], b)$. Note that the whole path of vertices from $([a_s, a, b], b)$ to $([a_s, a, b^2, c, d^2, e], a_f)$ is never analysed and is stripped from the constraint body, i.e. they are never inserted in C. Observe that all other arcs and constraints in the graph remain in the constraint body. As the example shows, choosing the right threshold level is essential, yet not trivial. For example, when using a threshold 0.35, some proper constraints, e.g. $[(a_s, a, b, c, d], f)$, are removed from the constraint body as well.

When applying ILP-based process discovery based on event store $\tilde{\Phi}'_1$ with sequence encoding filtering with heavy hitter filtering with threshold value 0.25, we indeed obtain the workflow net depicted in Figure 6.6a. As explained, the filter leaves out all constraints related to vertices on the path from $([a_s, a, b], b)$ to $([a_s, a, b^2, c, d^2, e], a_f)$. Hence, we find a similar model to the model found on event store $\tilde{\Phi}_1$ and are able to filter out infrequent exceptional behaviour.

6.5 Evaluation

algorithm 6.1, together with sequence encoding filtering, cf. algorithm 6.2, has a corresponding implementation in the ProM framework, which we describe in more detail in subsection 9.5.2. As a filter, we have implemented heavy hitter filtering. However, note that it is implemented as follows:

$$\kappa(v) = \{v' \mid (v, v') \in E \wedge \psi(v, v') \geq (1 - t) \cdot \max_{v'' \in V} \psi(v, v'')\}, \ t \in [0, 1] \tag{6.14}$$

This implies that, to obtain the same results as discussed in subsection 6.4.3, we need to use a threshold value of 0.75, rather than 0.25, i.e. $1 - 0.75 = 0.25$. To evaluate the approach, we conducted several experiments, using the aforementioned code as a basis. Experiments are conducted on machines with 8 Intel Xeon CPU E5-2407 v2 2.40 GHz processors and 64 GB RAM. In an artificial setting, we evaluated the quality of models discovered and the efficiency of applying sequence encoding filtering. We also compare sequence encoding to the Inductive Miner Infrequent (IMi) [79] algorithm, i.e. inductive miner with integrated filtering, and automaton-based filtering [40]. Finally, we assess the performance of sequence encoding filtering on real event data [81, 85]. Observe that in all cases, we use event logs in these experiments, rather than event streams.

6.5.1 Model Quality

The event logs used in the empirical evaluation of model quality are artificially generated event logs and originate from a study related to the impact of exceptional behaviour to rule-based approaches in process discovery [87]. Three event logs were generated out of three different process models, i.e. the *ground truth event logs*. These event logs do not consist of any exceptional behaviour, i.e. every trace fits the originating model. The ground truth event logs are called *a12f0n00*, *a22f0n00* and *a32f0n00*. The two digits behind the *a* character indicate the number of distinct activities present in the event log, i.e. *a12f0n00* contains 12 different activities. From each ground truth event log, by means of trace manipulation, four other event logs are created that do contain exceptional behaviour. Manipulation concerns tail/head of trace removal, random part of the trace body removal and interchanging two randomly chosen events [87]. The percentages of trace manipulation are 5%, 10%, 20% and 50%. The manipulation percentage is incorporated in the last two digits of the event log's name, i.e. the 5% manipulation version of the *a22f0n00* event log is called *a22f0n05*.

The existence of ground truth event logs, free of exceptional behaviour, is of utmost importance for the evaluation. We need to be able to distinguish *normal* from *exceptional* behaviour in an *unambiguous manner*. Within evaluation, these event logs, combined with the quality dimension precision, allow us to judge how well a technique is able to filter out exceptional behaviour. Recall that precision is defined as the number of traces producible by the process model that are also present in the event log. Thus if all traces producible by a process model are present in an event log, precision is maximal, i.e. the precision value is 1. If the model allows for traces that are not present in the event log, precision is lower than 1.

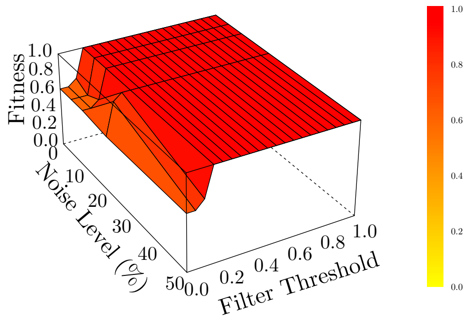
If exceptional behaviour is present in an event log, the conventional ILP-based process discovery algorithm produces a workflow net that allows for all exceptional behaviour. As a result, the algorithm is typically unable to find any meaningful patterns within the event log. This typically leads to places with a lot of self-loops. The acceptance of exceptional behaviour by the workflow net, combined with the inability to find meaningful patterns yields a low level of precision, *when using the ground truth log as a basis for precision computation*. On the other hand, if we discover models using an algorithm that is better in handling the presence of exceptional behaviour, we expect the algorithm to allow for less exceptional behaviour and find more meaningful patterns. Thus, with respect to the ground truth model, we expect higher precision values.

To evaluate the sequence encoding filtering approach, we have applied the ILP-based process discovery algorithm with sequence encoding filtering using filter thresholds $0, 0.05, 0.1, ..., 0.95, 1$. Moreover, we performed similar experiments for IMi [79] and automaton-based event log filtering [40] combined with ILP-based discovery. We measured precision [92] and replay-fitness [6] based on the *ground truth event logs*. The results for the different event logs are presented in Figure 6.8, Figure 6.9 and Figure 6.10. In the charts, we plot replay-fitness/precision against the noise level and filter threshold. We additionally use a colour scheme to highlight the differences in value.

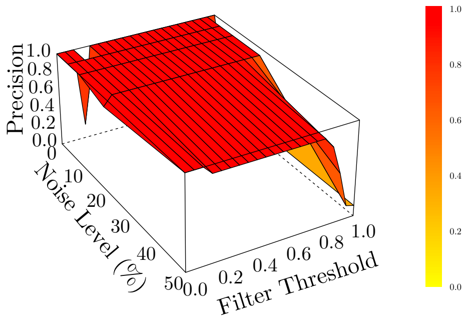
In Figure 6.8, on page 167, we present the replay-fitness and precision results of the experiments with the *a12f0nXX* event logs. Sequence encoding filtering shows low replay-fitness values for all event logs when using a filter threshold of 0. The replay-fitness values of the discovered models quickly rise to 1 and remain 1 for all filter thresholds above 0.2. In case of IMi, for a filter value of 1.0 we observe some values of 1 for replay-fitness.[6] Non-perfect replay-fitness seems to be more local, concentrated around noise levels 5% and 10% with corresponding threshold levels in-between 0.4 and 0.8. Finally, automaton-based filtering rapidly loses perfect replay-fitness when the filter threshold exceeds 0.2. Only for a noise-level of 0 it seems to retain high replay-values. Upon inspection, it turns out the filter returns empty event logs for the corresponding threshold and noise levels.

For the precision results, the charts of the sequence encoding filter show expected behaviour,

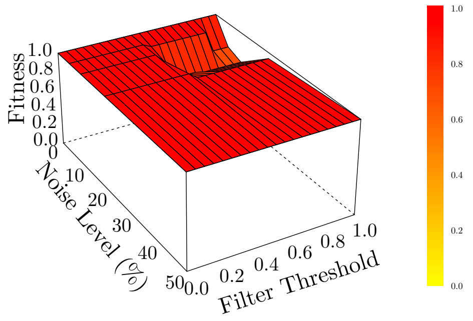
[6]Note that the IMi filter threshold works inverted with respect to sequence encoding filtering, i.e. a value of 1 implies most rigorous filtering, and thus a threshold value of 1 comparable to 0.0 for sequence encoding.



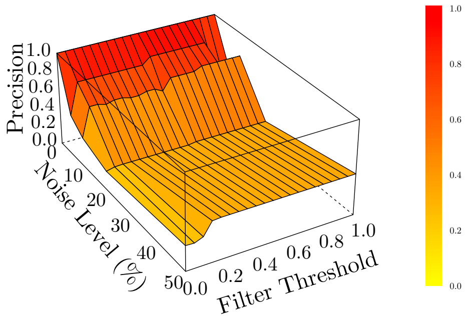
(a) Sequence Encoding - Replay-fitness.



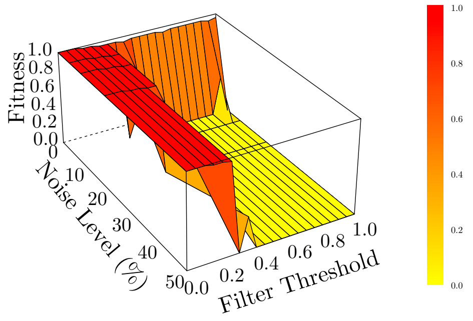
(b) Sequence Encoding - Precision.



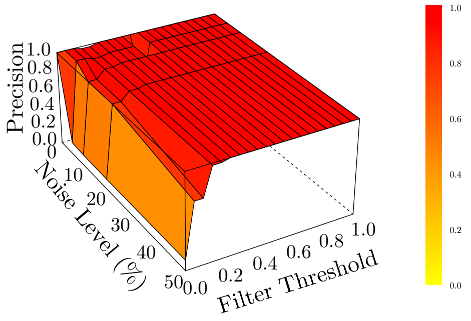
(c) IMi [79] - Replay-fitness.



(d) IMi [79] - Precision.



(e) ILP with AFA [40] - Replay-fitness.



(f) ILP with AFA [40] - Precision.

Figure 6.8: Replay-fitness and precision measurements based on *a12f0nXX* for Sequence Encoding, IMi [79] and ILP with AFA [40]. We observe that IMi quickly leads to imprecise models when noise increases. AFA quickly leads to non-fitting models when noise increases. Sequence encoding finds perfectly fitting and precise models, for most threshold values.

i.e. with high noise levels and high filter thresholds precision is low. There is, however, an unexpected drop in precision for noise-level 0 with a filter threshold around 0.2. The IMi filter behaves less predictable since the drop in precision seems mainly depending on the noise level rather than the filter setting. We expect the precision to be higher in case a filter threshold of 1.0 is chosen. There is only a slight increase for the 50% noise log when comparing a filter threshold of 0 to a filter threshold of 1. Finally, the precision of the automaton filter behaves as expected, i.e. precision rapidly increases together with an increase in the filter threshold.

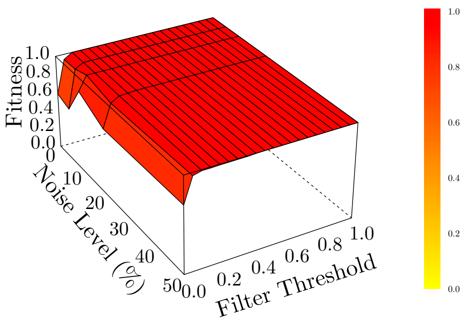
In Figure 6.9, on page 169, we present the replay-fitness and precision results of the experiments with the *a22f0nXX* event logs. For the sequence encoding filter (Figure 6.9a and Figure 6.9b) we again observe that replay-fitness is often 1, except for very rigorous levels of filtering, i.e. $t = 0$ and $t = 0.05$. When applying filtering as rigorous as possible, i.e. $t = 0$, we observe relatively stable replay-fitness values of around 0.6, for different levels of noise. The discovered model at 0% noise level has a precision value of 1. This implies that the filter, in the case of 0% noise, removes behaviour that is present in the ground-truth event log. Precision drops to around 0.7 for increasing levels of noise. The relatively stable levels of replay-fitness and precision for increasing levels of noise when using $t = 0$ suggest that the filter only incorporates a few branches of most frequent behaviour, which is the same throughout different levels of noise. Since the precision values are lower than 1, combined with the fact that parallelism exists in the original model, it seems that the most frequent branches do incorporate some form of parallelism that allow for behaviour not observed in the event log.

For the 5% and 10% noise levels we observe that threshold values in between 0 and 0.6 achieve acceptable levels of precision. These values are slightly lower than the precision values related to 0% noise, which implies that the filter in these cases is not able to remove all noise. The rapid trend towards precision values close to 0 for threshold levels above 0.6 suggests that the filter does not remove any or very little noise. For higher levels of noise, we observe a steeper drop in precision. Only very low threshold levels (up to 0.2) achieve precision values around 0.3. The results suggest that these levels of noise introduce levels of variety in the data that no longer allow the sequence encoding filter to identify (in)frequent behaviour. Hence, even for low threshold values the filter still incorporates noise into the resulting process models.

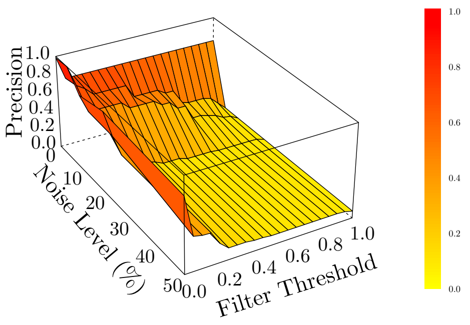
For IMi (Figure 6.9c and Figure 6.9d) we observe similar behaviour (as indicated, the filter threshold works inverted with respect to sequence encoding filtering). However, replay-fitness drops a little earlier compared to sequence encoding filtering. The drop in the precision of the sequence encoding filtering is smoother than the drop in precision of IMi, i.e. there exist some spikes within the graph. Hence, applying filtering within IMi, on these data, behaves less deterministic.

Finally, automaton based filtering (Figure 6.9e and Figure 6.9f) rapidly drops to replay-fitness values of 0. Upon inspection, it again turns out that the filter returns empty event logs for the corresponding threshold and noise levels. Hence, the filter seems to be very sensitive around a threshold value in-between 0 and 0.2. The precision results for the automaton based filter are as expected. For a low threshold value, we have very low precision, except when we have a 0% noise level. Towards a threshold level of 0.2, precision increases after which it maximizes out to a value of 1. This is in line with the replay-fitness measurements.

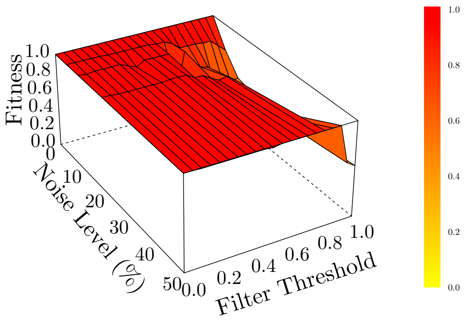
Finally, in Figure 6.10 we present the replay-fitness and the precision results of the experiments with the *a32f0nXX* event logs. Due to excessive computation time the automaton based filter [40] is left out of the analysis. We observe that sequence encoding filtering behaves similar to the experiments performed with the *a12f0nXX* and *a22f0nXX* event logs. Replay-fitness again quickly increase to 1 for increasing filter threshold values. We observe that IMi seems to filter out more behaviour related to the underlying system model when the filter threshold increases. Observe that, due to loop structures, the precision of a model that equals the originating model



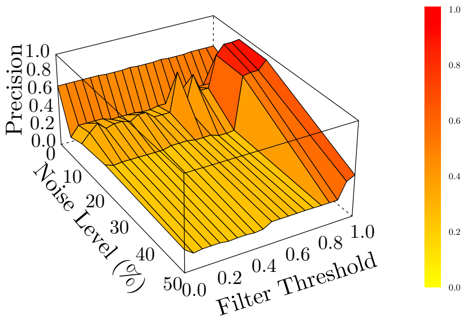
(a) Sequence Encoding - Replay-fitness.



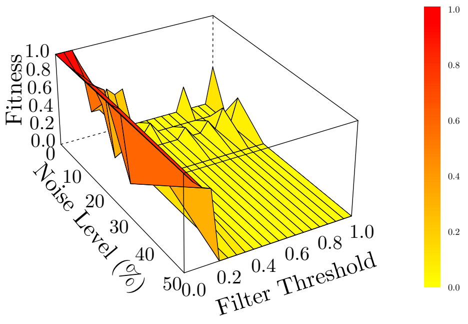
(b) Sequence Encoding - Precision.



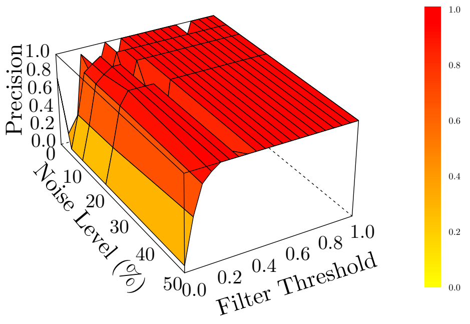
(c) IMi [79] - Replay-fitness.



(d) IMi [79] - Precision.

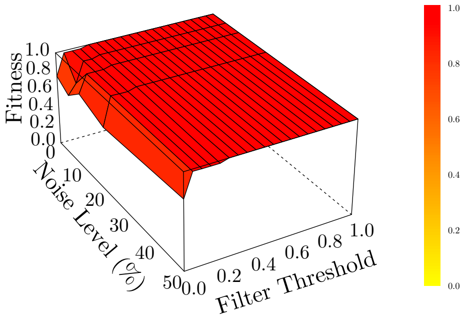


(e) ILP with AFA [40] - Replay-fitness.

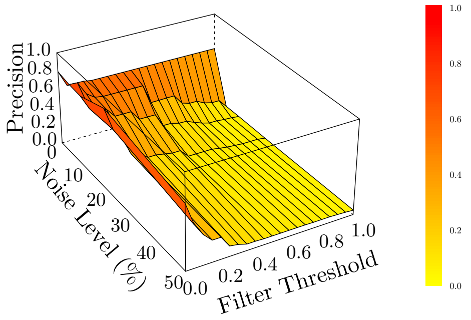


(f) ILP with AFA [40] - Precision.

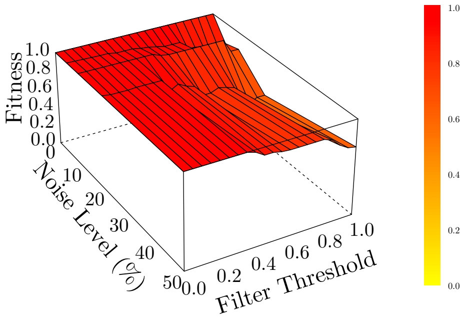
Figure 6.9: Replay-fitness and precision measurements based on *a22f0nXX* for Sequence Encoding, IMi [79] and ILP with AFA [40]. We observe that both Sequence encoding and IMi quickly lead to imprecise models when noise increases. AFA quickly leads to non-fitting models when noise increases.



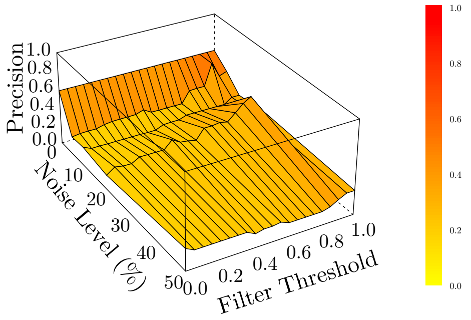
(a) Sequence Encoding - Replay-fitness.



(b) Sequence Encoding - Precision.



(c) IMi [79] - Replay-fitness.



(d) IMi [79] - Precision.

Figure 6.10: Replay-fitness and precision measurements based on *a32f0nXX* for Sequence Encoding and IMi [79]. We observe that both Sequence encoding and IMi quickly lead to imprecise models when noise increases.

is only roughly 0.6. Sequence encoding filtering shows a smooth decrease in precision when both noise and filter-thresholds are increased, which is as expected. With low noise levels and a low threshold value, sequence encoding seems to be able to filter out the infrequent behaviour, however, if there is too much noise and too little is removed we start finding workflow nets with self-loop places. IMi seems to result in models with a slightly higher precision compared to sequence encoding filtering. As is the case in the *a22f0nXX* event logs, we observe spike behaviour in precision of IMi based models hinting at non-deterministic behaviour of the filter.

We conclude that the sequence encoding filter and IMi lead to comparable results. However, the sequence encoding filter provides more expected results, i.e. IMi behaves somewhat less deterministic. The automaton based filter does provide good results, however, sensitivity of the filter threshold is much higher compared to sequence encoding filtering and IMi.

6.5.2 Computation time

Using sequence encoding filtering, we leave out constraints that refer to exceptional behaviour. Hence, we reduce the size of the core ILP constraint body and thus expect a decrease in computation time when applying rigorous filtering, i.e. heavy hitter filtering with filter threshold towards a value of 0. Using RapidMiner we repeated similar experiments to the experiments performed for model quality, and measured *cpu-execution* time for the three techniques. However, we only use threshold values 0, 0.25, 0.75 and 1.

In Figure 6.11, on page 172, we present the average cpu-execution time, based on 50 experiment repetitions, needed to obtain a process model from the *a22f0nXX* event logs. For each level of noise, we depict computation time for different filter threshold settings. For IMi, we measured the inductive miner algorithm with integrated filtering. For sequence encoding and automaton filtering, we measure the time needed to filter, discover a causal graph and solve underlying ILP problems. Observe that for IMi and the automaton-based filter, filtering most rigorously is performed with threshold levels of 1, as opposed to sequence encoding filtering which filters most rigorously at threshold 0.

We observe that IMi is fastest in most cases. Computation time slightly increases when the amount of noise increases within the event logs. For sequence encoding filtering we observe that lower threshold values lead to faster computation times. This is as expected since a low threshold value removes more constraints from the ILP constraint body than a high threshold value. The automaton-based filter is slowest in all cases. The amount of noise seems to have little impact on the computation time of the automaton-based filter, it seems to be predominantly depending on the filter threshold. From Figure 6.11 we conclude that IMi in general out-performs sequence encoding in terms of computation time. However, sequence encoding, in turn, out-performs automaton-based filtering.

6.5.3 Application to Real-Life Event Logs

We tested the applicability of sequence encoding filtering using real-life event logs. We used an event log related to a road fines administration process [81] and one regarding the treatment of patients suspected to have sepsis [85].

The results are presented in Figure 6.12 and Figure 6.13.

In case of the Road Fines event log, presented in Figure 6.12, we observe that replay-fitness is around 0.46 whereas precision is around 0.4 for threshold values from 0 to 0.5. The number of arcs for the models of these threshold values remains constant (as well as the number of places and the number of transitions) suggesting that the models found are the same. After this, the replay-fitness increases further to around 0.8 and reaches 1 for a threshold value of 1. Interestingly, precision shows a little increase around threshold values between 0.5 and 0.75 after which it drops slightly below its initial value. In this case, a threshold value in-between 0.5 and 0.75 seems most appropriate in terms of replay-fitness, precision and simplicity.

In case of the Sepsis event log, presented in Figure 6.13, we observe that replay-fitness and precision are roughly behaving as each-other's inverse, i.e. replay-fitness increases whereas precision decreases for increasing threshold values. We moreover observe that the number of arcs within the process models increases, for larger threshold values. In this case, a threshold value in-between 0.1 and 0.4 seems most appropriate in terms of replay-fitness, precision and simplicity.

Finally, for each experiment, we measured the associated computation time of solving all ILP problems. In case of the Road Fines event log, solving all ILP problems takes roughly 5 seconds. In case of the Sepsis event log, obtaining a model ILP problems takes less than 1 second.

As our experiments show, there is no specific threshold most suitable for sequence encoding, i.e. this greatly depends on the event log. However, we do observe that using lower threshold values, e.g. 0 – 0.4, leads to less complex models. We, therefore, in practical settings, advise to use a lower threshold value first, which also reduces computation time due to a smaller

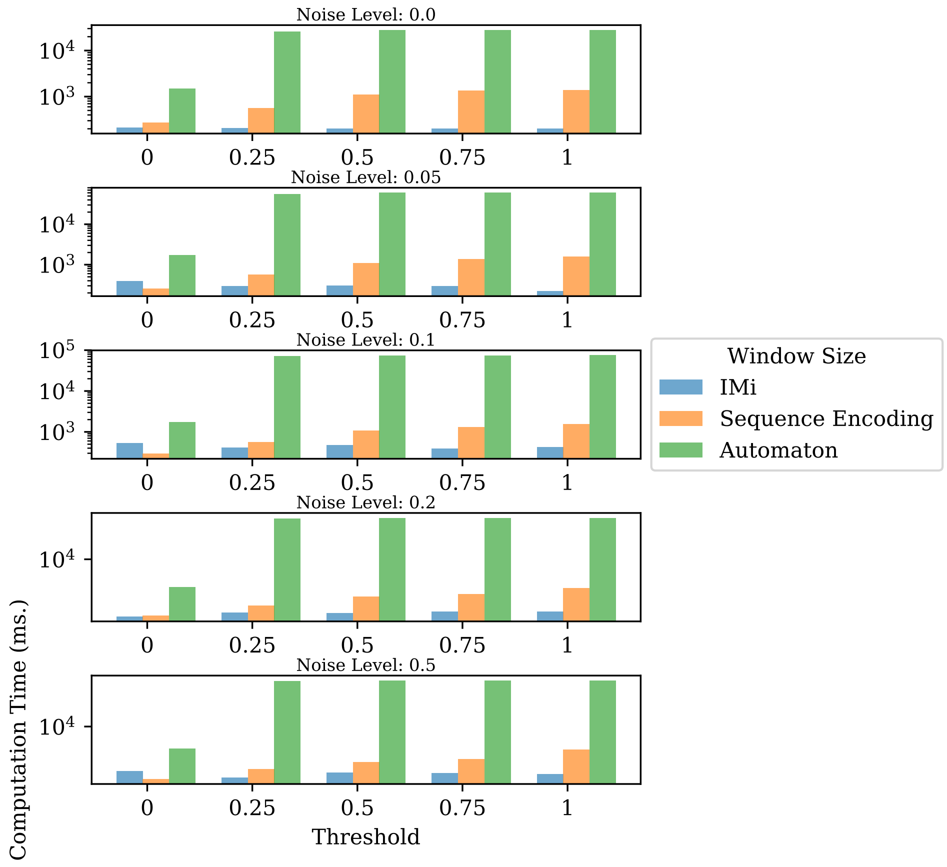
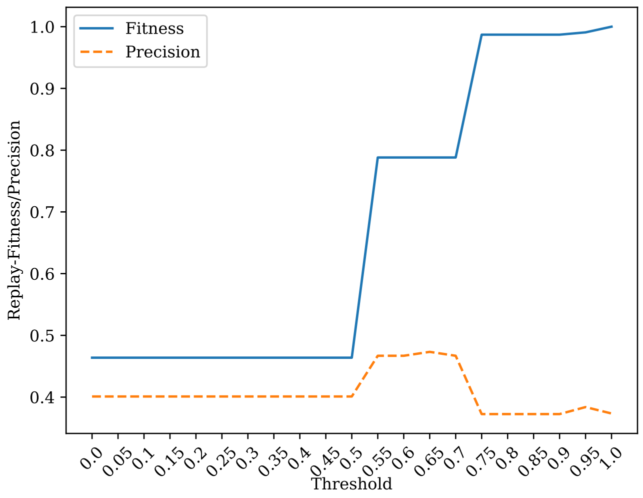
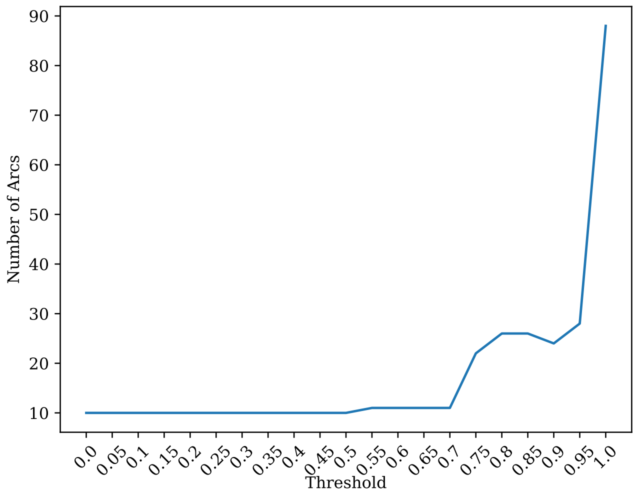


Figure 6.11: CPU-Execution Time (ms.) for a22f0nXX event logs (logarithmic scale) for different levels of noise. The percentage of noise is depicted on top of each bar chart.

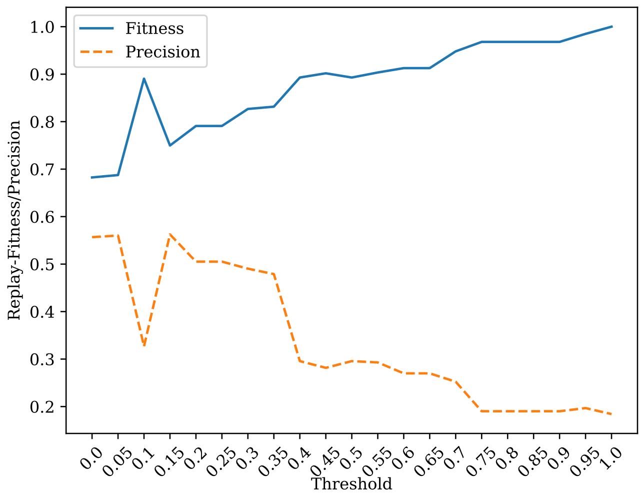


(a) Fitness (blue) and Precision (orange).

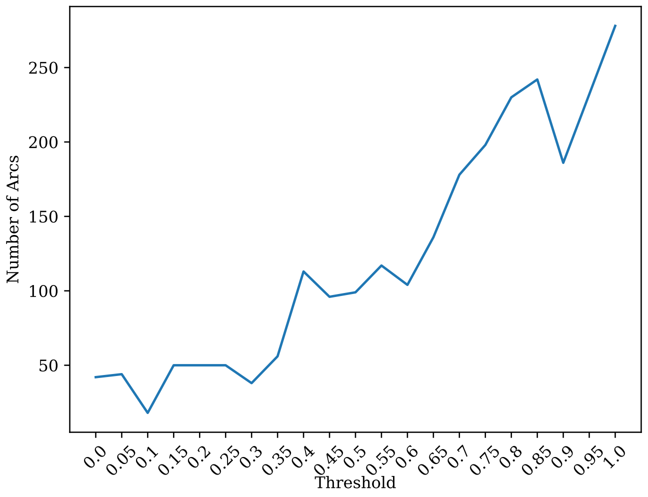


(b) Number of Arcs.

Figure 6.12: Replay-fitness, precision and complexity based on the `Road Fines` log [81].



(a) Fitness (blue) and Precision (orange).



(b) Number of Arcs.

Figure 6.13: Replay-fitness, precision and complexity based on the `Sepsis` log [85].

constraint body size, and based on the obtained result increase or decrease the threshold value if necessary.

6.6 Related Work

Several process discovery techniques, based on region theory, have been proposed based on region theory. Region theory comes in two forms, i.e. state-based region theory [24, 52, 53] using transition systems as an input and language-based region theory [19, 23, 43, 82, 83] using languages as an input. The main difference between the synthesis problem and process discovery is related to the generalization of the discovered models. Process models found by classical region theory approaches have perfect replay-fitness and maximal precision. Process discovery on the other hand, aims at extracting a generalizing process model, i.e. precision, and in some cases replay-fitness, need not to be maximized.

In [10] a process discovery approach is presented that transforms an event log into a transition system, after which state-based region theory is applied. Constructing the transition system is strongly parametrized, i.e. using different parameters yields different process discovery results. In [106] a similar approach is presented. The main contribution is a complexity reduction with respect to conventional region-based techniques. In [22] a process discovery approach is presented based on language-based region theory. The method finds a minimal linear basis of a polyhedral cone of integer points, based on the event log. It guarantees perfect replay-fitness, whereas it does not maximize precision. The worst-case time complexity of the approach is exponential in the size of the event log. In [37] a process discovery algorithm is proposed based on the concept of numerical abstract domains. Based on the event log's prefix-closure, a convex polyhedron is approximated by means of calculating a convex hull. The convex hull is used to compute causalities in the input event log by deducing a set of linear inequalities which represent places. In [123] a first design of a process discovery ILP-formulation is presented. An objective function is presented, which is generalized in [133], that allows for expressing a preference for finding certain Petri net places. The work also presents means to formulate ILP constraints that help finding more advanced Petri net-types, e.g. Petri nets with reset- and inhibitor arcs.

All aforementioned techniques leverage the strict implications of region theory with respect to process discovery, i.e. precision maximization, poor generalization and poor simplicity, to some extent. However, the techniques still perform suboptimal. Since the techniques guarantee perfect replay-fitness, they tend to fail if exceptional behaviour is present in the event log, i.e. they produce models that are incorporating infrequent behaviour (outliers).

6.7 Conclusion

The work presented in this chapter is motivated by the observation that existing region-based process discovery techniques are useful, as they are able to find non-local complex control flow patterns. However, the techniques do not provide any structural guarantees with respect to the resulting process models, and, they are unable to cope with infrequent, exceptional behaviour in event stores.

6.7.1 Contributions

The approach presented in this chapter, extends techniques presented in [123, 131, 133]. We have proven that our approach is able to discover relaxed sound workflow nets, i.e. we are now able to guarantee structural properties of the resulting process model. Additionally, we presented the sequence encoding filtering technique which enables us to filter out exceptional behaviour within the ILP-based process discovery algorithm. The experiments conducted in the context of this chapter confirm that the technique enables us to find meaningful Petri net structures in data consisting of exceptional behaviour, using ILP-based process discovery as an underlying technique. Sequence encoding filtering proves to be comparable to the IMi [79] approach, i.e. an integrated filter of the Inductive Miner [78], in terms of filtering behaviour. Moreover, it is considerably faster than the general purpose filtering approach of [40] and less sensitive to variations in the filter threshold.

6.7.2 Limitations

The algorithm as presented here, i.e. ILP-based process discovery on the basis of the causal relations present in a (simple) event store, are all defined on the notion of static data. As indicated, since we presented means to capture streaming data into a temporal finite static collection, i.e. by means of event stores as defined in chapter 3, we are able to perform the ILP-based process discovery approach presented here in a streaming setting. Note however that there are some challenges related to such adoption, regarding *trace completion* and *computational feasibility*.

Trace Completeness

Classical region theory assumes that the input system description is the complete behaviour of the given system. This assumption is partially adopted in ILP-based process discovery, i.e. we assume there exists behaviour in the process that is not reflected by the event store and we assume noise to exist within the data. Nonetheless, ILP-based process discovery does assume the traces present within the (simple) event store to refer to *completed process instances*. Note that this is mainly due to the constraints formed by $m\vec{1} + \mathbf{M}_{\tilde{\Phi}}(\vec{x} - \vec{y}) = \vec{0}$, which ensure us to find places that are empty after a trace is completed.

In a streaming setting, we observe incompleteness on both the start of cases and the termination, i.e. we typically store fragments of behaviour rather than full behaviour. As we have shown in chapter 3, we are partially able to account for incompleteness at the start of traces by using prefix-trees as an underlying data structure. However, incompleteness is an inherent challenge in event stream based process mining. To cope with this problem, two potential solutions are applicable, which work on different dimensions of the problem.

1. *Data dimension*; We integrate an additional component that monitors inactive cases. Such inactivity is an indicator for trace completeness. For those traces that are estimated to be incomplete, we predict the most likely remaining sequence of activities. In such a way, we partially base the constraint body based on predicted behaviour.
2. *Algorithmic dimension*; We are able to drop the constraints of the form $m\vec{1} + \mathbf{M}_{\tilde{\Phi}}(\vec{x} - \vec{y}) = \vec{0}$, in fact, in [123], these constraints are presented as an extension tailored towards finding workflow nets. By doing so, we lose the aforementioned property related to empty places after traces are completed. The main downside of this solution is related to the fact that we are no longer able to find workflow nets, nor guarantee *relaxed soundness*, which in

turn potentially hampers the use of the discovered models in subsequent process mining analyses.

Computational Feasibility

Aside from the potential challenges related to trace completeness, computational feasibility is another potential problem when adopting the techniques presented in this chapter in combination with an event store. ILP is known to be *NP-Complete*, which implies that finding a solution in general is time consuming. The main complexity is in the number of variables, which in turn depends on the number of activities observed. When positioned in the general data stream model, cf. section 1.2, applying ILP-based process discovery on top of a event store is in the *computation time* category. As such, it is advisable to apply it in a batch fashion, rather than in a continuous fashion.

6.7.3 Open Challenges & Future Work

An interesting direction for future work concerns combining ILP-based process discovery techniques with other process discovery techniques. The Inductive Miner discovers sound workflow nets, however, these models lack the ability to express complex control flow patterns such as a milestone pattern. Some of these patterns are however reconstructible using ILP-based process discovery. Hence, it is interesting to combine these approaches with possibly synergetic effects with respect to the process mining quality dimensions.

As discussed in subsection 6.7.2, ILP-based process discovery is not feasible to be applied from scratch after receiving a new event. However, given that we have discovered a model using ILP, we are able to use the underlying constraints for validation purposes. If a new event does not introduce any new behaviour, we are guaranteed that all of the places are still valid, according to region theory. Moreover, removal of behaviour never invalidates the places found, i.e. the constraint body gets reduced and is thus less restrictive. When new behaviour is added, at most two new constraints are added, i.e. one related to the main constraint body, and possibly one related to constraints of the form $m\vec{1} + \mathbf{M}_{\tilde{\Phi}}(\vec{x} - \vec{y}) = \vec{0}$. Hence, in $O(|P|)$ time we are able to verify whether the places within the current Petri net still accurately describe the behaviour within the event stream. Moreover, it serves as an indicator which specific ILP needs to be solved, i.e. to replace an invalid place. In such a way we are able to gradually shift ILP-based process discovery to a continuous incremental approach, rather than a batch approach. Furthermore, it is interesting to assess whether or not addition/removal of behaviour has an impact on the optimality of the regions found thus far.

Chapter 7

Online Conformance Checking using Prefix-Alignments

Thus far, we have presented online process mining techniques related to data storage and quality, cf. chapter 3 and chapter 4, and process discovery, cf. chapter 5 and chapter 6. In this chapter, we focus on *conformance checking*, i.e. we aim to compare a given process model with a stream of events and indicate whether or not the model accurately describes the behaviour captured by the event stream. In particular, we present an approach to incrementally compute *prefix-alignments*, which paves the way for accurate real-time online conformance checking and/or process monitoring. The corresponding experiments show that the reuse of previously computed prefix-alignments enhances memory efficiency, whilst preserving prefix-alignment optimality. Moreover, we show that, in the case of computing approximations of optimal prefix-alignments, there is a clear trade-off between memory efficiency and approximation error.

The contents presented in this chapter are based on the following publication:

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*, 2017 `https://doi.org/10.1007/s41060-017-0078-6`

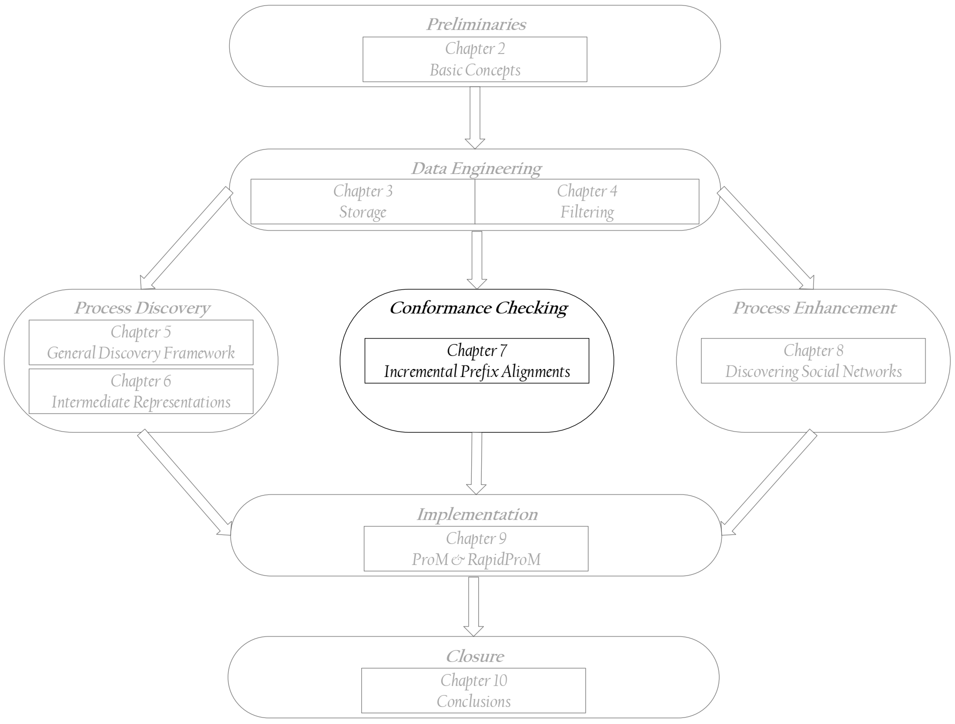


Figure 7.1: The contents of this chapter, i.e. event stream based conformance checking using incremental prefix-alignments, highlighted in the context of the general structure of this thesis.

