Knowl Inf Syst (2018) 54:407–435
https://doi.org/10.1007/s10115-017-1060-2

REGULAR PAPER

Event stream-based process discovery using abstract representations

Sebastiaan J. van Zelst[1] · Boudewijn F. van Dongen[1] · Wil M. P. van der Aalst[1]

Received: 20 July 2016 / Accepted: 27 April 2017 / Published online: 15 May 2017

Abstract The aim of process discovery, originating from the area of process mining, is to discover a process model based on business process execution data. A majority of process discovery techniques relies on an event log as an input. An event log is a static source of historical data capturing the execution of a business process. In this paper, we focus on process discovery relying on online streams of business process execution events. Learning process models from event streams poses both challenges and opportunities, i.e. we need to handle unlimited amounts of data using finite memory and, preferably, constant time. We propose a generic architecture that allows for adopting several classes of existing process discovery techniques in context of event streams. Moreover, we provide several instantiations of the architecture, accompanied by implementations in the process mining toolkit ProM (http://promtools.org). Using these instantiations, we evaluate several dimensions of stream-based process discovery. The evaluation shows that the proposed architecture allows us to lift process discovery to the streaming domain.

Keywords Process mining · Process discovery · Event streams · Abstract representations

1 Introduction

Process mining [1] aims at understanding and improving business processes. The field consists of three main branches, i.e. *process discovery*, *conformance checking* and *process enhancement*. Process discovery aims at discovering a process model based on event data.

✉ Sebastiaan J. van Zelst
s.j.v.zelst@tue.nl

Boudewijn F. van Dongen
b.f.v.dongen@tue.nl

Wil M. P. van der Aalst
w.m.p.v.d.aalst@tue.nl

1 Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Conformance checking is concerned with assessing whether a process model and event data conform to each other in terms of possible behaviour. Process enhancement is concerned with improvement of process models based on knowledge gained from event data, e.g. a process model is extended with performance diagnostics based on event data.

Several process discovery algorithms exist [2–7]. These algorithms all use an *event log* as an input. An event log is a static data source describing sequences of executed business process activities recorded over a historical time span. As the number of events recorded for operational processes is growing tremendously every year, so does the average event log size. Conventional process discovery techniques are not able to cope with such large data sets, i.e. they fail when the data do not fit main memory. Moreover, events are being generated at high rates, e.g. consider data originating from sensor networks, mobile devices and e-business applications. Since existing process discovery techniques use static data, they are not able to capture the dynamics of such event streams in an adequate manner.

In this paper, we focus on process discovery using streams of business process events, i.e. *event streams*, rather than event logs. Applying process discovery on event streams allows us to gain insights into the underlying business process in a live fashion. It furthermore allows us to deal with situations where (1) event logs are too large to fit main memory, (2) there is no time to access event data continuously, i.e. real-time constraints and (3) recent behaviour is more important, i.e. concept drift. A large class of existing process discovery algorithms transforms the event log into an *abstract representation*, i.e. an abstraction of the event log, which is subsequently used to discover a process model. To adopt these algorithms in a streaming context, it suffices to approximate the abstract representation based on the event stream. Using abstract representations has several advantages: (1) *reusability*: we *reuse* existing techniques by predominantly focusing on learning abstract representations from event streams. (2) *Extensibility*: once we design and implement a method for approximating a certain abstract representation, any (future) algorithm using the same abstract 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 abstract representations ignore large parts of the data, effectively storing a summary of the actual event data, and therefore comply to *anonymity* regulations.

We present the *stream-based abstract representation (S-BAR)* architecture that describes this mechanism in a generic way (Fig. 1). An event stream $\mathcal{S}$ represents an (in)finite sequence of *events*, emitted over time. An *event* is represented by a (c, a)-pair, stating that activity a is executed in context of case c. We maintain a data structure ($\mathcal{D}^{\mathcal{T}}$) that represents the past behaviour emitted onto stream $\mathcal{S}$. Each time a new event arrives the data structure is kept up to date by updating its previous state based on the newly received event ($\delta_{\mathcal{D}^{\mathcal{T}}}$). From the data structure, an *algorithm-specific abstract representation* ($\mathcal{A}^{\mathbf{T}}$) is deduced ($\lambda^{\mathcal{A}^{\mathbf{T}}}_{\mathcal{D}^{\mathcal{T}}}$). After

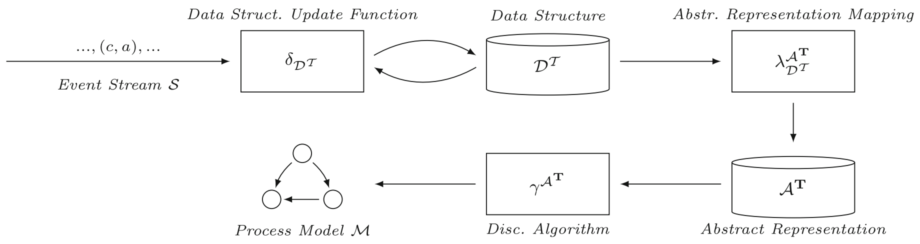

Data Struct. Update Function
Data Structure
Abstr. Representation Mapping
..., (c, a), ...
Event Stream $\mathcal{S}$
$\delta_{\mathcal{D}^{\mathcal{T}}}$
$\mathcal{D}^{\mathcal{T}}$
$\lambda^{\mathcal{A}^{\mathbf{T}}}_{\mathcal{D}^{\mathcal{T}}}$
$\gamma_{\mathcal{A}^{\mathbf{T}}}$
$\mathcal{A}^{\mathbf{T}}$
Process Model $\mathcal{M}$
Disc. Algorithm
Abstract Representation

Fig. 1 Schematic overview of the S-BAR architecture

learning the abstract representation, we reuse existing translations borrowed from conventional process discovery algorithms to return a process model ($\gamma^{\mathcal{A}^{\mathbf{T}}}$).

The S-BAR architecture is instantiated by designing a data structure, a data structure update mechanism and a data structure translation function. The actual implementation of the data structure and related update functions influences the behaviour described by the discovered process model, e.g. using a time-decaying data structure versus a data structure that approximates the most frequent cases on the stream. Several instantiations of the architecture have been implemented in the process mining toolkits ProM [8] and RapidProM [9,10]. Using these implementations, we conduct empirical experiments w.r.t. the behaviour of these algorithms in an event stream setting. The experiments show that the algorithms are able to capture the behaviour reflected by the event stream. Moreover, the experiments show that memory usage and processing times of the algorithms have non-increasing trends.

The remainder of this paper is organized as follows. In Sect. 2, we present background information regarding business processes and process discovery. In Sect. 3, we present event streams and the notion of event stream-based process discovery. In Sect. 4, we introduce the S-BAR architecture. In Sect. 5, we provide several instantiations of the architecture. In Sect. 6, we present an empirical evaluation of several instantiations of the architecture. In Sect. 7, we present related work. In Sect. 8, we discuss general challenges in event stream-based process discovery. Section 9 concludes the paper.

2 Background

In this section, we present general notation used throughout the paper and background concepts regarding business processes and process discovery.

$\mathbb{N}$ denotes the set of positive integers, and $\mathbb{N}_0$ includes 0. A *multiset* B over set X is a function $B : X \to \mathbb{N}_0$. We write a multiset as $[e_1^{k_1}, e_2^{k_2}, \ldots, e_n^{k_n}]$, where for $1 \leq i \leq n$ we have $e_i \in X$, $k_i \in \mathbb{N}$ and $e_i^{k_i} \equiv B(e_i) = k_i$. If for element e, $B(e) = 1$, we omit its superscript. If for element e, $B(e) = 0$, we omit e from the multiset notation. An empty multiset is denoted as [ ]. Element inclusion applies to multisets, i.e. if $e \in X$ and $B(e) > 0$ then $e \in B$.

A *sequence* σ of length n relates positions to elements $e \in X$, i.e. $\sigma : \{1, 2, \ldots, n\} \to X$. An empty sequence is denoted as ϵ. We write every non-empty sequence as $\langle e_1, e_2, \ldots, e_n \rangle$. The set of all possible sequences over a set X is denoted as X^*. We write *concatenation* of sequences σ_1 and σ_2 as $\sigma_1 \cdot \sigma_2$.

Let X, Y, Z and Z' be sets and let $f : X \to Y$ and $g : Y \to Z$. Function composition of f and g is defined as $g \circ f : X \to Z$, with $x \mapsto g(f(x))$ for $x \in X$. Moreover, given $h : Z \to Z'$, we write $h \circ g \circ f$ for $h \circ (g \circ f)$, i.e. $h \circ g \circ f : X \to Z'$, with $x \mapsto h(g(f(x)))$ for $x \in X$.

2.1 Business processes, models and event logs

Business processes represent the execution of related business activities leading to a business goal. Consider a bank offering loans to its customers. A business goal of the bank is to accept, reject or cancel a loan application. The bank's employees and its enterprise information system execute activities to achieve this goal, e.g. by checking a client's credit history and assessing the loan risk.