7.1 Introduction

The techniques discussed so far allow us to store event data, preferably free of noise. Moreover, in chapter 5 and chapter 6, we described how to learn process models from event data, i.e. process discovery. As we introduced in section 1.1, process mining also covers the area of *conformance checking*. In conformance checking, we aim at assessing to what degree the behaviour described by a process model is in line with the behaviour as captured within event data and thus, in the context of this thesis, an event stream.

Existing conformance techniques were developed based on the conventional, offline, process mining notion, i.e. on the basis of static event logs. Clearly, we are able to apply these techniques in a streaming setting by maintaining event stores as presented in chapter 3, and iteratively apply existing, conventional, conformance checking techniques. Applying these techniques in such a way, however, does not allow us to obtain conformance statistics on the fly, i.e. to observe deviations with respect to the reference model at the moment they occur. At the same time, early *process-oriented deviation detection* is critical for many organizations in different domains. Similarly, within highly complex administrative processes, e.g. provision of mortgages, notary processes and unemployment administration, deviant behaviour often leads to excessive process execution time and costs. Upon detection of a deviation, a process owner, or the supporting information system, is able to take adequate actions such as blocking the current process instance, assigning a case manager for an additional specialized intervention and/or even restarting the process instance.

In this chapter, we, therefore, focus on the incremental computation of *prefix-alignments* [13], cf. subsection 2.4.2. Recall that, prefix-alignments are a relaxed variant of conventional *alignments*. In particular, conventional alignments relate the traces of behaviour observed in an event log to a reference model by mapping them on the most likely corresponding execution path of the model. The main premise of prefix-alignments, as opposed to conventional alignments, is the fact that we explicitly take into account that the future behaviour of a process instance is unknown. Within prefix-alignments, we, therefore, map the behaviour seen thus-far onto an execution path of the model that most accurately describes the behaviour observed up until that point in time. Since we, in a streaming setting, typically consider incomplete process instance behaviour, computing prefix-alignments rather than conventional alignments is a natural fit. However, as we aim to obtain conformance checking results in an online fashion, exploiting the conventional, offline oriented, alignment computation is infeasible.

Consider Figure 7.2, in which we present a schematic overview of the proposed online conformance checking approach. We have two main sources of input, i.e. an event stream generated by the information system under study and a reference process model. Over time, we observe events emitted on the event stream which tell us what activity has been performed in the context of which case, i.e. we again focus on the *control-flow perspective*. For each case, we maintain a prefix-alignment. Whenever we receive a new event for a case, we recompute its prefix-alignment. We try to recompute prefix-alignments greedily, however, in some cases we need to resort to solving a shortest path problem. The main focus of this chapter is towards the efficiency of solving such shortest path problems.

The proposed approach entails an incremental algorithm that allows for computing both *optimal* and *approximate* prefix-alignments. We additionally show that the cost of an optimal prefix-alignment is always an underestimate for the cost of a conventional alignment of any of its possible suffixes, i.e. its future behaviour. As a consequence, when computing optimal prefix-alignments, our approach underestimates alignment costs for completed cases. This implies that once we detect a deviation from the reference model, we are guaranteed that the behaviour related to the case is not compliant with the reference model. Computing approximate

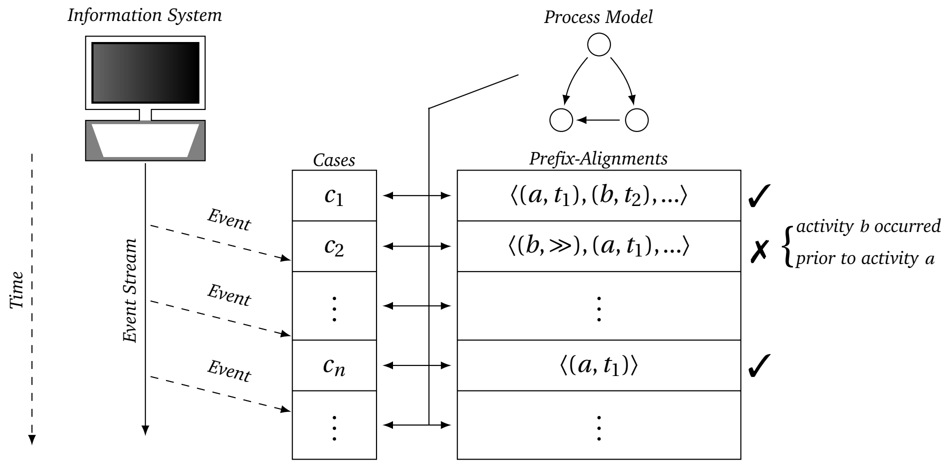

Information System
Process Model
Cases
Prefix-Alignments
Time
Event Stream
Event
Event
Event
c_1
c_2
$\vdots$
c_n
$\vdots$
$\langle(a,t_1),(b,t_2),...\rangle$ ✓
$\langle(b,\gg),(a,t_1),...\rangle$ ✗
activity b occurred prior to activity a
$\langle(a,t_1)\rangle$ ✓

Figure 7.2: Schematic overview of online conformance checking.

prefix-alignments leads to an improvement in memory efficiency, however, at the cost of losing prefix-alignment optimality.

We experimentally assess the trade-off between memory efficiency and optimality loss using several artificially generated process models. We additionally assess the applicability of the algorithm using real event data. The experiments show that reusing previously computed prefix-alignments positively impacts the efficiency of computing new prefix-alignments. Moreover, in case of using approximations, we observe a clear trade-off between memory usage and prefix-alignment optimality loss.

The remainder of this chapter is structured as follows. In section 7.2, we present an incremental algorithm to compute prefix-alignments, i.e. tailored towards event streams. In section 7.3, we evaluate the algorithm in terms of memory behaviour and optimality loss under approximation. In section 7.4 we discuss related work in the area of (event stream based) conformance checking. Finally, section 7.5 concludes this chapter.

7.2 Online Prefix-Alignments Computation

Alignments do not only provide us with an indication of behavioural deviations with respect to a given reference model, they also provide us with an explicit explanation of such deviations. This is of great interest in the context of conformance checking, as we are able to observe, *and explain*, deviations. When applying conventional alignments in a streaming setting, the underlying algorithm that computes conventional alignments, by design, "finishes" the behaviour in the model. This leads to falsely detecting, and explaining, deviations.

Assume that we are observing an event stream, and we receive the events related to a process instance of the running example with duplicate/invisible labels, i.e. Figure 2.13a. Furthermore, the process instance relates to correct behaviour, according to the model, e.g. it describes simple trace $\langle a, b, c, d, e\rangle$. However, since we use an event stream, we observe the corresponding events one-by-one, i.e. first an event describing activity a, secondly an event describing activity b, etc. Consider Figure 7.3, in which we depict what happens when computing conventional

γ_1:

a	$\gg$	$\gg$	$\gg$
t_1	t_3	t_4	t_7

(a) An optimal (conventional) alignment for simple trace $\langle a \rangle$, with respect to the running example model.

γ_2:

a	b	$\gg$	$\gg$	$\gg$
t_1	t_2	t_3	t_5	t_8

(b) An optimal (conventional) alignment for simple trace $\langle a, b \rangle$, with respect to the running example model.

γ_2:

a	b	c	$\gg$	$\gg$
t_1	t_2	t_3	t_5	t_7

(c) An optimal (conventional) alignment for simple trace $\langle a, b, c \rangle$, with respect to the running example model.

γ_2:

a	b	c	d	$\gg$
t_1	t_2	t_3	t_5	t_8

(d) An optimal (conventional) alignment for simple trace $\langle a, b, c, d \rangle$, with respect to the running example model.

γ_2:

a	b	c	d	e
t_1	t_2	t_3	t_5	t_7

(e) An optimal (conventional) alignment for simple trace $\langle a, b, c, d, e \rangle$, with respect to the running example model.

Figure 7.3: Example of continuously computing optimal conventional alignments in an online setting. We gradually observe that the alignment-costs of the optimal alignments for the trace of behaviour decreases and eventually becomes zero. As multiple optimal alignments exist, the explanation of the "missing behaviour", potentially changes across the different alignments computed, i.e. as represented by the alternating use of transitions t_7 and t_8.

alignments repeatedly, after each received event for the process instance. After receiving the first event (Figure 7.3a), i.e. resulting in trace $\langle a \rangle$, we obtain an optimal alignment that describes that we are missing an execution of transitions t_3, t_4 and t_7, representing activities c, d and e, respectively. After receiving the second event (Figure 7.3b), the alignment indicates that we are missing an execution of transitions t_3, t_5 and t_8, representing activities c, d and f, respectively. Observe that, when assuming unit-costs for the alignment moves, the estimated deviation is equal for both traces $\langle a \rangle$ and $\langle a, b \rangle$. However, the explanation of the expected missing behaviour in this case differs. After receiving the third event (Figure 7.3c), the alignment indicates that we are missing an execution of transitions t_5 and t_7, representing activities d and e. In this case, the estimated deviation is reduced, yet the explanation again differs with respect to the previously computed alignment. Finally, after receiving all behaviour related to the example trace, we observe that it is correct, and hence, we have consistently been overestimating the deviation, with different explanations.

When using prefix-alignments, we avoid the aforementioned problems. Consider Figure 7.4, where we present the optimal prefix-alignments, on the basis of aligning the same simple trace, i.e. $\langle a, b, c, d, e \rangle$ on the running example model. Observe that, in this case, we never observe any deviation. This is the case, because every prefix considered, indeed allows us to correctly finish the behaviour of the model. Moreover, we never observe any other type of deviation, e.g. a log

$$\gamma_1: \begin{array}{||c|} a \\ \hline t_1 \end{array}$$

(a) An optimal prefix-alignment for simple trace $\langle a \rangle$, with respect to the running example model.

$$\gamma_2: \begin{array}{||c|c|} a & b \\ \hline t_1 & t_2 \end{array}$$

(b) An optimal prefix-alignment for simple trace $\langle a, b \rangle$, with respect to the running example model.

$$\gamma_2: \begin{array}{||c|c|c|} a & b & c \\ \hline t_1 & t_2 & t_3 \end{array}$$

(c) An optimal prefix-alignment for simple trace $\langle a, b, c \rangle$, with respect to the running example model.

$$\gamma_2: \begin{array}{||c|c|c|c|} a & b & c & d \\ \hline t_1 & t_2 & t_3 & t_5 \end{array}$$

(d) An optimal prefix-alignment for simple trace $\langle a, b, c, d \rangle$, with respect to the running example model.

$$\gamma_2: \begin{array}{||c|c|c|c|c|} a & b & c & d & e \\ \hline t_1 & t_2 & t_3 & t_5 & t_7 \end{array}$$

(e) An optimal prefix-alignment, which also represents the final, conventional, alignment, cf. Figure 7.3e, for simple trace $\langle a, b, c, d, e \rangle$, with respect to the running example model.

Figure 7.4: Example of continuously computing optimal prefix-alignments in an online setting. We never get notified of any problems with the observed trace, which is in accordance with reality.

move. Hence, computing the prefix-alignments, in this case, reduces the number of false alarms during process monitoring.

The main aim of the algorithm presented in this chapter relates to online conformance checking, and, efficient prefix-alignment computation in particular. As explained in subsection 2.4.2, the minimal requirement to compute (prefix)-alignments, is that the Petri net is *easy sound*. Recall that this implies that at least one sequence of transition firings exists that, given the initial marking, allows us to reach the final marking of the Petri net. However, given a Petri net that is only easy sound, i.e. not weak sound nor sound, when computing an optimal prefix-alignment, the computational complexity increases.[1] This, specifically in an online setting, is disadvantageous. As in general, the models we use in conformance checking are considered to be designed by a human, preferably experienced, process designer, we assume them, arguably, to be of a certain level of quality. We therefore assume that the Petri nets used in this chapter are *sound workflow nets*, cf. 2.7. Under this assumption, given an arbitrary marking in the state space of such net, determining whether we are still able to reach the final marking is not needed, i.e. we are by definition able to reach such final marking.

In the remainder of this section, we present an algorithmic framework that allows us to compute prefix-alignments on the basis of event streams, in an incremental fashion. Subsequently, we present effective parametrization of the algorithm that allows us to reduce memory usage and computation time, in some cases at the cost of losing prefix-alignment optimality, i.e. resulting prefix-alignments that overestimate the cost with respect to the optimal prefix-alignment.

[1]This relates to the fact that we need to assess for each reachable state in the state-space of the underlying search problem, whether we are still able to reach the final marking.

7.2.1 An Incremental Framework

We aim at computing a prefix-alignment for each sequence of events seen so far on the event stream, for each case $c \in \mathscr{C}$. A basic approach to do this, is to use a (case-view) event store $\Phi_{\mathscr{C}}$ that allows us to query the recent history of behaviour for each case. Upon receiving the i^{th} event e on the event stream, we simply calculate $\overline{\omega}(N, \Phi^{i}_{\mathscr{C}}(\pi_{\mathsf{c}}(e)), M_i, M_f)$. The downside of this approach, however, is that we recompute a prefix-alignment from scratch, each time a new event arrives.

We, therefore, propose the use of a *prefix-alignment store*, defined in a similar fashion as event stores, of the form:

$$\Phi^{i}_{\overline{\Gamma}} \colon \mathscr{E}^* \times \mathscr{C} \rightarrow \overline{\Gamma} \tag{7.1}$$

Observe that $\Phi^{0}_{\overline{\Gamma}}(S, c) = \epsilon, \forall e \in \mathscr{E}$, i.e. initially there are no prefix-alignments. We additionally define a prefix-alignment store update function, that allows us to incrementally update the contents of the prefix-alignment store, i.e.

$$\overrightarrow{\Phi}_{\overline{\Gamma}} \colon \overline{\Gamma} \times \mathscr{E} \rightarrow \overline{\Gamma} \tag{7.2}$$

As such, we characterize $\Phi^{i}_{\overline{\Gamma}}(S, c) = \overrightarrow{\Phi}_{\overline{\Gamma}}(\Phi^{i-1}_{\overline{\Gamma}}(S, c), S(i))$.

Consider algorithm 7.1, in which we present a greedy algorithm for the purpose of prefix-alignment calculation, which includes an instantiation of the $\overrightarrow{\Phi}_{\overline{\Gamma}}$-function. Within the algorithm, we conceptually perform the following steps. When we receive an event related to a certain case, we check whether we previously computed a prefix-alignment for that case. In case we are guaranteed that the event refers to an activity move, i.e. because the activity simply has no corresponding label in the reference model, we append such activity move to the prefix-alignment. If this is not the case, we fetch the marking in the reference model, corresponding to the previous prefix-alignment. For example, given prefix-alignment $\langle (a, t_1) \rangle$ based on the running example net (Figure 2.13a), the corresponding marking is $[p_1, p_2]$. If the event is the first event received for the case, we simply obtain marking M_i. In case we are able to directly fire a transition within the obtained marking with the same label as the activity that the event refers to, we append a corresponding synchronous move to the previously computed prefix-alignment. Otherwise, we use a shortest path algorithm, of which we present some parametrization in subsection 7.2.2, to find a new (optimal) prefix-alignment. The algorithm expects a Petri net, initial- and final marking, an algorithm that computes optimal prefix-alignments and an event stream as an input. Note that, after receiving a new event, the prefix-alignment store for index $i-1$ is copied into the i^{th} version, i.e. line 5. This operation is $O(1)$ in practice.

In general, we are able to use any of the techniques described in chapter 3, e.g. sliding windows, reservoir sampling, on the level of cases. Thus, we maintain a sliding window of case identifiers rather than events, and an associated counter per case identifier present in the window. Whenever such a counter reaches zero, we set $\Phi^{i}_{\overline{\Gamma}}(S, c)$ to be ϵ, i.e. we remove the currently known prefix-alignment. Observe that we are also able to do this using other stream-based data storage techniques. Note however, that the proposed scheme works under the assumption that process instances are of a finite fashion. In the remainder of this chapter, we are primarily interested in the design of the $\overrightarrow{\Phi}_{\overline{\Gamma}}$-function, in terms of computing the actual new prefix-alignment of the newly arrived case.

Since optimal prefix-alignments underestimate conventional alignment costs (7.1), we are interested to what extent algorithm 7.1 guarantees optimality of the prefix-alignments stored in $\Phi_{\overline{\Gamma}}$.

Algorithm 7.1: `Incremental Prefix-Alignment Computation`

```
input: $N=(P,T,F,\lambda), M_i, M_f \in \mathscr{B}(P), \overline{\omega}\colon \mathscr{N} \times \mathscr{A}^* \times \mathscr{M} \times \mathscr{M} \rightarrow \overline{\Gamma}, S \in \mathscr{E}^*$
begin
    $i \leftarrow 0$;
    while $true$ do
        $i \leftarrow i+1$;
        $e \leftarrow S(i)$;
        copy all alignments of $\Phi^{i-1}_{\overline{\Gamma}}(S,c)$ to $\Phi^{i}_{\overline{\Gamma}}(S,c)$ for all $c \in \mathscr{C}$;
        $c \leftarrow \pi_{\mathrm{c}}(e)$;
        $a \leftarrow \pi_{\mathrm{a}}(e)$;
        $\overline{\gamma} \leftarrow \Phi^{i-1}_{\overline{\Gamma}}(S,c)$;
        let $M$ be the marking of $N$ obtained by $\overline{\gamma}$, i.e. such that $M_i \xrightarrow{(\pi_2(\overline{\gamma}))_{\downarrow T}} M$;
        if $\exists t \in T(\lambda(t)=a)$ then
            if $\exists t \in T(\lambda(t)=a \wedge (N,M)[t\rangle)$ then
                let $t$ denote such transition;
                $\Phi^{i}_{\overline{\Gamma}}(S,c) \leftarrow \overline{\gamma} \cdot \langle (a,t) \rangle$;
            else
                $\sigma \leftarrow (\pi_1(\overline{\gamma}))_{\downarrow \mathscr{A}}$;
                $\Phi^{i}_{\overline{\Gamma}}(S,c) \leftarrow \overline{\omega}(N, \sigma \cdot \langle a \rangle, M_i, M_f)$;
        else
            $\Phi^{i}_{\overline{\Gamma}}(S,c) \leftarrow \overline{\gamma} \cdot \langle (a,\gg) \rangle$;
```

Theorem 7.1 (algorithm 7.1 guarantees optimal prefix-alignments). *Let $\mathscr{E}$ denote the universe of events, let $\mathscr{C}$ denote the universe of case identifiers and let $\overline{\Gamma}$ denote the universe of prefix-alignments. Furthermore, given $N=(P,T,F,\lambda), M_i, M_f \in \mathscr{B}(P), \overline{\omega}\colon \mathscr{N} \times \mathscr{A}^* \times \mathscr{M} \times \mathscr{M} \rightarrow \overline{\Gamma}, S \in \mathscr{E}^*$, and, given $\Phi^{i}_{\overline{\Gamma}}\colon \mathscr{E}^* \times \mathscr{C} \rightarrow \overline{\Gamma}$ being a prefix-alignment store, cf. Equation 7.1, with $\Phi^{0}_{\overline{\Gamma}}(S,c)=\epsilon$, $\forall c \in \mathscr{C}$, which is updated according to algorithm 7.1. For any $S \in \mathscr{E}^*$, $c \in \mathscr{C}$, $i \in \mathbb{N}$ and $\overline{\gamma}=\Phi^{i}_{\overline{\Gamma}}(S,c)$ we have $\overline{\gamma} \in \overline{\Gamma}(N,\sigma,M_i,M_f)$ and $\overline{\gamma}$ is optimal for $(\pi_1(\overline{\gamma}))_{\downarrow \mathscr{A}}$.*
Proof *(Induction on i)*

- Base Case I: $i=0$*; All alignments are* ϵ.
- Base Case II: $i=1$*; Let* $e=(c,a,\ldots)=S(i)$. *We know* $\Phi^{i-1}_{\overline{\Gamma}}(S,c)=\Phi^{0}_{\overline{\Gamma}}(S,c)=\epsilon$. *In case we are able to fire some* t *with* $\lambda(t)=a$ *in* M_0, *we obtain alignment* $\langle (a,t) \rangle$, *which, under the unit-cost function is optimal. In case* $\nexists_{t \in T}(\lambda(t)=a)$, *we obtain* $\langle (a,\gg) \rangle$ *which is trivially an optimal prefix-alignment for trace* $\langle a \rangle$. *In any other case we compute* $\overline{\omega}(N,\langle a \rangle,M_i,M_f)$ *(optimal by definition).*
- Induction Hypothesis*; Let* $i>1$. *For any* $c \in \mathscr{C}$, *we assume that for* $\overline{\gamma}=\Phi^{i}_{\overline{\Gamma}}(S,c)$, *we have* $\overline{\gamma} \in \overline{\Gamma}$ *and* $\overline{\gamma}$ *is optimal.*
- Inductive Step*; We prove that, for any* $c \in \mathscr{C}$, *for* $\overline{\gamma}=\Phi^{i+1}_{\overline{\Gamma}}(S,c)$, *we have* $\overline{\gamma} \in \overline{\Gamma}$ *and* $\overline{\gamma}$ *is optimal.*

*Let $e = (c, a, ...) = S(i+1)$. In case $\Phi^{i}_{\overline{\Gamma}}(S, c) = \epsilon$ we know that $\overline{\gamma}$ is optimal (*Base Case $i = 1$*). Let $\Phi^{i}_{\overline{\Gamma}}(S, c) = \overline{\gamma}'$ s.t. $\overline{\gamma}' \neq \epsilon$. In case we are able to fire some t with $\lambda(t) = a$ in M, cf. line 11, we obtain $\overline{\gamma} = \overline{\gamma}' \cdot \langle(a, t)\rangle$. Since, under unit-cost function, $c(a, t) = 0$, if $\overline{\gamma}$ is non-optimal, then also $\overline{\gamma}'$ is non-optimal which contradicts the* IH. *A similar rationale holds in case $\nexists_{t \in T}(\lambda(t) = a)$. In any other case, we compute $\overline{\omega}(N, M_i, M_f, \sigma \cdot \langle a \rangle)$ which is optimal by definition.* □

Theorem 7.1 proves that algorithm 7.1 always computes optimal prefix-alignments for $(\pi_1(\overline{\gamma}))_{\downarrow_{\mathscr{A}}}$, i.e. the sequence of activities currently stored within $\Phi^{i}_{\overline{\Gamma}}$ for some $c \in \mathscr{C}$. Hence, combining this result with 7.1, we conclude that whenever the algorithm observes certain alignment costs exceeding 0, i.e. under unit cost function, the corresponding conventional alignment has at least the same costs, or higher.

Interestingly, any optimal prefix-alignment of any prefix of a trace is always underestimating the costs of the optimal alignment of any of its possible suffixes, and thus, of the eventually completed trace.

Proposition 7.1 (Prefix-alignments underestimate alignment costs). *Let $\sigma \in \mathscr{A}^*$ be a sequence of activities. Let $N = (P, T, F, \lambda)$ be a Petri net with labelling function $\lambda \colon T \to \Sigma \cup \{\tau\}$ and corresponding initial- and final marking $M_i, M_f \in \mathscr{B}(P)$. Let $\gamma^* \in \Gamma(N, \sigma, M_i, M_f)$ be an optimal alignment of σ and N. If $\overline{\sigma}$ is a prefix of σ and $\overline{\gamma}^* \in \overline{\Gamma}(N, \overline{\sigma}, M_i, M_f)$ is a corresponding optimal prefix-alignment, then $c(\overline{\gamma}) \leq c(\gamma)$.*

Proof *(Contradiction)*

Let us write γ^* as $\gamma^* = \gamma' \cdot \gamma''$, s.t. $(\pi_1(\gamma'))_{\downarrow_{\mathscr{A}}} = \overline{\sigma}$. Observe that by definition, γ' is a prefix-alignment of $\overline{\sigma}$. Furthermore, as $\overline{\gamma}^*$ is an optimal prefix-alignment for $\overline{\sigma}$, we know that $c(\overline{\gamma}^*) \leq c(\gamma')$. For $c(\overline{\gamma}^*) > c(\gamma^*)$ to hold, we deduce that $c(\gamma'') < 0$, which is impossible, i.e. as $rng(c) = \mathbb{R}_{\geq 0}$. Hence, in case $c(\overline{\gamma}^*) > c(\gamma^*)$, then also $c(\overline{\gamma}^*) > c(\gamma')$, which contradicts optimality of $\overline{\gamma}^*$. □

The property, presented in 7.1, is useful since, in an online setting, once an optimal prefix-alignment has non-zero costs, it guarantees that a deviation from the reference model is present. On the other hand, if a case is not properly terminated, and, will never terminate, yet the sequence of activities seen so far has a prefix-alignment cost of zero, we do not observe this type of deviation until we compute a corresponding conventional (optimal) alignment.

7.2.2 Parametrization

In the previous section, we used $\overline{\omega}$ completely as a black box, and, always solved a shortest path problem starting from M_i. In this section, we show that we are able to exploit the previously calculated alignment for a case c in order to prune the search state-space. Moreover, we show means to limit the search by changing its starting point.

Cost Upper Bounds

Assume that we receive the i^{th} event $e = (c, a, ...)$ on the stream and we let $\overline{\gamma}' = \Phi^{i-1}_{\overline{\Gamma}}(S, c)$ and $\overline{\gamma} = \Phi^{i}_{\overline{\Gamma}}(S, c)$. Let us write the corresponding sequence of activities as $\sigma = \sigma' \cdot \langle a \rangle$. By 7.1 we know that $\overline{\gamma}'$ is an optimal prefix-alignment for σ'. It is easy to see that the costs of $\overline{\gamma}'$ together with an activity move on a are an upper bound for the costs of $\overline{\gamma}$, i.e. $c(\overline{\gamma}) \leq c(\overline{\gamma}') + c(a, \gg)$. We are able to utilize this knowledge within the shortest path search algorithm $\overline{\omega}$. Whenever we encounter a path within the search that is (guaranteed to be) exceeding $c(\overline{\gamma}') + c(a, \gg)$, we simply ignore it, and all paths extending it.

a	b	x	c	d
t_1	t_2	$\gg$	t_3	t_5

$\xrightarrow{Receive\ b}$

a	b	x	c	d	$\gg$	b
t_1	t_2	$\gg$	t_3	t_5	t_6	t_2

Figure 7.5: Partially reverting ($k=2$) the prefix-alignment of $\langle a,b,x,c,d\rangle$ and N_1 in case of receiving new activity b. The grey coloured moves are not considered when computing the new alignment.

As indicated, in alignment computation, the A^* algorithm is often used as an instantiation for $\overline{\omega}$. The A^* algorithm traverses the state-space in an implicit manner, i.e. it expects each state it visits to tell which states are their neighbours, and, at what distance. Moreover, it assumes that each state is able to estimate its distance to the closest final state, i.e. each state has a *heuristic* distance estimation to the closest final state. For the purpose of computing (prefix-)alignments, there are two of these heuristic distance functions defined [13, Chapter 4]. The exact characterization of these heuristic functions is out of this chapter's scope, i.e. it suffices to know that we are able to, for each marking in the synchronous product net, compute the estimated distance (in terms of alignment costs) to final marking M_f. Moreover, such estimation is always underestimating the true distance. Thus, whenever we encounter a marking M in the state space of which the distance to reach M from M_i, combined with the estimated distance to M_f, exceeds $c(\overline{\gamma}') + c(a,\gg)$, we ignore it and all of its possible subsequent markings.

Limiting the Search

Again, assume we receive the i^{th} event $e = (c, a, ...)$ and we let marking M be the marking obtained by executing the transitions of $\overline{\gamma}' = \Phi_{\Gamma}^{i-1}(S,c)$. In case there exist transitions t with $\lambda(t) = a$, yet none of these transitions are enabled in M, the basic algorithm simply utilizes $\overline{\omega}(N, \sigma\cdot\langle a\rangle, M_i, M_f)$. In general, the shortest path algorithm does not need M_i as a start state, i.e. we are able to choose any marking of N as a start state. Hence, we propose to *partially revert* alignment $\overline{\gamma}'$ up to a maximal revert distance k and start the shortest path search from the corresponding marking. Doing so however no longer guarantees optimality as we are no longer searching for a global optimum in the state-space.

Consider Figure 7.5, where we depict a prefix-alignment for $\langle a,b,x,c,d\rangle$ and the running example Petri net (Figure 2.13a). Assume we receive a new event that states that activity b follows $\langle a,b,x,c,d\rangle$ and we use a revert window size of $k=2$. Note that the marking related to the alignment is $[p_5]$. In this marking, no transition with label b is enabled and the algorithm normally calls $\overline{\omega}(N_1, \langle a,b,x,c,d,b\rangle, [p_i], [p_o])$. However, we revert the alignment two moves, i.e. we revert (d,t_5) and (c,t_3) and call $\overline{\omega}(N_1, \langle c,d,b\rangle, [p_2,p_3], [p_o])$ instead. The result of this call is $\langle (c,t_3),(d,t_5),(\gg,t_6),(b,t_2)\rangle$, depicted on the right-hand-side of Figure 7.5. Note that after this call, the window shifts, i.e. the call appended two moves and thus (c,t_3) and (d,t_5) are no longer considered upon receiving of new events.

7.3 Evaluation

We have evaluated the proposed algorithm, including its parametrization, using a corresponding prototypical implementation. We primarily focus on the behaviour of the algorithm as a deviation monitoring tool, i.e. to what degree does the parametrization influence the calculated prefix-alignment costs? Moreover, we focus on the impact of the parametrization on the computational performance of the algorithm. As an underlying search algorithm, we use the A^*

Table 7.1: Parameters used in experiments.

Parameter	***Type***	***Values***
Use Upper Bound	`boolean`	`true`, `false`
Window Size	`integer`	$\{1,2,3,4,5,10,20,\infty\}$

algorithm implementation as provided by `hipster4j` [101]. To evaluate the proposed algorithm, we generated several process models with different characteristics, i.e. different degrees of parallelism, choice and loops. Additionally, we evaluated our approach using real event data, related to the treatment of hospital patients suspected of having sepsis. In this experiment, we additionally compare computing prefix-alignments with repeatedly computing conventional alignments on an event stream.

7.3.1 Experimental Set-up

We used a scientific workflow which, conceptually, performs the following steps:

1. Generate a (block-structured) workflow net with k labelled transitions, where k is drawn from a triangular distribution with parameters $\{10,20,30\}$, for increasing levels of Parallelism, Choice and Loops (from 0 to 50% in steps of 10%) [73].
2. For each workflow net, generate an event log with 1000 cases.
3. For each event log, add increasing levels (from 0 to 50% in steps of 10%) of one type of noise, i.e. *remove activity*, *add activity* or *swap activities*.
4. For each noisy event log, do incremental conformance checking against the workflow net it was generated from, using all parameter combinations presented in Table 7.1.

Observe that within the experiments we mimic event streams by visiting each event in a trace, one-by-one, e.g. if we consider simple event log $\tilde{L} = [\langle a,b,c\rangle, \langle a,c,b\rangle]$ we generate event stream $\langle(1,a),(1,b),(1,c),(2,a),(2,c),(2,b)\rangle$. Moreover, we align every trace-variant once, i.e. if $\langle a,b,c\rangle$ occurs multiple times in the event log, we only align it once. Note that this is feasible, since we aim to assess the impact of the algorithm's parametrization with respect to computational complexity and the prefix-alignment cost overestimation. As such, we to some degree, abstract from the explicit streaming notion, i.e. handling multiple process instances in parallel is not of influence on the global aim of the evaluation. In total, we have generated 18 different models, for which we generate 18 different event logs, each containing 1000 traces, yielding 18.000 noise-free traces. After applying the different types of noise we obtain a total of 324.000 traces. Clearly, the number of events per trace greatly varies depending on the generated model, however, within our experiments, in total 44.537.728 events were processed (with varying algorithm parametrization). Out of these events, 12.151.510 state-space searches were performed.

7.3.2 Results

Here, we present the results of the experiments, in line with the parametrization options as described in subsection 7.2.2. We first present results related to using cost upper bounds, later we present the results related to limited search.

Cost Upper Bounds

In this section, we present the results related to the performance of using cost upper bounds. Within these results we only focus on execution of $\overline{\omega}$ with M_0 as a start state, i.e. we do not incorporate results related to varying window sizes. In Figure 7.6, on page 191, we present, in terms of the level of introduced noise, the average number of enqueued states, queue size, visited nodes and traversed arcs for each search in the state-space.

Clearly, using the upper bound as a cut-off value in state-space traversal greatly enhances the memory efficiency. We observe that, when using the upper bound defined in subsection 7.2.2, the average number of states enqueued during the search is less than half compared to not using the upper bound. The average queue size, i.e. the average number of states in the queue throughout the search, is much lower in case of using a lower bound. We observe that the search efficiency, i.e. in terms of the number of visited states and traversed arcs, is positively affected by using the upper bound, however, the difference is smaller when compared to the number of enqueued states and average queue size, and in some cases negligible (0% noise level). Thus, using previously computed prefix-alignment values for a case allows effective states-space pruning in the shortest path algorithm.

In Figure 7.7, on page 192, we show the effect of the length of the prefix that needs to be aligned in terms of memory consumption. We only show results for length ≤ 100. Both in case of using and not using the upper bound we observe a linear increase in the number of states queued (Figure 7.7a) and average queue size (Figure 7.7b). However, the rate of growth is much lower when using an upper bound. We observe a small region of spikes in both charts around prefix-length 20-25. After investigating the distribution of prefix-length with respect to type of process model, i.e. containing loops versus not containing loops, we observed that most traces exceeding such length are related to the models containing loops. As the other group of models describes relatively more parallelism and/or choice constructs, the complexity of the underlying shortest path search is expected to be slightly more complex, which explains the spikes for relatively short prefix-lengths.

Reverting Alignments

In this section, we present results related to the performance and approximation quality of using revert windows as described in section 7.2.2. In Figure 7.8, we present performance results in terms of memory efficiency and approximation error, plotted against noise level. In Figure 7.8a, we show the average number of states enqueued when using different revert window sizes. Clearly, the memory usage increases when we increase the window size. Interestingly this increase seems linear. The approximation error (Figure 7.8b) shows an inverse pattern, however, the decrease in approximation error seems non-linear, when the window size increases. Moreover, in case we set the window size to 5 we observe that the approximation error, within this experiment, is negligible, whereas memory-wise window sizes of 10 and 20 use much more memory while hardly improving the quality of the result.

In Figure 7.9, we present performance results in terms of memory efficiency and approximation error, plotted against prefix length. We observe that in terms of enqueued nodes (Figure 7.9a), at first a rapid increase appears, after which a steep decline is present. Stabilization around lengths ≥ 25 is again due to the fact that all traces of such length originate from models with loops. The peak and decline behaviour is explained by the fact that the complexity of solving state-space based search within the models is most likely to be most complex in the middle of the trace. Towards the end of a model's behaviour, we expect less state-space complexity, which explains the decline in the chart around a prefix length of 10 to 20.

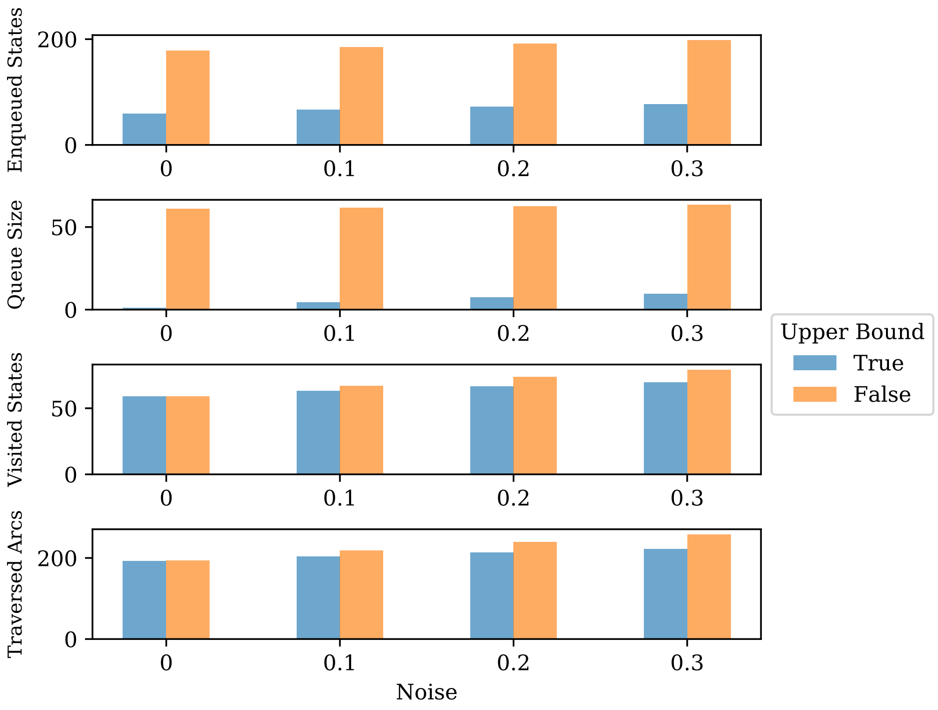
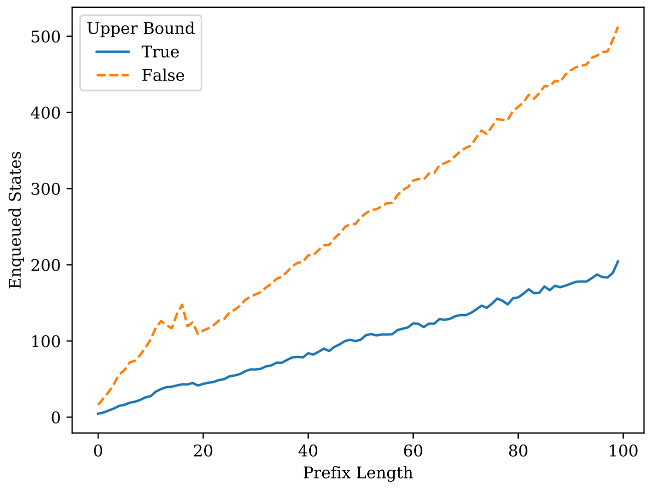
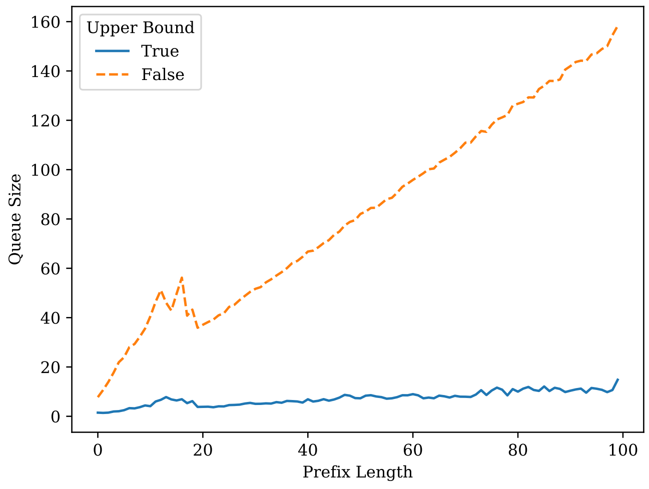


Figure 7.6: Performance results (averaged) of using-, versus not using prefix-alignment cost upper bounds while searching for optimal prefix-alignments. The difference in performance of using upper bounds is most notable in the number of *enqueued states* and the average *queue size*.

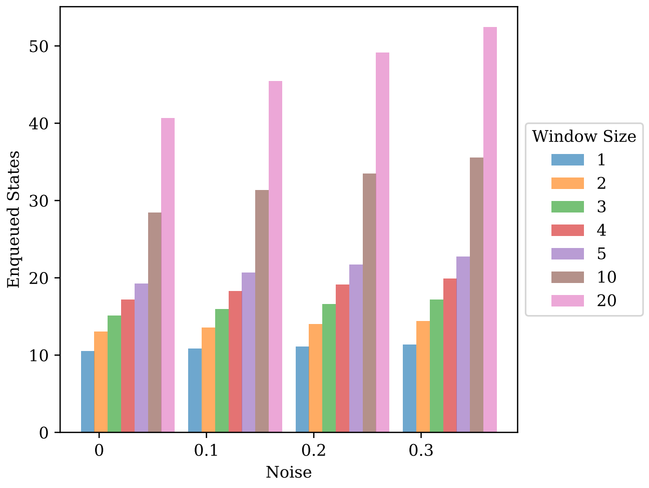


(a) Average number of states enqueued.

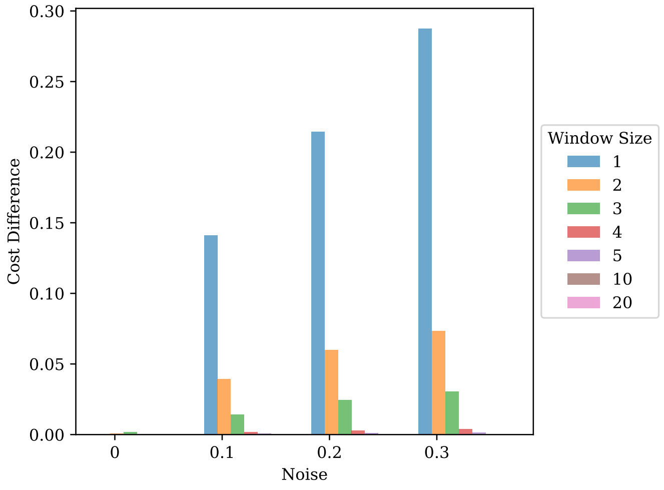


(b) Average queue size.

Figure 7.7: Memory performance per prefix-length ($1 \leq \text{length} \leq 100$).

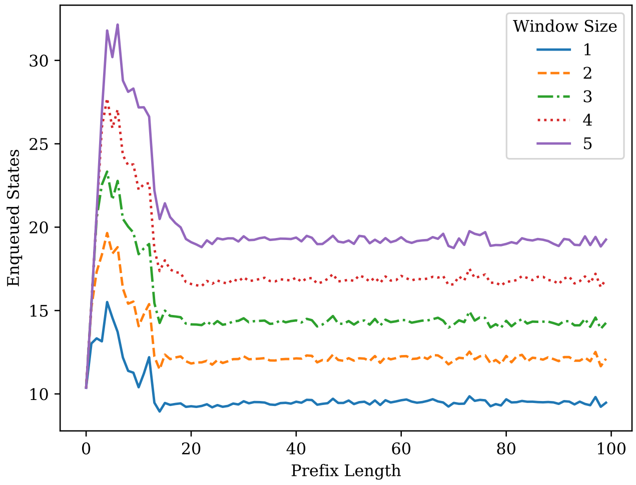


(a) Average number of states enqueued.

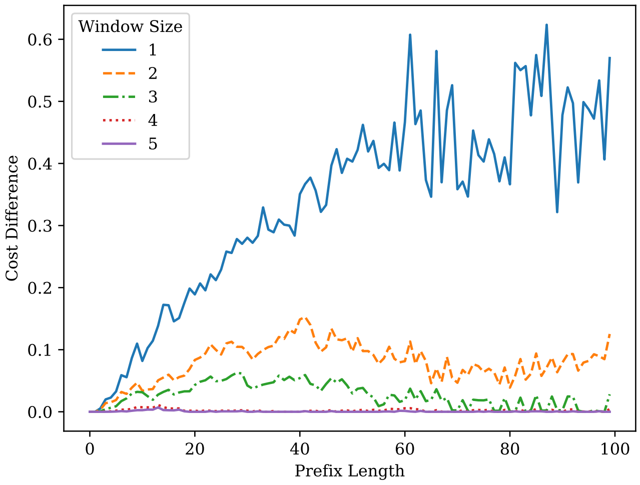


(b) Average cost difference.

Figure 7.8: Memory performance and cost difference with respect to optimal prefix-alignments when using different revert window sizes.



(a) Average number of states enqueued.



(b) Average cost difference.

Figure 7.9: Memory usage and cost difference with respect to optimal prefix-alignments per prefix-length (≥ 1), with different revert window sizes.

In Figure 7.9b we observe similar results as observed in Figure 7.8b. A window size of 1 is simply too small and it seems that, when the prefix length increases, the costs increase linearly. However, when using a window ≥ 2 we observe asymptotic behaviour in terms of approximation error. Again we observe that using a window of at least size 5 leads to negligible approximation errors.

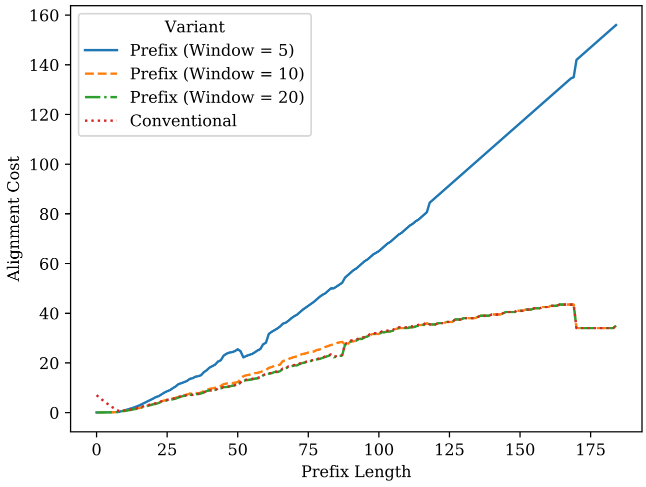
7.3.3 Evaluation using Real Event Data

In this section, we discuss the results of applying incremental alignment calculation based on real event data. We focus on the under/overestimation of true eventual conventional alignment cost, as well as the method's performance. Note that, the under/overestimation of the eventual conventional alignment cost allows us to judge the applicability of the presented algorithm (and its parametrization in particular) as a deviation monitoring tool. Therefore, as a baseline, we compute conventional alignments each time we receive a new event. We use an event log originating from a Dutch hospital related to the treatment of patients suspected of having sepsis [85]. Since we do not have a reference model, we generated one based on a subset of the data. This generated process model still describes around 90% of the behaviour within the event log (computed using conventional alignments). The dataset contains 15.214 events divided over 1.050 cases. Prefix-alignments were computed for 13.775 different events. We plot all results with respect to the aligned prefix length as noise percentages, i.e. used in Figure 7.6 and Figure 7.8, are unknown when using real event data. Finally note, that the distribution of trace length within the data is heavily skewed and has a long infrequent tail. The majority of the trace's length is below 30, hence, figures for prefix lengths above this value refer to a relatively limited set of cases. Nonetheless, we plot all results for all possible prefix lengths observed.

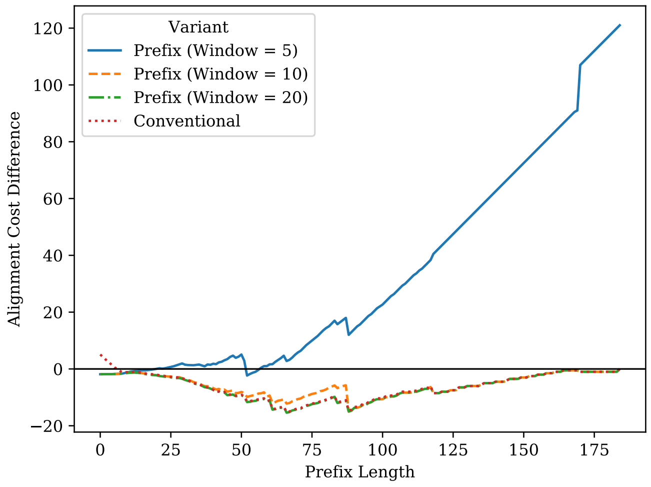
In Figure 7.10 we present results related to computed alignment costs. We show results for using the incremental scheme proposed in this chapter with window sizes 5, 10 and 20, and, the baseline ("*Conventional*"). In Figure 7.10a we show the average absolute alignment costs per prefix length. We observe that using a window size of 5, in general, leads to higher alignment costs. This is explained by the fact that the relatively little window size does not allow us to revert any choices made in previous alignments which consequently does not allow us to find an eventual global optimum. Interestingly, both window sizes 10 and 20 lead to, on average, comparable alignment costs to simply computing conventional alignments. However, at the beginning of cases, i.e. for small prefixes, as expected, computing conventional alignments leads to higher values. In Figure 7.10b, we show the average cost difference with respect to the eventual alignment costs, i.e. after case completion. Interestingly, after initially over-estimating eventual costs, conventional alignments underestimate the costs of conventional alignments quite severely. This can be explained by the fact that partial traces are aligned by a short path of model moves through the model combined with a limited set of activity moves.

In order to quantify the potential business impact of applying the (prefix-)alignment approach we derive several different measures of relevance for the three different window sizes and the baseline. These figures are presented in Table 7.2. To obtain the results as presented, for each received event we define:

- If the difference between the current (prefix-)alignment cost with the eventual alignment cost is zero, and the eventual costs exceed zero, we define a *True Positive*; i.e. we have an exact estimate of non-compliant behaviour.
- If the difference between the current (prefix-)alignment cost with the eventual alignment cost is greater than zero, we define a *False Positive*, i.e. we overestimate non-compliant behaviour.



(a) Average (prefix-)alignment cost.



(b) Average cost difference with respect to eventual optimal conventional alignment.

Figure 7.10: Average cost results per prefix-length (≥ 1), with different revert window sizes.

Table 7.2: Measures of relevance for different window-sized approaches versus computing conventional alignments.

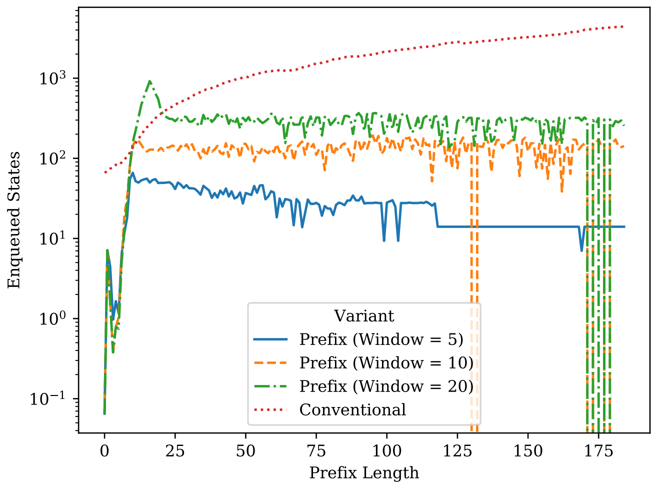
Variant	*Window ~ 5*	*Window ~ 10*	*Window ~ 20*	*Conventional*
True Positive	1517	2060	2110	3179
False Positive	2076	204	13	5215
True Negative	2962	3340	3377	1424
False Negative	7220	8171	8275	3957
Recall	0.17	0.20	0.20	0.45
Specificity	0.59	0.94	0.99	0.21
Precision	0.42	0.91	0.99	0.38
Negative Predictive Value	0.29	0.29	0.29	0.26
False Negative Rate	0.83	0.80	0.80	0.55
Fall-out	0.41	0.06	0.004	0.79
False Discovery Rate	0.58	0.09	0.006	0.62
False Omission Rate	0.71	0.71	0.71	0.74
Accuracy	0.33	0.40	0.40	0.33
F1-Score	0.25	0.33	0.34	0.41

- If the difference between the current (prefix-)alignment cost with the eventual alignment cost is zero, and the eventual costs equals zero, we define a *True Negative*; i.e. we have an exact estimate of the fact that no deviation occurs.
- If the difference between the current (prefix-)alignment cost with the eventual alignment cost is lower than zero, we define a *False Negative*, i.e. we underestimate non-compliant behaviour.

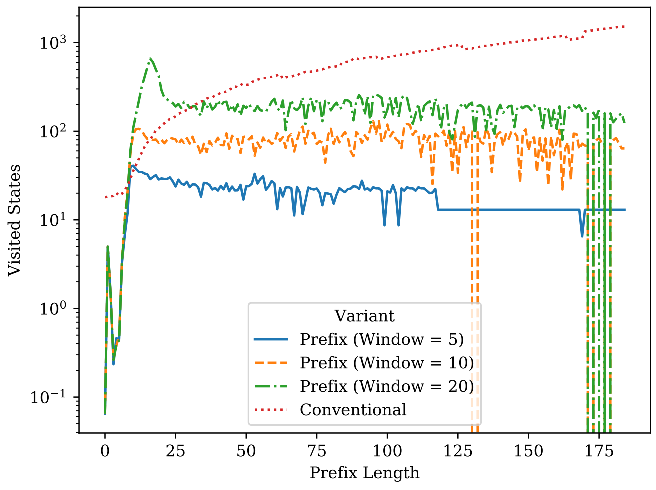
We acknowledge that alternative definitions of True/False Positives/Negatives are possible. Therefore, the results obtained are specific for the definition provided, as well as the data set we used. We observe that computing conventional alignments, for every event received, leads to a better *recall*, i.e. $\frac{TP}{TP+FN}$. This implies that the ratio of correctly observed deviations with respect to neglected deviations is better for the conventional approach. However, using the incremental scheme, leads to significantly higher *specificity* ($\frac{TN}{TN+FP}$) and *precision* values ($\frac{TP}{TP+FP}$). Specifically for window sizes 10 and 20 we observe very high precision values. This, in fact, is in line with 7.1 and, moreover, shows that the results obtained with these window sizes are close to results for an infinite window size. Finally, we observe that the accuracies of window sizes 10 and 20 are comparable and higher than the alternative approaches, i.e. window size 5 and conventional. However, in terms of $F1$-score, simply calculating conventional alignments outperforms using the incremental scheme as proposed.

In Figure 7.11, we show the performance of the different approaches in terms of enqueued states and visited states. Note that, the results presented consider the full incremental scheme, i.e. if we are able to execute a synchronous move directly, queued/visited states equals 0. As expected, using a window size of 5 is most efficient. Window sizes 10 and 20 are less efficient yet for longer prefix lengths, they outperform computing conventional alignments. For window size 20, we do observe peaks in terms of computational complexity for prefix lengths of 10-20. Such peaks can be explained by the relatively inaccurate heuristic used within the A^*-searches performed for prefix-alignment computation. The drops in the chart relate to purely incremental alignment updates. We observe that computational complexity of conventional alignment computation is in general increasing when prefix length increases. The incremental based approach seems not to suffer from this and shows relatively stabilizing behaviour.

Based on the experiments using real hospital data, we conclude that, for this specific dataset,



(a) Average number of states enqueued.



(b) Average number of traversed arcs.

Figure 7.11: Performance results based on the hospital dataset (logarithmic scales).

a window size of 10 is appropriate. As opposed to computing conventional alignments, it achieves precise results, i.e. whenever a deviation is detected it is reasonable to assume that this is indeed the case. Moreover, it outperforms computing conventional alignments in terms of computational complexity and memory usage.

7.4 Related Work

Early work in conformance checking uses token-based replay [102]. The techniques replay a trace of executed events in a process model (Petri net) and add missing tokens if transitions are not able to fire. After replay, remaining tokens are counted and a conformance statistic is computed based on missing and remaining tokens.

Alignments, as discussed in this chapter, were introduced in [6, 13] and have rapidly developed into the de-facto standard for conformance checking. In [3, 93] decomposition techniques are proposed together with computing alignments. Using decomposition techniques greatly enhances computation time, i.e. the techniques successfully apply the divide-and-conquer paradigm, however, the techniques provide lower bounds on conformance checking statistics, rather than computing alignments. More recently, general approximation schemes for alignments, i.e. computation of near-optimal alignments, have been proposed in [111].

In the realm of online conformance checking, limited work is done. In [32, 33] a framework (and corresponding implementation) for online conformance checking is proposed on the basis of explaining the behaviour observed on a stream in terms of an enriched transition system, on the basis of the process model. The authors propose to enrich the reachability graph of the process model in such way that for each possible combination of state in the graph and observable label, there is exactly one transition specified. Such transition potentially relates to modelled behaviour, in which it has zero-costs, in other cases, it relates to deviance and thus has an associated cost. The advantage of the approach is that, given a precomputed extended reachability graph, there are clear bounds on memory and processing time per event. The downside is that the technique merely gives an indication of conformance, and, computing the extended reachability graph is an expensive computational problem.

7.5 Conclusion

The work presented in this chapter is motivated by the observation that applying conventional conformance checking techniques, i.e. in a batch fashion on the basis of event stores as defined in chapter 3, does not lead to accurate results. In particular, *computing conventional alignments*, i.e. the de-facto standard conformance checking approach, does not account for the fact that the behaviour stored within the event store relates to unfinished process instances. As such, the results computed by means of conventional conformance checking techniques potentially lead to falsely rejecting correct process instances.

7.5.1 Contributions

We proposed an online, event stream based, conformance checking technique based on the use of prefix-alignments. The algorithm only performs a state-space search to compute a new prefix-alignment if no direct label- or synchronous move is possible. We presented two techniques to increase the search efficiency of the underlying shortest-path problems solved. The first technique, i.e. deriving prefix-alignment cost upper bounds, preserves optimality and allows for effective state-space pruning. The second technique uses an approximation scheme providing a balance between optimality and memory usage. In our evaluation, we primarily focused on the performance of the underlying shortest path problems solved. Our results show that we are able to effectively prune the state-space by using previously computed alignment results. Particularly in terms of memory efficiency, these results are promising. When using our approximation approach, we observe a linear trend in the amount of memory needed, for increasing window sizes used. However, the approximation error seems to decrease more rapidly, i.e. in a non-linear fashion when increasing window sizes.

7.5.2 Limitations

The aim of the incremental technique presented in this chapter is to compute approximations of alignments, by means of utilizing the concept of prefix-alignments, based on event streams. In particular, we aim at computing these approximations more efficiently with respect to simply computing conventional alignments, whilst at the same time limiting the loss in result accuracy.

In general, we conclude that the use of the technique proposed is justified in cases where computational resources are limited, and/or there is a need for high precision, i.e. we aim at high degrees of certainty when we observe a deviation. In cases where computational complexity is not an urgent issue, and/or high recall is more preferable, one can resort to computing conventional alignments. However, recall that conventional alignments tend to overestimate alignment costs, and thus, are not able to properly detect deviations in early stages of a case. Hence, when resorting to using conventional alignments, a buffering strategy, i.e. we only start aligning trace of which we have observed sufficient behaviour, is advisable. However, note that, to be able to do this, we need some trace-completeness estimator.

In the experiments performed using real data, we observe a certain unpredictability with respect to memory usage/computational efficiency of prefix-alignment computation, i.e. consider the peeks for window size 20 in Figure 7.11. In general, although the search algorithm used in prefix-alignment computation is A^* [68], the practical search performance is, however, equal to the performance of Dijkstra's shortest path algorithm [46]. This is mainly related to the fact that when computing prefix-alignments we need to resort to a rather inaccurate heuristic function. By partially reverting the alignments, combined with applying the upper bound pruning, we are able to reduce the search complexity, however, at the cost of losing accuracy. When computing conventional alignments, we are able to resort to a more accurate heuristic which explains the more predictable computational efficiency trends in Figure 7.11.

7.5.3 Open Challenges & Future Work

The main downside of the approach presented in this chapter, as briefly mentioned in subsection 7.5.2, is the fact that we have little guarantees on the memory bounds and/or processing time of the algorithm. This is mainly due to the fact that the heuristic function that we need to use is not able to use the marking equation internally, i.e. because of the fact that we consider incomplete trace behaviour. It is therefore interesting to study alternative techniques and methods, i.e. in the lines of [33], in which we are able to provide explicit bounds in terms of memory usage and/or processing time.

In this chapter, we primarily focus on computing conformance diagnostics on the basis of prefix-alignments. However, there exist other techniques for the purpose of conformance checking, cf. section 7.4, e.g. on the basis of simple token-based replay. Hence, an interesting direction for future work is a comparison of these different approaches in context of conformance checking oriented monitoring, i.e. to investigate which of these techniques is most accurately predicting non-conformance in a streaming setting.

Another interesting direction for future work, partly related to the aforementioned direction, concerns the development and definition of alternative accuracy measures regarding under/over-estimation of conventional alignments to more accurately measure the indicative behaviour of prefix-alignments. Within the current evaluation, we choose a rather straightforward classification of true/false positives/negatives. In some cases, however, a more advanced classification may be of interest.

Observe that the state of a prefix-alignment, in terms of the underlying reference model, carries some predictive value with respect to case termination. Thus, in cases we do not know explicit case termination, it is interesting to study the effect of using prefix-alignments as a case termination predictor.

Finally, observe that given the model, apart from deciding upon conformance, it is also interesting to consider the certainty of the current conformance value of a prefix-value.

Chapter 8

Discovering Social Networks

Thus far, we have considered event-stream-based techniques for the purpose of process discovery and conformance checking. Moreover, the main focus of the techniques presented, strongly leans towards the control-flow dimension. In this chapter, we turn our focus to the realm of *process enhancement*. In particular, we focus on discovering social networks, describing resources that perform the different events executed in the context of processes, together. We do so, by means of presenting a general framework that defines the problem in a generic way. Moreover, by empirically evaluating different social networks, we show the applicability of the framework, and, social networks in the streaming domain. As we focus on incrementally updating the networks, the techniques presented in this chapter enable us to effectively handle streaming event data.

The contents presented in this chapter are based on the following publication:

S.J. van Zelst, B.F. van Dongen, and W.M.P. van der Aalst. Online Discovery of Cooperative Structures in Business Processes. *CoopIS 2016*, volume 10033 of *Lecture Notes in Computer Science*, pages 210-228. Springer, 2016, `https://doi.org/10.1007/978-3-319-48472-3_12`

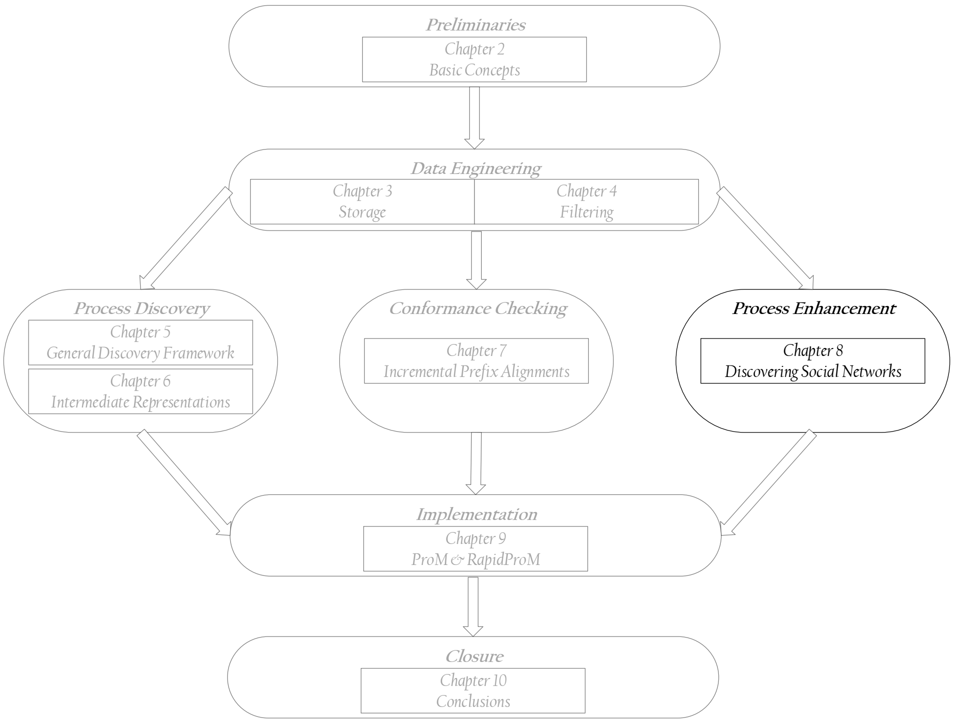


Figure 8.1: The contents of this chapter, i.e. online discovery of social networks from event stream data, highlighted in the context of the general structure of this thesis.

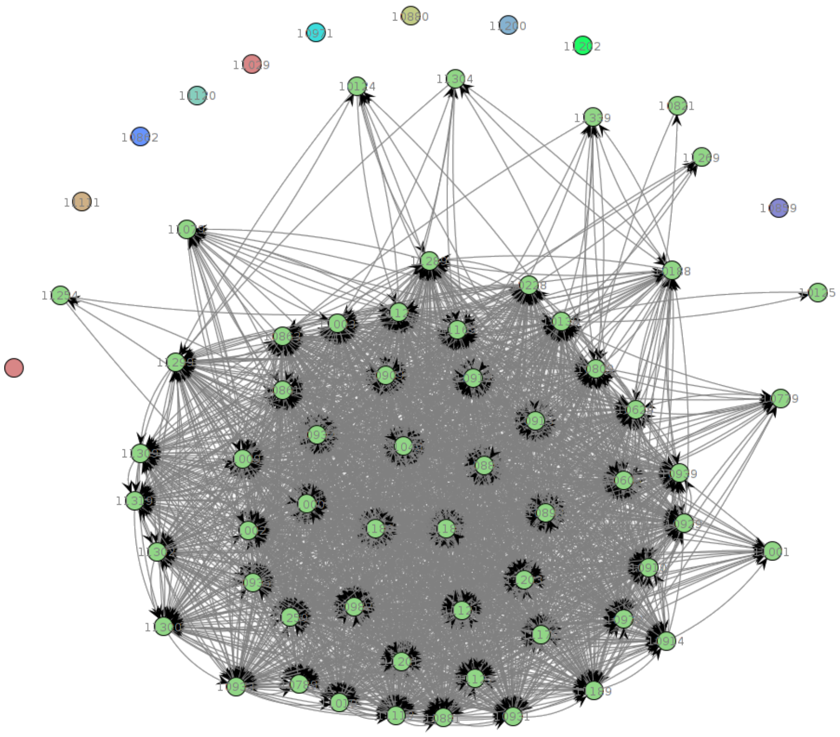


Figure 8.2: Result of calculating a handover-of-work-based social network on the BPI Challenge 2012 [47].

8.1 Introduction

The majority of the techniques presented in this thesis, apart from the general purpose storage techniques presented in chapter 3, focus on the *control-flow perspective*. For example, the filter as presented in chapter 4, uses control-flow to build the automata used within filtering, all discovery algorithms considered in chapter 5 are control-flow based, etc. However, as motivated in chapter 1, and exemplified in Table 1.1, typically many additional data attributes are available in the context of process mining data.

A variety of studies is applicable in the context of online/stream-based-process mining, e.g. bottleneck detection and/or prediction, real-time performance measurement, online KPI-based case routing etc. However, in this chapter, we focus on the discovery of social networks, describing resources that are active, and cooperate together, within a process, i.e. a topic that is well-studied in offline process mining. As an example, consider the *handover-of-work network* based on the BPI Challenge 2012 [47] event log, presented in Figure 8.2. Within the graph, each node represents a resource, whereas each arc represents a *handover of work*. Such handover of work, represented by a directed arc (r_1, r_2), indicates that at some point in time, resource r_2 started working on an activity for a given process instance, after resource r_1 finished an activity for the same process instance. The main problem of applying social network discovery in an

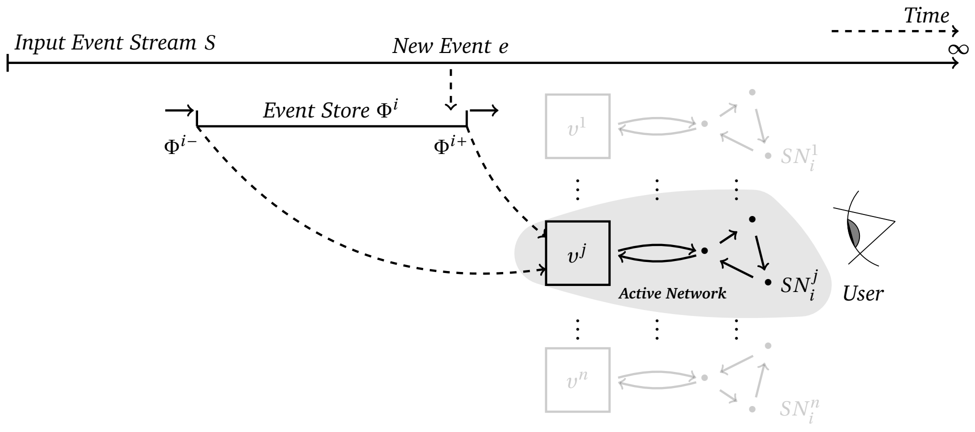


Figure 8.3: Schematic overview of the architecture proposed in this chapter. The active social networkis highlighted in grey and is incrementally updated by means of Φ^{i+} and Φ^{i-}.

offline setting is related to the interpretability of the result. In the case of Figure 8.2, apart from observing that various resources handover work to each other, we are not able to gain any valuable insights. Moreover, we are not able to deduce whether certain resources became more/less active, stopped working etc. Therefore, we aim to discover these social networks on the basis of event streams, as it additionally helps us to visualize the dynamics of the network.

The techniques presented in chapter 3, allow us to iteratively discover networks of resources, i.e. in a batch fashion. Even though this partially solves the aforementioned problem, still this results in a sequence of static resource networks. Therefore, in this chapter, we primarily focus on the design of an *efficient incremental architecture*, that allows us to seamlessly switch between different perspective on resource cooperation within the process. We present a generic architecture, cf. Figure 8.3, which allows us to incrementally build different networks of interacting resources. The framework allows the user to view multiple social networks, based on a single event stream. In this context, we assume that any event $e \in \mathscr{E}$ at least contains information about:

1. Case $c \in \mathscr{C}$ for which the event occurred, i.e. accessed by $\pi_{\mathtt{c}}(e)$, cf. 2.10.
2. Activity $a \in \mathscr{A}$ that was performed, i.e. accessed by $\pi_{\mathtt{a}}(e)$, cf. 2.10.
3. Resource $r \in \mathscr{R}$ that performed the activity, i.e. accessed by $\pi_{\mathtt{r}}(e)$.[1]

We assume that, there are n different types of resource networks that we aim to discover on the basis of event data. The user inspects a social network SN_i^j at time i, for some social network type identified by value $1 \leq j \leq n$. New events are incorporated within the network as fast as possible, i.e. in a real-time and incremental fashion. Such an update is performed by the corresponding social network update function v^j, which uses Φ^{i+} and Φ^{i-} to update the resource network. The information stored within the event store Φ allows the user to switch, after having received i events, the current network SN_i^j to some other type of social network

[1]Here, we let $\mathscr{R}$ denote the universe of resources.

$SN_i^{j'}$, e.g changing from a *handover of work* network to a *reassignment* network. As we use the event store as a whole, a property of the framework is that such switch does not need any form of *warm-up period*, i.e. after initialization of the network based on Φ_i, new events are again incorporated in the network in a real-time fashion.

The remainder of this chapter is organized as follows. In section 8.2, we generically introduce the notion of social networks, and provide a concrete instantiation. In section 8.3, we describe the proposed architecture for online social network discovery, i.e. the architecture. In section 8.4, we evaluate the scalability of a collection of specific resource networks. Finally, section 8.6 concludes this chapter.

8.2 Social Networks

In this chapter, we primarily focus on the construction of social networks as defined in [9], on the basis of event streams. We therefore first present a general definition of social networks, after which we illustrate different types of instantiations of such a network, from a streaming data perspective. In essence, all social network metrics are defined over pairs of resources, e.g given a certain pair of resources, what is the value for the *handover of work metric*?, how strong is the *working together metric*?, what is the *Minkowski distance* for the joint case metric of two resources? etc.

Definition 8.1 (Social Network Metric). *Let $\mathscr{R}$ denote the universe of resources. A social network metric μ is a function $\mu : \mathscr{R} \times \mathscr{R} \rightarrow \mathbb{R}_{\geq 0}$.*

Based on a social network metric, we define social networks, which are formally represented by means of (un)directed graphs. Whether such a graph is directed or not, depends on whether the metric is symmetric, i.e. if it is the case that, by definition of the metric, $\forall r_1, r_2 \in \mathscr{R} \left(\mu(r_1, r_2) = \mu(r_2, r_1)\right)$, it can be represented by undirected graph. Within a social network, the *active resources* of the process, i.e. as described within the underlying event stream, define the vertices. The metric itself defines the edges of the network, i.e. whenever we have a pair $(r_1, r_2) \in \mathscr{R} \times \mathscr{R}$, with $\mu(r_1, r_2) > 0$, there is a corresponding edge within the graph.

Definition 8.2 (Social Network). *Let $R \subseteq \mathscr{R}$ denote a set of resources, let $\mu : \mathscr{R} \times \mathscr{R} \rightarrow \mathbb{R}_{\geq 0}$ be a social network metric and let $E \subseteq R \times R$. A social network SN is an (un)directed graph $SN = (R, E)$, where:*

$$\mu(r_1, r_2) > 0 \Leftrightarrow (r_1, r_2) \in E \tag{8.1}$$

As an example of a social network metric, we consider the notion of the *handover of work* metric, i.e. a metric describing networks such as the social network presented in section 8.1, in Figure 8.2, in more detail. Consider an event store (case-projection) $\Phi^i_{\mathscr{C}}$ and a trace σ within the store, i.e. $\sigma \in \Phi^i_{\mathscr{C}}$. We define the absolute number of handovers from a given resource r_1 to resource r_2 as:

$$|r_1 \succ_\sigma r_2| = \sum_{i=1}^{|\sigma|-1} \begin{cases} 1, \text{ if } \pi_{\mathtt{r}}(\sigma(i)) = r_1 \wedge \pi_{\mathtt{r}}(\sigma(i+1)) = r_2 \\ 0, \text{ otherwise} \end{cases} \tag{8.2}$$

The relation $|r_1 \succ_\sigma r_2|$ denotes the number of times a handover occurs from resource r_1 to resource r_2 within σ. In general $|r_1 \succ_\sigma r_2|$ describes a *local trace-based metric*. To express handover of work in a global, event store based metric, we define $r_1 \rhd^i r_2$:

$$r_1 \rhd^i r_2 = \left(\sum_{\sigma \in \Phi^i_{\mathscr{C}}} |r_1 \succ_\sigma r_2| \right) \Big/ \left(\sum_{\sigma \in \Phi^i_{\mathscr{C}}} (|\sigma| - 1) \right) \tag{8.3}$$

Observe that, for the aforementioned relations, the traces considered are assumed to be of length greater than 1.

In [9], a more generalized definition is presented that allows us to incorporate relations between resources at longer distances. However, in this chapter, we primarily focus on distances of length 1, i.e. a direct succession of work, as it is most suitable in terms of performance in the context of online streams of events.

8.3 Architecture

In this section, we present a generic architecture, which allows us to discover and visualize a live view of different social networks, based on event streams. We discuss the main components it comprises of, and in particular, we focus on the incremental update complexity of social network metrics in a streaming context.

Given a stream of events, we are interested in constructing different types of resource networks. Since multiple social network metrics are defined, we aim at designing an architecture that allows us to discover several different networks on an event stream. Moreover, the user needs to be able to seamlessly switch between different networks. Hence, we present a general architecture, cf. Figure 8.3, which describes this in a generic way. The architecture comprises of the following main components:

1. An underlying event store Φ^i, cf. chapter 3, which temporarily stores the events emitted on the event stream.
2. Resource network builders v^1, v^2, ..., v^n, which comprise of two sub-functions:
   (a) Initialize a resource network, based on the event data stored in the event store Φ^i.
   (b) Perform real-time incremental updates on the resource network.
3. Corresponding resource networks SN^1, SN^2, ..., SN^n, as defined by social network builders v^1, v^2, ..., v^n.

In general, we assume only one network builder is active at a specific point in time (represented by v^j in Figure 8.3). In the remainder of this section, we therefore discuss the social network builders in more detail. In particular, we focus on the computational feasibility of different metrics in terms of incremental updates.

8.3.1 Networks & Builders

We maintain knowledge about the process instances observed so far in memory, i.e. within the underlying event store Φ, to be able to switch from one social networkto the other. Hence, as indicated, the network discovery components v^1,v^2,...,v^n, mainly consist of the following two separate tasks:

1. *Initialization* of the network based on the current state of Φ^i.
2. *Update* of the network based on a new event flowing in, as well as events that are removed due to receiving the new event.

The first step, i.e. initialization, equals the conventional computation of the resource network, since for any $i \in \mathbb{N}$, Φ^i describes a finite sequence of events and thus an event log. The second step, i.e. (incremental) updating of the network, is of particular interest. Consider that we obtain some network SN_{i-1}, based on Φ^{i-1}, and, we receive i^{th} event $e \in \mathscr{E}$. The new event either introduces or updates a network metric value for a pair of resources. Also, the metric value for a pair of resources usually depends on a global property, i.e. the divisor of $\rhd^i$, hence the new event potentially also affects the metric value of other pairs of resources within the network. Thus, network SN_i, based on Φ^i, is very likely to differ from the previous network SN_{i-1}, i.e. a new handover is added/removed due to the new event/removed data. From an implementation perspective, we need to refresh the internal data structures that maintain the network after each new event. The complexity of this operation is expected to grow in terms of the network size. For most metrics, we additionally need to design supporting data structures that allow us to recompute the actual metric value. For example, to be able to compute the $\rhd^i$ metric, for each resource pair, we need to maintain a denominator and a divisor. The denominator is resource pair specific, whereas the divisor is maintained globally.

Some supporting data structures turn out to be computable in an incremental fashion. If the incremental computation of the data structures is inexpensive, we are able to update them in a real-time fashion. However, some metrics do not support inexpensive incremental updates of their supporting data structures, i.e. we potentially need to recompute them from scratch after a new event. In such cases, we need a more periodic update scheme of the network in order to maintain the network in a real-time fashion. Hence, we propose two network update strategies: *right-time*, i.e. at the user's request, and *real-time*, i.e. updating after each event, which defines how we need to update our network over time.

8.3.2 Real-Time versus Right-Time Update Strategies

When we reconsider the *handover of work* metric as presented in section 8.2, we design the supporting data structures that allow us to maintain the metric as follows. We maintain a counter $\mathtt{count}_{i,\succ} : \mathscr{R} \times \mathscr{R} \rightarrow \mathbb{N}_0$, that maintains the denominator of $r_1 \rhd^i r_2$, i.e.

$$\mathtt{count}_{i,\succ}(r_1, r_2) = \sum_{\sigma \in \Phi^i_{\mathscr{C}}} |r_1 \succ_\sigma r_2| \tag{8.4}$$

Moreover, since the divisor of $r_1 \rhd^i r_2$ has the same value for all possible combinations of r_1 and r_2 we maintain it as a single integer $\mathtt{divisor}_i \in \mathbb{N}_0$, i.e.

$$\mathtt{divisor}_i = \sum_{\sigma \in \Phi^i_{\mathscr{C}}} (|\sigma| - 1) \tag{8.5}$$

Initially we have $\mathtt{count}_{i,\succ}(r_1, r_2) = 0,\ \forall r_1, r_2 \in \mathscr{R}$ and $\mathtt{divisor} = 0$. After receiving several events on the event stream, i.e. we have $\mathtt{divisor} \neq 0$, we are able to compute the handover of work metric for resource pair (r_1, r_2) as follows:

$$r_1 \rhd^i r_2 = \frac{\mathtt{count}_{i,\succ}(r_1, r_2)}{\mathtt{divisor}_i} \tag{8.6}$$

Assume that after observing the event stream for a while, we receive the i^{th} event e, such that we have $\Phi^{i-1}_{\mathscr{C}}(S, \pi_{\mathtt{c}}(e)) = \sigma'$, where $\sigma' \neq \epsilon$. Moreover, we assume that no event is removed from the event store yet. Thus, at time i, we have $\Phi^i(S, \pi_{\mathtt{c}}(e)) = \sigma' \cdot \langle e \rangle$. Observe that for any

$r_1, r_2 \in \mathscr{R}$, for the denominator of $r_1 \rhd^i r_2$, we have:

$$\sum_{\sigma \in \Phi^i_{\mathscr{C}}} |r_1 \succ_\sigma r_2| = \left(\sum_{\sigma \in \Phi^i_{\mathscr{C}} \setminus \{\sigma' \cdot \langle e \rangle\}} |r_1 \succ_\sigma r_2| \right) + |r_1 \succ_{\sigma' \cdot \langle e \rangle} r_2| \tag{8.7}$$

Since we assume that no events are removed from the event store, we deduce:

$$\sum_{\sigma \in \Phi^i_{\mathscr{C}} \setminus \{\sigma' \cdot \langle e \rangle\}} |r_1 \succ_\sigma r_2| = \sum_{\sigma \in \Phi^{i-1}_{\mathscr{C}} \setminus \{\sigma' \cdot \langle e \rangle\}} |r_1 \succ_\sigma r_2| \tag{8.8}$$

We furthermore have, $\forall r_1, r_2 \in \mathscr{R}$:

$$|r_1 \succ_{\sigma' \cdot \langle e \rangle} r_2| = |r_1 \succ_{\sigma'} r_2| + \begin{cases} 1 & \text{if } \pi_{\mathtt{r}}(e) = r_2 \wedge \pi_{\mathtt{r}}(\sigma'(|\sigma'|)) = r_1 \\ 0 & \text{otherwise} \end{cases} \tag{8.9}$$

Thus, the only action we need to perform is increasing the value of $\mathtt{count}_{i,\succ}(r, r')$ by one, where r denotes the resource that executed *the last event* of σ', and r' is the resource executing the newly received event. Furthermore, note that, for the divisor, we are able to deduce $\mathtt{divisor}_i = \mathtt{divisor}_{i-1} + 1$.

If we drop the assumption that no events are removed from the event store, we need to reduce the values of $\mathtt{count}_{i,\succ}$ and $\mathtt{divisor}$ accordingly. In general, we need to reduce the counters of those cases that are part of Φ^{i-}. Furthermore, we need to reduce the divisor by $|\Phi^{i-}|$, unless a removed event relates to a trace of length 1, as in such case, it does not contribute to the divisor (nor the counter of the corresponding case). Hence, the $\rhd$ metric is computable incrementally, as we are able to update the underlying counters in near constant time. As such, the $\rhd$ metric is an example of a metric that we classify as a *real-time metric*. We define social network metrics to be real-time if it is possible to update the metric's supporting data structures by means of an incremental update. However, in case the divisor value changes, we need to recompute all metric values for those $r_1, r_2 \in \mathscr{R}$ with a non-zero $\mathtt{count}_{i,\succ}$ value. Thus, in worst-case this operation has complexity $O(|R|^2)$.