A business process $\mathcal{P}$ defines a set of sequences over a set of activities A, i.e. $\mathcal{P} \subseteq A^*$. If $\sigma \in \mathcal{P}$, then the sequence of business activities σ leads to a business goal and belongs to the *behaviour* of $\mathcal{P}$. In this paper, we assume the execution of activities to be atomic and abstract from data attributes such as resource and timestamp. Hence, we only consider the sequential ordering of activities (the *control-flow perspective*). $\mathcal{U}_{\mathcal{P}}$ denotes the universe of business processes. A process model $\mathcal{M}$ *represents* a business process and, like a process, defines a set of sequences over a set of activities A, i.e. $\mathcal{M} \subseteq A^*$. $\mathcal{U}_{\mathcal{M}}$ denotes the universe of process models. In this paper, we consider process models that describe behaviour in a deterministic manner, e.g. Petri nets [12], BPMN [13] and workflow nets [14]. Consider the BPMN model of a loan application handling process in Fig. 2. It describes that after an application is received, the first activity to be executed is *"Check application completeness"*. Depending upon the completeness of the application, the corresponding form is *"Returned back to the applicant"*, or the client's *"credit history is checked"* and subsequently a *"loan risk assessment"* is performed. The two aforementioned activities can be executed concurrently with the *"appraise property"* activity. An *"eligibility assessment"* of the loan is performed, eventually leading to a *rejection*, *cancellation* or *approval* of the loan.

Today's information systems track the execution of business processes within a company. Such systems store the execution of activities in context of a *case*, i.e. an instance of the process. The data stored by the information system are often in the form of an *event log*. Consider Table 1 as an example. The execution of an activity in context of a case, e.g. *Approve application* executed for case *3*, is referred to as an *event*. A sequence of events, e.g. the sequence of events related to case *4*, ⟨*Check application form completeness, Check credit history, ..., Approve application*⟩, is referred to as a *trace* (written $\langle c_1, c_2, \ldots, a_4 \rangle$ when using abbreviated activity names).

An event log L is a multiset of sequences over a set of activities A, i.e. $L : A^* \rightarrow \mathbb{N}_0$, and describes the execution of some $\mathcal{P} \in \mathcal{U}_{\mathcal{P}}$. $\mathcal{U}_L$ denotes the universe of event logs. An event log is a *sample* of the underlying process. Therefore, there might exist process behaviour that is not present in the event log, e.g. caused by parallelism. In such case, an event log is *incomplete*. There might also exist traces in the event log that are not part of the process, i.e. *noisy* traces. Noisy traces can be caused by faulty execution of the process, incomplete specifications or technical issues such as incorrect logging, system errors and mixed time granularity.

2.2 Process discovery

The goal of process discovery is to discover a process model based on an event log. Several process discovery algorithms exist [2–7]. These algorithms differ in terms of their underlying computational schemes and data structures as well as their resulting process modelling formalism. We refer to [1,15,16] for a detailed overview of process discovery algorithms.

A process discovery algorithm γ_L discovers a process model based on an event log, i.e. $\gamma_L : \mathcal{U}_L \rightarrow \mathcal{U}_{\mathcal{M}}$. The challenge is to design γ_L in such way that $\gamma_L(L)$ is an *appropriate representation* of the underlying process $\mathcal{P}$. Appropriateness of $\gamma(L)$ depends on the aim of the process discovery analysis, e.g. ensuring that all behaviour in the event log is present in the model versus ensuring that the most frequent behaviour is present. Given the different aims of process discovery analyses, several quality measures are defined in order to judge their resulting model's appropriateness. Ideally, $\mathcal{P}$ is used as a basis to compute these metrics; however, as L is the only tangible sample of $\mathcal{P}$, we typically compute the quality of $\gamma_L(L)$ using L. The four essential process mining quality dimensions are *replay fitness*, *precision*, *simplicity* and *generalization* [1,17]. Replay fitness describes what fraction of the behaviour

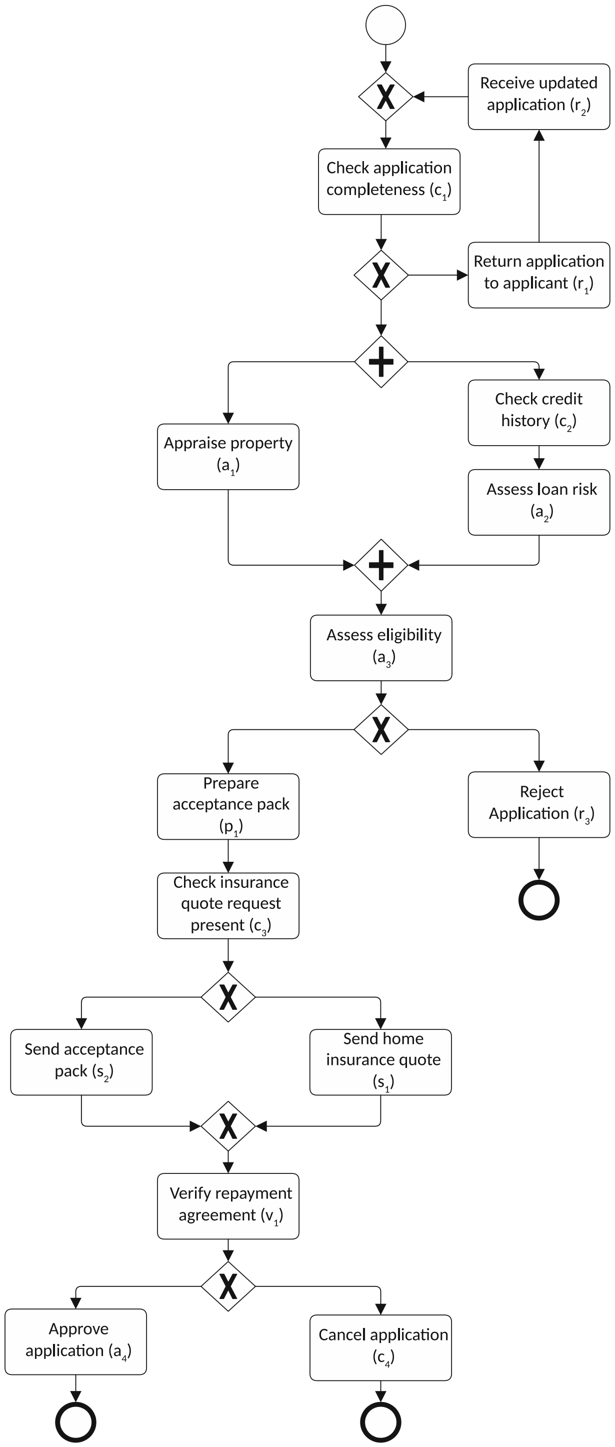

Receive updated application (r2)
Check application completeness (c1)
Return application to applicant (r1)
Check credit history (c2)
Appraise property (a1)
Assess loan risk (a2)
Assess eligibility (a3)
Prepare acceptance pack (p1)
Reject Application (r3)
Check insurance quote request present (c3)
Send acceptance pack (s2)
Send home insurance quote (s1)
Verify repayment agreement (v1)
Approve application (a4)
Cancel application (c4)

Fig. 2 BPMN model of a loan application process (adopted from [11])

Table 1 Fragment of an event log

Case	Activity	Resource	Timestamp
…	…	…	…
3	*Approve application (a4)*	*John*	*2015-05-08 08:45*
4	*Check application completeness (c1)*	*Lucy*	*2015-05-08 09:13*
5	*Check application completeness (c1)*	*John*	*2015-05-08 09:14*
5	*Return application to applicant (r1)*	*Pete*	*2015-05-08 10:11*
5	*Receive updated application (r2)*	*Pete*	*2015-05-08 10:28*
6	*Check application completeness (c1)*	*Lucy*	*2015-05-08 10:33*
4	*Check credit history (c2)*	*Rob*	*2015-05-08 10:43*
5	*Appraise property (a1)*	*Pete*	*2015-05-08 11:00*
4	*Appraise property (a1)*	*Rob*	*2015-05-08 11:14*
4	*Assess loan risk (a2)*	*Rob*	*2015-05-08 11:35*
4	*Assess eligibility (a3)*	*Lucy*	*2015-05-08 11:55*
5	*Check credit history (c2)*	*John*	*2015-05-08 11:57*
4	*Prepare acceptance pack (p1)*	*Lucy*	*2015-05-08 12:25*
4	*Check insurance quote request present (c3)*	*Lucy*	*2015-05-08 12:23*
4	*Send acceptance pack (s2)*	*Lucy*	*2015-05-08 12:28*
5	*Assess loan risk (a2)*	*John*	*2015-05-08 12:37*
4	*Verify repayment agreement (v1)*	*Lucy*	*2015-05-09 13:05*
4	*Approve application (a4)*	*John*	*2015-05-09 14:15*
…	…	…	…

present in L is also described by $\gamma_L(L)$. Precision describes what fraction of the behaviour described by $\gamma_L(L)$ is also present in L. Simplicity describes the (perceived) complexity of the process model. Since it is unlikely that the event log contains all behaviour (incompleteness), generalization describes how well the process model generalizes for behaviour not present in L. Due to noise, an algorithm guaranteeing perfect replay fitness, i.e. all behaviour in the event log is present in the discovered model, captures behaviour that is not part of the process. In practice, this leads to very complex models that are impossible to be interpreted by a human analyst. Hence, a process discovery algorithm needs to strike *an adequate balance* between the four essential quality dimensions.