There are examples of social network metrics that do not meet the real-time requirement. An example of such metric is the *boolean-causal variation* of the handover of work metric [9]. Given a sequence σ of events, we define relation $r_1 \succeq_\sigma r_2$ which specifies that resource r_1 hands over work of a case to resource r_2, given that the underlying corresponding executed activities are in a causal relation. In this context, we assume such causal relation to be similar to the causal relation used within the alpha miner, i.e. based on the directly follows relation. Moreover, relation $r_1 \succeq_\sigma r_2$ does not count the number of occurrences of such handover, i.e. it only captures the fact that such handover occurred at least once in the trace.

$$r_1 \succeq_\sigma r_2 = \begin{cases} 1 & \text{iff } \exists 1 \le i < |\sigma| \left(\pi_{\mathtt{r}}(\sigma(i)) = r_1 \wedge \pi_{\mathtt{r}}(\sigma(i+1)) = r_2 \wedge \pi_{\mathtt{a}}(\sigma(i)) \rightarrow \pi_{\mathtt{a}}(\sigma(i+1)) \right) \\ 0 & \text{otherwise} \end{cases} \tag{8.10}$$

We now define a corresponding global social network metric $r_1 \unrhd^i r_2$ that captures this for event streams. Let i denote the index of the latest received event on the event stream, then:

$$r_1 \unrhd^i r_2 = \left(\sum_{\sigma \in \Phi^i_{\mathscr{C}}} r_1 \succeq_\sigma r_2 \right) \Big/ |\{c \in \mathscr{C} \mid \Phi^i_{\mathscr{C}}(S, c) \neq \epsilon\}| \tag{8.11}$$

Maintaining the divisor is in this case trivial as it just reflects the number of process instances that have a non-empty entry within the underlying event store. For the denominator, we again maintain $\texttt{counter}_{i,\succeq} \colon (\mathscr{R} \times \mathscr{R}) \rightarrow \mathbb{N}_0$, which represents the denominator of the $\trianglerighteq$ metric. However, to maintain the metric we need to additionally keep track of the causal relations present within the traces. To maintain the causal relations, we maintain a counter $\texttt{counter}_{i,>} \colon (\mathscr{A} \times \mathscr{A}) \rightarrow \mathbb{N}_0$ that maintains the $>$ relation. Based on $\texttt{counter}_{>}$ we maintain a set $X_{\rightarrow} \in \mathscr{P}(\mathscr{A} \times \mathscr{A})$ that describes causal pairs. In case we have $\texttt{counter}(a,a') = \texttt{counter}(a',a) = 0$, or $\texttt{counter}(a,a') > 0 \wedge \texttt{counter}(a',a) > 0$ then $(a,a'),(a',a) \notin X_{\rightarrow}$. However in case $\texttt{counter}(a,a') > 0 \wedge \texttt{counter}(a',a) = 0$ then $(a,a') \in X_{\rightarrow}, (a',a) \notin X_{\rightarrow}$.

The main problem that arises when maintaining a causal-flavoured social network metric is related to changes of the $X_{\rightarrow}$ set and its impact on the metric(s). Consider, the following four scenario's, which highlight the aforementioned problem.

1. We receive a new event e.

   (a) The event generates a *new causal relation* (a,a'), such that a is executed by some resource r and a' is executed by resource r'.

   In this scenario, we are guaranteed that there is no other trace that contains an activity a directly followed by activity a'. Moreover, this is also the case for the reverse relation, i.e. (a',a). Hence, the only action we need to take here is to check whether the trace σ related to the new event already contributed to the $\succeq_{\sigma}$-relation on r and r'. If so, we do not update $\texttt{counter}_{i,\succeq}(r,r')$, otherwise, we increase $\texttt{counter}_{i,\succeq}(r,r')$ with one.

   (b) The event *invalidates an existing causal relation* (a,a').

   In this scenario, we observe that the event generates a directly follows pair (a',a) which is not yet observed, yet its reverse, i.e. (a,a'), is already observed. Thus, we are guaranteed that the corresponding events contributed to the $\succeq_{\sigma}$-relation for different resources. Hence, we need to traverse all traces in $\Phi^i_{\mathscr{C}}$ and potentially reduce some $\texttt{counter}_{i,\succeq}$-values.

2. An event is removed from $\Phi^i_{\mathscr{C}}$, i.e. $e \in \Phi^{i-}$.

   (a) The removal of the event generates a *new causal relation* (a,a').

   This implies that there is no other occurrence of the (a,a') directly follows relation any more, yet there are occurrences of the reverse i.e. (a',a). Hence, like in case 1.(b), we need to traverse all traces in $\Phi^i_{\mathscr{C}}$ and potentially reduce some $\texttt{counter}_{i,\succeq}$-values.

   (b) The removal of the event *invalidates an existing causal relation* (a,a').

   This implies that there is no more occurrence of the (a,a')-, nor the (a',a) directly follows relation. Hence, like in case 1.(a), we only need to check whether the removal of the event triggers a reduction in the corresponding $\texttt{counter}_{i,\succeq}$-relation.

As indicated, there are two cases that have a negative computational impact with respect to the social network. In case we receive a new event which leads to invalidating an existing causal relation, we need to traverse the whole event store, or any form of underlying data structure, to update the maintained $\texttt{counter}_{i,\succeq}$-values. Symmetrically, if due to event removal, a new causal relation is created, we again need to traverse the event store as a whole. The effect of such traversal is the potential removal/addition of edges within the resource network. Moreover, several metric values are likely to be changed. Hence, a change in the underlying causal structure has a global impact, both from a computational perspective as from a social network perspective.

The $\trianglerighteq$ metric is an example of a *right-time* metric, i.e. instead of continuously maintaining the supporting data structures, it should be recomputed at regular intervals, or, at the user's request. In section 8.4, we empirically assess the computational complexity of both metrics $\triangleright$ and $\trianglerighteq$. We also assess how often we need to recompute the internal data structures of $\trianglerighteq$ based on real data, i.e. how often does a causal change happen?

8.4 Evaluation

In this section, we assess the scalability of the two types of metrics discussed in this chapter, i.e. $\triangleright$ and $\trianglerighteq$. For the experiments we created an event stream based on the BPI Challenge 2012 event log [47]. In total, the event log contains 262.200 events. Out of these events, 244.190 events actually contain resource information, i.e. describing what resource executed the corresponding activity. Within the event stream, events are ordered based on timestamps. Using the event stream, for each metric we measured:

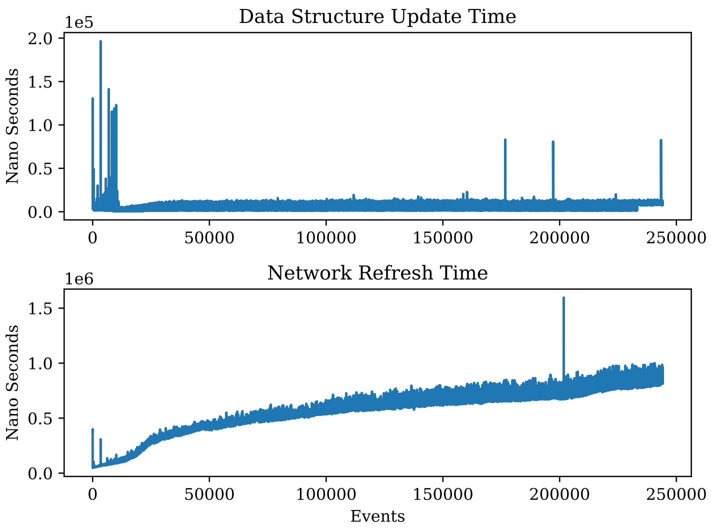
1. Time needed to refresh the supporting data structure(s) (in *nano-seconds*)
2. Time needed to refresh the whole network (in *nano-seconds*)
3. Total memory consumption of the data structure(s) (in *bytes*)
4. The global network size (in the *number of edges*).

Apart from the four metrics, we investigate and compare two different scenario's.

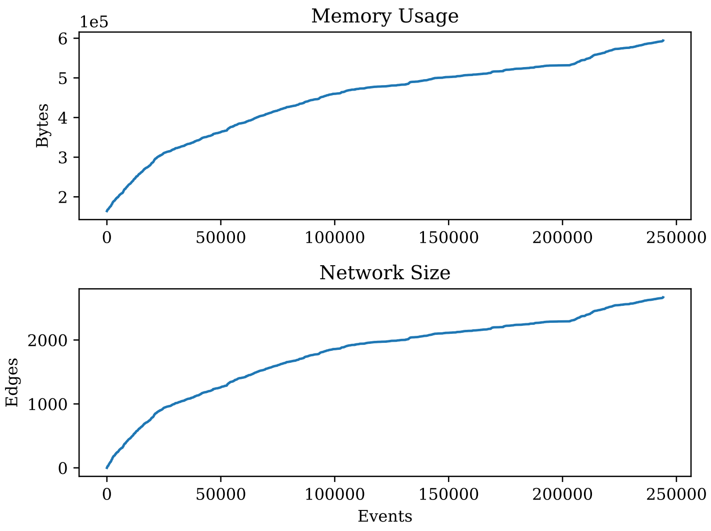
1. No restriction on the event store's available memory.
2. As an underlying event store, we us a forward decay model [42], with an *exponential decay function* with a *decay rate* of 0.01 and threshold for removal of 0.01. The events in the event store are represented by a prefix-tree, in which the nodes represent pairs of the form $(a, r) \in \mathscr{A} \times \mathscr{R}$. We *do not* extend the memory life of the process instances within the prefix-tree, i.e. as described in chapter 3.

To reduce the effects of outliers in terms of run-time measurements, e.g. caused by the *java garbage collector*, we ran each experiment ten times, removed the minimal and maximal measurement values, and, averaged the remaining eight measurement values. Additionally, in some charts the y-axes are restricted in order to highlight the overall trends, instead of showing outliers. Finally note that, as the implementation is of a prototypical fashion, the evaluation results are primarily indicative in terms of trends.

In Figure 8.4, on page 213, the results of discovering the $\triangleright$ metric, without any restriction on the available memory, are depicted. In Figure 8.4a, we present the time-performance of the approach, whereas in Figure 8.4b, we present memory-related performance. Note that, since we do not restrict memory usage, events are never removed from the event store. As a result, the memory usage is steadily increasing. The data structure update time only consists of incrementally updating the divisor and the denominator values of the resource pairs affected by the update. We observe that initially there are some high values in terms of update time with respect to the overall, relatively constant trend. This relative constant trend, is likely to be related to the initialization of the underlying data structure. Then, after a short period of increasing update time values, the update time seems to stabilize. When comparing the network size with the memory usage, we observe that they follow the exact same pattern. This makes sense since the larger the network, the more absolute and relative values we need to store. The time to refresh the network follows the same shape. This is as expected as we need to calculate more relative values as the network size increases.



(a) Time performance, measured in terms of internal data structure update time and network refresh time.



(b) Memory performance, measured in terms of the global memory usage and social network size.

Figure 8.4: Results of discovering the $\triangleright$ metric without memory restrictions.

In Figure 8.5, on page 215, the results of discovering the $\triangleright$ metric, with finite memory restriction, are depicted. Similarly to Figure 8.4, in Figure 8.5a, we present the time-performance of the approach, whereas in Figure 8.5a, we present memory-related performance. In this case, we observe that memory usage is fluctuating. When we compare the time needed for data structure updates, we observe that this is now slightly higher than the time performance reported in Figure 8.4a. This is explained by the fact that when restricting memory to be finite, i.e. we also need to account for the removal of events. Again, the network refresh rate follows the behaviour of the memory/network size. Due to the restrictions on memory usage, the refresh rate of the network is now comparable to the data structure update time.

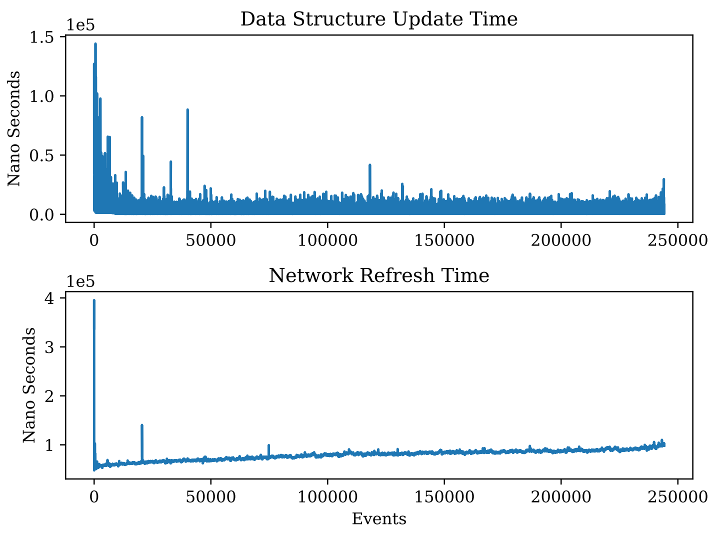
As explained in subsection 8.3.2, we consider the $\trianglerighteq$ as a *right-time* metric, as we have no control over the degree of the applicability of incremental updates. However, it is interesting to assess to what degree such a metric is still feasible from an online/real-time perspective. In Figure 8.6, on page 216, we depict the results of computing the $\trianglerighteq$ metric with restricted memory.

The measured data structure update time, again shows some outliers, yet overall shows a relative constant trend. Upon inspection, when ignoring outliers, we observe that on average the update time is 10-fold with respect to the measured times for the $\triangleright$ metric, cf. Figure 8.4 and Figure 8.5. This is explained by the fact that we need to perform significantly more bookkeeping, i.e. we need to keep track of the underlying causalities as well. Interestingly, we observe quite some deviations in the network refresh time, indicating that at some times, the network needs to be recomputed due to a change in the underlying causalities. We observe that the network size again shows a similar pattern as the measured memory, however, it seems to fluctuate more with respect to the measurements of the $\triangleright$ metric. This, again, is explained by the fact that we need to additionally keep track of the underlying causalities. These results indicate that, even though identified as a right-time metric, the $\trianglerighteq$ metric can also be adopted in real time scenarios, i.e. depending on the velocity of the stream. The memory footprint, in particular, shows an acceptable bounded trend.

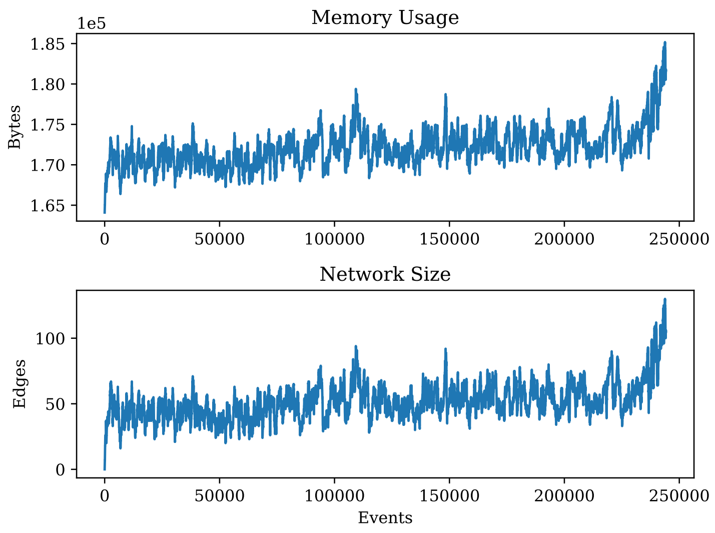
8.5 Related Work

In this section, we cover related work in the area of online/offline organizational analysis techniques in process mining.

In [9] a collection of social network metrics is defined in the context of conventional process mining, i.e. metrics based on event log data. The work can be regarded as one of the foundational works of the application of social network analysis in the context of process mining. In [107] the authors extend the work of [9] with organizational model mining and information flow mining. In [62] the authors identify that applying conventional techniques as defined in [9, 107] result in complex networks that are not easy to understand. The authors propose to solve this by applying hierarchical clustering on the social networks to form clusters of similar resources. In [98] an extensible framework is proposed that allows for extracting resource behaviour time series. It allows process owners to visualize their resource's performance over time, using event logs as an objective source of information. Recently, in [17], Appice et al. propose a method to analyze evolving communities within event logs. The method transforms the event log into a finite stream of events, and, subsequently divides the stream into windows. The method computes a resource community for each window and assesses the changes within the communities of successive windows.

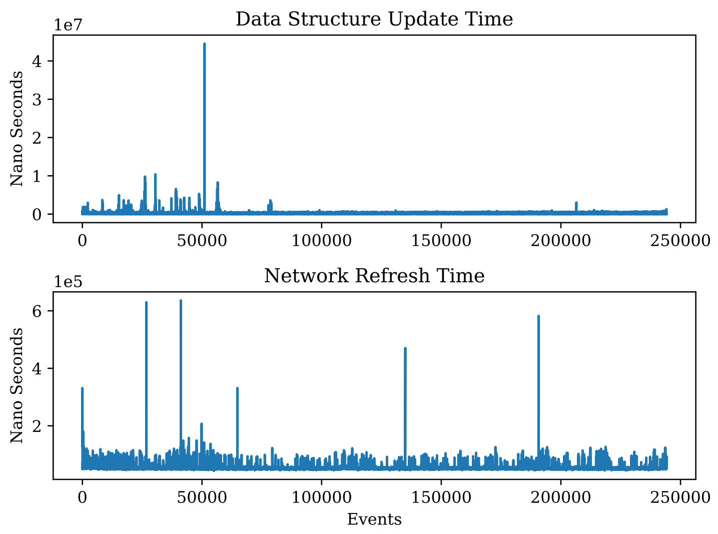


(a) Time performance, measured in terms of internal data structure update time and network refresh time.

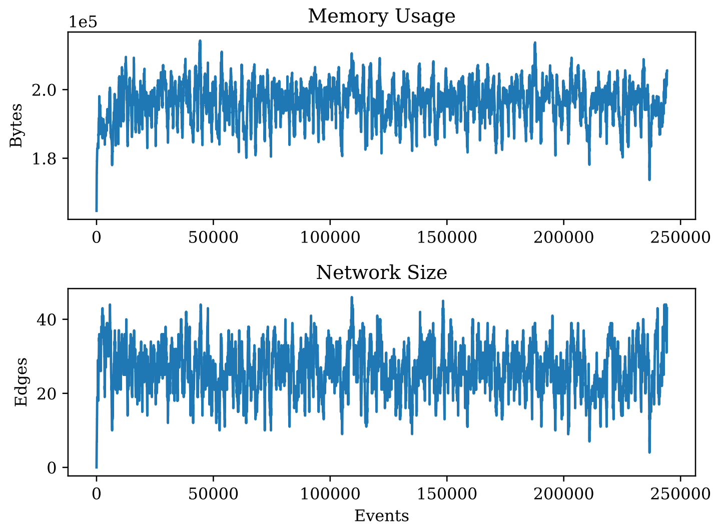


(b) Memory performance, measured in terms of the global memory usage and social network size.

Figure 8.5: Results of discovering the ▷ metric with memory restrictions.



(a) Time performance, measured in terms of internal data structure update time and network refresh time.



(b) Memory performance, measured in terms of the global memory usage and social network size.

Figure 8.6: Results of discovering the $\trianglerighteq$ metric with memory restrictions.

8.6 Conclusion

In this section, we conclude the work presented in this chapter. We discuss the contributions, limitations and provide directions for future work.

8.6.1 Contributions

In this chapter, we presented a generic framework, which allows us to discover several social networks on the basis of event streams. The architecture in generic and in principle allows for many different types of analysis based on online/real-time event stream data. A prototypical implementation of the architecture, including several different social network metrics, is available in the ProM framework. Moreover, it enables us to gain insights in social networks over time, rather than providing one monolithic view. Due to the assumptions on the data, i.e. event streams might be infinite, the architecture additionally allows us to analyse event logs that exceed a computer's physical memory. Therefore, it can be used to speed up the analysis of large event logs and is not limited to streams.

Within the experiments performed, we in particular focus on different types of *handover of work* networks. The experiments show that we are able to compute social networks in an event stream setting. We have shown that there are social network metrics suitable for real-time incremental update strategies. For these metrics, updating of the supporting data structures converges to a constant amount of time. The time needed to update the social network, however, grows in terms of the size of the network. Additionally, we have shown that limiting the available memory has a positive impact both on the use of memory as on the network refresh times. However, as a consequence of the limited available memory, we need to remove cases, which slightly increases the data structure update times.

8.6.2 Limitations

The framework described in this chapter covers the use of general event stores, i.e. it is not tied to a specific instantiation of an event store. In some cases, however, using a specific type of event store is likely to reduce the computational complexity of some types of networks. Observe that when we use a *sliding window*, once it is full, we know that $|\Phi^{i+}| = |\Phi^{i-}| = 1$, i.e. one event flows in, one flows out. Observe that, in such case, the $\texttt{count}_{i,>}$-value of at most two resource pairs changes. However, what is more interesting, is the fact that the divisor potentially remains the same. In the case an event arrives and another event is dropped that are both part of a trace that has length > 1, the divisor does not change. Observe that this is very beneficial for the computational complexity of computing the network, as in such case, only two arcs of the network attain a slightly different value.

8.6.3 Open Challenges & Future Work

The time needed to refresh the network is strongly related to the network size. However, in some cases, the effect of an increase in a metric's denominator/divisor might have little effect on the value of the metric, e.g. $\frac{1.001.337}{2.133.700} = 0,47 \approx \frac{1.001.338}{2.133.700}$. An interesting direction for future work is to quickly detect whether the network should be recalculated.

In this chapter, we have primarily focussed on the feasibility of adopting social network metrics in an event stream setting. To gain more value out of stream-based social network analysis, an interesting direction for future work is the assessment of different visualization methods of the evolution of the networks. Also, extensions such as (online) community detection etc. are of interest.

Chapter 9

Implementation

The techniques presented in this thesis have a corresponding implementation in the widely used process mining framework `ProM` [113]. Moreover, the majority of these implementations have been ported to `RapidProM` [7], i.e. an extension of the RapidMiner (`http://rapidminer.org`) platform for designing and executing data science analysis workflows. In this chapter, we describe the implementations related to the techniques presented in this thesis in more detail. We discuss how the implantations allow users to use process mining in an event stream context, as well as how to conduct (large scale) experiments.

The contents presented in this chapter are based on the following publications:

S.J. van Zelst, Andrea Burattin, B.F. van Dongen, and H.M.W. Verbeek. Data Streams in ProM 6: A Single-node Architecture. *BPM (Demos) 2014*, volume 1295 of *CEUR Workshop Proceedings*, pages 81 . 2014, `http://ceur-ws.org/Vol-1295/paper6.pdf`

S.J. van Zelst, B.F. van Dongen, and W.M.P. van der Aalst. Know What You Stream: Generating Event Streams from CPN Models in ProM 6. *BPM (Demos) 2015*, volume 1418 of *CEUR Workshop Proceedings*, pages 85–89 . 2015, `http://ceur-ws.org/Vol-1418/paper18.pdf`

W.M.P. van der Aalst, A. Bolt, S.J. van Zelst. RapidProM: Mine Your Processes and Not Just Your Data. *(Informal)*, abs/1703.03740 *CoRR*. 2017, `http://arxiv.org/abs/1703.03740`

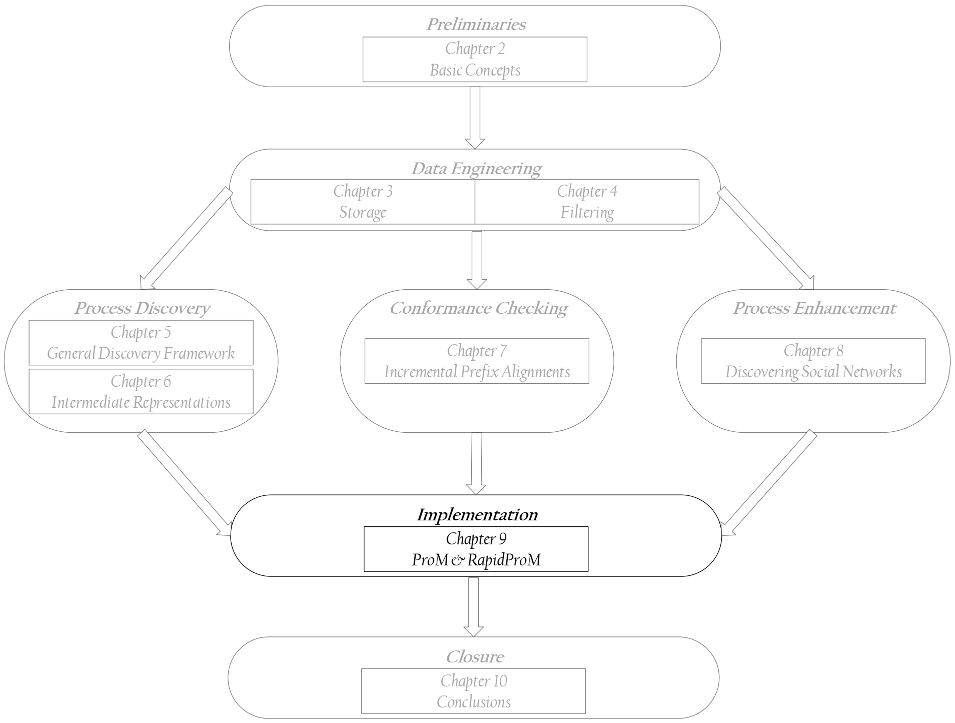


Figure 9.1: The contents of this chapter, i.e. the implementation of the various algorithms, techniques and frameworks proposed in this thesis, highlighted in the context of the general structure of this thesis.

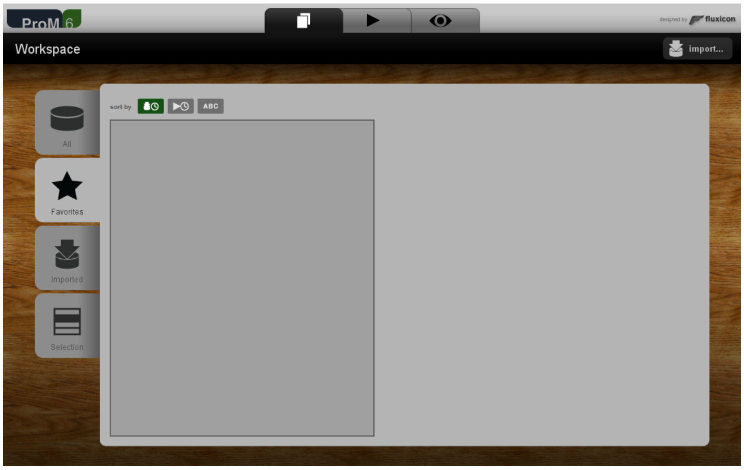


Figure 9.2: Screen shot of the ProM Framework [113].

9.1 Introduction

Within the process mining community, major efforts and time are spent in the actual development of implementations corresponding to the algorithms developed in numerous academic works. The development of the ProM framework [113], the de-facto standard academic process mining tool, to which several researchers contribute worldwide, can be regarded as a stimulating source of these efforts. ProM is a general purpose tool, which allows us to import event logs, discover process models, compute conformance checking statistics etc. Consider Figure 9.2 in which we provide a screen shot of the ProM framework. For most techniques discussed in this thesis, a corresponding implementation is developed in ProM, which is used to obtain the various results as presented in the *Evaluation* sections of the different chapters. Furthermore, some of the techniques are additionally ported to the process mining extension of RapidMiner[1], i.e. RapidProM [7]. RapidMiner is a data science platform that allows us to create *scientific workflows*, e.g. consider Figure 9.3 in which we show such scientific workflow for the purpose of creating an event stream from an event log. In particular, the RapidProM extension allows us to construct process mining analyses using scientific workflows, which is particularly suitable for large-scale experiments [28].

In this chapter, we provide a brief summary of these implementations, and present details on how to use them. As such, the content of this chapter is to be regarded as a high-level documentation of the different implementations developed in the context of this thesis. We additionally describe how to generate event streams from a variety of different sources. The remainder of this chapter is structured as follows. In section 9.2, we explain how to generate event streams from several static and non-static sources. In section 9.3, we discuss the implementation of the techniques related to general purpose storage as presented in chapter 3. In section 9.4, we discuss the implementation of the automaton based filter as presented in chapter 4. In section 9.5, we discuss the implementation of the algorithms presented in chapter 5, i.e. related to intermediate representation based discovery, as well as the ILP-Based discovery approach in chapter 6. In section 9.6, we discuss the implementation of the incremental prefix-alignment

[1]`http://rapidminer.org`

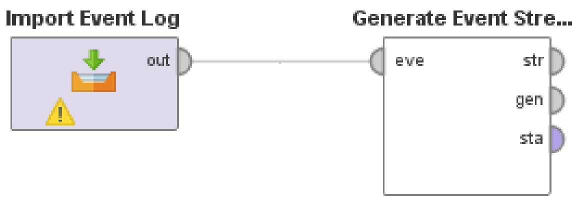


Figure 9.3: Example scientific workflow, depicting the generation of an event stream from an event log.

algorithm as described in chapter 7. In section 9.7, we discuss the implementation related to the discovery of collaborative resource networks as presented in chapter 8. Finally, section 9.9, concludes this chapter and, in particular, discusses limitations of the implementations presented here, as well as future work.

9.2 Event Stream Generation

In this section, we describe how to generate streams of events from various sources of (event) data. We first detail on stream generation in ProM, after which we describe how to generate event streams in RapidProM.

ProM The majority of the code related to the generation of event streams is located in the *EventStream* package, available through `http://svn.win.tue.nl/repos/prom/Packages/EventStream`. We identify three ways to generate an event stream within ProM.

1. Generating event streams from static finite data sources, e.g. from event logs
2. Generating event streams from live, possible infinite data sources.
3. Generating event stream from Coloured Petri Nets (CPN) models.

Finite Data Sources

To generate a stream out of an event log, we import an event log and select it. Consider Figure 9.4, in which we depict the BPI Challenge 2012 [47] event log, imported in the ProM framework. In order to generate an event stream, we click the *play button* (highlighted by means of a red circle in Figure 9.4). When we click the play button, there are two subsequent possible actions:

1. Directly generate a *live event stream* (visualized in Figure 9.5a).
2. Generate a *static event stream*, i.e. a sequence of events, effectively equal to event logs as defined in this thesis (visualized in Figure 9.5b).

In case we decide to generate an event stream, either live or static, we are able to set several properties of the stream, which we visualize in Figure 9.6. These properties entail:

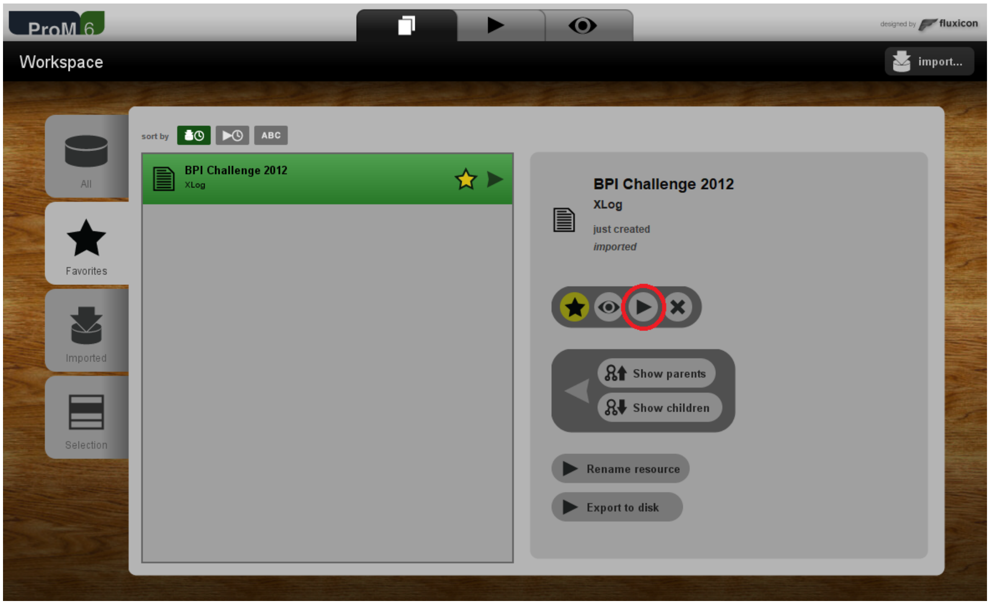
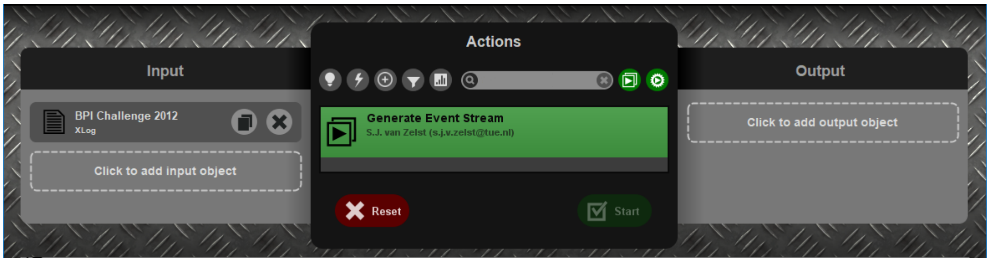
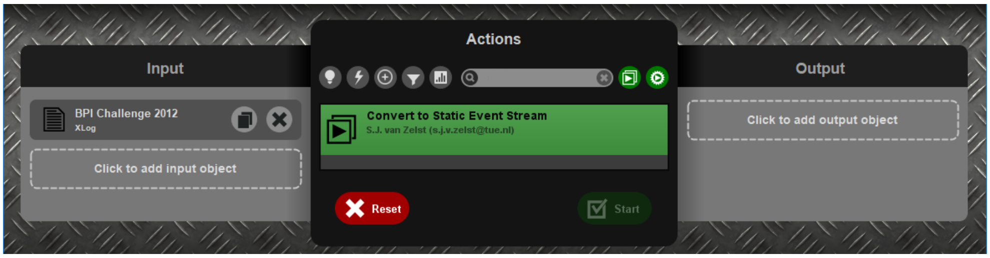


Figure 9.4: Event log of the BPI Challenge 2012 [47], imported in ProM. The first action to perform in order to generate an event stream is clicking the *play button*, highlighted by means of the red circle.

- *Case Identifier*
  We indicate which *trace-level attribute*, i.e. a data attribute stored at a trace level in the event log, is eligible to act as a case identifier. Usually, this is the `concept:name` attribute.
- *Activity Name*
  We indicate how to deduce the activity name of an event. An event log typically includes a collection of *classifiers* which allows us to map an event onto an activity. The widget allows us to select which classifiers, as defined in the event log, to use for this purpose.
- *Event Index*
  We are able to add the index of the event within the given trace as data payload within the event.
- *Termination Indication*
  We are able to add a boolean flag to the payload of the event, specifying whether or not the event is the last event of the trace.
- *Event Ordering*
  By default, the generated stream contains the events in order of occurrence in the event log, on a trace-by-trace basis. As such, process instances are not simulated to run in parallel. In case that time stamps are omni-present within the event log, we are able to sort the events based on time-stamp, rather than trace-by-trace.
- *Emission Rate* (Only for live stream generation)
  In case of live event stream generation, we are able to specify an emission rate in terms of the number of events on the event stream per second.


Actions
Input
Output
BPI Challenge 2012
XLog
Click to add input object
Generate Event Stream
S.J. van Zelst (s.j.v.zelst@tue.nl)
Click to add output object
Reset
Start

(a) ProM action for generating a live event stream, i.e. plug-in entitled *Generate Event Stream*.


Actions
Input
Output
BPI Challenge 2012
XLog
Click to add input object
Convert to Static Event Stream
S.J. van Zelst (s.j.v.zelst@tue.nl)
Click to add output object
Reset
Start

(b) ProM action for generating a static event stream, i.e. plug-in entitled *Convert to Static Event Stream*.

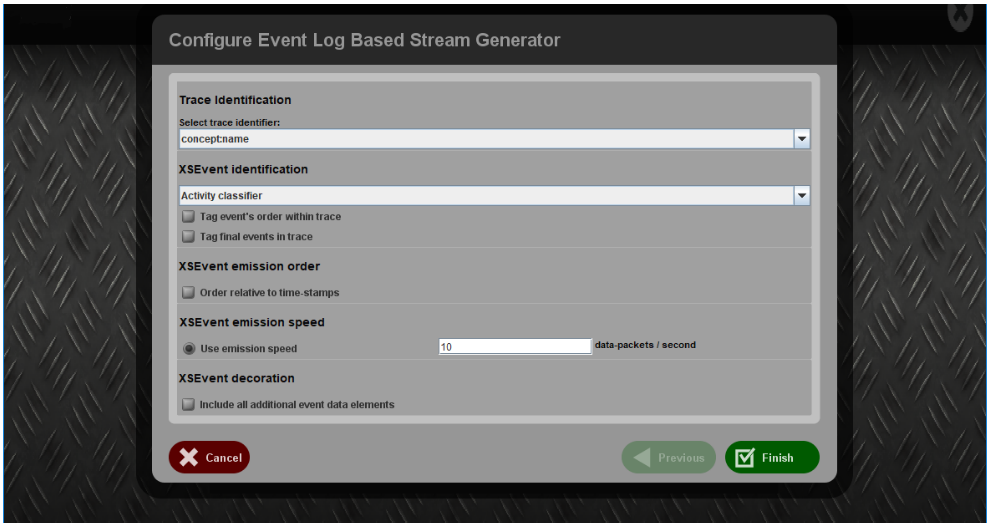
Figure 9.5: ProM actions for generating live- and static event streams.

- *Event Decoration*
  We are able to add all payload present in the event, i.e. as part of the event log, to the event emitted on the stream.

Within an event stream, an event is represented by means of a tuple of *key-value* pairs. For the different decoration options as presented earlier, we identify the following keys and/or key-prefixes that enable us to access them.

- *Case Identifier*
  The case identifier is accessible by means of a key entitled `trace`.
- *Activity Name*
  The activity name of an event is accessible by means of a key entitled `concept:name`.
- *Event Index*
  The index of the event within its original trace is accessible by means of a key entitled `xsevent:index`.
- *Termination Indication*
  The trace termination indication attribute is accessible by means of a key entitled `xsevent:final`.
- *Event Decoration*
  Given any additional payload that we want to add, originally accessible in the event log by some key `k`. Within the generated event, the same payload is accessible by means of a key entitled `xsevent:data:k`.

The main difference between creating a live- and a static event stream is related to the emission rate as well as the output of the plugin. When we create a static event stream, i.e. a


Configure Event Log Based Stream Generator

Trace Identification

Select trace identifier: concept:name

XSEvent identification

Activity classifier

☐ Tag event's order within trace

☐ Tag final events in trace

XSEvent emission order

☐ Order relative to time-stamps

XSEvent emission speed

● Use emission speed 10 data-packets / second

XSEvent decoration

☐ Include all additional event data elements

Figure 9.6: Wizard intended to set parameters related to a generated event stream.

sequence of events, we just receive a single ProM object representing the static stream entity. We are able to store such event stream on disk, resulting in a `.evst` file. Such file is a text file in which each line represents an event. Within the *EventStream* package we also provide code that allows us to import arbitrary `.evst` files. From a static event stream ProM object, we are able to generate a live event stream. We do so by means of selecting the static event stream object and subsequently calling the *Generate Event Stream* plug-in, cf. Figure 9.7.

Invoking the aforementioned action yields two artefacts, i.e. a *Stream Generator* object and an *Event Stream* object. These artefacts are also created when directly creating a live event stream from a finite data source, e.g. an event log. The Event Stream object represents a live version of the (static) event stream. The Stream Generator allows us to manipulate the event stream and is the primary output of the *Generate Event Stream* plug-in. Initially, the stream is *inactive*, i.e. there are no events emitted on the stream, cf. Figure 9.8a on page 227.

The stream generator's visualization has a custom *play button*, depicted on the top-right of its visualization, i.e. highlighted by means of a red circle in Figure 9.8a. When we click the play button, the stream generator starts emitting events on the stream. As such, any algorithm connected to the event stream object gets triggered to handle the newly arriving packets. An example visualization of a stream generator, when active, is depicted in Figure 9.8b.

Live Data Sources

Within the *EventStream* ProM package, we additionally provide the possibility to connect to a live stream of events. In this context, code is provided to connect to a live stream of JSON objects (`http://json.org`). JSON, which stands for JavaScript Object Notation, is a data-interchange format using key-value pairs and array-typed objects.

As an explanatory proof-of-concept, we have used the basic classes provided in the *Event-Stream* package infrastructure in combination with real/live event data. The data used as an input consists of a publicly available stream of `http://www.meetup.com` (`http://www.meetup.`

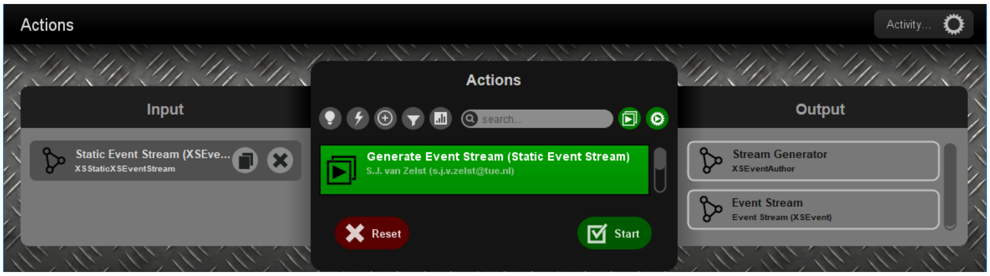
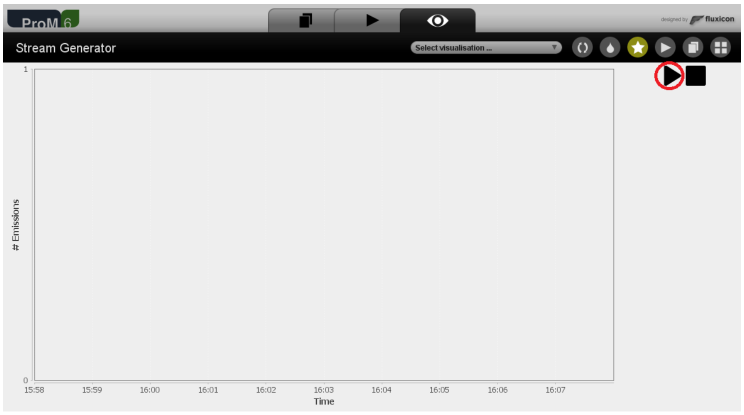


Figure 9.7: Converting a static event stream to a live event stream. Applying this action yields a *Stream Generator* object and an *Event Stream* object which actually represents the live stream.

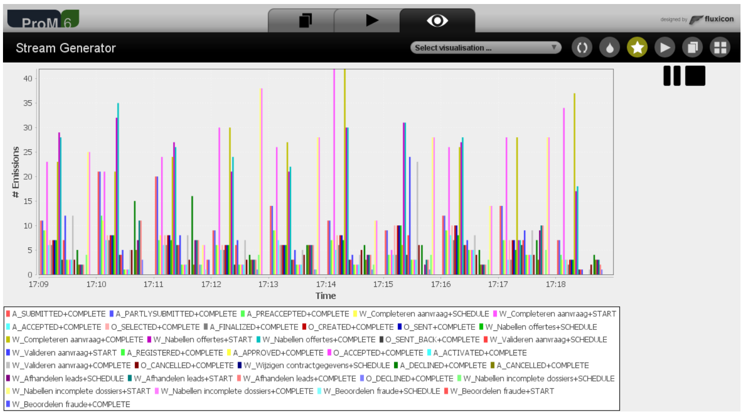
com/meetup_api/docs/2/open_events/). meetup.com is a website where people plan *happenings*, e.g. a jam-session for musicians or an afternoon walk through Central Park in New York. Within meetup.com these are called *events*. However, to avoid ambiguity with respect to the well-defined notion of events in this thesis, we use the term *happenings* when we refer to meetup events. A happening has several parameters (name, status, venue etc.). When a user has created a happening, he/she is allowed to change some parameters of the happening at a later stage, e.g. after creating a happening we decide to increase the total number of accepted guests. Every data-manipulation of a happening which has a public access level will be published on the meetup.com event stream.

Consider Figure 9.9 on page 228, in which we show two results after applying the inductive miner, on the basis of the generic architecture as presented in chapter 5, using the meetup.com event stream as an input. In Figure 9.9a, we show the resulting Petri net after receiving just 224 events, i.e. after just tapping in on the stream. At this point in time, the model describes that first a happening needs to be created, i.e. represented by the activity *create new meetup event*. After this, either the name is changed, by means of a *change name* activity. Otherwise, a large flower pattern of behaviour is discovered, i.e. everything is possible, including activities such as *change time*, *change venue*, *change status* etc. The final activity for the branch including the flower pattern is *change event_url*.

In Figure 9.9b, we depict the resulting Petri net after receiving a total of 27.988 events. In this case, the model still starts with the creation of a new happening. However, after this, either the process terminates immediately, i.e. related to the creation of happenings that are not changed afterwards, or we enter a large flower construct in which all kinds of manipulations of the happening are possible. Observe that this is as expected, i.e. from a global perspective, we do not assume that creating and/or changing happenings is a very structured process.

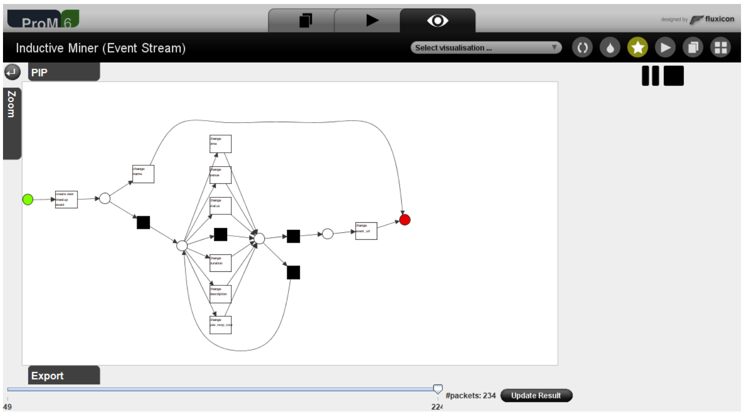


(a) Visualization of the Stream Generator object in which the stream is inactive. We are able to start the stream by clicking the *play button* in the top-right of the visualizer, highlighted by means of a red circle.

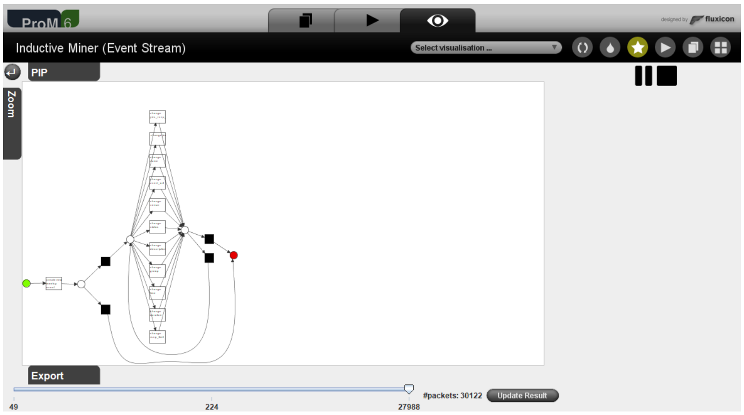

ProM 6
designed by fluxicon
Stream Generator
Select visualisation ...
# Emissions
40
35
30
25
20
15
10
5
0
17:09
17:10
17:11
17:12
17:13
17:14
17:15
17:16
17:17
17:18
Time
A_SUBMITTED+COMPLETE
A_PARTLYSUBMITTED+COMPLETE
A_PREACCEPTED+COMPLETE
W_Completeren aanvraag+SCHEDULE
W_Completeren aanvraag+START
A_ACCEPTED+COMPLETE
O_SELECTED+COMPLETE
A_FINALIZED+COMPLETE
O_CREATED+COMPLETE
O_SENT+COMPLETE
W_Nabellen offertes+SCHEDULE
W_Completeren aanvraag+COMPLETE
W_Nabellen offertes+START
W_Nabellen offertes+COMPLETE
O_SENT_BACK+COMPLETE
W_Valideren aanvraag+SCHEDULE
W_Valideren aanvraag+START
A_REGISTERED+COMPLETE
A_APPROVED+COMPLETE
O_ACCEPTED+COMPLETE
A_ACTIVATED+COMPLETE
W_Valideren aanvraag+COMPLETE
O_CANCELLED+COMPLETE
W_Wijzigen contractgegevens+SCHEDULE
A_DECLINED+COMPLETE
A_CANCELLED+COMPLETE
W_Afhandelen leads+SCHEDULE
W_Afhandelen leads+START
W_Afhandelen leads+COMPLETE
O_DECLINED+COMPLETE
W_Nabellen incomplete dossiers+SCHEDULE
W_Nabellen incomplete dossiers+START
W_Nabellen incomplete dossiers+COMPLETE
W_Beoordelen fraude+SCHEDULE
W_Beoordelen fraude+START
W_Beoordelen fraude+COMPLETE

(b) Visualization of the Stream Generator object in which the stream is active. In this visualization, a bar chart is shown where each bar indicates the number of emitted events of a certain activity type per minute.

Figure 9.8: Visualizations of the Stream Generator, both showing an inactive- and active stream.

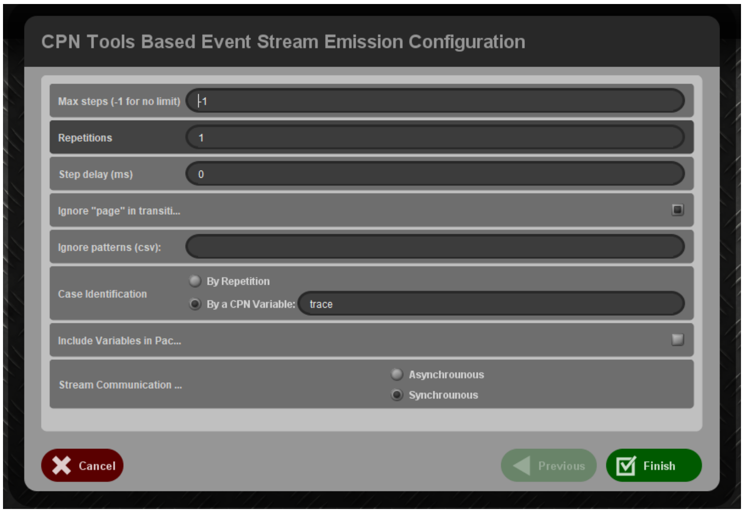


(a) Petri net discovered after receiving 224 events.



(b) Petri net discovered after receiving 27.988 events.

Figure 9.9: Petri nets discovered using the Inductive Miner instantiation of the discovery architecture presented in chapter 5, in combination with a live event stream originating from `http://meetup.com`.


CPN Tools Based Event Stream Emission Configuration

Max steps (-1 for no limit)	-1
Repetitions	1
Step delay (ms)	0
Ignore "page" in transiti...	■
Ignore patterns (csv):	
Case Identification	By Repetition [✓ By a CPN Variable:] trace
Include Variables in Pac...	☐
Stream Communication ...	Asynchrounous [✓ Synchrounous]

Figure 9.10: Configuration options of a CPN-based event stream generator.

Coloured Petri Nets

As a third method to generate event streams, the *EventStream* package supports event stream generation from Coloured Petri nets, e.g. as used in the evaluation of chapter 5. The underlying connection to CPN Tools, i.e. for the purpose of simulation of the CPN Model, is handled by the Access/CPN framework [124, 125].

Generating Event Streams In order to generate events, the underlying event stream generator needs a CPN model and an initial marking of the CPN model. Additionally we need to specify certain parameters of the generator (cf. Figure 9.10):

1. *Maximal number of steps*

   Represents the maximum number of steps within a single instance of the process, denoted here by s_{max}.

2. *Maximal number of repetitions*

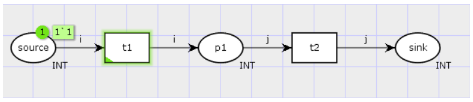
   Represents the total number of instances of the process that need to be simulated, denoted here by r_{max}.

3. *Step delay*

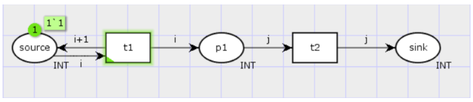
   Represents the minimal time desired in-between executions of consecutive events. If set to 0, no delay is applied.

4. *Page ignore*

   Boolean option that allows us to ignore the hierarchy of the CPN model used. CPN models allow for hierarchies, i.e. we are able to construct a CPN model within a transition. In such case, the name of such higher level transition is a prefix of the actual transition, e.g. a transition t within a transition t' is identified as $t'.t$. When using the page ignore option, the name of the transition is just t rather than $t'.t$.



(a) CPN Model suitable for repetition-based case identification technique.



(b) CPN Model suitable for CPN variable based case identification technique.

Figure 9.11: Two CPN Model fragments representing different examples of case identification technique.

5. *Ignore Patterns*

   CPN does not allow to construct transitions without an explicit unique name. In some cases, we need invisible transitions, e.g. for routing constructs. When we provide a (comma-separated) list of ignore patterns, all transitions having such pattern in their name are not emitted onto the event stream.

6. *Case Identification*

   This parameter specifies how we identify instances of the process. Currently, we have implemented two approaches being *repetition-based*, i.e. we first finish a full process instance after starting the next one, and *CPN variable based*, which supports parallel instances.

7. *Event decoration*

   We are able to choose whether we want to emit all variables associated to the firing of a transition within a data packet or only the core elements, being the trace identifier and the event name.

As indicated, we provide two ways to identify process instances within the CPN model. In the *repetition-based* case, each repetition of an execution of the CPN Model is used as a basis for identifying a case. Thus, all transitions fired in the first repetition will have *1* as a case identifier, all transitions fired in the second repartition will have *2* as a case identifier etc. In this identification technique, every transition that is fired is emitted as an event where the transition name acts as an event name. As an example of a CPN Model suitable for a repetition-based case identification technique, consider Figure 9.11a. Within the CPN model we have defined two variables of type `INT`, i.e. `var i,j: INT;`. An example stream originating from the CPN Model, where $r_{max}, s_{max} \geq 2$, including event decoration could be: ⟨ `{trace=1, concept:name=t1, i=1}`, `{trace=1, concept:name=t2, j=1}`, `{trace=2, concept:name=t1, i=1}`, `{trace=2, concept:name=t2, j=1}`, ...⟩. Note that within the repetition-based case, first all events related to trace *1* are emitted before events related to trace *2* are emitted, i.e. cases do not run concurrently. An instance of the process stops if either the maximal number of steps within the trace is reached or when no more transitions are enabled.

In the CPN *variable based* approach, the user specifies a specific variable present within the CPN model to act as a case identifier. In this case, only those transitions that fire and that have the specified variable associated are emitted to the event stream. Consider Figure 9.11b which depicts a CPN model suitable for CPN variable based case identification. Again we have defined two variables, i.e. `var i,j: INT;`. When we define variable `i` as the trace identification variable, given $r_{max} \geq 1$, $s_{max} \geq 3$, a possible stream originating from the CPN Model could be ⟨ `{trace=1, concept:name=t1, i=1}`, `{trace=2, concept:name=t1, i=2}`, `{trace=3, concept:name=t1, i=3}`, ...⟩. Observe that we never observe transition *t2* in this case, as it uses variable *j*, i.e. using variable based case identification allows us to hide certain

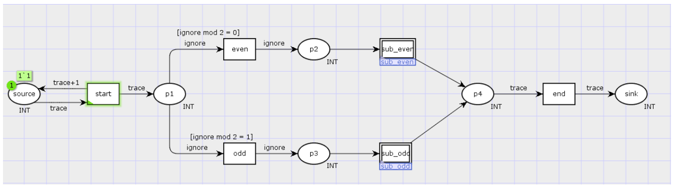
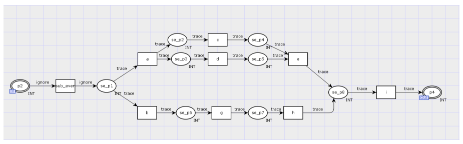
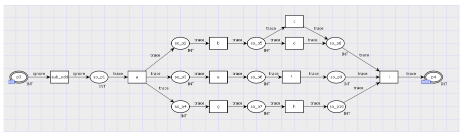


Figure 9.12: Root CPN Model of the hierarchical model used in the case study.



(a) CPN sub-model executed in case of a token with an even `INT` value.



(b) CPN sub-model executed in case of a token with an odd `INT` value.

Figure 9.13: Two CPN sub-models used within the case study.

transitions present within the model from the stream. Furthermore, it allows us to simulate multiple process instances in parallel. In particular, when using time within the CPN model, we are able to simulate arrival patterns, bottlenecks etc.

Proof of Concept As an explanatory proof of concept we have designed a hierarchical CPN model that is used as a basis for stream generation. The model consists of one root model and two sub models. The root model is depicted in Figure 9.12.

The CPN model consists of two variables, i.e. `var trace, ignore: INT`. The initial marking of the root model is one token of `colset INT`, i.e. `1‘1`, in place `source`. The transition labeled `start` is connected to place `source` and acts as a token generator. In its binding it uses the `trace` variable. If transition `start` fires, it produces a token with the value of `trace` in place `p1` and it produces a token with value `trace + 1` in place `source`. All tokens with an even `INT` value will be routed to the sub-model named `sub_even` whereas all tokens with an odd `INT` value will be routed to the sub-model named `sub_odd`. In this case, in routing to the sub-models the variable `ignore` is used. The two sub-models are depicted in Figure 9.13.

After importing the hierarchical model in the ProM framework, we configure an event stream with the following parameters: $r_{max} = 1$, $s_{max} = \infty$, case identification = *CPN Variable* with value `trace` and event decoration is `true`. After the event stream object is created we connect the stream-based implementation of the Inductive Miner. After receiving a number of events, the stream-based Inductive Miner returns the Petri net depicted in Figure 9.14.

Although the stream-based discovery algorithm is not able to discover hierarchy in the resulting model, its result matches with the input model, i.e. from a control-flow perspective it exactly describes all possible traces that are emitted onto the event stream.

RapidProM The code related to generation of streams of events is ported to the RapidProM plug-in, i.e. the ProM based extension for the RapidMiner data science platform. We briefly discuss these elements, and provide an indicative screen shot on how to use them. RapidProM supports the following actions, (based on the ProM code presented in the previous section).

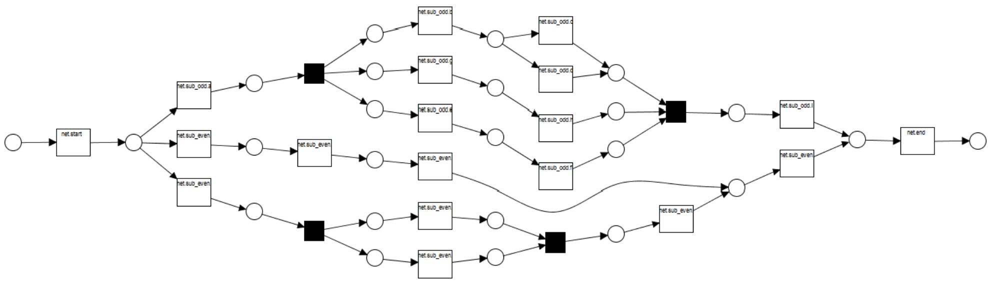
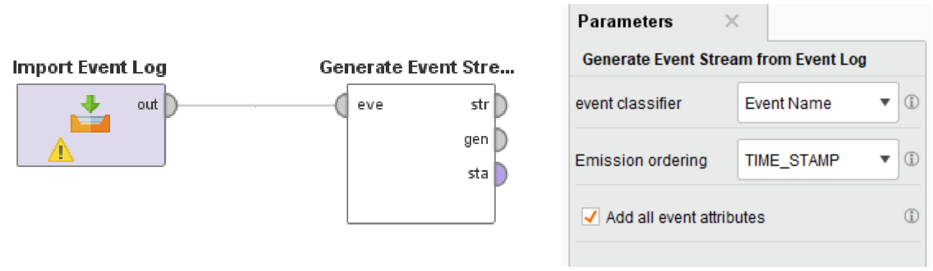


Figure 9.14: Result of applying a stream-based implementation of the Inductive Miner to the event stream generated by the hierarchical CPN model.

1. Generation of a live event stream from an event log, cf. Figure 9.15a. Supported parameters are: *event classifier*, *ordering of events* and *additional payload inclusion*.
2. Generation of a static event stream from an event log, cf. Figure 9.15b. Supported parameters are: *event classifier*, *ordering of events* and *additional payload inclusion*.
3. Generation of a live event stream from a CPN model, cf. Figure 9.15c. Supported parameters are: *maximal number of steps per process instance*, *number of repetitions*, *delay in-between events*, *case identification*, *CPN variable* (when using CPN variable based case identification), *additional payload inclusion*, *communication type*, *page ignoring* and *ignore patterns*.
4. Generation of a live event stream from a static stream, cf. Figure 9.16. This operator has no specific parameters. Note that we are able to convert an imported event log into a static stream and subsequently create a live event stream, cf. Figure 9.16a, or, from an imported static event stream, cf. Figure 9.16b.

Observe that, when we generate a live event stream from an event log, cf. Figure 9.15a, additionally, a static event stream object is generated. Moreover, whenever we generate a live stream object, like in ProM, we obtain both a *Generator* object and a stream object. To start the streaming process, we visualize the generator, which is equal to the visualization in ProM, cf. Figure 9.8, and click the start button. Thus, when using the streaming framework within RapidProM, we typically first design the experiment as a whole, i.e. generation of the stream, the subsequent filtering and/or mining algorithms etc. After this, we start the RapidMiner workflow. Note, however, that after the workflow is completed, all the objects, i.e. streams, generators, filters, algorithms etc. are created. However, these are inactive. To actually start the analysis, we manually start the stream by means of the aforementioned play button in the visualization of the respective object(s).

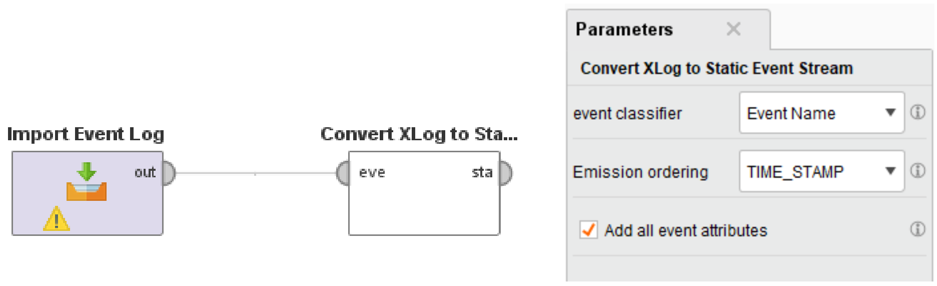


Parameters

Generate Event Stream from Event Log

event classifier	Event Name
Emission ordering	TIME_STAMP
✓ Add all event attributes	

(a) Generation of a live event stream from an event log.

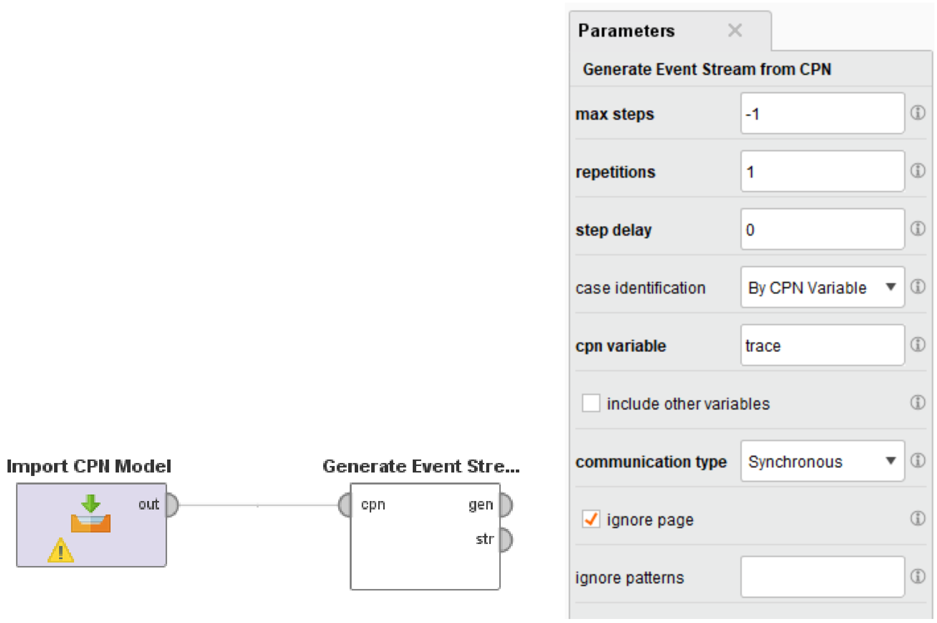


Parameters

Convert XLog to Static Event Stream

event classifier	Event Name
Emission ordering	TIME_STAMP
✓ Add all event attributes	

(b) Generation of a static event stream from an event log.



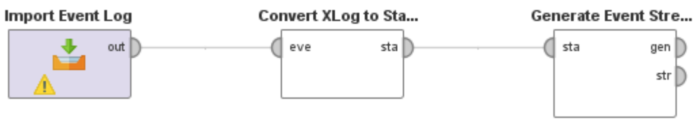
Parameters

Generate Event Stream from CPN

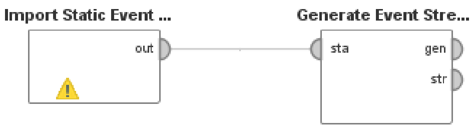
max steps	-1
repetitions	1
step delay	0
case identification	By CPN Variable
cpn variable	trace
include other variables	
communication type	Synchronous
✓ ignore page	
ignore patterns	

(c) Generation of a live event stream from a CPN model.

Figure 9.15: Scientific workflows and associated parameters for the purpose of event stream generation in RapidProM.



(a) Generation of a live event stream from a static event stream, directly converted from an event log.



(b) Generation of a live event stream from an imported static event stream.

Figure 9.16: Scientific workflows for the purpose of event stream generation from static event streams in RapidProM.

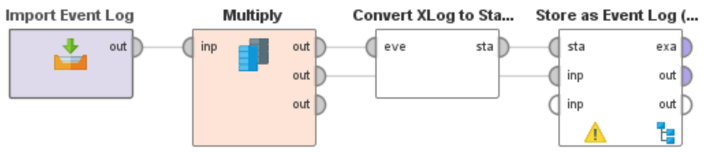


Figure 9.17: Scientific workflow to apply conventional process discovery on the basis of event stores.

9.3 Storage

The techniques presented in chapter 3, allow us to temporarily store the events emitted onto the event stream, in terms of a conventional event log. An implementation of the techniques discussed in chapter 3 is available in the *StreamBasedEventLog* ProM package[2]. The main component of the implementation, accessible for users, however resides in the RapidProM framework. It comprises of an operator that is intended for constructing experiments with the different storage techniques.

Consider Figure 9.17, in which we depict a simple workflow for the purpose of stream based discovery on the basis of conventional process discovery techniques applied on top of event stores. The RapidProM operator entitled *Store as Event Log (Finite)* allows us to iteratively store the input event stream as an event store, i.e. by converting it to a conventional event log. It needs a static event stream as an input. The operator iterates over the events in the static stream and passes each event to the underlying storage component. It supports the following storage techniques:

- Sliding Window, cf. subsection 3.3.1 (page 63)
- Reservoir Sampling (Event Level), cf. subsection 3.3.2 (page 65)
- Reservoir Sampling (Case Level), cf. subsection 3.3.2 (page 65)
- Prefix-Tree based storage (Using a Sliding Window as underlying storage technique), cf. section 3.4 (page 75)

For all techniques, we are able to specify a size parameter, i.e. the length of the sliding window/size of the reservoir. For the case-level reservoir we need to specify an additional size parameter for the subsequences stored for each case identifier. We are additionally able to set a *stop criterion*, which specifies after what number of received events the operator needs to terminate.

The *Store as Event Log (Finite)* operator is of a hierarchical type, i.e. it is allowed to contain a scientific workflow internally. Consider Figure 9.18, in which we depict an example internal workflow of the *Store as Event Log (Finite)* operator. After each event, a new event log is generated, based on the events stored in the sliding window, reservoir sample or the prefix-tree. The event log is delivered as the left-most upper input object (circular shape labelled `xlo`). In the workflow depicted in Figure 9.18, we forward the input event log to the implementation of the Alpha Miner [11] in RapidProM. Subsequently, we compute replay-fitness of the discovered model with respect to the event log used to generate the static event stream as a whole. Note that, in Figure 9.17, we provide the event log used to generate the static event stream, as a

[2]`http://svn.win.tue.nl/repos/prom/Packages/StreamBasedEventLog`

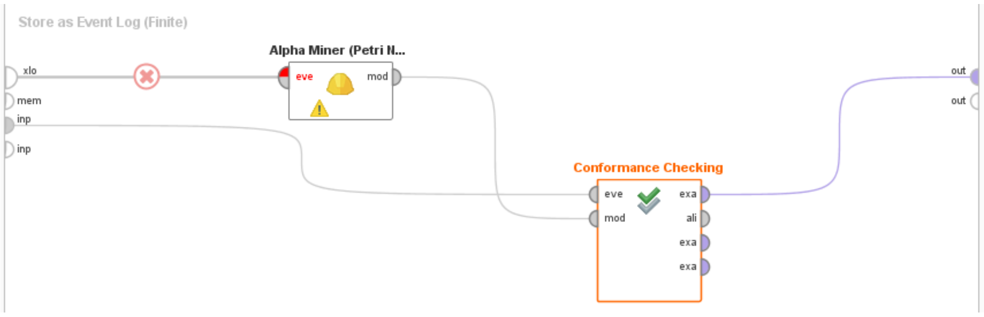


Figure 9.18: Example internal workflow in the *Store Event Log (Finite) Operator*.

second input of the *Store as Event Log (Finite)* operator. In this way, we are able to assess the result of the Alpha Miner, with respect to the original event log as a whole.

9.4 Filtering

In this section, we present details regarding the implementation of the filtering algorithm, as presented in chapter 4.

ProM The automaton-based filtering technique as presented in chapter 4 is implemented in ProM, i.e. in the *StreamBasedEventFilter* (`http://svn.win.tue.nl/repos/prom/Packages/StreamBasedEventFilter/`) package. Moreover, a part of the code is ported to RapidProM, in a similar fashion as the storage implementation, cf. section 9.4. The filter acts as an event processor, i.e. it takes a live event stream object as an input and returns a live event stream object. The resulting event stream represents the filtered event stream. The corresponding ProM plug-in is entitled *Spurious Event Filter*, cf. Figure 9.19. When invoking the ProM plug-in, the filter is automatically started, i.e. it is actively listening to incoming events and actively filtering. Within the ProM implementation, the parameters are not exposed to the user, i.e. they are only manipulated from the source code.

RapidProM The automaton filter is also ported to the RapidProM framework, primarily for the purpose of performing repeated experiments with the filter. The filter is implemented as a RapidProM operator that takes a static event stream as an input. It supports the following parameters:

- *Sliding window size*

  Represents the size of the underlying sliding window on which the behaviour in the automaton is based.

- *Abstraction*

  Represents the abstraction to use within the filter, as presented in section 4.4, i.e. the identity-, Parikh- or set abstraction.

- *Filter Direction*

  Allows us to specify whether we need to look back and/or forward, given the current event when filtering. In chapter 4 we only describe the backward scenario. The implementation

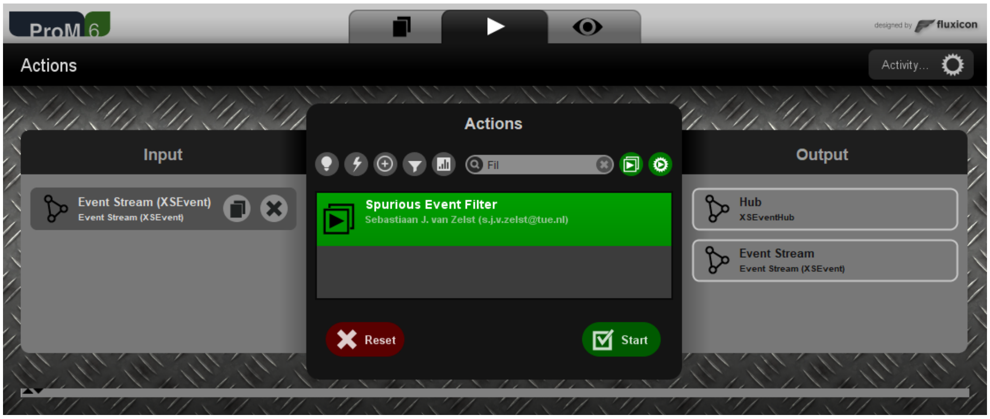


Figure 9.19: ProM plug-in of the automaton based filter as presented in chapter 4.

also allows us to “look forward”, i.e. given that we receive a new event d for a simple trace $\langle a, b, c \rangle$, we are also able to assess the probability of $\langle c, d \rangle$ following activity b. We are also able to use both forward and backward within filtering. Note that we have not studied the effect of forward-based filtering in depth in chapter 4.

- *Filter*
  Allows us to specify what filter to use as defined in subsection 4.4.3, i.e. *Fractional*, *Heavy Hitter* and *Smoothened Heavy Hitter*.
- *Filter Threshold*
  Represents the threshold to use in combination with the filter of choice.
- *Maximum Abstraction Window Size*
  Specifies the maximum abstraction size to use within the filter. Recall that this parameter determines the number of automata that are used within the filter.
- *Emission Delay*
  Specifies the use of an emission delay. If set to value k, every event is emitted with an emission delay of k events. The filter is however updated immediately, i.e. the delay only relates to the effective filtering of the event.
- *Experiment*
  Boolean parameter indicating whether the operator is used in an experimental setting. If selected, the user needs to specify the two following additional parameters (only possible in synthetic experimental settings):
  – *Noise Label Key*
    Name of the event attribute that specifies whether the event is noise or not.
  – *Noise Label Value*
    The value of the attribute as specified by the *Noise Label Key* in case the event actually relates to noise.

As indicated, the operator in RapidProM is mainly designed for experiments to assess the accuracy of the filter. As such, the operator results in a table with quality results, rather than an

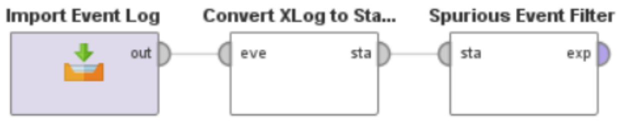


Figure 9.20: Example RapidProM workflow including automaton based filtering.

output stream. This table is delivered at the operator's output port entitled *exp*, cf. Figure 9.20. The resulting table contains the following quality metrics:

- *True Positives*
  Number of events that are predicted to be noise and where indeed noise, i.e. as specified by the noise label key/value.
- *False Positives*
  Number of events that are predicted to be noise whereas they are in fact not specified by the noise label key/value as being noise.
- *True Negatives*
  Number of events that are not predicted as being noise, which are indeed not noise.
- *False Negatives*
  Number of events that are not predicted as being noise which are however noise according to the noise label key/value.
- A collection of derived quality metrics, i.e. *Recall*, *Precision*, *Specificity*, *Negative Predictive Value*, *Accuracy* and *F1 Score*.

9.5 Discovery

In line with the chapters presented in this thesis related to process discovery, i.e. chapter 5 and chapter 6, we present the corresponding implementations in a twofold fashion. We first present the implementation of the intermediate representation based architecture as presented in chapter 5, after which we present the implementation related to ILP-based process discovery, cf. chapter 6.

9.5.1 Intermediate Representations

In this section, we present details regarding the implementation of the event-stream-based process discovery architecture, as presented in chapter 5.

ProM Several instantiations of the intermediate representation based architecture are implemented in ProM. These implementations entail:

- *Alpha Miner*
  An implementation of the Alpha Miner based instantiation of the architecture is available through the *StreamAlphaMiner* package (`http://svn.win.tue.nl/repos/prom/Packages/StreamAlphaMiner/`). Within ProM, given a live event stream, we are able to invoke the implementation using the ProM plug-in entitled:

   - *Discover Accepting Petri Net(s) (Alpha Miner)*

- *Inductive Miner*

  An implementation of the inductive miner based instantiation of the architecture is available through the *StreamInductiveMiner* package (`http://svn.win.tue.nl/repos/prom/Packages/StreamInductiveMiner/`). Within ProM, given a live event stream, we are able to invoke the implementation using the ProM plug-in entitled:

   - *Discover Accepting Petri Net(s) (Inductive Miner)*
   - *Discover Process Tree(s) (Inductive Miner)*

- *ILP Miner*[3]

  An implementation of the ILP miner based instantiation of the architecture is available through the *StreamILPMiner* package (`http://svn.win.tue.nl/repos/prom/Packages/StreamILPMiner/`). Within ProM, given a live event stream, we are able to invoke the implementation using the ProM plug-in entitled:

   - *Discover Accepting Petri Net(s) (ILP Miner)*

- *Transition Systems Miner*

  An implementation of the transition system's miner based instantiation of the architecture is available through the *StreamTransitionSystemsMiner* package (`http://svn.win.tue.nl/repos/prom/Packages/StreamTransitionSystemsMiner/`). Within ProM, given a live event stream, we are able to invoke the implementation using the ProM plug-in entitled:

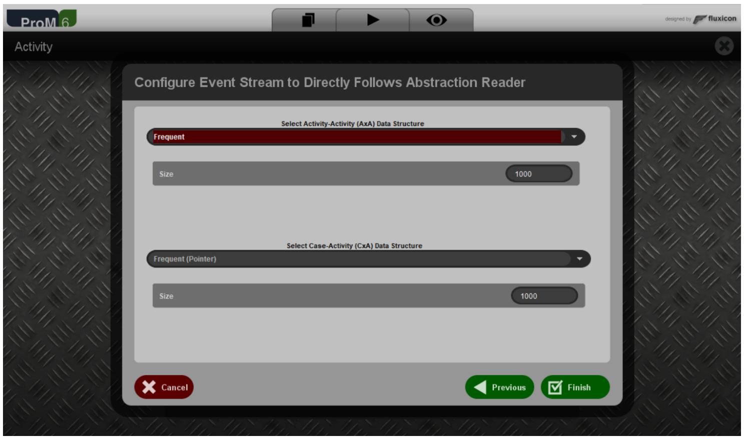
   - *Discover Accepting Petri Net(s) (Transition Systems Miner)*

From a user perspective, the different instantiations behave the same. Using the Alpha Miner as a basis, we show how to invoke the plug-ins. To obtain a live event stream, we refer back to section 9.2. After obtaining a live event stream, we select it in the ProM Workspace and invoke the plug-in of choice. All implementations use frequent item approximation techniques as an underlying storage technique, i.e. the `Frequent`, `Space Saving` and `Lossy Counting` algorithms. Thus, before visualizing the algorithm, the user is asked to select the preferred underlying storage technique and associated number of elements that the algorithms is allowed to store, cf. Figure 9.21.

After the user has specified the desired underlying event store, the visualization of the online discovery technique is shown. An example of this visualization is presented in Figure 9.22. We are able to pause the discovery algorithm by means of the *pause button* in the top-right of the visualization. When paused, the discovery algorithm temporarily stops receiving events on the stream. When clicking the *stop button*, located right to the pause button, the discovery algorithm is terminated. The main component of the visualization is a canvas depicting a discovered process model. In this case, we show a Petri net, as this is the result of the Alpha Miner. However, when we use the inductive miner variant that results in process trees, the canvas visualizes process trees, rather than Petri nets.[4] Underneath the canvas, we observe a slider and a click-able button, labelled "Update Result". When we click the button, the underlying

[3]Observe that the implementation of the stream based ILP Miner does not include finding Workflow nets with artificial start/end activities as described in chapter 6. It uses classical region theory to find places related to suspected causal relations.

[4]A process tree is a process modelling formalism in which the components of the process model are structured hierarchically, by means of a tree structure.



Configure Event Stream to Directly Follows Abstraction Reader

Select Activity-Activity (AxA) Data Structure: Frequent

Size: 1000

Select Case-Activity (CxA) Data Structure: Frequent (Pointer)

Size: 1000

Figure 9.21: User interaction dialog, allowing the user to specify what underlying storage method needs to be used.