3 Event stream-based process discovery

Existing process discovery techniques discover process models in an a posteriori fashion, i.e. they provide a historical view of the data. However, most information systems allow us to capture the execution of activities at the moment they occur. Discovering and analysing process models from such continuous streams of events allows us to get a real-time view of the process under study. Such view paves the way for new types of process mining analysis, i.e. we are able to answer more advanced questions such as “What is the current status of the process?” and “What running cases are likely to cause problems?”. It also allows us to inspect and visualize recent behaviour and evolution of behaviour in the process, i.e. concept drift.

There are several other advantages of studying streams of events rather than event logs. Trends such as *Big Data* and *Data Science* signify the spectacular growth and omnipresence of data. Typically, real event logs do not fit main memory. Since we assume event streams to be potentially infinite, analysing them enables us to handle event data of arbitrary size. In other cases, we do not have the time or are not allowed to access event data continuously and, hence, need to analyse events at the moment they occur.

In this section, we formalize *event streams* and *event stream-based process discovery*. Additionally, we quantify high-level requirements for the design of event stream- based process discovery algorithms.

3.1 Event streams

An event stream is a continuous stream of events executed in context of an underlying business process. We represent an event stream as a sequence of pairs consisting of a *case identifier* and an *activity*. Hence, for each event we know what activity was performed in context of what process instance. When comparing event streams to event logs, we identify two main differences: (1) an event stream is potentially *infinite*, and (2) behaviour seen for a case is *incomplete*, i.e. in the future new events may be executed in context of a case.

Definition 1 (*Event stream*) Let A be a set of activities and let C denote the set of all possible case identifiers. An event stream $\mathcal{S}$ is a sequence over $C \times A$, i.e. $\mathcal{S} \in (C \times A)^*$.

A pair $(c, a) \in C \times A$ represents an event, i.e. activity a was executed in context of case c. $\mathcal{S}(1)$ denotes the first event that we receive, whereas $\mathcal{S}(i)$ denotes the ith event. Consider stream $\mathcal{S}_1$ in Fig. 3 as an example, where event $(3, a_4)$ is emitted first ($\mathcal{S}_1(1) = (3, a_4)$), event $(4, c_1)$ is emitted second and event $(5, a_1)$ is the eight and last event emitted onto the stream *up until now*. We receive multiple events related to the same case at different points in time, e.g. the second and seventh event on $\mathcal{S}_1$ are related to case 4. Hence, handling such type of data needs new types of data structures and event processing techniques compared to conventional process discovery.

3.2 Process discovery

The goal of event stream- based process discovery is to discover a process model using an event stream as an input. A first step is to approximate, based on $\mathcal{S}$, the presence of some $\sigma \in \mathcal{P}$ and possibly σ's frequency w.r.t. $\mathcal{S}$. Given such approximation, the next step is to deploy a process discovery algorithm onto the approximation in order to obtain a process model.

A naive approach is to construct an event log based on the event stream by using a data structure that stores case-sequence pairs $(c, \sigma) \in C \times A^*$. For every event (c, a) we receive, we check whether the data structure contains entry (c, σ'). If so, we update this entry to $(c, \sigma' \cdot \langle a \rangle)$. If not, we insert new entry $(c, \langle a \rangle)$. Whenever we want to discover a new process model based on the current state of the event stream, we transform the data structure into an event log and provide it to any conventional process discovery algorithm. Observe that, since the stream is potentially infinite, this procedure needs *infinite memory*. Moreover, the approach includes *redundancy*, i.e. several (partial) traces that were already analysed in a

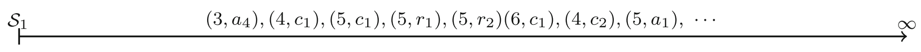

$\mathcal{S}_1$
$(3, a_4), (4, c_1), (5, c_1), (5, r_1), (5, r_2)(6, c_1), (4, c_2), (5, a_1), \cdots$
∞

Fig. 3 Example event stream $\mathcal{S}_1$

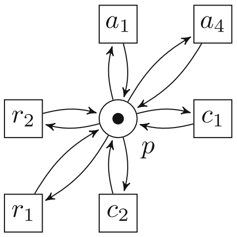


Fig. 4 Example "flower" model

previous call to a discovery algorithm, and are still in memory at the next call, are analysed twice. Hence, we want the data structure to either represent, or be easily translatable to, some minimal form of data needed in order to discover a process model.

An example of an algorithm using a minimal data representation is the *flower miner*. The flower miner produces a process model that allows for every possible sequence over the observed activities. Reconsider example stream $\mathcal{S}_1$ (Fig. 3) which consists of activities labelled a_1, a_4, c_1, c_2, r_1 and r_2. In Fig. 4, we depict a flower model, in terms of a Petri net [12], that allows for all activities on $\mathcal{S}_1$.

To ensure that the flower miner uses finite memory, we just need to deploy any finite memory-based data structure that keeps track of the activities seen on the stream. A wide variety of such data structures exits, e.g. count-based frequent item data structures [18], reservoirs [19,20] and time-decay-based models [21]. Whenever we receive a new event (c, a), we just add a to the data structure. Translating the data structure to a process model is trivial, i.e. every activity present in the data structure is adopted in the flower model.

The flower miner works, yet it has deficiencies from a process discovery perspective. It generalizes the behaviour represented by the event stream as much as possible. The resulting process model very likely allows for *much more behaviour* than actually present in the underlying process. Hence, we need techniques that are *more precise*.

The event log-based approach and the flower miner represent two extremes. Storing the event stream as an event log requires us to reuse a large part of the data several times. The flower miner on the other hand neglects a large quantity of information carried by the event stream and greatly overgeneralizes the stream's behaviour. We therefore need a scheme that is in the middle of both extremes, i.e. it does not store the complete event log, yet it stores enough data to provide meaningful output.

4 The S-BAR architecture

When analysing conventional process discovery algorithms, we observe that a majority shares a common underlying algorithmic mechanism. The event log is transformed into an *abstract representation*, which is subsequently used to construct a resulting process model. Moreover, several algorithms use the same abstract representation. In Example 1, we illustrate the *directly follows abstraction*, used by the *α-Miner* [3].

Example 1 (The directly follows abstraction and the α-Miner) Consider event log $L = [\langle a, b, c, d\rangle, \langle a, c, b, d\rangle]$. The α-Miner computes a *directly follows abstraction* based on the event log. Activity a is directly followed by b, written as $a > b$, if there exists some sequence $\sigma \in L$ of the form $\sigma = \sigma' \cdot \langle a, b\rangle \cdot \sigma''$. In case of event log L, we deduce $a > b$, $a > c$,

$b > c, b > d, c > b, c > d$. Using these relations as a basis, the α-Miner constructs a Petri net.

As Example 1 shows, the event log is translated into a *directly follows abstraction*, which is subsequently used to construct a process model. Other discovery algorithms like the Inductive Miner [5] and the ILP Miner [7] use *the same mechanism* to discover a process model. To adopt these algorithms to an event stream context, it suffices to determine whether we are able to learn the corresponding abstract representation from the event stream and, if possible, design a data structure that supports this.

In the remainder of this section, we formalize the notion of abstract representations. Subsequently, we introduce the *stream-based abstract representation* (S-BAR) architecture that captures the notion of event stream-based abstract representation computation in a generic manner.

4.1 Abstract representations in conventional process discovery

We refine conventional process discovery by splitting γ_L into two steps. In the first step, the event log is translated into the abstraction used by the discovery algorithm. In the second step, the abstraction is translated into a process model. In the remainder, we let $\mathbf{T}$ denote an abstract representation type. $\mathcal{A}^{\mathbf{T}}$ denotes an abstract representation of type $\mathbf{T}$, and $\mathcal{U}_{\mathcal{A}^{\mathbf{T}}}$ denotes the universe of abstract representations of type $\mathbf{T}$.

Definition 2 (*Abstraction function—event log*) Let $\mathbf{T}$ denote an abstract representation type. An abstraction function $\lambda_L^{\mathcal{A}^{\mathbf{T}}}$ is a function mapping an event log to an abstract representation of type $\mathbf{T}$.

$$\lambda_L^{\mathcal{A}^{\mathbf{T}}} : \mathcal{U}_L \rightarrow \mathcal{U}_{\mathcal{A}^{\mathbf{T}}} \tag{1}$$

Using Definition 2, we define process discovery in terms of abstract representations.

Definition 3 (*Process discovery algorithm—abstract representation*) Let $\mathbf{T}$ denote an abstract representation type. An abstract representation-based process discovery algorithm $\gamma^{\mathcal{A}^{\mathbf{T}}}$ maps an abstract representation of type $\mathbf{T}$ to a process model.

$$\gamma^{\mathcal{A}^{\mathbf{T}}} : \mathcal{U}_{\mathcal{A}^{\mathbf{T}}} \rightarrow \mathcal{U}_{\mathcal{M}} \tag{2}$$

Every discovery algorithm that uses an abstract representation internally can be expressed as a composition of $\lambda_L^{\mathcal{A}^{\mathbf{T}}}$ and $\gamma^{\mathcal{A}^{\mathbf{T}}}$. Thus, given event log $L \in \mathcal{U}_L$ and abstract representation type $\mathbf{T}$, we obtain $\gamma_L = (\gamma^{\mathcal{A}^{\mathbf{T}}} \circ \lambda_L^{\mathcal{A}^{\mathbf{T}}})(L)$. For example, consider Fig. 5 depicting the α-Miner in terms of $\gamma^{\mathcal{A}^{\mathbf{T}}}$ and $\lambda_L^{\mathcal{A}^{\mathbf{T}}}$.

4.2 Abstract representations in event stream-based process discovery

In this section, we present the S-BAR architecture which captures the use of abstract representations in an event stream context in a generic manner. In Fig. 6, the S-BAR architecture is

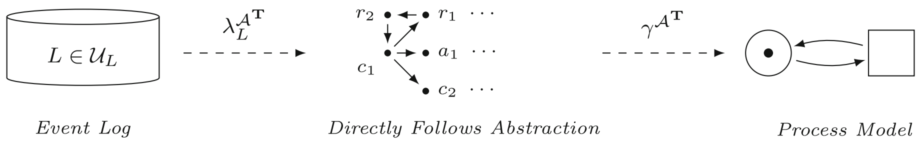
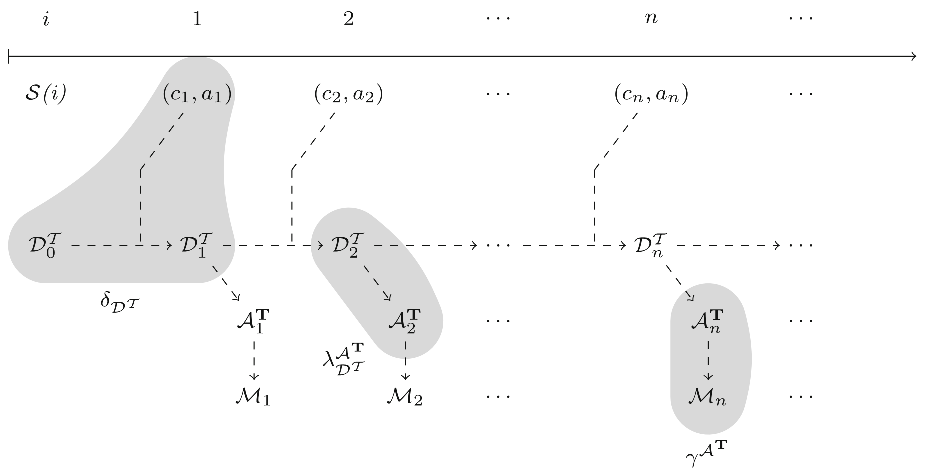

$L \in \mathcal{U}_L$
$\lambda_L^{\mathcal{A}^{\mathbf{T}}}$
r_2
r_1 ···
a_1 ···
c_1
c_2 ···
$\gamma^{\mathcal{A}^{\mathbf{T}}}$
Event Log
Directly Follows Abstraction
Process Model

Fig. 5 The α-Miner in terms of its abstract representation



i
1
2
$\cdots$
n
$\cdots$
$\mathcal{S}(i)$
(c_1, a_1)
(c_2, a_2)
(c_n, a_n)
$\mathcal{D}_0^{\mathcal{T}}$
$\mathcal{D}_1^{\mathcal{T}}$
$\mathcal{D}_2^{\mathcal{T}}$
$\mathcal{D}_n^{\mathcal{T}}$
$\delta_{\mathcal{D}^{\mathcal{T}}}$
$\mathcal{A}_1^{\mathbf{T}}$
$\mathcal{A}_2^{\mathbf{T}}$
$\mathcal{A}_n^{\mathbf{T}}$
$\lambda_{\mathcal{D}^{\mathcal{T}}}^{\mathcal{A}^{\mathbf{T}}}$
$\mathcal{M}_1$
$\mathcal{M}_2$
$\mathcal{M}_n$
$\gamma^{\mathcal{A}^{\mathbf{T}}}$

Fig. 6 Detailed overview of the S-BAR architecture

depicted schematically. The S-BAR architecture conceptually splits event stream-based process discovery into three components, highlighted in grey in Fig. 6. We explain the purpose of each component, i.e. $\delta_{\mathcal{D}^{\mathcal{T}}}$, $\lambda_{\mathcal{D}^{\mathcal{T}}}^{\mathcal{A}^{\mathbf{T}}}$ and $\gamma^{\mathcal{A}^{\mathbf{T}}}$, by means of an example.

Consider maintaining the directly follows abstraction, introduced in Example 1, on a stream. To do this, we need a data structure that tracks the most recent activity for each case. Given such data structure, if we receive new event (c, a), we check whether we already received an activity a' for case c or whether a is the first activity received for case c. If we already received activity a' for case c, we deduce $a' > a$. Subsequently, we update our data structure such that it now assigns a to be the last activity received for case c.

The first component, i.e. $\delta_{\mathcal{D}^{\mathcal{T}}}$, maintains and updates a (collection of) data structure(s) that together form a *sufficient representation* of the behaviour entailed by the event stream. In context of our example, the first component is mainly concerned with keeping track of pairs of activities that are in a $a' > a$ relation. The second component, i.e. $\lambda_{\mathcal{D}^{\mathcal{T}}}^{\mathcal{A}^{\mathbf{T}}}$, translates the data structure to an abstract representation. In context of our example, this consists of translating the pairs of activities that are in a $a' > a$ relation into the directly follows abstraction. The third component, i.e. $\gamma^{\mathcal{A}^{\mathbf{T}}}$, translates the abstract representation to a process model and is inherited from conventional process discovery.

In the remainder, given an arbitrary *data structure type* $\mathcal{T}$, we let $\mathcal{U}_{\mathcal{D}^{\mathcal{T}}}$ denote the universe of data structures of type $\mathcal{T}$. A data type $\mathcal{T}$ 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 [22]. We assume any $\mathcal{D}^{\mathcal{T}} \in \mathcal{U}_{\mathcal{D}^{\mathcal{T}}}$ to use finite memory.

Definition 4 (*Data structure update function*) Let A be a set of activities and let C denote the set of all possible case identifiers. We define a data structure update function $\delta_{\mathcal{D}^{\mathcal{T}}}$ as:

$$\delta_{\mathcal{D}^{\mathcal{T}}} : \mathcal{U}_{\mathcal{D}^{\mathcal{T}}} \times C \times A \to \mathcal{U}_{\mathcal{D}^{\mathcal{T}}} \tag{3}$$

The data structure update function $\delta_{\mathcal{D}^{\mathcal{T}}}$ allows us to update a given data structure $\mathcal{D}^{\mathcal{T}} \in \mathcal{U}_{\mathcal{D}^{\mathcal{T}}}$ based on any newly arrived event. In practice, the function typically consists of two components. One component keeps track of the cases that were already active before and maps them in some way to a second (collection of) data structure(s). Such second component allows us to construct the abstract representation. Thus, when abstracting this mechanism, given some event stream-based data structure, we need a mechanism to translate the data structure, i.e. the range of $\delta_{\mathcal{D}^{\mathcal{T}}}$, to an abstract representation.

Definition 5 (*Abstraction function—data structure*) An abstraction function $\lambda_{\mathcal{D}^{\mathcal{T}}}^{\mathcal{A}^{\mathbf{T}}}$ is a function mapping a data structure of type $\mathcal{T}$ to an abstract representation of type $\mathbf{T}$.

$$\lambda_{\mathcal{D}^{\mathcal{T}}}^{\mathcal{A}^{\mathbf{T}}} : \mathcal{U}_{\mathcal{D}^{\mathcal{T}}} \rightarrow \mathcal{U}_{\mathcal{A}^{\mathbf{T}}} \tag{4}$$

Ideally, translating the data structure is computationally inexpensive. However, in some cases translating the data structure to the intended abstract representation might be expensive. This is acceptable, as long as we (re)-compute the abstraction in a periodic fashion or at the user's request.

Assume that we have seen $i \geq 0$ events on an event stream $\mathcal{S}$ and let $\mathcal{D}_i^{\mathcal{T}} \in \mathcal{U}_{\mathcal{D}^{\mathcal{T}}}$ denote the data structure that approximates the behaviour in the event stream $\mathcal{S}$ after receiving i events. When new event $(c, a) \in C \times A$ arrives, we are able to discover a new process model $\mathcal{M}_{i+1}$ by applying $\left(\gamma^{\mathcal{A}^{\mathbf{T}}} \circ \lambda_{\mathcal{D}^{\mathcal{T}}}^{\mathcal{A}^{\mathbf{T}}} \circ \delta_{\mathcal{D}^{\mathcal{T}}}\right)\left(\mathcal{D}_i^{\mathcal{T}}, c, a\right)$. In practice, $\delta_{\mathcal{D}^{\mathcal{T}}}$ is applied continuously and whenever, after receiving a new ith event, we are interested in finding a process model we apply $\left(\gamma^{\mathcal{A}^{\mathbf{T}}} \circ \lambda_{\mathcal{D}^{\mathcal{T}}}^{\mathcal{A}^{\mathbf{T}}}\right)\left(\mathcal{D}_i^{\mathcal{T}}\right)$ to obtain the process model.

The main challenge in instantiating the framework is designing a data structure $\mathcal{D}^{\mathcal{T}} \in \mathcal{U}_{\mathcal{D}^{\mathcal{T}}}$ that allows us to approximate an abstract representation together with accompanying $\delta_{\mathcal{D}^{\mathcal{T}}}$ and $\lambda_{\mathcal{D}^{\mathcal{T}}}^{\mathcal{A}^{\mathbf{T}}}$ functions.