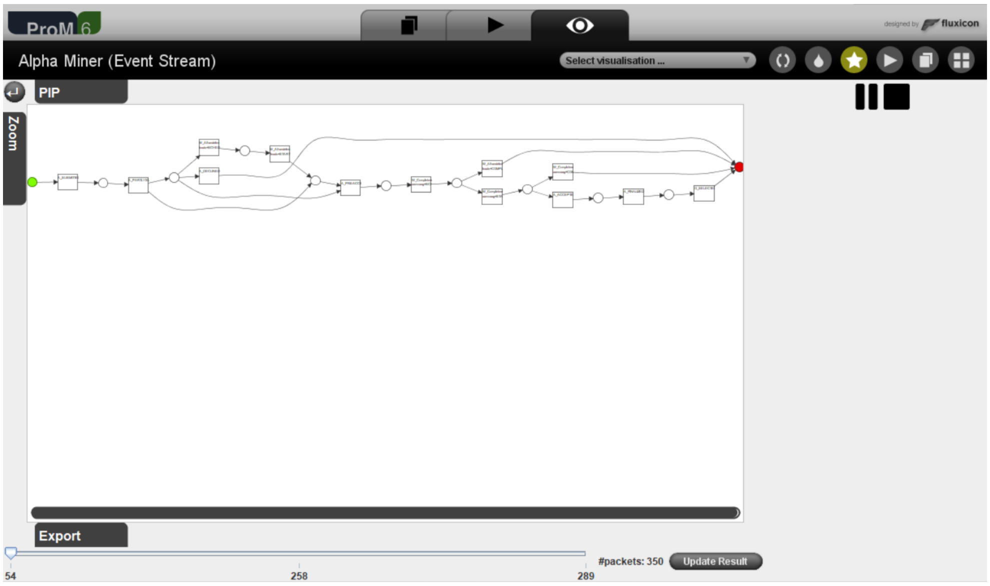


Figure 9.22: User interface of a stream based discovery algorithm.

discovery algorithm is invoked, i.e. translating the intermediate representation maintained in memory into a process model. The slider allows the user to browse between different discovered process models at different points in time. This allows the user to visually inspect potential changes in the underlying stream, i.e. visual inspection of concept drift.

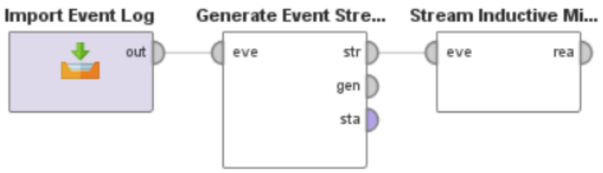


Figure 9.23: Example workflow of applying process discovery in RapidProM.

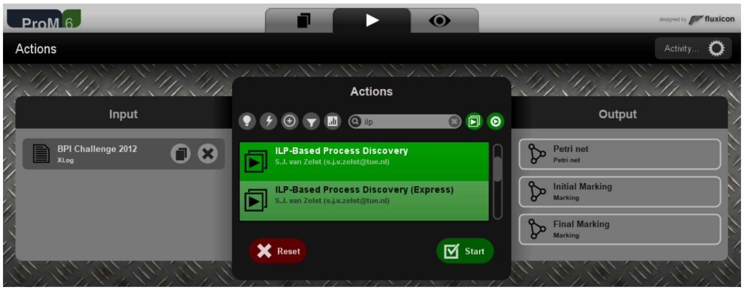


Figure 9.24: Invoking the *ILP-Based Process Discovery* plug-in in ProM.

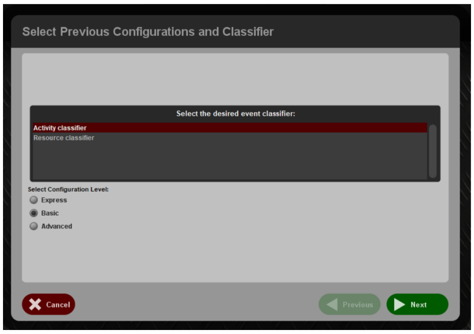
RapidProM Within RapidProM, the Alpha Miner and the Inductive Miner are made available. For the Inductive Miner, both the version resulting in (accepting) Petri nets as well as process trees is available. The general workflow, using an event log as a data source, to use any of the discovery algorithms is depicted in Figure 9.23.

As indicated in section 9.2, we first need to run the workflow, after which we obtain an inactive generator object, and an active discovery algorithm, ports *gen* and *rea* in Figure 9.23 respectively. To perform the analysis, we need to start the event stream as described in section 9.2, i.e. RapidProM uses the same visualizations as the ProM framework. Likewise, the discovery algorithm visualization is similar to the visualizations shown in Figure 9.9 and Figure 9.22.

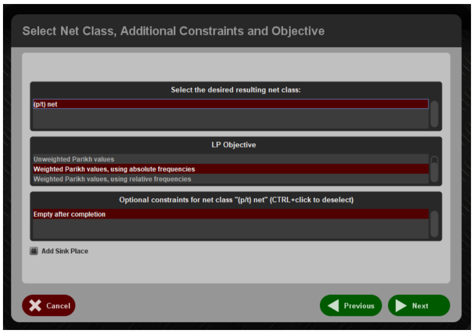
9.5.2 ILP-Based Process Discovery

In this section, we discuss the implementation of the ILP-based process discovery approach as described in chapter 6. As the algorithm is designed to work with conventional simple event stores, i.e. in a streaming setting obtainable by the techniques presented in chapter 3, the implementation only supports using conventional event logs rather than event streams. The code related to the implementation of the ILP-based process discovery is available in the *HybridILPMiner* package (`http://svn.win.tue.nl/repos/prom/Packages/HybridILPMiner`). Again, the implementation is available both in the ProM framework and in RapidProM.

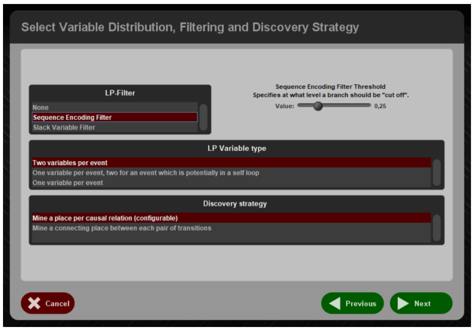
ProM To invoke the implementation of the ILP-based process discovery algorithm, after loading an event log into ProM, we use the *ILP-Based Process Discovery* plugin, cf. Figure 9.24. Observe that there is a second version of the plug-in, entitled *ILP-Based Process Discovery (Express)*, i.e. the second plug-in visualized in Figure 9.24. This plug-in simply applies default


Select Previous Configurations and Classifier
Select the desired event classifier:
Activity classifier
Resource classifier
Select Configuration Level:
Express
Basic
Advanced
Cancel
Previous
Next

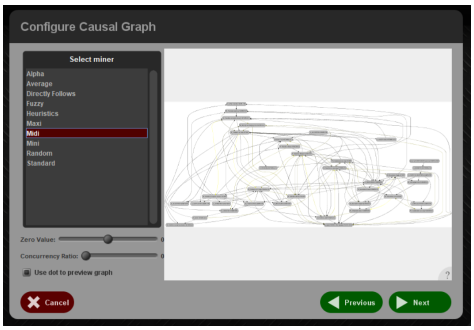
(a) Parameter screen for selecting the event classifier to use and the configuration, i.e. express, basic or advanced.


Select Net Class, Additional Constraints and Objective
Select the desired resulting net class:
(p/t) net
LP Objective
Unweighted Parikh values
Weighted Parikh values, using absolute frequencies
Weighted Parikh values, using relative frequencies
Optional constraints for net class "(p/t) net" (CTRL+click to deselect)
Empty after completion
Add Sink Place
Cancel
Previous
Next

(b) Parameter screen (*advanced configuration only*) including selection of the objective function and addition of *empty after completion constraints*.


Select Variable Distribution, Filtering and Discovery Strategy
LP-Filter
None
Sequence Encoding Filter
Slack Variable Filter
Sequence Encoding Filter Threshold
Specifies at what level a branch should be "cut off".
Value:
0,25
LP Variable type
Two variables per event
One variable per event, two for an event which is potentially in a self loop
One variable per event
Discovery strategy
Mine a place per causal relation (configurable)
Mine a connecting place between each pair of transitions
Cancel
Previous
Next

(c) Parameter screen (*advanced configuration only*) including selection of internal filtering technique, variable distribution and discovery.



(d) Parameter screen (*advanced & basic configuration*) allowing us to select the causal graph to use.

Figure 9.25: Different user interaction screens allowing the user to set different parameters of the ILP-based process discovery algorithm.

settings of the algorithm, i.e. after invoking the plug-in, a process model is obtained immediately.

When selecting the basic plug-in, i.e. *ILP-Based Process Discovery*, the user needs to specify several parameters, distributed over four different screens, cf. Figure 9.25

The parameters of the ILP-based process discovery algorithm entail:

- *Classifier* (Figure 9.25a)

  Allows us to specify how to identify the activity that is described by the event. The list of classifiers is based on information provided by the underlying event log object.

- *Configuration Level*

  Allows us to specify to what degree we are interested in configuring the discovery algorithm. There are three configuration levels:

  – *Express* (Figure 9.25a)

    Requires no additional configuration. The express setting incorporates *emptiness after trace completion* constraints, uses the sequence encoding filter as described in section 6.4 with a threshold value of 0.25 and uses a causal graph as discovered by the Flexible Heuristics Miner [122]. It is equal to the plug-in *ILP-Based Process Discovery (Express)*.

- *Basic*

  Adopts the same settings as the express configuration, however, allows the user to select a different causal graph, cf. Figure 9.25d.

- *Advanced*

  Allows the user to specify all the different parameters as presented as follows.

- *LP objective function* (Figure 9.25b)

  Allows us to specify what objective function is used when searching for places in the Petri net. The parameter allows the user to select both z_1 and $z_{\overline{L}}$. Moreover, an alternative variant of $z_{\overline{L}}$ is implemented that scales the coefficients back to the range $[0,1]$.

- *Emptiness after trace completion* (Figure 9.25b)

  When selected, the LP constraint body includes the constraints related to emptiness after trace completion, i.e. $m\vec{1} + \mathbf{M}_L(\vec{x} - \vec{y}) = \vec{0}$ in 6.3. Observe that when we select this option, we are able to additionally search for a sink place. This option is vital when we aim at finding a workflow net.

- *Internal LP Filter* (Figure 9.25c)

  Allows us to specify what filter to use (if the user aims at using internal filtering). Options include applying no filtering, sequence encoding filtering as presented in section 6.4 and *slack variable filtering*. The notion of slack variable filtering is presented in [132]. It adds a boolean variable for each trace present in the constraint body, which allows the ILP to ignore the constraint when finding a solution. The total number of traces to ignore is bounded by an additional user-specified threshold. Although the technique works on small example data, the resulting constraint body is too large to be solved efficiently when using real event data in combination with slack variable filtering.

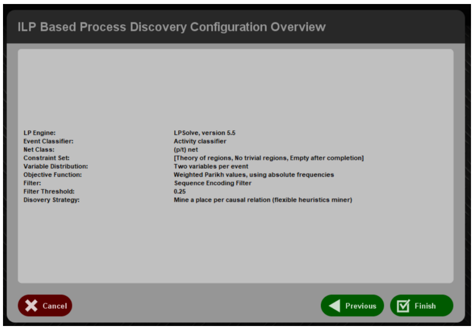
- *Variable distribution* (Figure 9.25c)

  Allows us to specify whether we aim to use one or two variables per activity in the event log. As presented in [133], we are able to use only one variable per activity, taking any of the three values in $\{-1, 0, 1\}$. A variable assignment of -1 implies an outgoing arc to the corresponding transition, whereas a value of 1 indicates an incoming arc from the transition. It is however not clear whether a variable assignment of 0 indicates a self-loop on the corresponding transition or not. There additionally exists a hybrid variant which uses only two variables for an activity a if there exists a simple trace of the form $\langle ...a, a, ...\rangle$ in the input event log.

- *Discovery strategy* (Figure 9.25c) Allows us to specify in what way places need to be searched for. We are able to use an underlying causal structure, i.e. the result of a causal oracle ϱ. Alternatively, we are able to search for a place between transitions related to each possible pair of activities that are present in the input traces.

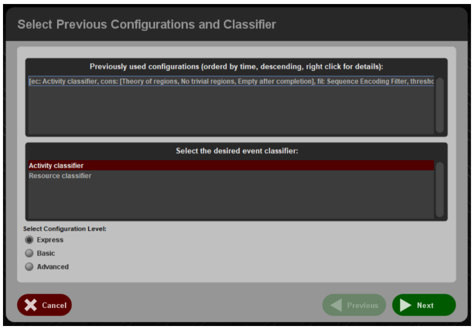
- *Causal graph selection* (Figure 9.25d, conditional to causal graph discovery strategy)

  Allows us to specify what causal graph to use. In the left-hand panel of the user interface a list of available causal oracles is listed. On the right-hand panel, a canvas is shown in which the corresponding causal graph is visualized. Additionally, sliders are present that allow the user to interactively change the thresholds of the causal oracle of choice, allowing the user to alter the resulting causal graph. *It is important to note that it is the user's responsibility to verify whether the causal graph of choice is compliant with the requirements specified in section 6.3 in order to obtain a workflow net.*



LP Engine:	LPSolve, version 5.5
Event Classifier:	Activity classifier
Net Class:	(p/t) net
Constraint Set:	[Theory of regions, No trivial regions, Empty after completion]
Variable Distribution:	Two variables per event
Objective Function:	Weighted Parikh values, using absolute frequencies
Filter:	Sequence Encoding Filter
Filter Threshold:	0.25
Disovery Strategy:	Mine a place per causal relation (flexible heuristics miner)

(a) Overview screen presenting a summary of the parameter configuration.



Previously used configurations (orderd by time, descending, right click for details):

[ec: Activity classifier, cons: [Theory of regions, No trivial regions, Empty after completion], fil: Sequence Encoding Filter, thresho

Select the desired event classifier:

Activity classifier
Resource classifier

Select Configuration Level:
● Express
○ Basic
○ Advanced

(b) Initial screen allowing the user to load a previously used parameter setting.

Figure 9.26: Summary screen and screen to load previously used parameter settings.

After clicking the *next button* in the final parameter screen (subject to the configuration level of choice), the user is presented with an overview of the chosen parameters, cf. Figure 9.26a. When the user subsequently clicks the *finish* button, the discovery phase is started, and finally a Petri net (with associated initial and final marking) is constructed.

Due to the relatively large amount of parameters, the plug-in additionally allows the user to load a previously used parameter combination. If the user runs the plug-in for the second, third, ..., n^{th} time, the first parameter screen additionally shows a list of previously used parameter combinations, cf. Figure 9.26b. When the user selects one of these, all values of that specific collection of parameter instances is loaded. Doing so, always requires the user to go through the advanced parameter setting option. This is intended as such, as we expect the user to iteratively tweak certain parameters to gain a better/desired result. Finally, observe that the ILP based process discovery implementation *always appends an artificial start- and end activity to all traces in the input event log*.

RapidProM The ILP-based process discovery algorithm is ported to RapidProM. Within RapidProM, the algorithm is however less configurable, i.e. many of the discussed before is fixed. For example, the discovery strategy is causal, i.e. it is not possible to use all pairs of activities. Moreover, instead of being a parameter, the causal graph needs to be provided as an additional input. The only two parameters provided in the RapidProM instance of the ILP based process discovery algorithm are:

- *Classifier*

  Allows us to specify how to identify the activity that is described by the event. The list of classifiers is based on information provided by the underlying event log object.

- *Internal LP Filter*

  Only includes applying no filter or the sequence encoding filter, i.e. as described for the ProM-based variant.

Consider Figure 9.27 in which we present an example workflow using the ILP based process discovery algorithm. The left-most RapidProM operator imports the event log. Both the causal graph operator and the ILP based process discovery algorithm use an event log as an input, hence we need a multiplier operator to (graphically) duplicate the event log. We use the causal

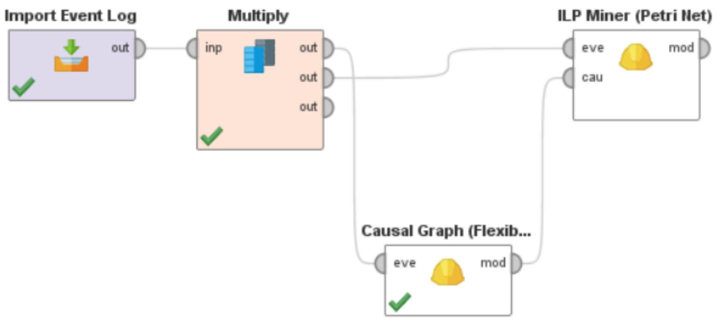


Figure 9.27: Example workflow of the ILP based process discovery approach in RapidProM.

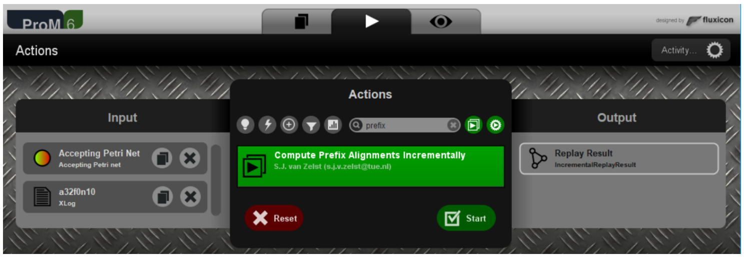


Figure 9.28: ProM plug-in for the purpose of computing incremental prefix alignments.

graph object as an input for the ILP based process discovery algorithm, i.e. represented by the connection between the *mod* and *cau* ports.

9.6 Conformance Checking

For the purpose of event stream based conformance checking, the incremental prefix-alignment algorithm, cf. chapter 7, has been implemented in ProM and was ported to RapidProM. The code is available in the OnlineConformance package (`http://svn.win.tue.nl/repos/prom/Packages/OnlineConformance`). The implementation is mainly intended for performing experiments related to the quality of the computed prefix-alignments, i.e. it does not provide a graphical interface. As such, it uses a conventional event log as an input rather than a (static) stream. To invoke the plug-in, we need an *Accepting Petri net* object and an *event log object*, cf. Figure 9.28. The plug-in is entitled *Compute Prefix Alignments Incrementally*. The code iteratively traverses each prefix of each *trace variant* in the event log. It stores the result related to each prefix and performs a search based on the algorithm as presented in chapter 7.

In RapidProM, cf. Figure 9.29, apart from an event log and a *regular Petri net* object, the operator needs an additional conventional conformance checking result object. This object is used to compare the prefix-alignment value of each prefix to the actual eventual alignment value, i.e. after trace completion. The RapidProM operator returns a detailed table containing alignment costs (and differences to eventual alignment costs) for all prefixes of all trace variants in the input event log.

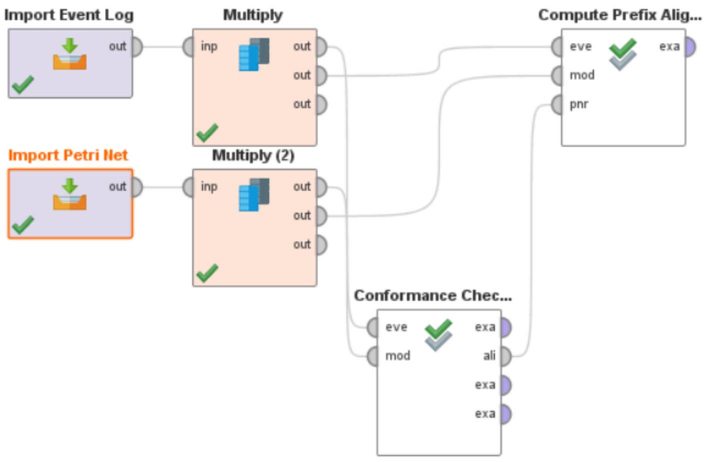


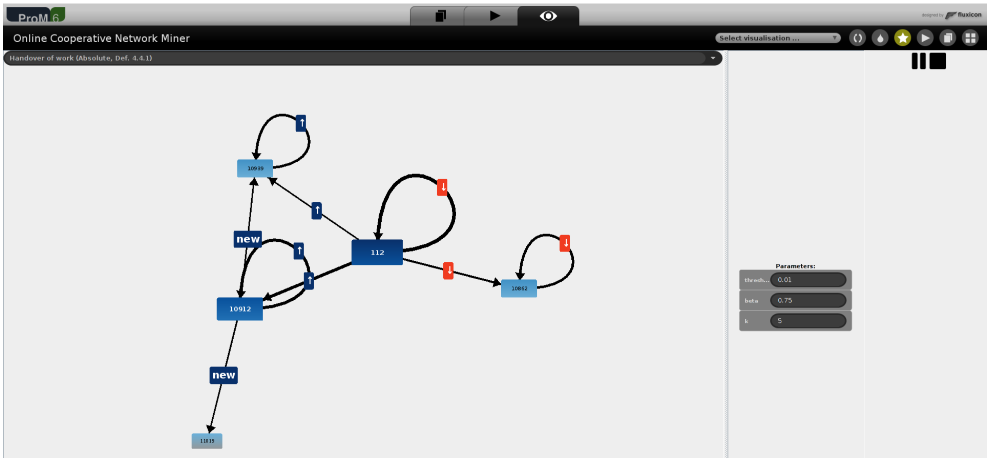
Figure 9.29: Example workflow of using computing incremental prefix alignments in RapidProM.

9.7 Enhancement

For the purpose of process enhancement, different event-stream-based social networks are implemented in ProM, within an interactive tool. Due to the interactive nature of the tool, it has not been ported to RapidProM. The implementation of the Online Cooperative Network (OCN), cf. chapter 8, is available within the `StreamSocialNetworks` package (`https://svn.win.tue.nl/repos/prom/Packages/StreamSocialNetworks/`). Consider Figure 9.30, in which we depict an example screen shot. The implementation provides support for several different notions of social networks, i.e. *Generalized Handover of Work Metrics* [9, Definition 4.4], *Generalized In-Between Metrics* [9, Definition 4.7], *Working Together Metrics* [9, Definition 4.8], and, *Joint Activity Metrics* based on *Minkowski distance*, *Hamming distance* and *Pearson's correlation coefficient* [9, Definition 4.10]. Moreover, the implementation is *easily extensible* to support more cooperative network metrics.

Within the implementation, a *prefix-tree* of (activity,resource)-pairs is built in memory. Observe however, that this implementation does not use the additional functionality of the prefix-tree based storage as described in section 3.4 of this thesis. The visualization within the implementation makes use of windows in order to *visualize changes* within the network(s). Instead of visualizing each network individually, the visualizer stores two windows each containing w networks. For each link within the network it computes the average corresponding metric value for both windows. The widths of the links within the network, are based on the relative change of the average link values. The size and colour of the resources within the network are based on their relative involvement within the network as a whole.

To illustrate the usefulness and applicability of the framework, consider Figure 9.31 where we depict a sequence of subsequent window based cooperative networks. We generated the networks using the handover of work metric (maximum distance of 5 events), using a fading factor of 0.75. We used a window-size w of 50, and, a threshold value of 0.01, i.e. only those links are displayed that within the second window have an average value of at least 0.01. In the second network, we observe a new handover of work relation of resource 11203 to *itself*.



Parameters:
thresh... 0.01
beta 0.75
k 5

Figure 9.30: Example screen shot of the OCN framework within ProM.

Thus, resource 11203 executes multiple activities in sequence. In general, we observe that most resources either execute activities in sequence, or execute multiple activities within a span of at most five activities.

9.8 Overview

In this section, we present a schematic overview of the availability of all the different implementations discussed in this chapter. Consider Table 9.1, on page 248, in which we present the availability of different functionality as presented in this chapter. Note that the overview table only considers event stream based techniques, i.e. we omit the stream generation techniques as presented in section 9.2 and the ILP based discovery approach presented in subsection 9.5.2. Furthermore, as the code related to computing prefix-alignments works on the basis of conventional event logs, we indicate $\sim$ for static streams, rather than $\checkmark$, i.e. it works with static data. Furthermore, note that we explicitly list the contributions in terms of implementations in the context of this thesis, e.g. there exists an offline implementation of the Alpha Miner in the ProM framework[5].

[5] Since this implementation is not developed in the context of this thesis, we use a $\times$-symbol for the static variant of the Alpha Miner

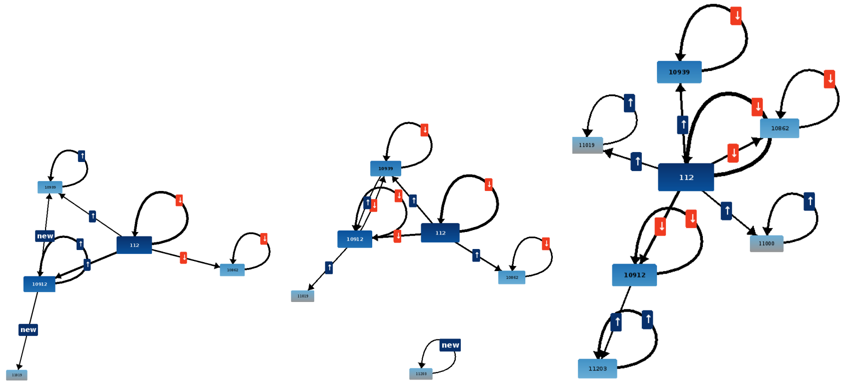


Figure 9.31: Three consecutive window-based cooperative networks based on an event stream of the BPI Challenge 2012 event log [47].

Table 9.1: Schematic overview of the availability of the implementations in ProM and RapidProM, developed in the context of this thesis. Each cell indicates on what type of input data the different techniques work.

Category	***Technique***	***ProM***		***RapidProM***	
		Live	*Static*	*Live*	*Static*
Storage					
	Sliding Window	✓	×	×	✓
	Reservoir Sampling (Event Level)	✓	×	×	✓
	Reservoir Sampling (Case Level)	✓	×	×	✓
	Reservoir Sampling (Prefix-tree)	✓	×	×	✓
Filtering					
	Automaton Based Filter	✓	×	×	✓
Discovery					
	Alpha	✓	×	✓	×
	Inductive	✓	×	✓	×
	ILP	✓	×	×	×
	Transition Systems	✓	×	×	×
Conformance Checking					
	Prefix-Alignments	×	∼	×	∼
Enhancement					
	Social Networks	✓	×	×	×

9.9 Conclusion

In this section, we conclude and reflect on the various implementations presented in this chapter. We summarize the main contributions, after which we discuss the limitations of the implementations described. Finally, we discuss open challenges and identify possible avenues for future research.

9.9.1 Contributions

We presented several implementations of the different concepts, algorithms and techniques discussed in this chapter. Most techniques are available both in ProM and RapidProM, however, in some cases, the type of input data used differs. In particular, most of the techniques ported to RapidProM are intended to be used for (repeated) scientific experiments and are therefore typically based on a strictly finite sequence of events, i.e. a static event stream. As indicated in each section, the code related to the implementations is distributed across several different *ProM packages*.

9.9.2 Limitations

It is important to note that the implementations presented here are of prototypical fashion and serve as a basis for research. In particular, most studies performed in this thesis are of a conceptual nature, i.e. we study the behaviour of several algorithms under the assumption that streams of events are used, rather than event logs. As such, a direct adoption of the code in enterprise-level software most likely leads to performance and/or scalability issues.

In the light of the aforementioned, the implementations of the different algorithms presented in this chapter primarily serve a two-fold goal:

- *Proof of concept*

  The implementation serves as a tangible proof of the applicability of the algorithms as presented in this thesis in the context of event streams.

- *Experimentation*

  The main purpose of the implementations is to aid in setting up experiments to test the quality and/or performance of the algorithms presented in this thesis. For example, the implementation related to conformance checking is purely intended to allow us to verify how the computation of prefix alignments behaves with respect to conventional alignment computation. For this particular task, an actual adoption of actual streams of events is not needed.

9.9.3 Open Challenges & Future Work

In this thesis, we have not considered the integration of the algorithms implemented in ProM, with streaming data architectures such as apache kafka (`http://kafka.apache.org/`), apache spark (`https://spark.apache.org/`) and/or apache flink (`https://flink.apache.org/`). Providing such integration is interesting in order to adopt the techniques presented here into enterprise-level solutions. It is furthermore interesting to build dedicated implementations of the algorithms in these (sometimes distributed) environments. In some cases this allows us to exploit the typically distributed nature of these architectures.

Chapter 10

Conclusions

This chapter concludes the thesis. We discuss the main contributions and their relation to the research questions as introduced in chapter 1. Whereas the individual chapters highlighted the limitations of the individual contributions, section 10.2 presents the limitations and boundaries of the work presented, from a broader perspective. Similarly, in section 10.3, we discuss the open issues that are not covered in this thesis. Finally, in section 10.4, we highlight several interesting avenues for future work.

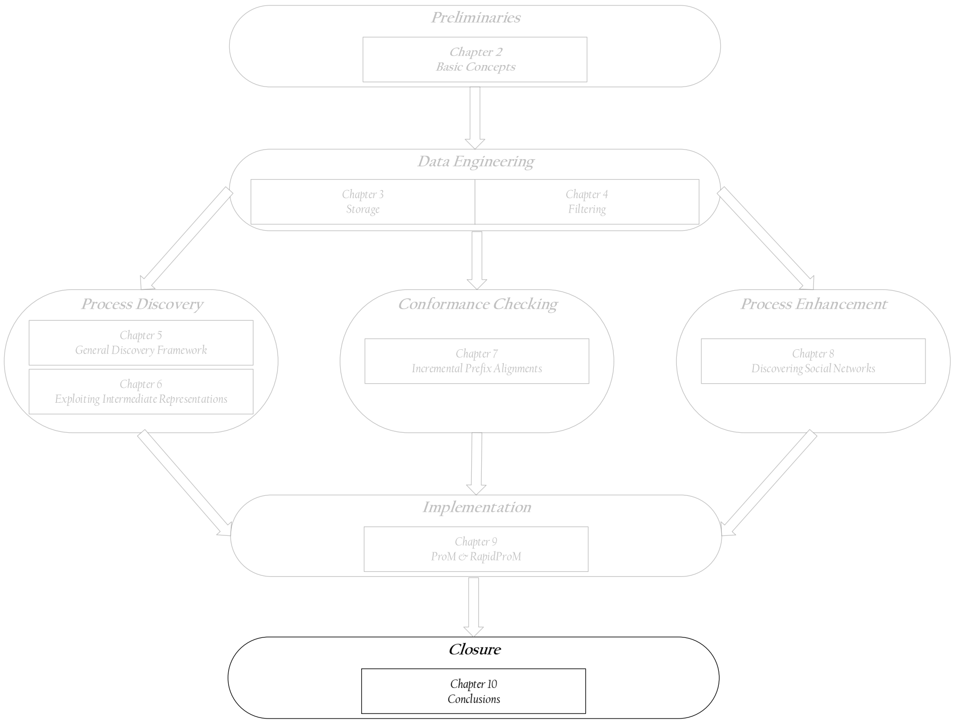


Figure 10.1: The contents of this chapter, i.e. a conclusion regarding the research covered in this thesis, highlighted in the context of the general structure of this thesis.

10.1 Contributions

In this thesis, we proposed several techniques that enable us to apply process mining on the basis of event streams. The techniques cover event stream based process discovery, conformance checking and process enhancement. Moreover, most of the techniques presented have an accompanying implementation in the process mining software platforms ProM and/or RapidProM.

The main research goals of this thesis can be grouped using the main pillars of process mining, i.e. process discovery, conformance checking and enhancement. Here, we revisit these research goals and indicate in what way the contents presented in this thesis contribute to the fulfilment of these respective research goals.

- ***Research Goal I;*** *Development/design of general purpose techniques for high-quality, efficient event data storage.*

  In chapter 3, we presented and formalized the notion of an *event store*, which represents a finite view of the event data observed on the event stream. We provided several possible instantiations of event stores using existing data storage techniques originating from different areas of data stream processing. Moreover, we compared and implemented different proposed instantiations. Additionally, we presented a novel storage technique, i.e. storing control-flow oriented prefix trees, that exploits control-flow similarity amongst different process instances. The experiments conducted in the context of chapter 3 indicate that prefix-tree based storage allows us to decrease the overall memory footprint, when compared to alternative approaches. Moreover, we are able to slightly improve the quality of the process mining analyses applied on top of prefix-tree based storage, yet the increase is not significant.

  In chapter 4, we presented a filtering technique that allows us to identify and remove *infrequent behaviour* from event streams. The filter acts as a *stream processor*, i.e. both its input and output are a stream of events. The experiments conducted in the context of chapter 4 show that we are able to achieve high filtering accuracy for different instantiations of the filter. Moreover, they indicate that the proposed filter significantly increases the accuracy of state-of-the-art online drift detection techniques.

- ***Research Goal II;*** *Development/design of specialized techniques for efficient, event stream based, process discovery.*

  In chapter 5, we presented a general framework for the purpose of online process discovery. The framework conceptually describes a high-level architecture on the basis of the data abstractions (intermediate representations) used by existing process discovery algorithms. Different instantiations of the architecture are provided, which use a minimal memory footprint, i.e. by storing the least amount of data needed to reconstruct the algorithm's intermediate representation. The experiments conducted in the context of chapter 5, show that instantiations of the architecture are able to accurately capture process behaviour originating from a steady state-based process. Furthermore, in the case of concept drift, we observe that the size of the internal data structure used impacts both the resulting process model quality and the drift detection point.

  In chapter 6, we have shown that we are able to, for ILP-based process discovery, exploit its internal representation to such extent that we are able to guarantee both structural and behavioural properties of the discovered process models. Moreover, we presented an internal filtering method, built on top of the underlying intermediate representation, that allows us to increase the overall quality of the discovered process models. The experiments conducted in the context of chapter 6, confirm that the techniques presented enable us to find meaningful Petri net structures in data consisting of exceptional behaviour, using

ILP-based process discovery as an underlying technique. The proposed internal filter proves to be comparable to an alternative state-of-the-art integrated discovery approach.

- ***Research Goal III;*** *Development/design of specialized techniques for efficient, event stream based, conformance checking.*

  In chapter 7, we presented a novel algorithm for the purpose of online, event stream based, conformance checking. The technique uses a greedy incremental algorithm, which computes prefix-alignments, i.e. the most likely explanation of observed behaviour in terms of a given reference model, whilst accounting for future behaviour. We have proven that, under certain conditions, the cost of the computed prefix-alignments are underestimating the final deviation costs, i.e. upon completion of the process instance. The experiments conducted in the context of chapter 7, show that the greedy algorithm is able to effectively prune the state-space of the underlying prefix-alignment search.

- ***Research Goal IV;*** *Development/design of specialized techniques for efficient, event stream based, process enhancement.*

  In chapter 8, we presented an assessment of the computation of social networks in event-stream-based settings. In particular, we focussed on the computational feasibility in terms of incremental network updates of *handover-of-work networks*. The experiments conducted in the context of chapter 8, show that we are able to compute social networks in an event stream setting. As such, these results indicate that there are social network metrics suitable for real-time incremental update strategies.

10.2 Limitations

Throughout the different chapters of this thesis, we identified different limitations related to different aspects of the different techniques presented. In this section, we present these limitations in a global, event-stream-oriented context.

- The algorithms and techniques presented in this thesis aim to both study and enable the application of process mining in the context of event streams. Currently, however, the demand for such techniques in real-life applications is rather limited. As such, apart from using existing event logs as a basis for streaming data, no actual “live” streams of event data have been used in the experiments performed in the context of this research.[1] It is therefore hard to assess the potential business impact and/or feasibility of the techniques presented in this thesis. With the potential advent and/or availability of event streams from actual business processes, we are able to more accurately assess the actual business impact and/or feasibility of the techniques presented in this thesis.

- Observe that the implementations as presented in chapter 9, are, as indicated, intended to serve as a proof of concept. As such, in some cases the code is only used in the context of static streams, and/or event logs. Observe that such implementation allows us to study several interesting phenomena of the algorithms presented, i.e. both in terms of (memory) efficiency as in terms of the quality of the results produced. However, it is likely that the implementations, as presented in this thesis, do not scale accordingly when applied in actual online (business) process contexts. As such, the implementations need to be used as a guiding basis for further development of industry level, highly scalable applications.

[1] In this regard, the `meetup.com` stream can be regarded as a real live stream. However, there is no clear underlying notion of process within that stream.

10.3 Open Challenges

The work presented in this thesis considers the application of process mining techniques in the context of streaming event data. Within the field of process mining, this is a relatively new avenue of research. As such, during the course of conducting the research as reported on in this thesis, several interesting open challenges came to light. In this section, we discuss these challenges in more detail.

- *Purely incremental process discovery*;

  In chapter 5, we presented a generic architecture that aims at minimizing the memory footprint of online process discovery algorithms, by storing/approximating the intermediate representation used by the respective algorithm. Within the architecture, we advocate the reuse of the existing intermediate representation as used by the underlying conventional discovery algorithm. However, in several cases, such an approach can lead to unnecessary rework. For example, it is possible that a sequence of subsequently discovered process models is in fact exactly equal. It, therefore, remains an open challenge to assess what class of existing algorithms allow for a truly incremental discovery scheme.

  When we consider the Inductive Miner [78], it seems a natural approach to assess what impact a new directly follows relation has on the underlying hierarchy of the currently discovered process model. Likewise, in case of ILP-based mining, we are able to relatively efficiently decide whether new behaviour generates new constraints and/or violates already existing places in the discovered process model. Using such type of analyses allows us to determine whether the previous model is still valid or not. Furthermore, it is interesting to study, in case we observe that the previously discovered model is no longer valid, to what degree we are able to reuse certain parts of the previous model and only perform local repair actions in the previous model.

- *Providing guarantees on trace initialization and termination*;

  The vast majority of existing process mining algorithms heavily relies on the notion of explicit knowledge of trace initialization and trace termination. For example, the Alpha algorithm uses this explicit knowledge to deduce what transitions in the discovered Petri net need to be connected to the source/sink place. Similarly, the Inductive Miner uses this knowledge to identify a partitioning of the directly follows abstraction. Furthermore, note that the formal properties that we are able to guarantee with ILP-based process discovery, cf. chapter 6, heavily rely on trace termination, i.e. by means of the emptiness after completion constraints.

  Also in the context of conformance checking, explicit knowledge of trace initialization and termination is essential. Clearly, when we perform conformance checking on trace behaviour that is in fact not describing the process instance from the start, we typically indicate that some part of the behaviour is missing. Furthermore, when we are guaranteed that the trace behaviour observed is final, we are able to assess whether termination of the process instance, as captured by the event stream, is in fact in line with the reference model.

  As the aforementioned examples indicate, explicit knowledge of trace initialization and termination, is essential. It remains an open challenge to develop dedicated techniques, possibly on top of the notion of event stores as presented in chapter 3, that allow us to improve guarantees with respect to the traces captured. Moreover, it is interesting to subdivide the traces stored into two subsets, i.e. traces that have a high certainty of being complete versus ongoing cases.

- *Advanced process monitoring*;

  The streaming model, and, correspondingly, event streams, are a natural fit with the notion of *process monitoring*, i.e. observing and potentially steering the process under study. In the context of this thesis, process monitoring is only partially covered in chapter 7, i.e. it allows us to determine potential non-conformance of running process instances. However, there are a multitude of additional interesting monitoring-related questions, e.g. in the context of compliance, it is interesting to study whether non-compliance is increasing or decreasing over time. Other interesting aspects are, for example, related to performance, i.e. is a certain case likely to meet a given deadline, and concept drift, i.e. investigating whether there is a clear indication that the execution of the process changed. To answer such questions, we need intelligent data storage solutions, that integrate the additional (derived) data within the current view of the process. It is particularly interesting, to study the impact of approximation errors, i.e. inherently present in most data storage techniques developed for the purpose of streaming data, on the computed statistics.

10.4 Future Work

From the limitations and challenges presented in section 10.2 and section 10.3, we identify the following avenues for future work.

First, we identify the need for performing a set of case studies regarding the adoption of event stream based process discovery in industry settings. Using such case studies, we are able to asses, to what degree the different main branches of process mining actually lead to novel results and insights, from a business perspective. In particular, online conformance seems to be a promising enabler for control-flow oriented process monitoring. More specifically, it enables the process owner to pro-actively alter the execution of the ongoing process. In line with these case studies, it is interesting to adopt the techniques mentioned in this thesis in *big data ecosystems*, e.g. Apache Spark (`https://spark.apache.org/`), tailored towards large-scale distributed stream processing.

Within the techniques presented in this thesis, the basic requirements of data-stream-based mining, cf. section 1.2, are often taken into account explicitly. Recall that we aim to limit memory usage/process mining to be polylogarithmic in the size of the stream. As most of the algorithms presented here are built on top of existing algorithms, these requirements are often met by design. However, in this thesis, the justification of the aforementioned claim is most often based on experimental results and evidence, rather than theoretical results. It is hence interesting to provide and/or more thoroughly investigate the common complexity classes of event stream based process mining techniques.

Bibliography

[1] IEEE Standard for eXtensible Event Stream (XES) for Achieving Interoperability in Event Logs and Event Streams. *IEEE Std 1849-2016*, pages 1–50, Nov 2016.

[2] W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[3] W. M. P. van der Aalst. Decomposing Petri Nets for Process Mining: A Generic Approach. *Distributed and Parallel Databases*, 31(4):471–507, 2013.

[4] W. M. P. van der Aalst. *Process Mining - Data Science in Action, Second Edition*. Springer, 2016.

[5] W. M. P. van der Aalst. Relating Process Models and Event Logs - 21 Conformance Propositions. In W. M. P. van der Aalst, R. Bergenthum, and J. Carmona, editors, *Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data 2018 Satellite event of the conferences: 39th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2018 and 18th International Conference on Application of Concurrency to System Design ACSD 2018, Bratislava, Slovakia, June 25, 2018.*, volume 2115 of *CEUR Workshop Proceedings*, pages 56–74. CEUR-WS.org, 2018.