5 Instantiating S-BAR

In this section, we show the applicability of the S-BAR framework by presenting several instantiations for different existing process discovery algorithms. A large class of algorithms, e.g. the α-Miner [3], the Heuristics Miner [6,23] and the Inductive Miner [5], is based on the directly follows abstraction. Therefore, we first present how to compute this abstraction. 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 w.r.t. 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 [2,7]. These techniques work fundamentally different compared to the aforementioned class of algorithms and use different abstract representations.

5.1 Directly follows abstraction

The *directly follows abstraction* describes pairs of activities (a, b), written as $a > b$, if there exists some sequence $\sigma \in L$ of the form $\sigma = \sigma' \cdot \langle a, b \rangle \cdot \sigma''$. To approximate the relation, we let data structure $\mathcal{D}^{\mathcal{T}} \in \mathcal{U}_{\mathcal{D}^{\mathcal{T}}}$ consist of two internal data structures $\mathcal{D}^{\mathcal{C}}$ and $\mathcal{D}^{\mathcal{A}}$. Within $\mathcal{D}^{\mathcal{C}}$, we store (case, activity)-pairs, i.e. $(c, a) \in C \times A$, that represent the last activity a seen for case c. Within $\mathcal{D}^{\mathcal{A}}$, we store (activity, activity)-pairs $(a, a') \in A \times A$, where $(a, a') \in \mathcal{D}^{\mathcal{A}} \Leftrightarrow a > a'$. The basic scheme works as follows. When a new event (c, a) arrives, we check whether $\mathcal{D}^{\mathcal{C}}$ already contains some pair (c, a'). If so, we add (a', a) to $\mathcal{D}^{\mathcal{A}}$, remove (c, a') from $\mathcal{D}^{\mathcal{C}}$ and add (c, a) to $\mathcal{D}^{\mathcal{C}}$. If not, we just add (c, a) to $\mathcal{D}^{\mathcal{C}}$. $\mathcal{D}^{\mathcal{A}}$ represents the directly follows abstraction by means of a collection of pairs; thus, function

Algorithm 1: $\mathcal{D}^{\mathcal{C}}$ (Space Saving)

```
input : $k \in \mathbb{N}, \mathcal{S} \in (C \times A)^*, \mathcal{D}^{\mathcal{A}}$
begin
1    $X \leftarrow \emptyset; i \leftarrow 0;$
2    while $true$ do
3        $i \leftarrow i + 1;$
4        $(c, a) \leftarrow \mathcal{S}(i);$
5        if $\exists_{(c',a') \in X}(c' = c)$ then
6            $v_c \leftarrow v_c + 1;$
7            $\mathcal{D}^{\mathcal{A}} \uplus \{(a', a)\};$
8            $X \leftarrow (X \cup \{(c, a)\}) \setminus \{(c, a')\};$
9        else if $|X| < k$ then
10           $X \leftarrow X \cup \{(c, a)\};$
11           $v_c \leftarrow 1;$
12       else
13           $(c', a') \leftarrow \arg\min_{(c',a') \in X} (v_{c'});$
14           $v_c \leftarrow v_{c'} + 1;$
15           $X \leftarrow (X \cup \{(c, a)\}) \setminus \{(c', a')\};$
```

Algorithm 2: $\mathcal{D}^{\mathcal{C}}$ (Lossy)

```
input : $k \in \mathbb{N}, \mathcal{S} \in (C \times A)^*, \mathcal{D}^{\mathcal{A}}$
begin
1    $i, \Delta \leftarrow 0; X \leftarrow \emptyset;$
2    while $true$ do
3        $i \leftarrow i + 1;$
4        $(c, a) \leftarrow \mathcal{S}(i);$
5        if $\exists_{(c',a') \in X}(c' = c)$ then
6            $v_c \leftarrow v_c + 1;$
7            $\mathcal{D}^{\mathcal{A}} \uplus \{(a', a)\};$
8            $X \leftarrow (X \cup \{(c, a)\}) \setminus \{(c, a')\};$
9        else
10           $X \leftarrow X \cup \{(c, a)\};$
11           $v_c \leftarrow \Delta;$
12       if $\lfloor i/k \rfloor \neq \Delta$ then
13           foreach $(c', a') \in X$ do
14               if $v_{c'} \leq \Delta$ then
15                   $X \leftarrow X \setminus (c', a');$
16           $\Delta \leftarrow \lfloor i/k \rfloor;$
```

Algorithm 3: $\mathcal{D}^{\mathcal{A}}$ (Frequent)

```
input : $k \in \mathbb{N}, \mathcal{S}_A \in (A \times A)^*$
begin
1    $X \leftarrow \emptyset, i \leftarrow 0;$
2    while $true$ do
3        $i \leftarrow i + 1;$
4        $(a, a') \leftarrow \mathcal{S}_A(i);$
5        if $(a, a') \in X$ then
6            $v_{(a,a')} \leftarrow v_{(a,a')} + 1;$
7        else if $|X| < k$ then
8            $X \leftarrow X \cup \{(a, a')\};$
9            $v_{(a,a')} \leftarrow 1;$
10       else
11           foreach $(x, y) \in X$ do
12               $v_{(x,y)} \leftarrow v_{(x,y)} - 1;$
13               if $v_{(x,y)} = 0$ then
14                   $X \leftarrow X \setminus \{(x, y)\};$
```

$\lambda^{\mathcal{A}^{\mathbf{T}}}_{\mathcal{D}^{\mathcal{T}}}$ consists of translating $\mathcal{D}^{\mathcal{A}}$ to the appropriate underlying data type used by the discovery algorithm of choice.

As an example, consider Algorithms 1 and 2 describing a design of $\mathcal{D}^{\mathcal{C}}$ based on the SpaceSaving algorithm [24] and Lossy Counting [22], respectively.

Both algorithms have three inputs, i.e. a maximum size $k \in \mathbb{N}$, an event stream $\mathcal{S} \in (C \times A)^*$ and a finite memory data structure implementing $\mathcal{D}^{\mathcal{A}}$. The algorithms maintain a set of (case, activity)-pairs X, initialized to $\emptyset$ (line 1). For each case c present in X, an associated counter v_c is maintained which is used for memory management. When a new event (c, a) appears on the event stream, the algorithms check whether some pair (c', a') s.t. $c = c'$ is stored in X (line 5). If this is the case, c's counter is increased, (a', a) is added to data structure $\mathcal{D}^{\mathcal{A}}$, and (c, a') is replaced by (c, a) in X (lines 6–8). The algorithms differ in the way they process events (c, a) for which $\nexists_{(c',a') \in X}(c' = c)$. The Space Saving-based algorithm (Algorithm 1) either adds the element to X if $|X| < k$ or replaces pair $(c', a') \in X$ with the lowest corresponding counter ($v_{c'}$) value (Algorithm 1, lines 9–15). The Lossy Counting- based algorithm cleans up its X-set after each block of k consecutive events and removes all those entries that have a counter value lower than variable Δ (lines 9–16).

Both algorithms insert a new element in data structure $\mathcal{D}^{\mathcal{A}}$ in line 7. Conceptually, the algorithms generate a stream of (activity, activity)-pairs. Hence, in Algorithm 3 we present a basic design for $\mathcal{D}^{\mathcal{A}}$ based on the Frequent Algorithm [25,26] which uses an activity pair stream $\mathcal{S}_A \in (A \times A)^*$ as an input. Thus, $\mathcal{D}^{\mathcal{A}} \uplus \{(a', a)\}$ in line 7 of Algorithms 1 and 2 represents adding pair (a, a') at the end of stream $\mathcal{S}_A$.

The algorithm stores pairs of activities in its internal set X. Whenever a new pair (a, a') arrives, the algorithm checks whether it is already present in X, if so, it updates the corresponding counter $v_{(a,a')}$. If the pair is not yet present in X, the size of X is evaluated. If $|X| < k$, the new pair is added to X and a new counter is created for the pair. If $|X| \geq k$, the new pair is not added; moreover, each counter is decreased by one and if a counter gets value 0 the corresponding pair is removed.

The general mechanism of Algorithm 3 is very similar to Algorithm 1. The main difference consists of how to update X when $|X| \geq k$. All three algorithms use a parameter k which, in a way, represents the (maximum) size of X. Hence, when we write $|\mathcal{D}^{\mathcal{C}}|$, $|\mathcal{D}^{\mathcal{A}}|$, respectively, we implicitly refer to the value of k. It should be clear that we are also able to implement $\mathcal{D}^{\mathcal{C}}$ based on the Frequent Algorithm, i.e. we just adopt a different updating mechanism for X. Likewise, we are also able to design $\mathcal{D}^{\mathcal{A}}$ based on the Space Saving/Lossy Counting algorithm. Moreover, for $\mathcal{D}^{\mathcal{C}}$ we are able to use other types of stream-aware data structures, i.e. techniques adopting a different scheme to ensure finite memory. Examples of such types of techniques are Reservoir Sampling [19], and/or Decay- Based Schemes [21]. In the next sections, we briefly explain how the α-Miner, Heuristics Miner and Inductive Miner use the directly follows abstraction and what changes to the base scheme must be applied in order to adopt them in a streaming setting.

5.1.1 The α-miner

The α-Miner [3] transforms the directly follows abstraction into a Petri net. When adopting the α-Miner 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 a new case, the corresponding activity represents a start activity. However, given that we at some point remove (case, activity)-pairs from $\mathcal{D}^{\mathcal{C}}$, we might designate some activities falsely as start activities, i.e. a new case may in fact refer to a previously removed case. Approximating the end activities is more 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 cases that are dropped out of $\mathcal{D}^{\mathcal{C}}$ are terminated. However, since we approximate case termination, using this approach may lead 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 $a \in A$ with $\nexists_{a' \in A}(a' \neq a \mid a' > a)$, and an end activity is an $a \in A$ with $\nexists_{a' \in A}(a' \neq a \mid a > a')$. 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 Sect. 8.2, we discuss this problem in depth.

5.1.2 The Heuristics Miner

The Heuristics Miner [6,23,27] 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 tasks a and b ($|a > b|$ denotes the number of occurrences of $a > b$). The basic scheme presented in Sect. 5.1 suffices for computing $a \Rightarrow b$, as long as $\mathcal{D}^{\mathcal{A}}$ explicitly tracks or approximates, the frequencies of its elements (in the scheme this is achieved by the internal counters).

5.1.3 The Inductive Miner

The Inductive Miner [5], like the α-Miner, uses the directly follows abstraction and start and end activities. It tries to find patterns within the directly follows abstraction that indicate certain behaviour, e.g. parallelism. Using these patterns, it splits the event log into several smaller logs and repeats the procedure. Due to its iterative nature, the Inductive Miner guarantees to find *sound workflow nets* [14]. The Inductive Miner has also been extended to handle noise and/or infrequent behaviour [28]. This requires, like the Heuristics Miner, to count the $>$-relation. In [29], a version of the Inductive Miner is presented in which the inductive steps are directly performed on the directly follows abstraction. In context of event streams, this is the most adequate version to use as we only need to maintain a (counted) directly follows abstraction.

5.2 Region theory

Several process discovery algorithms [2,7,30–32] are based on *region theory* which solve the Petri net synthesis problem [33]. Classical region theory techniques ensure strict formal properties for the resulting process models. Process discovery algorithms based on region theory relax these properties. We identify two different region theory approaches, i.e. *language-based* and *state-based* region theory, which use different forms of abstract representations.

5.2.1 Language-based approaches

Algorithms based on *language-based* region theory [7,30] rely on a *prefix closure* of the input event log, i.e. the set of all prefixes of all traces. It is trivial to adapt the scheme presented to compute the directly follows abstraction (Sect. 5.1) to prefix closures. In stead of storing (case, activity)-pairs in $\mathcal{D}^{\mathcal{C}}$, we store pairs $(c, \sigma) \in C \times A^*$. We additionally use a data structure $\mathcal{D}^{pc}$ which approximates the prefix closure. Whenever we receive an event (c, a), we look for a pair $(c, \sigma) \in \mathcal{D}^{\mathcal{C}}$. If such pair exist, we subsequently add $\sigma' = \sigma \cdot \langle a \rangle$ to $\mathcal{D}^{pc}$ and update (c, σ) to (c, σ'). If there is no such pair (c, σ), we add ϵ and $\langle a \rangle$ to $\mathcal{D}^{pc}$ and $(c, \langle a \rangle)$ to $\mathcal{D}^{\mathcal{C}}$. In case of [7], which uses integer linear programming where (an abstraction of) the prefix closure forms the constraint body, we simply store the constraints in $\mathcal{D}^{pc}$, rather than the prefix closure.

5.2.2 State-based approaches

Within process discovery based on state-based regions [2], a transition system is constructed based on *a view* of a trace. Examples of a view are the complete prefix of the trace, the multiset projection of the prefix, 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 is unknown.

As an example of a transition system based on a simple event log $L = [\langle a, b, c, d\rangle, \langle a, c, b, d\rangle]$, consider Fig. 7. In Fig. 7a 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 Fig. 7a, we do not limit the maximum size of the multisets. Figure 7b 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.

Consider the following scheme, similar to the scheme presented in Sect. 5.1. Given a view type V, e.g. a set view, we design $\mathcal{D}^{\mathcal{C}}$ to maintain pairs (c, v_c), s.t. v_c is the last view constructed for case c. Moreover, we maintain a collection of views $\mathcal{D}^V$. Updating $\mathcal{D}^V$ is straightforward. Given new event (c, a), based on v_c we compute some new view v'_c, add it to $\mathcal{D}^V$ and update (c, v_c) to (c, v'_c) in $\mathcal{D}^{\mathcal{C}}$, e.g. updating the size-1 set view means that the new view based on new event (c, a) is simply the set $\{a\}$. However, just maintaining size-1 sets in $\mathcal{D}^V$ does not suffice as the relations between those sets, i.e. the transitions in the transition system, are not present in $\mathcal{D}^V$.

The problem is fixed by maintaining the transition system in memory, rather than $\mathcal{D}^V$, and updating it directly when we receive new events. Given some latest view v_c for case c, i.e. $(c, v_c) \in \mathcal{D}^{\mathcal{C}}$, activity a of new event (c, a) represents the transition from v_c to the newly derived v'_c. Without a limit on the view size, translating the transition system into a Petri net is rather slow. Hence, in a streaming setting we limit the maximum size of the views. This, in turn, causes some challenges w.r.t. $\mathcal{D}^{\mathcal{C}}$ and translation function $\lambda^{\mathcal{A}^{\mathbf{T}}}_{\mathcal{D}^{\mathcal{T}}}$. Consider the case where we maintain a multiset/set view of traces with some arbitrary finite capacity k. Moreover, given $k = 2$, assume we receive event (c, a) and $(c, \{a', a''\}) \in \mathcal{D}^{\mathcal{C}}$. The question is whether the new view for c is $\{a, a'\}$ or $\{a, a''\}$? Only if we store the last two events observed for c, in order, we are able to answer this question, i.e. if $(c, \langle a', a''\rangle) \in \mathcal{D}^{\mathcal{C}}$ we deduce the new view to be $\{a, a''\}$. Finally note that when we aim at removing paths from the transitions system, for example when we remove cases from c from $\mathcal{D}^{\mathcal{C}}$, we need to store the whole trace for c in order to be able to reduce all states and transitions related to case c.

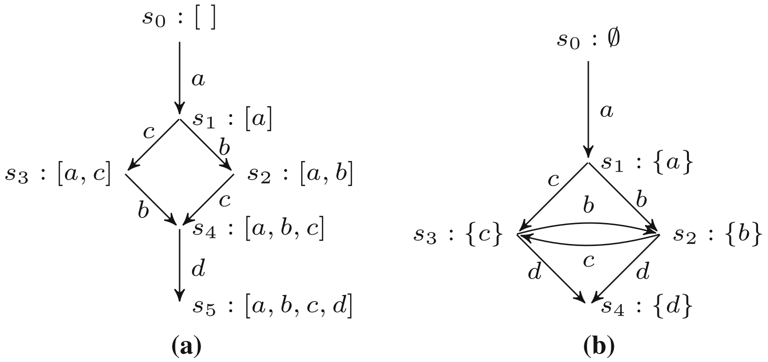


Fig. 7 Example transition systems based on $L = [\langle a, b, c, d\rangle, \langle a, c, b, d\rangle]$, **a** multiset abstraction (unbounded), **b** set abstraction (max. set size 1)

6 Evaluation

In this section, we present an evaluation of several instantiations of the architecture. We also consider performance aspects of the implementation. All five algorithms, i.e. α-Miner, Heuristics Miner, Inductive Miner, ILP (language- based regions) and Transition System Miner (state-based regions), have been implemented using the schemes presented in Sect. 5 in the ProM [8] framework (http://www.promtools.org). ProM is the de facto standard academic toolkit for process mining algorithms and is additionally used by practitioners in the field. Some of the implementations are ported to RapidProM [9] (http://www.rapidprom.org), i.e. a plugin of RapidMiner (http://www.rapidminer.com), which allows for designing large-scale repetitive experiments by means of scientific workflows [10]. Source code of the implementations is available via the *stream*-related packages within the ProM code base, i.e. *StreamAbstractRepresentation*, *StreamAlphaMiner*, *StreamHeuristicsMiner*, *StreamILPMiner*, *StreamInductiveMiner* and *StreamTransitionSystemsMiner* (code for a package X is located at http://svn.win.tue.nl/repos/prom/Packages/X). Experiment results, event streams and generating process models used, are available at https://github.com/s-j-v-zelst/research/releases/download/kais1/2016_kais1_experiments.tar.gz.

6.1 Structural analysis

As a first visual experiment, we investigate the steady-state behaviour of the *Inductive Miner* [5]. For both $\mathcal{D}^{\mathcal{C}}$ and $\mathcal{D}^{\mathcal{A}}$, we use the Lossy Counting scheme (Sect. 5.1). To create an event stream, we created a timed Coloured Petri Net [34] in CPN tools [35] which simulates the BPMN model depicted in Fig. 2 and emits the corresponding events. The event stream and all other event streams used for experiments are free of noise. The model is able to simulate multiple cases being executed simultaneously. The ProM streaming framework [36,37] is used to generate an event stream out of the process model.