[6] W. M. P. van der Aalst, A. Adriansyah, and B. F. van Dongen. Replaying History on Process Models for Conformance Checking and Performance Analysis. *Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery*, 2(2):182–192, 2012.

[7] W. M. P. van der Aalst, A. Bolt, and S. J. van Zelst. RapidProM: Mine Your Processes and Not Just Your Data. *CoRR*, abs/1703.03740, 2017.

[8] 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 Asp. Comput.*, 23(3):333–363, 2011.

[9] W. M. P. van der Aalst, H. A. Reijers, and M. Song. Discovering Social Networks from Event Logs. *Computer Supported Cooperative Work*, 14(6):549–593, 2005.

[10] W. M. P. van der Aalst, V. Rubin, H. M. W. Verbeek, B. F. van Dongen, E. Kindler, and C. W. Günther. Process Mining: A Two-Step Approach to Balance Between Underfitting and Overfitting. *Software and System Modeling*, 9(1):87–111, 2010.

[11] W. M. P. van der Aalst, A. J. M. M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Trans. Knowl. Data Eng.*, 16(9):1128–1142, 2004.

[12] H. Abdi. The kendall rank correlation coefficient. *Encyclopedia of Measurement and Statistics. Sage, Thousand Oaks, CA*, pages 508–510, 2007.

[13] A. Adriansyah. *Aligning Observed and Modeled Behavior*. PhD thesis, Eindhoven University of Technology, Department of Mathematics and Computer Science, July 2014.

[14] C. C. Aggarwal. On Biased Reservoir Sampling in the Presence of Stream Evolution. In U. Dayal, K. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y. Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 607–618. ACM, 2006.

[15] C. C. Aggarwal. *Data Mining - The Textbook*. Springer, 2015.

[16] G. Alonso, P. Dadam, and M. Rosemann, editors. *Business Process Management, 5th International Conference, BPM 2007, Brisbane, Australia, September 24-28, 2007, Proceedings*, volume 4714 of *Lecture Notes in Computer Science*. Springer, 2007.

[17] A. Appice, M. Di Pietro, C. Greco, and D. Malerba. Discovering and Tracking Organizational Structures in Event Logs. In M. Ceci, C. Loglisci, G. Manco, E. Masciari, and Z. W. Ras, editors, *New Frontiers in Mining Complex Patterns - 4th International Workshop, NFMCP 2015, Held in Conjunction with ECML-PKDD 2015, Porto, Portugal, September 7, 2015, Revised Selected Papers*, volume 9607 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2015.

[18] 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. *CoRR*, abs/1705.02288, 2017.

[19] E. Badouel, L. Bernardinello, and P. Darondeau. Polynomial Algorithms for the Synthesis of Bounded Nets. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*, volume 915 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 1995.

[20] E. Badouel, L. Bernardinello, and P. Darondeau. *Petri Net Synthesis*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2015.

[21] E. Badouel and P. Darondeau. Theory of Regions. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 529–586. Springer, 1996.

[22] R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Process Mining Based on Regions of Languages. In Alonso et al. [16], pages 375–383.

[23] R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Synthesis of Petri Nets from Finite Partial Languages. *Fundam. Inform.*, 88(4):437–468, 2008.

[24] L. Bernardinello. Synthesis of Net Systems. In M. A. Marsan, editor, *Application and Theory of Petri Nets 1993, 14th International Conference, Chicago, Illinois, USA, June 21-25, 1993, Proceedings*, volume 691 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 1993.

[25] A. Bifet and R. Gavaldà. Learning from Time-Changing Data with Adaptive Windowing. In *Proceedings of the Seventh SIAM International Conference on Data Mining, April 26-28, 2007, Minneapolis, Minnesota, USA*, pages 443–448. SIAM, 2007.

[26] V. Bloemen, S. J. van Zelst, W. M. P. van der Aalst, B. F. van Dongen, and J. van de Pol. Maximizing Synchronization for Aligning Observed and Modelled Behaviour. In *Business Process Management - 16th International Conference, BPM 2018, Sydney, NSW, Australia, September 9-14, 2018, Proceedings*, pages 233–249, 2018.

[27] A. Bolt and W. M. P. van der Aalst. Multidimensional Process Mining Using Process Cubes. In Gaaloul et al. [63], pages 102–116.

[28] A. Bolt, M. de Leoni, and W. M. P. van der Aalst. Scientific Workflows for Process Mining: Building Blocks, Scenarios, and Implementation. *STTT*, 18(6):607–628, 2016.

[29] R. P. J. C. Bose, W. M. P. van der Aalst, I. Zliobaite, and M. Pechenizkiy. Dealing With Concept Drifts in Process Mining. *IEEE Trans. Neural Netw. Learning Syst.*, 25(1):154–171, 2014.

[30] S. Boushaba, M. I. Kabbaj, and Z. Bakkoury. Process Discovery - Automated Approach for Block Discovery. In J. Filipe and L. A. Maciaszek, editors, *ENASE 2014 - Proceedings of the 9th International Conference on Evaluation of Novel Approaches to Software Engineering, Lisbon, Portugal, 28-30 April, 2014*, pages 204–211. SciTePress, 2014.

[31] S. Boushaba, M. I. Kabbaj, Z. Bakkoury, and S. M. Matais. Process Mining: On the Fly Process Discovery. In A. El Oualkadi, F. Choubani, and A. El Moussati, editors, *Proceedings of the Mediterranean Conference on Information & Communication Technologies 2015*, pages 79–89, Cham, 2016. Springer International Publishing.

[32] A. Burattin. Online Conformance Checking for Petri Nets and Event Streams. In Clarisó et al. [39].

[33] A. Burattin and J. Carmona. A Framework for Online Conformance Checking. In *Proceedings of the 13th International Workshop on Business Process Intelligence (BPI'17)*, 2017.

[34] A. Burattin, M. Cimitile, F. M. Maggi, and A. Sperduti. Online Discovery of Declarative Process Models from Event Streams. *IEEE Trans. Services Computing*, 8(6):833–846, 2015.

[35] A. Burattin, A. Sperduti, and W. M. P. van der Aalst. Control-flow Discovery from Event Streams. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2014, Beijing, China, July 6-11, 2014*, pages 2420–2427. IEEE, 2014.

[36] A. Burattin, S. J. van Zelst, A. Armas-Cervantes, B. F. , van Dongen, and J. Carmona. Online Conformance Checking Using Behavioural Patterns. In *Business Process Management - 16th International Conference, BPM 2018, Sydney, NSW, Australia, September 9-14, 2018, Proceedings*, pages 250–267, 2018.

[37] J. Carmona and J. Cortadella. Process Discovery Algorithms Using Numerical Abstract Domains. *IEEE Trans. Knowl. Data Eng.*, 26(12):3064–3076, 2014.

[38] V. Chandola, A. Banerjee, and V. Kumar. Anomaly Detection for Discrete Sequences: A Survey. *IEEE Trans. Knowl. Data Eng.*, 24(5):823–839, 2012.

[39] R. Clarisó, H. Leopold, J. Mendling, W. M. P. van der Aalst, A. Kumar, B. T. Pentland, and M. Weske, editors. *Proceedings of the BPM Demo Track and BPM Dissertation Award co-located with 15th International Conference on Business Process Modeling (BPM 2017), Barcelona, Spain, September 13, 2017*, volume 1920 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017.

[40] R. Conforti, M. La Rosa, and A. H. M. ter Hofstede. Filtering Out Infrequent Behavior from Business Process Event Logs. *IEEE Trans. Knowl. Data Eng.*, 29(2):300–314, 2017.

[41] G. Cormode and M. Hadjieleftheriou. Methods for Finding Frequent Items in Data Streams. *VLDB J.*, 19(1):3–20, 2010.

[42] G. Cormode, V. Shkapenyuk, D. Srivastava, and B. Xu. Forward Decay: A Practical Time Decay Model for Streaming Systems. In Y. E. Ioannidis, D. L. Lee, and R. T. Ng, editors, *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 138–149. IEEE Computer Society, 2009.

[43] P. Darondeau. Deriving Unbounded Petri Nets from Formal Languages. In D. Sangiorgi and R. de Simone, editors, *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*, volume 1466 of *Lecture Notes in Computer Science*, pages 533–548. Springer, 1998.

[44] M. Dees, M. de Leoni, and F. Mannhardt. Enhancing Process Models to Improve Business Performance: A Methodology and Case Studies. In H. Panetto, C. Debruyne, W. Gaaloul, M. P. Papazoglou, A. Paschke, C. A. Ardagna, and R. Meersman, editors, *On the Move to Meaningful Internet Systems. OTM 2017 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2017, Rhodes, Greece, October 23-27, 2017, Proceedings, Part I*, volume 10573 of *Lecture Notes in Computer Science*, pages 232–251. Springer, 2017.

[45] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Frequency Estimation of Internet Packet Streams with Limited Space. In R. H. Möhring and R. Raman, editors, *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2461 of *Lecture Notes in Computer Science*, pages 348–360. Springer, 2002.

[46] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959.

[47] B. F. van Dongen. BPI Challenge 2012, 2012.

[48] B. F. van Dongen. BPI Challenge 2015, 2015.

[49] B. F. van Dongen, A. K. A. de Medeiros, and L. Wen. Process Mining: Overview and Outlook of Petri Net Discovery Algorithms. *Trans. Petri Nets and Other Models of Concurrency*, 2:225–242, 2009.

[50] B. F. van Dongen and S. Shabani. Relational XES: Data Management for Process Mining. In J. Grabis and K. Sandkuhl, editors, *Proceedings of the CAiSE 2015 Forum at the 27th International Conference on Advanced Information Systems Engineering co-located with 27th International Conference on Advanced Information Systems Engineering (CAiSE 2015), Stockholm, Sweden, June 10th, 2015.*, volume 1367 of *CEUR Workshop Proceedings*, pages 169–176. CEUR-WS.org, 2015.

[51] M. Dumas, M. La Rosa, J. Mendling, and H. A. Reijers. *Fundamentals of Business Process Management*. Springer, 2013.

[52] A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures. Part I: Basic Notions and the Representation Problem. *Acta Inf.*, 27(4):315–342, 1990.

[53] A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures. Part II: State Spaces of Concurrent Systems. *Acta Inf.*, 27(4):343–368, 1990.

[54] J. Evermann. Scalable Process Discovery Using Map-Reduce. *IEEE Trans. Services Computing*, 9(3):469–481, 2016.

[55] J. Evermann, J. Rehse, and P. Fettke. Process Discovery from Event Stream Data in the Cloud - A Scalable, Distributed Implementation of the Flexible Heuristics Miner on the Amazon Kinesis Cloud Infrastructure. In *2016 IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2016, Luxembourg, December 12-15, 2016*, pages 645–652. IEEE Computer Society, 2016.

[56] D. Fahland and W. M. P. van der Aalst. Model Repair - Aligning Process Models to Reality. *Inf. Syst.*, 47:220–243, 2015.

[57] W. Fan and A. Bifet. Mining Big Data: Current Status, and Forecast to the Future. *SIGKDD Explorations*, 14(2):1–5, 2012.

[58] M. Fani Sani, S. J. van Zelst, and W. M. P. van der Aalst. Improving Process Discovery Results by Filtering Outliers using Conditional Behavioural Probabilities. In *Proceedings of the 13th International Workshop on Business Process Intelligence (BPI'17)*, 2017.

[59] M. Fani Sani, S. J. van Zelst, and W. M. P. van der Aalst. Improving Process Discovery Results by Filtering Outliers Using Conditional Behavioural Probabilities. In *Business Process Management Workshops - BPM 2017 International Workshops, Barcelona, Spain, September 10-11, 2017, Revised Papers*, pages 216–229, 2017.

[60] M. Fani Sani, S. J. van Zelst, and W. M. P. van der Aalst. Applying Sequence Mining for Outlier Detection in Process Mining. In *On the Move to Meaningful Internet Systems. OTM 2018 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22-26, 2018, Proceedings, Part II*, pages 98–116, 2018.

[61] M. Fani Sani, S. J. van Zelst, and W. M. P. van der Aalst. Repairing Outlier Behaviour in Event Logs. In W. Abramowicz and A. Paschke, editors, *Business Information Systems - 21st International Conference, BIS 2018, Berlin, Germany, July 18-20, 2018, Proceedings*, volume 320 of *Lecture Notes in Business Information Processing*, pages 115–131. Springer, 2018.

[62] D. R. Ferreira and C. Alves. Discovering User Communities in Large Event Logs. In F. Daniel, K. Barkaoui, and S. Dustdar, editors, *Business Process Management Workshops - BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I*, volume 99 of *Lecture Notes in Business Information Processing*, pages 123–134. Springer, 2011.

[63] K. Gaaloul, R. Schmidt, S. Nurcan, S. Guerreiro, and Q. Ma, editors. *Enterprise, Business-Process and Information Systems Modeling - 16th International Conference, BPMDS 2015, 20th International Conference, EMMSAD 2015, Held at CAiSE 2015, Stockholm, Sweden, June 8-9, 2015, Proceedings*, volume 214 of *Lecture Notes in Business Information Processing*. Springer, 2015.

[64] J. Gama. *Knowledge Discovery from Data Streams*. Chapman and Hall / CRC Data Mining and Knowledge Discovery Series. CRC Press, 2010.

[65] A. Gandomi and M. Haider. Beyond the Hype: Big Data Concepts, Methods, and Analytics. *Int J. Information Management*, 35(2):137–144, 2015.

[66] C. W. Günther and W. M. P. van der Aalst. Fuzzy Mining - Adaptive Process Simplification Based on Multi-perspective Metrics. In Alonso et al. [16], pages 328–343.

[67] M. Gupta, J. Gao, C. C. Aggarwal, and J. Han. Outlier Detection for Temporal Data: A Survey. *IEEE Trans. Knowl. Data Eng.*, 26(9):2250–2267, 2014.

[68] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968.

[69] M. Hassani, S. Siccha, F. Richter, and T. Seidl. Efficient Process Discovery From Event Streams Using Sequential Pattern Mining. In *IEEE Symposium Series on Computational Intelligence, SSCI 2015, Cape Town, South Africa, December 7-10, 2015*, pages 1366–1373. IEEE, 2015.

[70] S. Hernández, J. Ezpeleta, S. J. van Zelst, and W. M. P. van der Aalst. Assessing Process Discovery Scalability in Data Intensive Environments. In I. Raicu, O. F. Rana, and R. Buyya, editors, *2nd IEEE/ACM International Symposium on Big Data Computing, BDC 2015, Limassol, Cyprus, December 7-10, 2015*, pages 99–104. IEEE Computer Society, 2015.

[71] S. Hernández, S. J. van Zelst, J. Ezpeleta, and W. M. P. van der Aalst. Handling Big(ger) Logs: Connecting ProM 6 to Apache Hadoop. In F. Daniel and S. Zugal, editors, *Proceedings of the BPM Demo Session 2015 Co-located with the 13th International Conference on Business Process Management (BPM 2015), Innsbruck, Austria, September 2, 2015.*, volume 1418 of *CEUR Workshop Proceedings*, pages 80–84. CEUR-WS.org, 2015.

[72] K. Jensen and L. M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

[73] Jouck, T. and Depaire, B. PTandLogGenerator: A Generator for Artificial Event Data. In L. Azevedo and C. Cabanillas, editors, *Proceedings of the BPM Demo Track 2016 Co-located with the 14th International Conference on Business Process Management (BPM 2016), Rio de Janeiro, Brazil, September 21, 2016.*, volume 1789 of *CEUR Workshop Proceedings*, pages 23–27. CEUR-WS.org, 2016.

[74] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A Simple Algorithm for Finding Frequent Elements in Streams and Bags. *ACM Trans. Database Syst.*, 28:51–55, 2003.

[75] M. G. Kendall. Rank Correlation Methods. 1955.

[76] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 3rd Edition*. Addison-Wesley, 1998.

[77] S. J. J. Leemans, F. Dirk, and W. M. P. van der Aalst. Scalable Process Discovery with Guarantees. In Gaaloul et al. [63], pages 85–101.

[78] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst. Discovering Block-Structured Process Models from Event Logs - A Constructive Approach. In J. M. Colom and J. Desel, editors, *Application and Theory of Petri Nets and Concurrency - 34th International Conference, PETRI NETS 2013, Milan, Italy, June 24-28, 2013. Proceedings*, volume 7927 of *Lecture Notes in Computer Science*, pages 311–329. Springer, 2013.

[79] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst. Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour. In N. Lohmann, M. Song, and P. Wohed, editors, *Business Process Management Workshops - BPM 2013 International Workshops, Beijing, China, August 26, 2013, Revised Papers*, volume 171 of *Lecture Notes in Business Information Processing*, pages 66–78. Springer, 2013.

[80] M. de Leoni and W. M. P. van der Aalst. Data-Aware Process Mining: Discovering Decisions in Processes using Alignments. In S. Y. Shin and J. C. Maldonado, editors, *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pages 1454–1461. ACM, 2013.

[81] M. de Leoni and F. Mannhardt. Road Traffic Fine Management Process, 2015.

[82] R. Lorenz and G. Juhás. Towards Synthesis of Petri Nets from Scenarios. In S. Donatelli and P. S. Thiagarajan, editors, *Petri Nets and Other Models of Concurrency - ICATPN 2006, 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, Turku, Finland, June 26-30, 2006, Proceedings*, volume 4024 of *Lecture Notes in Computer Science*, pages 302–321. Springer, 2006.

[83] R. Lorenz, S. Mauser, and G. Juhás. How to Synthesize Nets from Languages - A Survey. In S. G. Henderson, B. Biller, M. H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, editors, *Proceedings of the Winter Simulation Conference, WSC 2007, Washington, DC, USA, December 9-12, 2007*, pages 637–647. WSC, 2007.

[84] G. S. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 346–357. Morgan Kaufmann, 2002.

[85] F. Mannhardt. Sepsis Cases - Event Log, 2016.

[86] F. Mannhardt. XESLite - Managing Large XES Event Logs in ProM. Technical report, 2016.

[87] L. Maruster, A. J. M. M. Weijters, W. M. P. van der Aalst, and A. van den Bosch. A Rule-Based Approach for Process Discovery: Dealing with Noise and Imbalance in Process Logs. *Data Min. Knowl. Discov.*, 13(1):67–87, 2006.

[88] A. I. McLeod and D. R. Bellhouse. A Convenient Algorithm for Drawing a Simple Random Sample. *Applied Statistics*, 32(2):182–184, 1983.

[89] A. K. A. de Medeiros, B. F. van Dongen, W. M. P. van der Aalst, and A. J. M. M. Weijters. Process Mining for Ubiquitous Mobile Systems: An Overview and a Concrete Algorithm. In L. Baresi, S. Dustar, H. C. Gall, and M. Matera, editors, *Ubiquitous Mobile Information and Collaboration Systems, Second CAiSE Workshop, UMICS 2004, Riga, Latvia, June 7-8, 2004, Revised Selected Papers*, volume 3272 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2004.

[90] J. Mendling and W. M. P. van der Aalst. Formalization and Verification of EPCs with OR-Joins Based on State and Context. In J. Krogstie, A. L. Opdahl, and G. Sindre, editors, *Advanced Information Systems Engineering, 19th International Conference, CAiSE 2007, Trondheim, Norway, June 11-15, 2007, Proceedings*, volume 4495 of *Lecture Notes in Computer Science*, pages 439–453. Springer, 2007.

[91] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In T. Eiter and L. Libkin, editors, *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings*, volume 3363 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.

[92] J. Munoz-Gama. *Conformance Checking and Diagnosis in Process Mining - Comparing Observed and Modeled Processes*, volume 270 of *Lecture Notes in Business Information Processing*. Springer, 2016.

[93] J. Munoz-Gama, J. Carmona, and W. M. P. van der Aalst. Single-Entry Single-Exit Decomposed Conformance Checking. *Inf. Syst.*, 46:102–122, 2014.

[94] T. Murata. Petri nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

[95] S. Muthukrishnan. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[96] Object Management Group. Business Process Model and Notation. Technical Report formal/2011-01-03, Object Management Group, 2011.

[97] A. Ostovar, A. Maaradji, M. La Rosa, A. H. M. ter Hofstede, and B. F. van Dongen. Detecting Drift from Event Streams of Unpredictable Business Processes. In *Proceedings of the 35th International Conference on Conceptual Modeling ER'16*, volume 9974 of *Lecture Notes in Computer Science*, pages 330–346. Springer, 2016.

[98] A. Pika, M. T. Wynn, C. J. Fidge, A. H. M. ter Hofstede, M. Leyer, and W. M. P. van der Aalst. An Extensible Framework for Analysing Resource Behaviour Using Event Logs. In M. Jarke, J. Mylopoulos, C. Quix, C. Rolland, Y. Manolopoulos, H. Mouratidis, and J. Horkoff, editors, *Advanced Information Systems Engineering - 26th International Conference, CAiSE 2014, Thessaloniki, Greece, June 16-20, 2014. Proceedings*, volume 8484 of *Lecture Notes in Computer Science*, pages 564–579. Springer, 2014.

[99] D. Redlich, T. Molka, W. Gilani, G. S. Blair, and A. Rashid. Constructs Competition Miner: Process Control-Flow Discovery of BP-Domain Constructs. In S. W. Sadiq, P. Soffer, and H. Völzer, editors, *Business Process Management - 12th International Conference, BPM*

2014, Haifa, Israel, September 7-11, 2014. Proceedings, volume 8659 of *Lecture Notes in Computer Science*, pages 134–150. Springer, 2014.

[100] D. Redlich, T. Molka, W. Gilani, G. S. Blair, and A. Rashid. Scalable Dynamic Business Process Discovery with the Constructs Competition Miner. In R. Accorsi, P. Ceravolo, and B. Russo, editors, *Proceedings of the 4th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2014), Milan, Italy, November 19-21, 2014.*, volume 1293 of *CEUR Workshop Proceedings*, pages 91–107. CEUR-WS.org, 2014.

[101] P. Rodríguez-Mier, A. Gonzalez-Sieira, M. Mucientes, M. Lama, and A. Bugarin. Hipster: An Open Source Java Library for Heuristic Search. In *2014 9th Iberian Conference on Information Systems and Technologies (CISTI)*. IEEE, June 2014.

[102] A. Rozinat and W. M. P. van der Aalst. Conformance Checking of Processes Based on Monitoring Real Behavior. *Inf. Syst.*, 33(1):64–95, 2008.

[103] G. Da San Martino, N. Navarin, and A. Sperduti. A Lossy Counting Based Approach for Learning on Streams of Graphs on a Budget. In F. Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 1294–1301. IJCAI/AAAI, 2013.

[104] J. C. Schlimmer and R. H. Granger. Beyond Incremental Processing: Tracking Concept Drift. In T. Kehler, editor, *Proceedings of the 5th National Conference on Artificial Intelligence. Philadelphia, PA, August 11-15, 1986. Volume 1: Science.*, pages 502–507. Morgan Kaufmann, 1986.

[105] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999.

[106] M. Solé and J. Carmona. Process Mining from a Basis of State Regions. In J. Lilius and W. Penczek, editors, *Applications and Theory of Petri Nets, 31st International Conference, PETRI NETS 2010, Braga, Portugal, June 21-25, 2010. Proceedings*, volume 6128 of *Lecture Notes in Computer Science*, pages 226–245. Springer, 2010.

[107] M. Song and W. M. P. van der Aalst. Towards Comprehensive Support for Organizational Mining. *Decision Support Systems*, 46(1):300–317, 2008.

[108] A. Syamsiyah, B. F. van Dongen, and W. M. P. van der Aalst. DB-XES: Enabling Process Discovery in the Large. In P. Ceravolo, C. Guetl, and S. Rinderle-Ma, editors, *Data-Driven Process Discovery and Analysis - 6th IFIP WG 2.6 International Symposium, SIMPDA 2016, Graz, Austria, December 15-16, 2016, Revised Selected Papers*, volume 307 of *Lecture Notes in Business Information Processing*, pages 53–77. Springer, 2016.

[109] N. Tax, X. Lu, N. Sidorova, D. Fahland, and W. M. P. van der Aalst. The Imprecisions of Precision Measures in Process Mining. *Inf. Process. Lett.*, 135:1–8, 2018.

[110] N. Tax, S. J. van Zelst, and I. Teinemaa. An Experimental Evaluation of the Generalizing Capabilities of Process Discovery Techniques and Black-Box Sequence Models. In *Enterprise, Business-Process and Information Systems Modeling - 19th International Conference, BPMDS 2018, 23rd International Conference, EMMSAD 2018, Held at CAiSE 2018, Tallinn, Estonia, June 11-12, 2018, Proceedings*, pages 165–180, 2018.

[111] F. Taymouri and J. Carmona. A Recursive Paradigm for Aligning Observed Behavior of Large Structured Process Models. In M. La Rosa, P. Loos, and O. Pastor, editors, *BPM 2016, Rio de Janeiro, Brazil, September 18-22, 2016. Proceedings*, volume 9850 of *Lecture Notes in Computer Science*, pages 197–214. Springer, 2016.

[112] B. Vázquez-Barreiros, S. J. van Zelst, J. C. A. M. Buijs, M. Lama, and M. Mucientes. Repairing Alignments: Striking the Right Nerve. In *Enterprise, Business-Process and Information Systems Modeling - 17th International Conference, BPMDS 2016, 21st International Conference, EMMSAD 2016, Held at CAiSE 2016, Ljubljana, Slovenia, June 13-14, 2016, Proceedings*, pages 266–281, 2016.

[113] E. Verbeek, J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. ProM 6: The Process Mining Toolkit. In M. La Rosa, editor, *Proceedings of the Business Process Management 2010 Demonstration Track, Hoboken, NJ, USA, September 14-16, 2010*, volume 615 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.

[114] H. M. W. Verbeek, J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. XES, XESame, and ProM 6. In P. Soffer and E. Proper, editors, *Information Systems Evolution - CAiSE Forum 2010, Hammamet, Tunisia, June 7-9, 2010, Selected Extended Papers*, volume 72 of *Lecture Notes in Business Information Processing*, pages 60–75. Springer, 2010.

[115] J. S. Vitter. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[116] T. Vogelgesang. Improving Interactivity in Multidimensional Process Mining: The Interactive PMCube Explorer Tool. In Clarisó et al. [39].

[117] T. Vogelgesang and H. Appelrath. PMCube: A Data-Warehouse-Based Approach for Multidimensional Process Mining. In M. Reichert and H. A. Reijers, editors, *Business Process Management Workshops - BPM 2015, 13th International Workshops, Innsbruck, Austria, August 31 - September 3, 2015, Revised Papers*, volume 256 of *Lecture Notes in Business Information Processing*, pages 167–178. Springer, 2015.

[118] J. Wang, S. Song, X. Lin, X. Zhu, and J. Pei. Cleaning Structured Event Logs: A Graph Repair Approach. In J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman, editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 30–41. IEEE Computer Society, 2015.

[119] B. Weber, M. Reichert, and S. Rinderle-Ma. Change Patterns and Change Support Features - Enhancing Flexibility in Process-Aware Information Systems. *Data Knowl. Eng.*, 66(3):438–466, 2008.

[120] J. de Weerdt, 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. *Inf. Syst.*, 37(7):654–676, 2012.

[121] A. J. M. M. Weijters and W. M. P. van der Aalst. Rediscovering Workflow Models from Event-Based Data using Little Thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.

[122] A. J. M. M. Weijters and J. T. S. Ribeiro. Flexible Heuristics Miner (FHM). In *Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2011, part of the IEEE Symposium Series on Computational Intelligence 2011, April 11-15, 2011, Paris, France*, pages 310–317, 2011.

[123] J. M. E. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, and A. Serebrenik. Process Discovery using Integer Linear Programming. *Fundam. Inform.*, 94(3-4):387–412, 2009.

[124] M. Westergaard. Access/CPN 2.0: A High-Level Interface to Coloured Petri Net Models. In L. M. Kristensen and L. Petrucci, editors, *Applications and Theory of Petri Nets - 32nd International Conference, PETRI NETS 2011, Newcastle, UK, June 20-24, 2011. Proceedings*, volume 6709 of *Lecture Notes in Computer Science*, pages 328–337. Springer, 2011.

[125] M. Westergaard and L. M. Kristensen. The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator. In G. Franceschinis and K. Wolf, editors, *Applications and Theory of Petri Nets, 30th International Conference, PETRI NETS 2009, Paris, France, June 22-26, 2009. Proceedings*, volume 5606 of *Lecture Notes in Computer Science*, pages 313–322. Springer, 2009.

[126] M. Westergaard and T. Slaats. CPN Tools 4: A Process Modeling Tool Combining Declarative and Imperative Paradigms. In M. Fauvet and B. F. van Dongen, editors, *Proceedings of the BPM Demo sessions 2013, Beijing, China, August 26-30, 2013*, volume 1021 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[127] S. J. van Zelst, A. Bolt, and B. F. van Dongen. Tuning Alignment Computation: An Experimental Evaluation. In W. M. P. van der Aalst, R. Bergenthum, and J. Carmona, editors, *Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data 2017 Satellite event of the conferences: 38th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2017 and 17th International Conference on Application of Concurrency to System Design ACSD 2017, Zaragoza, Spain, June 26-27, 2017.*, volume 1847 of *CEUR Workshop Proceedings*, pages 6–20. CEUR-WS.org, 2017.

[128] S. J. van Zelst, A. Bolt, and B. F. van Dongen. Computing Alignments of Event Data and Process Models. *T. Petri Nets and Other Models of Concurrency*, 13:1–26, 2018.

[129] 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*, Oct 2017.

[130] S. J. van Zelst, A. Burattin, B. F. van Dongen, and H. M. W. Verbeek. Data Streams in ProM 6: A Single-node Architecture. In *Proceedings of the BPM Demo Sessions 2014 Co-located with the 12th International Conference on Business Process Management (BPM 2014), Eindhoven, The Netherlands, September 10, 2014.*, page 81, 2014.

[131] S. J. van Zelst, B. F. van Dongen, and W. M. P. van der Aalst. Avoiding Over-Fitting in ILP-Based Process Discovery. In H. R. Motahari-Nezhad, J. Recker, and M. Weidlich, editors, *Business Process Management - 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 - September 3, 2015, Proceedings*, volume 9253 of *Lecture Notes in Computer Science*, pages 163–171. Springer, 2015.

[132] S. J. van Zelst, B. F. van Dongen, and W. M. P. van der Aalst. *Filter Techniques for Region-Based Process Discovery*. BPM reports. BPMcenter.org, 2015.

[133] S. J. van Zelst, B. F. van Dongen, and W. M. P. van der Aalst. ILP-Based Process Discovery Using Hybrid Regions. In W. M. P. van der Aalst, R. Bergenthum, and J. Carmona, editors, *Proceedings of the ATAED 2015 Workshop, Satellite event of Petri Nets/ACSD 2015, Brussels, Belgium, June 22-23, 2015.*, volume 1371 of *CEUR Workshop Proceedings*, pages 47–61. CEUR-WS.org, 2015.

[134] S. J. van Zelst, B. F. van Dongen, and W. M. P. van der Aalst. Know What You Stream: Generating Event Streams from CPN Models in ProM 6. In *Proceedings of the BPM Demo Session 2015 Co-located with the 13th International Conference on Business Process Management (BPM 2015), Innsbruck, Austria, September 2, 2015*, pages 85–89, 2015.

[135] S. J. van Zelst, B. F. van Dongen, and W. M. P. van der Aalst. Online Discovery of Cooperative Structures in Business Processes. In C. Debruyne, H. Panetto, R. Meersman, T. S. Dillon, e. Kühn, D. O'Sullivan, and C. Agostino Ardagna, editors, *On the Move to Meaningful Internet Systems: OTM 2016 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2016, Rhodes, Greece, October 24-28, 2016, Proceedings*, volume 10033 of *Lecture Notes in Computer Science*, pages 210–228, 2016.

[136] S. J. van Zelst, B. F. van Dongen, and W. M. P. van der Aalst. Event Stream-Based Process Discovery Using Abstract Representations. *Knowl. Inf. Syst.*, 54(2):407–435, 2018.

[137] S. J. van Zelst, B. F. van Dongen, W. M. P. van der Aalst, and H. M. W. Verbeek. Discovering Workflow Nets Using Integer Linear Programming. *Computing*, 100(5):529–556, 2018.

[138] S. J. van Zelst, M. Fani Sani, A. Ostovar, R. Conforti, and M. La Rosa. Filtering Spurious Events from Event Streams of Business Processes. In J. Krogstie and H. A. Reijers, editors, *Advanced Information Systems Engineering - 30th International Conference, CAiSE 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings*, volume 10816 of *Lecture Notes in Computer Science*, pages 35–52. Springer, 2018.

Index

abstraction function
    event log, 46
    event stream, 122
abstraction update function, 122
alignment, 48
    cost function, 49
    prefix-alignment, 50
    unit cost function, 49
automaton, 38
    deterministic, 38
    non-deterministic, 38
    probabilistic, 39

bag, 26
binary relation, 25
    functional, 25
    injective, 25
    left-total, 25
boundedness, 36

Cartesian product, 25
case identifier, 5
causal relation oracle, 154
control-flow perspective, 43

data point, 5
data stream model, 13
    (ϵ, δ)-approximation, 16
    cash register, 13
    computation time, 15
    insert-delete, 14
    insertion-only, 13
    memory consumption, 14
    processing time, 14
    turnstile, 14
directly follows relation, 45

easy soundness, 37
endorelation, 26
    antisymmetric, 26
    irreflexive, 26
    partial order, 26
    reflexive, 26
    strict partial order, 26
    symmetric, 26
    transitive, 26
event, 40, 41
    universe, 41
event data, 4
event log, 42
    simple event log, 43
event store, 60
    update function, 61
event stream, 44

firing rule, 35
Frequent algorithm, 69
function
    codomain, 25
    domain, 25
    partial, 25
    range, 25

generalization, 13
generalized weighted objective function, 151
graph, 29
    directed, 29
    path, 29

    undirected, 29

ilp, 31
integer linear programming, 31
interval, 25

left-total, 25
liveness, 36
Lossy Counting algorithm, 69

marking, 34
matrix, 28
minimal region, 150
multiset, 26
    difference, 26
    intersection, 26
    sum, 26
    union, 26

natural numbers, 25

objective function
    generalized weighted, 151
online filter, 104
    fractional, 104
    heavy hitter, 104
    smoothed heavy hitter, 105

Parikh, 27
partial order, 26
Petri net, 33, 34
    boundedness, 36
    deadlock, 35
    firing rule, 35
    language, 35
    livelock, 35
    liveness, 36
    marking, 34
    reachable markings, 35
    safe, 36
    workflow net, 36
power set, 25
precision, 13
prefix, 27
prefix tree, 30
prefix-alignment, 50
    oracle, 51
prefix-alignment oracle, 51
prefix-closed, 27
prefix-closure, 27
probabilistic automaton, 39
process, 3
process discovery algorithm, 44
    intermeidate representation based, 46
process discovery ilp formulation, 150
process instance, 5
process mining, 3

region, 149
    matrix form, 149
    minimal, 150
relation, 25
    binary, 25
relaxed soundness, 37
replay-fitness, 13
reservoir sampling, 65

sequenc encoding
    filter, 164
sequence, 27
sequence encoding, 161
    graph, 162
sequence encoding filter, 164
sequence encoding graph, 162
set, 25
simple event log, 43
simplicity, 13
sliding window, 63
social network, 207
social network metric, 207
soundness, 37
SpaceSaving algorithm, 69
strict partial order, 26
strictly increasing, 26
subsequence, 27
    prefix, 27
    strict, 27
suffix, 28

time decay, 74
total function, 25
trace, 5, 42
tuple, 25

USE-store, 155

vector, 28
    column vector, 28
    row vector, 28

workflow net, 36
easy soundness, 37
relaxed soundness, 37
soundness, 37

Summary

Process Mining with Streaming Data

Modern information systems allow us to track, often in great detail, the execution of processes within companies. Such event data is typically stored in a company's information system and describes the execution of the process at hand. The field of *process mining* aims to translate the event data into actionable insights. As such, we identify three main branches within the field, i.e. *process discovery*, *conformance checking* and *process enhancement*. In process discovery, we aim at discovering a process model, i.e. a formal behavioural description, which describes the process as captured by the event data. In conformance checking, we aim to assess to what degree the event data is in correspondence with a given reference model, i.e. a model describing how the process ought to be executed. Finally, in process enhancement, the main goal is to improve the view of the process, i.e. by enhancing process models on the basis of facts derived from event data.

The ever-increasing automation of processes, combined with the increased interconnectivity of devices, generates enormous amounts of data, both in terms of size and velocity. The sheer complexity of such data requires us to rethink and re-engineer the way we manipulate and study it, in order to obtain meaningful insights out of it. To cope with this problem, this thesis proposes a first investigation of the application of the field of process mining on the basis of streaming event data. When applying process mining on the basis of streaming event data we assume that we are able to tap-in to a digital stream of events, on which we keep receiving newly executed events. As opposed to the conventionally used static event logs, the available data of the underlying process is constantly changing. Moreover, we assume the event streams to be potentially infinite, which implies that at some point we need to erase some parts of the stream based data previously stored.

The techniques presented in this thesis cover all aspects of process mining, i.e. process discovery, conformance checking and process enhancement, using event streams as a basis. We present techniques and tools to efficiently store events emitted on the input event stream, as well as ways to effectively remove noise. For the purpose of process discovery, we propose a generic architecture, covering a wide variety of process discovery algorithms, that emphasizes storage of common algorithmic data structures, to minimize the corresponding memory footprint. For a specific class of algorithms, we furthermore show that its corresponding algorithmic data structure can be exploited to guarantee both behavioural and structural properties of the discovered process models. We present a greedy conformance checking algorithm that allows us to compute conformance checking statistics for running instances of the process, i.e. accounting for the fact that some instances are not completed yet. In the context of process enhancement, we assess the applicability of cooperative networks in the context of streaming event data.

In summary, this thesis covers a multitude of process dimensions in the context of streaming data, whilst at the same time identifying several interesting open challenges and directions for future work. There is a need for a large set of case studies, particularly studying event stream based process mining and verify the business impact of the techniques proposed. Furthermore, a new family of fully incremental process discovery algorithms, building on top of the discovery architecture as presented in this thesis, is a promising direction. Finally, an extension of the techniques presented in this thesis related to event storage, i.e. focussing on storage of complete (high frequent) behaviour, is of high interest.

Acknowledgements

Writing a thesis, as well as conducting the research that allows one to do so, are arguably relatively lonely endeavours. Nonetheless, it is beyond dispute, that numerous individuals have had a remarkable and positive impact on these aforementioned endeavours.

First of all, I am grateful for the guidance of my supervisors, prof.dr.ir. Wil M.P. van der Aalst and prof.dr.ir Boudewijn F. van Dongen, who have helped me in becoming the researcher I am today. Wil's feedback both on my scientific ideas and writing, are of unprecedented quality as well as level of detail. Seemingly effortless, Wil discovers the weaknesses of an idea, experiments or text and provides you with the tools and guidance to improve upon them, in the right direction. I am grateful for Boudewijn's openness and constructive attitude towards the, from time-to-time not so eminent, problems (and proposed solutions) I was facing throughout my research. Often, his ideas as well as theoretical and technical skill and knowledge, helped me to go forward. There have been times where me, "supporting Boudewijn's door post", was a daily routine, and, I am grateful that in the majority of cases, my support to his door post was welcome. Thank you both, for your guidance, when it comes to performing sound research as well as teaching. Even more so, for your patience with my passion for solving the complex problems first, rather than starting with the (seemingly) simple, sometimes logical, first step towards solving these more complex problems.

Next to the scientific-oriented problems one needs to master during the PhD journey, there are, off course, several administrative and other non-science-related hurdles that need to be tackled. In any of these cases, the secretaries were always kind to provide support and even valuable guidance. I would therefore like to express my gratitude to Ine, Jose and Riet, for always being ready to help, wherever needed. Furthermore, I would explicitly like to thank Ine for proof reading this thesis.

There are several individuals that I have met during the course of my PhD that made the journey an unforgettable, and, enjoyable one. I have greatly enjoyed the time spent during the weekly "VriMiBo" (and subsequent visits to (karaoke)bars), as well as the numerous BBQ sessions in sunny Eindhoven. I would like to thank all (former) members of the TU/e AIS research group, with which I have had the pleasure to work, and socialize, with throughout the past years. Many thanks for this go out to, Alfredo, Alok, Alifah, Arya, Bart, Cong, Dennis, Dirk, Eduardo, Elham, Eric, Farideh, Felix, Guangming, Hajo, Han, Hilda, Joos, Long, Marcus, Maikel v. E., Maikel L., Marie, Marwan, Massimiliano, Mohammadreza, Murat, Natalia, Niek, Nour, Patrick, Petar, Rafal, Renata, Remi, Richard, Ronny, Sander, Shengnan, Shiva, Vadim, Wim and Xixi.

Finally, I would like to highlight that, the contents of chapter 4, are based on [138], which was a result of a research visit at Queensland University of Technology, Brisbane, Australia, in the context of the Rise BPM project[2]. I would explicitly like to thank Mohammadreza Fani Sani, Alireza Ostovar and Raffaele Conforti for their contribution to the paper corresponding to the aforementioned chapter. Furthermore, I would like to thank Marcello La Rosa for both his contributions to the paper as well as for facilitating the amazing research visit in amazing Brisbane. Similarly, the contents of chapter 8 are based on [135], which was a partial result of a research visit at Pohang University of Science and Technology (POSTECH), Pohang, South-Korea in the context of the Rise BPM project. I gratefully thank Minseok Song and Minsu Cho (and my travel buddy Maikel), for facilitating the aforementioned research visit and the very pleasant stay in Pohang and Jeju Island.

Sebastiaan J. van Zelst
Aachen, January 2019

[2]This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 645751

Curriculum Vitae

Sebastiaan J. (Bas) van Zelst was born in the Wilrijk district of the municipality of Antwerp, Belgium, on September 30^{th}, 1989. After obtaining his high-school degree in 2007, he obtained a BSc. degree in *Computer Science* in 2011 at the *Eindhoven University of Technology*. Subsequently, at the same university, he finished his computer science MSc. degree (ir. in Dutch) with a specialization in *Business Information Systems* (a hybrid study program combining computer science and industrial engineering).

After a short period in industry, at *Sioux Embedded Systems B.V.*, he started as a PhD candidate in the *Architecture of Information Systems* research group, headed by prof.dr.ir. Wil M.P. van der Aalst at the *Eindhoven University of Technology*, in early 2014. Three years within his PhD project, he was appointed lecturer at the Eindhoven University of Technology, for 0.5 FTE. Within this role, he was involved in several courses, primarily covering the topic of process mining, both as an instructor and as a responsible lecturer.

From July 1^{st} 2018, he continued his academic career as a post-doctoral researcher/project leader at the *Fraunhofer Gesellschaft*, in cooperation with the *Process and Data Science* research group, headed by prof.dr.ir. Wil M.P. van der Aalst at the *RWTH Aachen University*. In this new position, he mainly focusses on the study, development and industrial application of techniques related to *purely data-driven process improvement*.

List of Publications

Sebastiaan J. van Zelst has published the following articles:

Journals

- S. J. van Zelst, A. Bolt, and B. F. van Dongen. Computing Alignments of Event Data and Process Models. *T. Petri Nets and Other Models of Concurrency*, 13:1–26, 2018
- S. J. van Zelst, B. F. van Dongen, W. M. P. van der Aalst, and H. M. W. Verbeek. Discovering Workflow Nets Using Integer Linear Programming. *Computing*, 100(5):529–556, 2018
- S. J. van Zelst, B. F. van Dongen, and W. M. P. van der Aalst. Event Stream-Based Process Discovery Using Abstract Representations. *Knowl. Inf. Syst.*, 54(2):407–435, 2018
- 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*, Oct 2017

Proceedings and Conference Contributions

- M. Fani Sani, S. J. van Zelst, and W. M. P. van der Aalst. Applying Sequence Mining for Outlier Detection in Process Mining. In *On the Move to Meaningful Internet Systems. OTM 2018 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22-26, 2018, Proceedings, Part II*, pages 98–116, 2018
- M. Fani Sani, S. J. van Zelst, and W. M. P. van der Aalst. Repairing Outlier Behaviour in Event Logs. In W. Abramowicz and A. Paschke, editors, *Business Information Systems - 21st International Conference, BIS 2018, Berlin, Germany, July 18-20, 2018, Proceedings*, volume 320 of *Lecture Notes in Business Information Processing*, pages 115–131. Springer, 2018
- A. Burattin, S. J. van Zelst, A. Armas-Cervantes, B. F. , van Dongen, and J. Carmona. Online Conformance Checking Using Behavioural Patterns. In *Business Process Management - 16th International Conference, BPM 2018, Sydney, NSW, Australia, September 9-14, 2018, Proceedings*, pages 250–267, 2018
- V. Bloemen, S. J. van Zelst, W. M. P. van der Aalst, B. F. van Dongen, and J. van de Pol. Maximizing Synchronization for Aligning Observed and Modelled Behaviour. In *Business Process Management - 16th International Conference, BPM 2018, Sydney, NSW, Australia, September 9-14, 2018, Proceedings*, pages 233–249, 2018

- N. Tax, S. J. van Zelst, and I. Teinemaa. An Experimental Evaluation of the Generalizing Capabilities of Process Discovery Techniques and Black-Box Sequence Models. In *Enterprise, Business-Process and Information Systems Modeling - 19th International Conference, BPMDS 2018, 23rd International Conference, EMMSAD 2018, Held at CAiSE 2018, Tallinn, Estonia, June 11-12, 2018, Proceedings*, pages 165–180, 2018
- S. J. van Zelst, M. Fani Sani, A. Ostovar, R. Conforti, and M. La Rosa. Filtering Spurious Events from Event Streams of Business Processes. In J. Krogstie and H. A. Reijers, editors, *Advanced Information Systems Engineering - 30th International Conference, CAiSE 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings*, volume 10816 of *Lecture Notes in Computer Science*, pages 35–52. Springer, 2018
- M. Fani Sani, S. J. van Zelst, and W. M. P. van der Aalst. Improving Process Discovery Results by Filtering Outliers Using Conditional Behavioural Probabilities. In *Business Process Management Workshops - BPM 2017 International Workshops, Barcelona, Spain, September 10-11, 2017, Revised Papers*, pages 216–229, 2017
- S. J. van Zelst, A. Bolt, and B. F. van Dongen. Tuning Alignment Computation: An Experimental Evaluation. In W. M. P. van der Aalst, R. Bergenthum, and J. Carmona, editors, *Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data 2017 Satellite event of the conferences: 38th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2017 and 17th International Conference on Application of Concurrency to System Design ACSD 2017, Zaragoza, Spain, June 26-27, 2017.*, volume 1847 of *CEUR Workshop Proceedings*, pages 6–20. CEUR-WS.org, 2017
- B. Vázquez-Barreiros, S. J. van Zelst, J. C. A. M. Buijs, M. Lama, and M. Mucientes. Repairing Alignments: Striking the Right Nerve. In *Enterprise, Business-Process and Information Systems Modeling - 17th International Conference, BPMDS 2016, 21st International Conference, EMMSAD 2016, Held at CAiSE 2016, Ljubljana, Slovenia, June 13-14, 2016, Proceedings*, pages 266–281, 2016
- S. J. van Zelst, B. F. van Dongen, and W. M. P. van der Aalst. Online Discovery of Co-operative Structures in Business Processes. In C. Debruyne, H. Panetto, R. Meersman, T. S. Dillon, e. Kühn, D. O'Sullivan, and C. Agostino Ardagna, editors, *On the Move to Meaningful Internet Systems: OTM 2016 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2016, Rhodes, Greece, October 24-28, 2016, Proceedings*, volume 10033 of *Lecture Notes in Computer Science*, pages 210–228, 2016
- S. J. van Zelst, B. F. van Dongen, and W. M. P. van der Aalst. ILP-Based Process Discovery Using Hybrid Regions. In W. M. P. van der Aalst, R. Bergenthum, and J. Carmona, editors, *Proceedings of the ATAED 2015 Workshop, Satellite event of Petri Nets/ACSD 2015, Brussels, Belgium, June 22-23, 2015.*, volume 1371 of *CEUR Workshop Proceedings*, pages 47–61. CEUR-WS.org, 2015
- S. Hernández, S. J. van Zelst, J. Ezpeleta, and W. M. P. van der Aalst. Handling Big(ger) Logs: Connecting ProM 6 to Apache Hadoop. In F. Daniel and S. Zugal, editors, *Proceedings of the BPM Demo Session 2015 Co-located with the 13th International Conference on Business Process Management (BPM 2015), Innsbruck, Austria, September 2, 2015.*, volume 1418 of *CEUR Workshop Proceedings*, pages 80–84. CEUR-WS.org, 2015
- S. J. van Zelst, B. F. van Dongen, and W. M. P. van der Aalst. Know What You Stream: Generating Event Streams from CPN Models in ProM 6. In *Proceedings of the BPM Demo Session 2015 Co-located with the 13th International Conference on Business Process Management (BPM 2015), Innsbruck, Austria, September 2, 2015*, pages 85–89, 2015

- S. J. van Zelst, B. F. van Dongen, and W. M. P. van der Aalst. Avoiding Over-Fitting in ILP-Based Process Discovery. In H. R. Motahari-Nezhad, J. Recker, and M. Weidlich, editors, *Business Process Management - 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 - September 3, 2015, Proceedings*, volume 9253 of *Lecture Notes in Computer Science*, pages 163–171. Springer, 2015
- S. J. van Zelst, A. Burattin, B. F. van Dongen, and H. M. W. Verbeek. Data Streams in ProM 6: A Single-node Architecture. In *Proceedings of the BPM Demo Sessions 2014 Co-located with the 12th International Conference on Business Process Management (BPM 2014), Eindhoven, The Netherlands, September 10, 2014.*, page 81, 2014

Technical Reports (Non-Refereed)

- W. M. P. van der Aalst, A. Bolt, and S. J. van Zelst. RapidProM: Mine Your Processes and Not Just Your Data. *CoRR*, abs/1703.03740, 2017

SIKS Dissertations

2011

01 Botond Cseke (RUN), *Variational Algorithms for Bayesian Inference in Latent Gaussian Models*.
02 Nick Tinnemeier(UU), *Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language*.
03 Jan Martijn van der Werf (TUE), *Compositional Design and Verification of Component-Based Information Systems*.
04 Hado van Hasselt (UU), *Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference*.
05 Base van der Raadt (VU), *Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.*.
06 Yiwen Wang (TUE), *Semantically-Enhanced Recommendations in Cultural Heritage*.
07 Yujia Cao (UT), *Multimodal Information Presentation for High Load Human Computer Interaction*.
08 Nieske Vergunst (UU), *BDI-based Generation of Robust Task-Oriented Dialogues*.
09 Tim de Jong (OU), *Contextualised Mobile Media for Learning*.
10 Bart Bogaert (UvT), *Cloud Content Contention*.
11 Dhaval Vyas (UT), *Designing for Awareness: An Experience-focused HCI Perspective*.
12 Carmen Bratosin (TUE), *Grid Architecture for Distributed Process Mining*.
13 Xiaoyu Mao (UvT), *Airport under Control. Multiagent Scheduling for Airport Ground Handling*.
14 Milan Lovric (EUR), *Behavioral Finance and Agent-Based Artificial Markets*.
15 Marijn Koolen (UvA), *The Meaning of Structure: the Value of Link Evidence for Information Retrieval*.
16 Maarten Schadd (UM), *Selective Search in Games of Different Complexity*.
17 Jiyin He (UVA), *Exploring Topic Structure: Coherence, Diversity and Relatedness*.
18 Mark Ponsen (UM), *Strategic Decision-Making in complex games*.
19 Ellen Rusman (OU), *The Mind ' s Eye on Personal Profiles*.
20 Qing Gu (VU), *Guiding service-oriented software engineering - A view-based approach*.
21 Linda Terlouw (TUD), *Modularization and Specification of Service-Oriented Systems*.
22 Junte Zhang (UVA), *System Evaluation of Archival Description and Access*.
23 Wouter Weerkamp (UVA), *Finding People and their Utterances in Social Media*.
24 Herwin van Welbergen (UT), *Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior*.
25 Syed Waqar ul Qounain Jaffry (VU)), *Analysis and Validation of Models for Trust Dynamics*.
26 Matthijs Aart Pontier (VU), *Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots*.
27 Aniel Bhulai (VU), *Dynamic website optimization through autonomous management of design patterns*.
28 Rianne Kaptein(UVA), *Effective Focused Retrieval by Exploiting Query Context and Document Structure*.
29 Faisal Kamiran (TUE), *Discrimination-aware Classification*.
30 Egon van den Broek (UT), *Affective Signal Processing (ASP): Unraveling the mystery of emotions*.
31 Ludo Waltman (EUR), *Computational and Game-Theoretic Approaches for Modeling Bounded Rationality*.
32 Nees-Jan van Eck (EUR), *Methodological Advances in Bibliometric Mapping of Science*.
33 Tom van der Weide (UU), *Arguing to Motivate Decisions*.
34 Paolo Turrini (UU), *Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations*.
35 Maaike Harbers (UU), *Explaining Agent Behavior in Virtual Training*.
36 Erik van der Spek (UU), *Experiments in serious game design: a cognitive approach*.
37 Adriana Burlutiu (RUN), *Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference*.
38 Nyree Lemmens (UM), *Bee-inspired Distributed Optimization*.
39 Joost Westra (UU), *Organizing Adaptation using Agents in Serious Games*.
40 Viktor Clerc (VU), *Architectural Knowledge Management in Global Software Development*.
41 Luan Ibraimi (UT), *Cryptographically Enforced Distributed Data Access Control*.
42 Michal Sindlar (UU), *Explaining Behavior through Mental State Attribution*.
43 Henk van der Schuur (UU), *Process Improvement through Software Operation Knowledge*.
44 Boris Reuderink (UT), *Robust Brain-Computer Interfaces*.
45 Herman Stehouwer (UvT), *Statistical Language Models for Alternative Sequence Selection*.

46 Beibei Hu (TUD), *Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work*.
47 Azizi Bin Ab Aziz(VU), *Exploring Computational Models for Intelligent Support of Persons with Depression*.
48 Mark Ter Maat (UT), *Response Selection and Turn-taking for a Sensitive Artificial Listening Agent*.
49 Andreea Niculescu (UT), *Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality*.

2012

01 Terry Kakeeto (UvT), *Relationship Marketing for SMEs in Uganda*.
02 Muhammad Umair(VU), *Adaptivity, emotion, and Rationality in Human and Ambient Agent Models*.
03 Adam Vanya (VU), *Supporting Architecture Evolution by Mining Software Repositories*.
04 Jurriaan Souer (UU), *Development of Content Management System-based Web Applications*.
05 Marijn Plomp (UU), *Maturing Interorganisational Information Systems*.
06 Wolfgang Reinhardt (OU), *Awareness Support for Knowledge Workers in Research Networks*.
07 Rianne van Lambalgen (VU), *When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions*.
08 Gerben de Vries (UVA), *Kernel Methods for Vessel Trajectories*.
09 Ricardo Neisse (UT), *Trust and Privacy Management Support for Context-Aware Service Platforms*.
10 David Smits (TUE), *Towards a Generic Distributed Adaptive Hypermedia Environment*.
11 J.C.B. Rantham Prabhakara (TUE), *Process Mining in the Large: Preprocessing, Discovery, and Diagnostics*.
12 Kees van der Sluijs (TUE), *Model Driven Design and Data Integration in Semantic Web Information Systems*.
13 Suleman Shahid (UvT), *Fun and Face: Exploring non-verbal expressions of emotion during playful interactions*.
14 Evgeny Knutov(TUE), *Generic Adaptation Framework for Unifying Adaptive Web-based Systems*.
15 Natalie van der Wal (VU), *Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes.*.
16 Fiemke Both (VU), *Helping people by understanding them - Ambient Agents supporting task execution and depression treatment*.
17 Amal Elgammal (UvT), *Towards a Comprehensive Framework for Business Process Compliance*.
18 Eltjo Poort (VU), *Improving Solution Architecting Practices*.
19 Helen Schonenberg (TUE), *What's Next? Operational Support for Business Process Execution*.
20 Ali Bahramisharif (RUN), *Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing*.
21 Roberto Cornacchia (TUD), *Querying Sparse Matrices for Information Retrieval*.
22 Thijs Vis (UvT), *Intelligence, politie en veiligheidsdienst: verenigbare grootheden?*.
23 Christian Muehl (UT), *Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction*.
24 Laurens van der Werff (UT), *Evaluation of Noisy Transcripts for Spoken Document Retrieval*.
25 Silja Eckartz (UT), *Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application*.
26 Emile de Maat (UVA), *Making Sense of Legal Text*.
27 Hayrettin Gurkok (UT), *Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games*.
28 Nancy Pascall (UvT), *Engendering Technology Empowering Women*.
29 Almer Tigelaar (UT), *Peer-to-Peer Information Retrieval*.
30 Alina Pommeranz (TUD), *Designing Human-Centered Systems for Reflective Decision Making*.
31 Emily Bagarukayo (RUN), *A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure*.
32 Wietske Visser (TUD), *Qualitative multi-criteria preference representation and reasoning*.
33 Rory Sie (OUN), *Coalitions in Cooperation Networks (COCOON)*.
34 Pavol Jancura (RUN), *Evolutionary analysis in PPI networks and applications*.
35 Evert Haasdijk (VU), *Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics*.
36 Denis Ssebugwawo (RUN), *Analysis and Evaluation of Collaborative Modeling Processes*.
37 Agnes Nakakawa (RUN), *A Collaboration Process for Enterprise Architecture Creation*.
38 Selmar Smit (VU), *Parameter Tuning and Scientific Testing in Evolutionary Algorithms*.
39 Hassan Fatemi (UT), *Risk-aware design of value and coordination networks*.
40 Agus Gunawan (UvT), *Information Access for SMEs in Indonesia*.
41 Sebastian Kelle (OU), *Game Design Patterns for Learning*.
42 Dominique Verpoorten (OU), *Reflection Amplifiers in self-regulated Learning*.
43 Withdrawn, .
44 Anna Tordai (VU), *On Combining Alignment Techniques*.
45 Benedikt Kratz (UvT), *A Model and Language for Business-aware Transactions*.
46 Simon Carter (UVA), *Exploration and Exploitation of Multilingual Data for Statistical Machine Translation*.
47 Manos Tsagkias (UVA), *Mining Social Media: Tracking Content and Predicting Behavior*.
48 Jorn Bakker (TUE), *Handling Abrupt Changes in Evolving Time-series Data*.
49 Michael Kaisers (UM), *Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions*.
50 Steven van Kervel (TUD), *Ontologogy driven Enterprise Information Systems Engineering*.
51 Jeroen de Jong (TUD), *Heuristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching*.

2013

01 Viorel Milea (EUR), *News Analytics for Financial Decision Support*.
02 Erietta Liarou (CWI), *MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing*.
03 Szymon Klarman (VU), *Reasoning with Contexts in Description Logics*.
04 Chetan Yadati(TUD), *Coordinating autonomous planning and scheduling*.
05 Dulce Pumareja (UT), *Groupware Requirements Evolutions Patterns*.
06 Romulo Goncalves(CWI), *The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience*.
07 Giel van Lankveld (UvT), *Quantifying Individual Player Differences*.
08 Robbert-Jan Merk(VU), *Making enemies: cognitive modeling for opponent agents in fighter pilot simulators*.
09 Fabio Gori (RUN), *Metagenomic Data Analysis: Computational Methods and Applications*.
10 Jeewanie Jayasinghe Arachchige(UvT), *A Unified Modeling Framework for Service Design.*.
11 Evangelos Pournaras(TUD), *Multi-level Reconfigurable Self-organization in Overlay Services*.
12 Marian Razavian(VU), *Knowledge-driven Migration to Services*.
13 Mohammad Safiri(UT), *Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly*.
14 Jafar Tanha (UVA), *Ensemble Approaches to Semi-Supervised Learning Learning*.
15 Daniel Hennes (UM), *Multiagent Learning - Dynamic Games and Applications*.
16 Eric Kok (UU), *Exploring the practical benefits of argumentation in multi-agent deliberation*.
17 Koen Kok (VU), *The PowerMatcher: Smart Coordination for the Smart Electricity Grid*.
18 Jeroen Janssens (UvT), *Outlier Selection and One-Class Classification*.
19 Renze Steenhuizen (TUD), *Coordinated Multi-Agent Planning and Scheduling*.
20 Katja Hofmann (UvA), *Fast and Reliable Online Learning to Rank for Information Retrieval*.
21 Sander Wubben (UvT), *Text-to-text generation by monolingual machine translation*.
22 Tom Claassen (RUN), *Causal Discovery and Logic*.
23 Patricio de Alencar Silva(UvT), *Value Activity Monitoring*.
24 Haitham Bou Ammar (UM), *Automated Transfer in Reinforcement Learning*.
25 Agnieszka Anna Latoszek-Berendsen (UM), *Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System*.
26 Alireza Zarghami (UT), *Architectural Support for Dynamic Homecare Service Provisioning*.
27 Mohammad Huq (UT), *Inference-based Framework Managing Data Provenance*.
28 Frans van der Sluis (UT), *When Complexity becomes Interesting: An Inquiry into the Information eXperience*.
29 Iwan de Kok (UT), *Listening Heads*.
30 Joyce Nakatumba (TUE), *Resource-Aware Business Process Management: Analysis and Support*.
31 Dinh Khoa Nguyen (UvT), *Blueprint Model and Language for Engineering Cloud Applications*.
32 Kamakshi Rajagopal (OUN), *Networking For Learning; The role of Networking in a Lifelong Learner's Professional Development*.
33 Qi Gao (TUD), *User Modeling and Personalization in the Microblogging Sphere*.
34 Kien Tjin-Kam-Jet (UT), *Distributed Deep Web Search*.
35 Abdallah El Ali (UvA), *Minimal Mobile Human Computer Interaction*.
36 Than Lam Hoang (TUe), *Pattern Mining in Data Streams*.
37 Dirk Börner (OUN), *Ambient Learning Displays*.
38 Eelco den Heijer (VU), *Autonomous Evolutionary Art*.
39 Joop de Jong (TUD), *A Method for Enterprise Ontology based Design of Enterprise Information Systems*.
40 Pim Nijssen (UM), *Monte-Carlo Tree Search for Multi-Player Games*.
41 Jochem Liem (UVA), *Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning*.
42 Léon Planken (TUD), *Algorithms for Simple Temporal Reasoning*.
43 Marc Bron (UVA), *Exploration and Contextualization through Interaction and Concepts*.

2014

01 Nicola Barile (UU), *Studies in Learning Monotone Models from Data*.
02 Fiona Tuliyano (RUN), *Combining System Dynamics with a Domain Modeling Method*.
03 Sergio Raul Duarte Torres (UT), *Information Retrieval for Children: Search Behavior and Solutions*.
04 Hanna Jochmann-Mannak (UT), *Websites for children: search strategies and interface design - Three studies on children's search performance and evaluation*.
05 Jurriaan van Reijsen (UU), *Knowledge Perspectives on Advancing Dynamic Capability*.
06 Damian Tamburri (VU), *Supporting Networked Software Development*.
07 Arya Adriansyah (TUE), *Aligning Observed and Modeled Behavior*.
08 Samur Araujo (TUD), *Data Integration over Distributed and Heterogeneous Data Endpoints*.
09 Philip Jackson (UvT), *Toward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language*.
10 Ivan Salvador Razo Zapata (VU), *Service Value Networks*.
11 Janneke van der Zwaan (TUD), *An Empathic Virtual Buddy for Social Support*.
12 Willem van Willigen (VU), *Look Ma, No Hands: Aspects of Autonomous Vehicle Control*.

13 Arlette van Wissen (VU), *Agent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains*.
14 Yangyang Shi (TUD), *Language Models With Meta-information*.
15 Natalya Mogles (VU), *Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare*.
16 Krystyna Milian (VU), *Supporting trial recruitment and design by automatically interpreting eligibility criteria*.
17 Kathrin Dentler (VU), *Computing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability*.
18 Mattijs Ghijsen (UVA), *Methods and Models for the Design and Study of Dynamic Agent Organizations*.
19 Vinicius Ramos (TUE), *Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support*.
20 Mena Habib (UT), *Named Entity Extraction and Disambiguation for Informal Text: The Missing Link*.
21 Kassidy Clark (TUD), *Negotiation and Monitoring in Open Environments*.
22 Marieke Peeters (UU), *Personalized Educational Games - Developing agent-supported scenario-based training*.
23 Eleftherios Sidirourgos (UvA/CWI), *Space Efficient Indexes for the Big Data Era*.
24 Davide Ceolin (VU), *Trusting Semi-structured Web Data*.
25 Martijn Lappenschaar (RUN), *New network models for the analysis of disease interaction*.
26 Tim Baarslag (TUD), *What to Bid and When to Stop*.
27 Rui Jorge Almeida (EUR), *Conditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty*.
28 Anna Chmielowiec (VU), *Decentralized k-Clique Matching*.
29 Jaap Kabbedijk (UU), *Variability in Multi-Tenant Enterprise Software*.
30 Peter de Cock (UvT), *Anticipating Criminal Behaviour*.
31 Leo van Moergestel (UU), *Agent Technology in Agile Multiparallel Manufacturing and Product Support*.
32 Naser Ayat (UvA), *On Entity Resolution in Probabilistic Data*.
33 Tesfa Tegegne (RUN), *Service Discovery in eHealth*.
34 Christina Manteli(VU), *The Effect of Governance in Global Software Development: Analyzing Transactive Memory Systems.*.
35 Joost van Ooijen (UU), *Cognitive Agents in Virtual Worlds: A Middleware Design Approach*.
36 Joos Buijs (TUE), *Flexible Evolutionary Algorithms for Mining Structured Process Models*.
37 Maral Dadvar (UT), *Experts and Machines United Against Cyberbullying*.
38 Danny Plass-Oude Bos (UT), *Making brain-computer interfaces better: improving usability through post-processing.*.
39 Jasmina Maric (UvT), *Web Communities, Immigration, and Social Capital*.
40 Walter Omona (RUN), *A Framework for Knowledge Management Using ICT in Higher Education*.
41 Frederic Hogenboom (EUR), *Automated Detection of Financial Events in News Text*.
42 Carsten Eijckhof (CWI/TUD), *Contextual Multidimensional Relevance Models*.
43 Kevin Vlaanderen (UU), *Supporting Process Improvement using Method Increments*.
44 Paulien Meesters (UvT), *Intelligent Blauw. Met als ondertitel: Intelligence-gestuurde politiezorg in gebiedsgebonden eenheden.*.
45 Birgit Schmitz (OUN), *Mobile Games for Learning: A Pattern-Based Approach*.
46 Ke Tao (TUD), *Social Web Data Analytics: Relevance, Redundancy, Diversity*.
47 Shangsong Liang (UVA), *Fusion and Diversification in Information Retrieval*.

2015

01 Niels Netten (UvA), *Machine Learning for Relevance of Information in Crisis Response*.
02 Faiza Bukhsh (UvT), *Smart auditing: Innovative Compliance Checking in Customs Controls*.
03 Twan van Laarhoven (RUN), *Machine learning for network data*.
04 Howard Spoelstra (OUN), *Collaborations in Open Learning Environments*.
05 Christoph Bösch(UT), *Cryptographically Enforced Search Pattern Hiding*.
06 Farideh Heidari (TUD), *Business Process Quality Computation - Computing Non-Functional Requirements to Improve Business Processes*.
07 Maria-Hendrike Peetz(UvA), *Time-Aware Online Reputation Analysis*.
08 Jie Jiang (TUD), *Organizational Compliance: An agent-based model for designing and evaluating organizational interactions*.
09 Randy Klaassen(UT), *HCI Perspectives on Behavior Change Support Systems*.
10 Henry Hermans (OUN), *OpenU: design of an integrated system to support lifelong learning*.
11 Yongming Luo(TUE), *Designing algorithms for big graph datasets: A study of computing bisimulation and joins*.
12 Julie M. Birkholz (VU), *Modi Operandi of Social Network Dynamics: The Effect of Context on Scientific Collaboration Networks*.
13 Giuseppe Procaccianti(VU), *Energy-Efficient Software*.
14 Bart van Straalen (UT), *A cognitive approach to modeling bad news conversations*.
15 Klaas Andries de Graaf (VU), *Ontology-based Software Architecture Documentation*.
16 Changyun Wei (UT), *Cognitive Coordination for Cooperative Multi-Robot Teamwork*.
17 André van Cleeff (UT), *Physical and Digital Security Mechanisms: Properties, Combinations and Trade-offs*.
18 Holger Pirk (CWI), *Waste Not, Want Not! - Managing Relational Data in Asymmetric Memories*.
19 Bernardo Tabuenca (OUN), *Ubiquitous Technology for Lifelong Learners*.
20 Loïs Vanhée(UU), *Using Culture and Values to Support Flexible Coordination*.
21 Sibren Fetter (OUN), *Using Peer-Support to Expand and Stabilize Online Learning*.

23 Luit Gazendam (VU), *Cataloguer Support in Cultural Heritage*.
24 Richard Berendsen (UVA), *Finding People, Papers, and Posts: Vertical Search Algorithms and Evaluation*.
25 Steven Woudenberg (UU), *Bayesian Tools for Early Disease Detection*.
26 Alexander Hogenboom (EUR), *Sentiment Analysis of Text Guided by Semantics and Structure*.
27 Sándor Héman (CWI), *Updating compressed colomn stores*.
28 Janet Bagorogoza(TiU), *KNOWLEDGE MANAGEMENT AND HIGH PERFORMANCE; The Uganda Financial Institutions Model for HPO*.
29 Hendrik Baier (UM), *Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains*.
30 Kiavash Bahreini(OU), *Real-time Multimodal Emotion Recognition in E-Learning*.
31 Yakup Koç (TUD), *On the robustness of Power Grids*.
32 Jerome Gard(UL), *Corporate Venture Management in SMEs*.
33 Frederik Schadd (TUD), *Ontology Mapping with Auxiliary Resources*.
34 Victor de Graaf(UT), *Gesocial Recommender Systems*.
35 Jungxao Xu (TUD), *Affective Body Language of Humanoid Robots: Perception and Effects in Human Robot Interaction*.

2016

01 Syed Saiden Abbas (RUN), *Recognition of Shapes by Humans and Machines*.
02 Michiel Christiaan Meulendijk (UU), *Optimizing medication reviews through decision support: prescribing a better pill to swallow*.
03 Maya Sappelli (RUN), *Knowledge Work in Context: User Centered Knowledge Worker Support*.
04 Laurens Rietveld (VU), *Publishing and Consuming Linked Data*.
05 Evgeny Sherkhonov (UVA), *Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers*.
06 Michel Wilson (TUD), *Robust scheduling in an uncertain environment*.
07 Jeroen de Man (VU), *Measuring and modeling negative emotions for virtual training*.
08 Matje van de Camp (TiU), *A Link to the Past: Constructing Historical Social Networks from Unstructured Data*.
09 Archana Nottamkandath (VU), *Trusting Crowdsourced Information on Cultural Artefacts*.
10 George Karafotias (VUA), *Parameter Control for Evolutionary Algorithms*.
11 Anne Schuth (UVA), *Search Engines that Learn from Their Users*.
12 Max Knobbout (UU), *Logics for Modelling and Verifying Normative Multi-Agent Systems*.
13 Nana Baah Gyan (VU), *The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach*.
14 Ravi Khadka (UU), *Revisiting Legacy Software System Modernization*.
15 Steffen Michels (RUN), *Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments*.
16 Guangliang Li (UVA), *Socially Intelligent Autonomous Agents that Learn from Human Reward*.
17 Berend Weel (VU), *Towards Embodied Evolution of Robot Organisms*.
18 Albert Meroño Peñuela (VU), *Refining Statistical Data on the Web*.
19 Julia Efremova (Tu/e), *Mining Social Structures from Genealogical Data*.
20 Daan Odijk (UVA), *Context & Semantics in News & Web Search*.
21 Alejandro Moreno Célleri (UT), *From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground*.
22 Grace Lewis (VU), *Software Architecture Strategies for Cyber-Foraging Systems*.
23 Fei Cai (UVA), *Query Auto Completion in Information Retrieval*.
24 Brend Wanders (UT), *Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach*.
25 Julia Kiseleva (TU/e), *Using Contextual Information to Understand Searching and Browsing Behavior*.
26 Dilhan Thilakarathne (VU), *In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains*.
27 Wen Li (TUD), *Understanding Geo-spatial Information on Social Media*.
28 Mingxin Zhang (TUD), *Large-scale Agent-based Social Simulation - A study on epidemic prediction and control*.
29 Nicolas Höning (TUD), *Peak reduction in decentralised electricity systems -Markets and prices for flexible planning*.
30 Ruud Mattheij (UvT), *The Eyes Have It*.
31 Mohammad Khelghati (UT), *Deep web content monitoring*.
32 Eelco Vriezekolk (UT), *Assessing Telecommunication Service Availability Risks for Crisis Organisations*.
33 Peter Bloem (UVA), *Single Sample Statistics, exercises in learning from just one example*.
34 Dennis Schunselaar (TUE), *Configurable Process Trees: Elicitation, Analysis, and Enactment*.
35 Zhaochun Ren (UVA), *Monitoring Social Media: Summarization, Classification and Recommendation*.
36 Daphne Karreman (UT), *Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies*.
37 Giovanni Sileno (UvA), *Aligning Law and Action - a conceptual and computational inquiry*.
38 Andrea Minuto (UT), *MATERIALS THAT MATTER - Smart Materials meet Art & Interaction Design*.
39 Merijn Bruijnes (UT), *Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect*.
40 Christian Detweiler (TUD), *Accounting for Values in Design*.
41 Thomas King (TUD), *Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance*.
42 Spyros Martzoukos (UVA), *Combinatorial and Compositional Aspects of Bilingual Aligned Corpora*.
43 Saskia Koldijk (RUN), *Context-Aware Support for Stress Self-Management: From Theory to Practice*.
44 Thibault Sellam (UVA), *Automatic Assistants for Database Exploration*.

45 Bram van de Laar (UT), *Experiencing Brain-Computer Interface Control*.
46 Jorge Gallego Perez (UT), *Robots to Make you Happy*.
47 Christina Weber (UL), *Real-time foresight - Preparedness for dynamic innovation networks*.
48 Tanja Buttler (TUD), *Collecting Lessons Learned*.
49 Gleb Polevoy (TUD), *Participation and Interaction in Projects. A Game-Theoretic Analysis*.
50 Yan Wang (UVT), *The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains*.

2017

01 Jan-Jaap Oerlemans (UL), *Investigating Cybercrime*.
02 Sjoerd Timmer (UU), *Designing and Understanding Forensic Bayesian Networks using Argumentation*.
03 Daniël Harold Telgen (UU), *Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines*.
04 Mrunal Gawade (CWI), *MULTI-CORE PARALLELISM IN A COLUMN-STORE*.
05 Mahdieh Shadi (UVA), *Collaboration Behavior*.
06 Damir Vandic (EUR), *Intelligent Information Systems for Web Product Search*.
07 Roel Bertens (UU), *Insight in Information: from Abstract to Anomaly*.
08 Rob Konijn (VU), *Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery*.
09 Dong Nguyen (UT), *Text as Social and Cultural Data: A Computational Perspective on Variation in Text*.
10 Robby van Delden (UT), *(Steering) Interactive Play Behavior*.
11 Florian Kunneman (RUN), *Modelling patterns of time and emotion in Twitter #anticipointment*.
12 Sander Leemans (TUE), *Robust Process Mining with Guarantees*.
13 Gijs Huisman (UT), *Social Touch Technology - Extending the reach of social touch through haptic technology*.
14 Shoshannah Tekofsky (UvT), *You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior*.
15 Peter Berck, Radboud University (RUN), *Memory-Based Text Correction*.
16 Aleksandr Chuklin (UVA), *Understanding and Modeling Users of Modern Search Engines*.
17 Daniel Dimov (UL), *Crowdsourced Online Dispute Resolution*.
18 Ridho Reinanda (UVA), *Entity Associations for Search*.
19 Jeroen Vuurens (TUD), *Proximity of Terms, Texts and Semantic Vectors in Information Retrieval*.
20 Mohammadbashir Sedighi (TUD), *Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility*.
21 Jeroen Linssen (UT), *Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)*.
22 Sara Magliacane (VU), *Logics for causal inference under uncertainty*.
23 David Graus (UVA), *Entities of Interest— Discovery in Digital Traces*.
24 Chang Wang (TUD), *Use of Affordances for Efficient Robot Learning*.
25 Veruska Zamborlini (VU), *Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search*.
26 Merel Jung (UT), *Socially intelligent robots that understand and respond to human touch*.
27 Michiel Joosse (UT), *Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors*.
28 John Klein (VU), *Architecture Practices for Complex Contexts*.
29 Adel Alhuraibi (UVT), *From IT-Business Strategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT*.
30 Wilma Latuny (UVT), *The Power of Facial Expressions*.
31 Ben Ruijl (UL), *Advances in computational methods for QFT calculations*.
32 Thaer Samar (RUN), *Access to and Retrievability of Content in Web Archives*.
33 Brigit van Loggem (OU), *Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity*.
34 Maren Scheffel (OUN), *The Evaluation Framework for Learning Analytics*.
35 Martine de Vos (VU), *Interpreting natural science spreadsheets*.
36 Yuanhao Guo (UL), *Shape Analysis for Phenotype Characterisation from High-throughput Imaging*.
37 Alejandro Montes García (TUE), *WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy*.
38 Alex Kayal (TUD), *Normative Social Applications*.
39 Sara Ahmadi (RUN), *Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR*.
40 Altaf Hussain Abro (VUA), *Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems*.
41 Adnan Manzoor (VUA), *Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle*.
42 Elena Sokolova (RUN), *Causal discovery from mixed and missing data with applications on ADHD datasets*.
43 Maaike de Boer (RUN), *Semantic Mapping in Video Retrieval*.
44 Garm Lucassen (UU), *Understanding User Stories - Computational Linguistics in Agile Requirements Engineering*.
45 Bas Testerink (UU), *Decentralized Runtime Norm Enforcement*.
46 Jan Schneider (OU), *Sensor-based Learning Support*.
47 Yie Yang (TUD), *Crowd Knowledge Creation Acceleration*.

48 Angel Suarez (OU), *Colloborative inquiry-based learning*.

2018

01 Han van der Aa (VUA), *Comparing and Aligning Process Representations*.
02 Felix Mannhardt (TUE), *Multi-perspective Process Mining*.
03 Steven Bosems (UT), *Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction*.
04 Jordan Janeiro (TUD), *Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks*.
05 Hugo Huurdeman (UVA), *Supporting the Complex Dynamics of the Information Seeking Process*.
06 Dan Ionita (UT), *Model-Driven Information Security Risk Assessment of Socio-Technical Systems*.
07 Jieting Luo (UU), *A formal account of opportunism in multi-agent systems*.
08 Rick Smetsers (RUN), *Advances in Model Learning for Software Systems*.
09 Xu Xie (TUD), *Data Assimilation in Discrete Event Simulations*.
10 Julienka Mollee (VUA), *Moving forward: supporting physical activity behavior change through intelligent technology*.
11 Mahdi Sargolzaei (UVA), *Enabling Framework for Service-oriented Collaborative Networks*.
12 Xixi Lu (TUE), *Using behavioral context in process mining*.
13 Seyed Amin Tabatabaei (VUA), *Computing a Sustainable Future: Exploring the added value of computational models for increasing the use of renewable energy in the residential sector*.
14 Bart Joosten (UVT), *Detecting Social Signals with Spatiotemporal Gabor Filters*.
15 Naser Davarzani (UM), *Biomarker discovery in heart failure*.
16 Jaebok Kim (UT), *Automatic recognition of engagement and emotion in a group of children*.
17 Jianpeng Zhang (TUE), *On Graph Sample Clustering*.
18 Henriette Nakad (UL), *De Notaris en Private Rechtspraak*.
19 Minh Duc Pham (VUA), *Emergent relational schemas for RDF*.
20 Manxia Liu (RUN), *Time and Bayesian Networks*.
21 Aad Slootmaker (OU), *EMERGO: a generic platform for authoring and playing scenario-based serious games*.
22 Eric Fernandes de Mello Araújo (VUA), *Contagious: Modeling the Spread of Behaviours, Perceptions and Emotions in Social Networks*.
23 Kim Schouten (EUR), *Semantics-driven Aspect-Based Sentiment Analysis*.
24 Jered Vroon (UT), *Responsive Social Positioning Behaviour for Semi-Autonomous Telepresence Robots*.
25 Riste Gligorov (VUA), *Serious Games in Audio-Visual Collections*.
26 Roelof de Vries (UT), *Theory-Based And Tailor-Made: Motivational Messages for Behavior Change Technology*.
27 Maikel Leemans (TUE), *Hierarchical Process Mining for Scalable Software Analysis*.
28 Christian Willemse (UT), *Social Touch Technologies: How they feel and how they make you feel*.
29 Yu Gu (UVT), *Emotion Recognition from Mandarin Speech*.

2019

1 Rob van Eijk (UL), *Web Privacy Measurement in Real-Time Bidding Systems, A Graph-Based Approach to RTB system classification*.
2 Emmanuelle Beauxis- Aussalet (CWI, UU), *Statistics and Visualizations for Assessing Class Size Uncertainty*.
3 Eduardo Gonzalez Lopez de Murillas (TUE), *Process Mining on Databases: Extracting Event Data from Real Life Data Sources*.
4 Ridho Rahmadi (RUN), *Finding stable causal structures from clinical data*.
5 Sebastiaan J. van Zelst (TUE), *Process Mining with Streaming Data*.