In Fig. 8, we show the behaviour of the Inductive Miner over time, configured with $|\mathcal{D}^{\mathcal{C}}| = 75$, $|\mathcal{D}^{\mathcal{A}}| = 75$, based on a random simulation of the CPN model. 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, not enough 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 enough behaviour is emitted onto the stream, Model 5 shows a Petri net version of example process model of Fig. 2.

Figure 8 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 we can *convert to* a Petri net. The techniques however differ in terms of guarantees w.r.t. the resulting process model. The Inductive Miner guarantees that the resulting Petri nets are *sound*, 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.

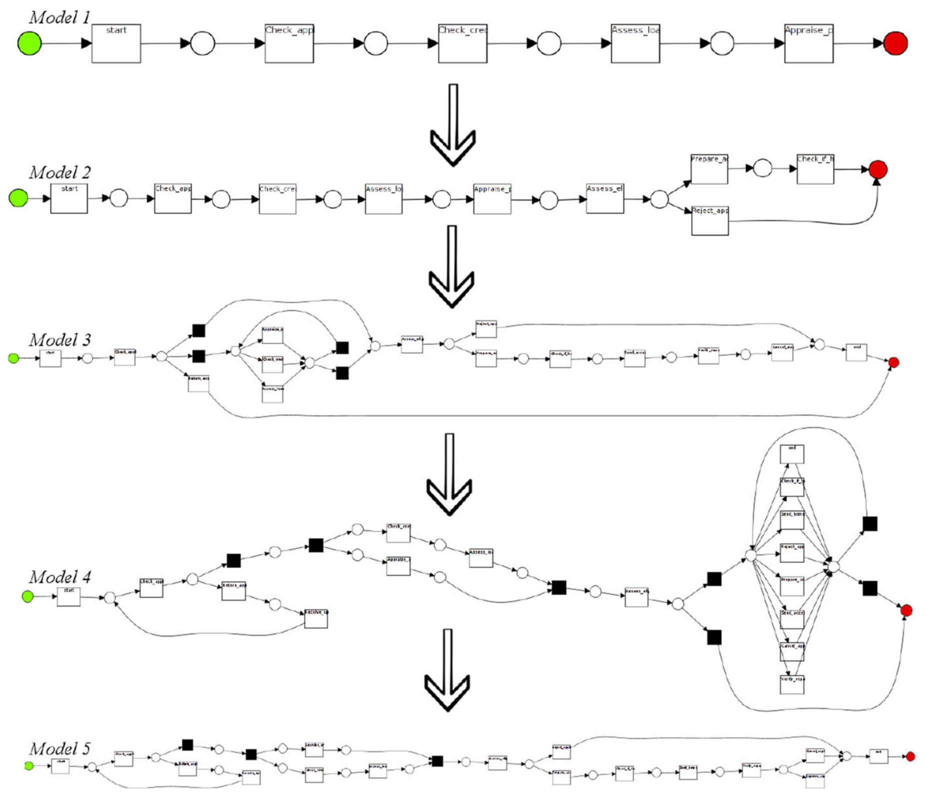


Fig. 8 Visual results of applying the Inductive Miner on a stream

Using the offline variant of each algorithm, we first compute a reference Petri net. We generated an event log L which contains *enough behaviour* such that the discovered Petri nets describe all behaviour of the BPMN model of Fig. 2. Based on the reference Petri net, we create a 15-by-15 matrix in which each row/column corresponds to an activity in the BPMN model. If, in the Petri net, two labelled transitions are connected by means of a place, the corresponding cells in the matrix get value 1. For example, given the first Petri net of Fig. 8, the labels *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 construct the 15-by-15 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 Fig. 8 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: $d_{M,M_R} = \sqrt{\sum_{i,j \in \{1,2,\ldots,15\}}((M(i,j) - M_R(i,j))^2}$. For all algorithms, the internal data structures used were based on Lossy Counting, with size 100.

Since the Inductive Miner and the α-Miner are completely based on the same abstraction, we expect them to behave similar. Hence, we plot their corresponding results together in Fig. 9a. 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 Fig. 9b. We observe that the algorithms behave similar to the α- 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 abstract 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.

6.2 Behavioural analysis

Although the previous experiments provide interesting insights w.r.t. the functioning of the algorithms in a streaming setting, they only consider structural model quality. A distance value of 0 in Fig. 9 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 w.r.t. 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 S-BAR framework, using the scheme described in Sect. 5.1, combined with the modifications described in Sect. 5.1.3. For finding start and end activities, we traverse the directly follows abstraction and select activities that have no predecessor, or, successor, respectively. We again use Lossy Counting [22] to implement both $\mathcal{D}^{\mathcal{C}}$ and $\mathcal{D}^{\mathcal{A}}$ (Algorithm 2, Sect. 5.1).

We assess under what conditions the Inductive Miner instantiation is able to discover a process model with the same behaviour as the BPMN model in Fig. 2. In the experiment, after each received event, we query the miner for its current result and compute replay fitness

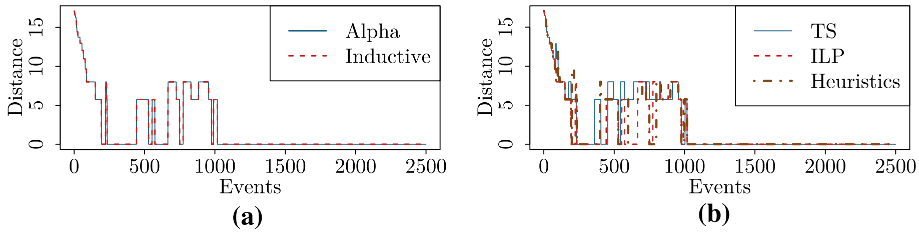


Fig. 9 Distance measurements for the α-Miner, Inductive Miner (IM), ILP Miner (ILP), Transition Systems Miner (TS) and Heuristics Miner, **a** distances for α and IM, **b** distances for TS, ILP and HM

and precision measures based on a complete corresponding event log. In Fig. 10, the results are presented for varying capacity sizes of the underlying data structure (Lossy Counting).

For the smallest data structure sizes, i.e. Fig. 10a, we identify that the replay fitness does not stabilize. When the data structure size increases, i.e. Fig. 10b, we identify the replay fitness to reach a value of 1 rapidly. The high variability in the precision measurements present in Fig. 10c 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. The stable pattern present in Fig. 10d 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 $\mathcal{D}^{\mathcal{C}}$ and $\mathcal{D}^{\mathcal{A}}$. Here we study the influence of the individual sizes of $\mathcal{D}^{\mathcal{C}}$ and $\mathcal{D}^{\mathcal{A}}$. In Fig. 11 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 11a depicts the results for a fixed value $|\mathcal{D}^{\mathcal{C}}| = 100$ and varying sizes $|\mathcal{D}^{\mathcal{A}}| = 10, 20, \ldots, 50$. Figure 11b depicts the

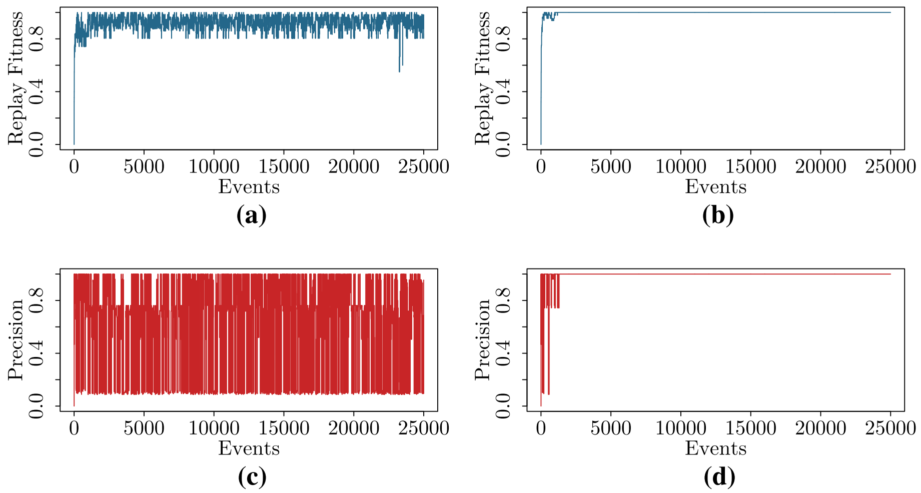


Fig. 10 Replay fitness and Precision measures based on applying the Stream Inductive Miner: increasing memory helps to improve fitness and precision, **a** $|\mathcal{D}^{\mathcal{C}}| = 25$, $|\mathcal{D}^{\mathcal{A}}| = 25$, **b** $|\mathcal{D}^{\mathcal{C}}| = 75$, $|\mathcal{D}^{\mathcal{A}}| = 75$, **c** $|\mathcal{D}^{\mathcal{C}}| = 25$, $|\mathcal{D}^{\mathcal{A}}| = 25$, **d** $|\mathcal{D}^{\mathcal{C}}| = 75$, $|\mathcal{D}^{\mathcal{A}}| = 75$

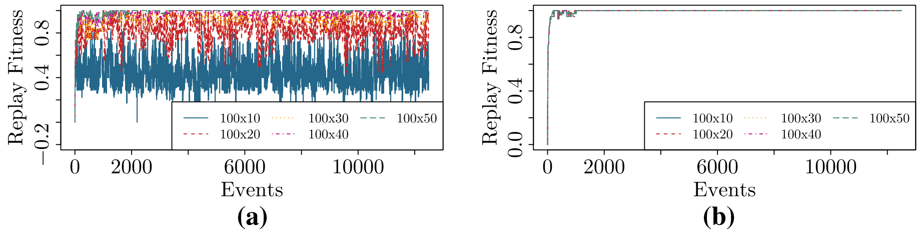


Fig. 11 Replay Fitness measures for the Stream Inductive Miner, **a** $|\mathcal{D}^{\mathcal{C}}| = 100$, $|\mathcal{D}^{\mathcal{A}}| = 10, 20, \ldots, 50$, **b** $|\mathcal{D}^{\mathcal{C}}| = 10, 20, \ldots, 50$, $|\mathcal{D}^{\mathcal{A}}| = 100$

results for a fixed value $|\mathcal{D}^{\mathcal{A}}| = 100$ and varying sizes $|\mathcal{D}^{\mathcal{C}}| = 10, 20, \ldots, 50$. As the results show, the lack of conversion to a replay fitness value of 1 mostly depends on the size of $\mathcal{D}^{\mathcal{A}}$ and is relatively independent of the size of $\mathcal{D}^{\mathcal{C}}$. Intuitively, this makes sense as we only need one entry $(c, a) \in \mathcal{D}^{\mathcal{C}}$ to deduce $a > b$, given that the newly received event is (c, b). Even if case c is dropped at some point in time, and reinserted later, still information regarding the directly follows abstraction can be deduced. However, if not enough space is reserved for the $\mathcal{D}^{\mathcal{A}}$ data structure, then the data structure is incapable of storing the complete directly follows abstraction.

6.3 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 extend the Inductive Miner-based instantiation of the framework is able to handle *concept drift* [38,39]. We focus on *gradual drift*, i.e. the behaviour of the process model changes at some point in time, though the change is only applicable for new cases, already active cases follow the old behaviour. In order to obtain a gradual drift, we manipulated the CPN simulation model of the process model presented in Fig. 2. The first five hundred cases that are simulated follow the original model. All later cases are routed to a model in which we swap the parallel and choice structures within the model (Fig. 12).

Figure 13 depicts the results of applying the Inductive Miner on the described gradual drift. In Fig. 13a, we depict the results using data structure sizes $|\mathcal{D}^{\mathcal{C}}| = 100$ and $|\mathcal{D}^{\mathcal{A}}| = 50$ (Lossy Counting). The blue solid line depicts the replay fitness w.r.t. an event log containing

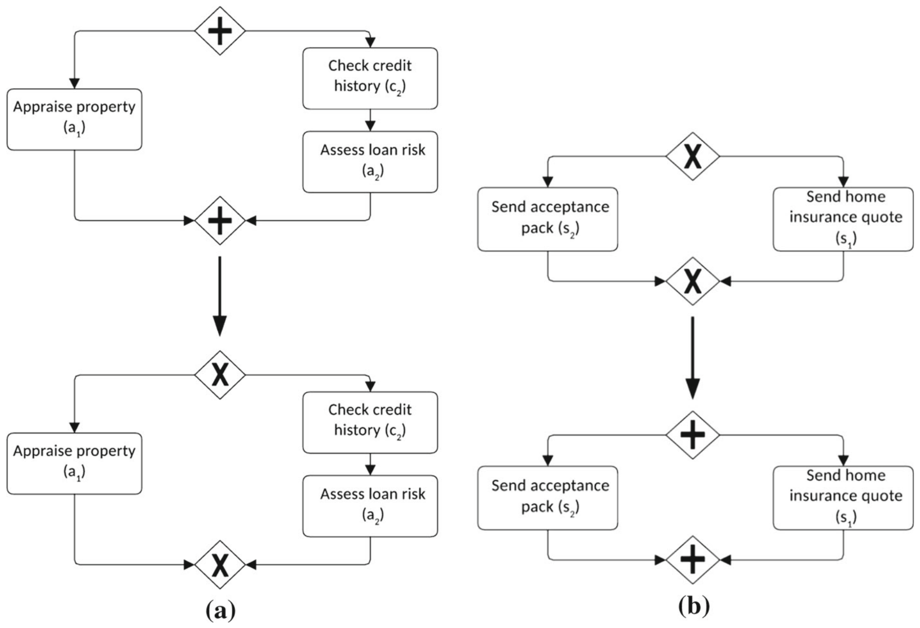

Check credit history (c2)
Appraise property (a1)
Assess loan risk (a2)
Check credit history (c2)
Appraise property (a1)
Assess loan risk (a2)
(a)
Send acceptance pack (s2)
Send home insurance quote (s1)
Send acceptance pack (s2)
Send home insurance quote (s1)
(b)

Fig. 12 Changes made to the business process model presented in Fig. 2, **a** parallel to choice, **b** choice to parallel

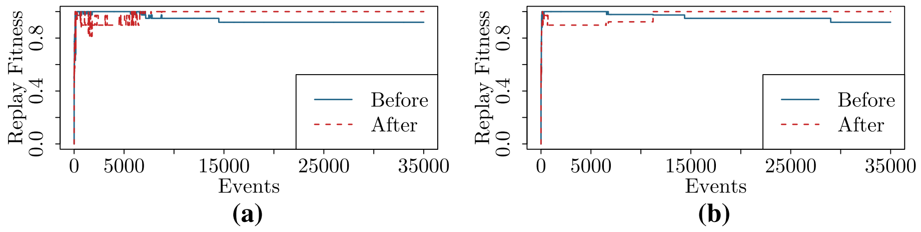


Fig. 13 Replay Fitness measures for the Stream Inductive Miner, given an event stream containing concept drift, **a** $|\mathcal{D}^{\mathcal{C}}| = 100$, $|\mathcal{D}^{\mathcal{A}}| = 50$, **b** $|\mathcal{D}^{\mathcal{C}}| = 100$, $|\mathcal{D}^{\mathcal{A}}| = 100$

behaviour *prior* to the drift, the red dashed line represents replay fitness w.r.t. an event log containing behaviour *after* the drift. We observe that the algorithm again needs some time to stabilize in terms of behaviour w.r.t. the pre-drift model. Interestingly, at the moment that the algorithm seems to be stabilized w.r.t. the pre-drift model, the replay fitness w.r.t. 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 post-drift model. The first event in the stream related to the new execution of the process is the 6.415th event. Indeed, the blue solid line drops around this point in Fig. 13a. Likewise, the red dashed line rapidly increases to value 1.0. Finally, around event 15.000 the replay fitness w.r.t. 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 Fig. 13b, we depict results for the Inductive Miner using sizes $|\mathcal{D}^{\mathcal{C}}| = 100$ and $|\mathcal{D}^{\mathcal{A}}| = 100$. 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 k-value of the Lossy Counting Algorithm, the drift is reflected longer in the replay fitness values. Only after roughly the 30.000th event the replay fitness w.r.t. the pre-drift model stabilizes.

6.4 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.

Within the 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): $|\mathcal{D}^{\mathcal{C}}| = 25$ and $|\mathcal{D}^{\mathcal{A}}| = 25$, $|\mathcal{D}^{\mathcal{C}}| = 50$ and $|\mathcal{D}^{\mathcal{A}}| = 50$ and $|\mathcal{D}^{\mathcal{C}}| = 75$, $|\mathcal{D}^{\mathcal{A}}| = 75$ (represented in the figures as 25×25, 50×50 and 75×75, respectively). We measured the time the algorithm needs to update both $\mathcal{D}^{\mathcal{C}}$ and $\mathcal{D}^{\mathcal{A}}$. The memory measured is the combined size of $\mathcal{D}^{\mathcal{C}}$ and $\mathcal{D}^{\mathcal{A}}$ in bytes. The results of the experiments are depicted in Fig. 14. Both figures depict the total number of events received on the x axis. In Fig. 14a, the processing time in nanoseconds is shown on the y axis, whereas in Fig. 14b, the memory usage in bytes is depicted. The aggregates of the experiments are depicted in Table 2.

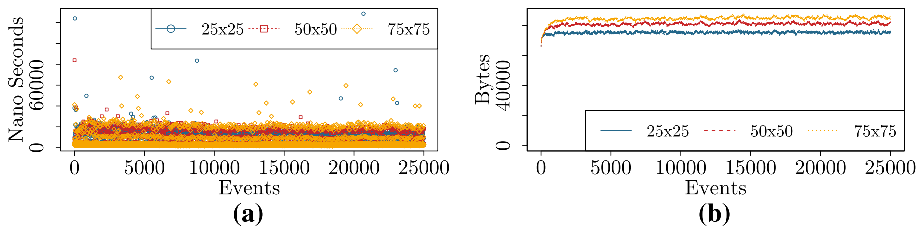


Fig. 14 Performance measurements based on the Stream Inductive Miner, **a** processing times in nanoseconds, **b** memory usage in bytes

Table 2 Aggregate performance measures for the Stream Inductive Miner

	25 × 25	50 × 50	75 × 75
Avg. processing time (ns.)	4.168	3.866	3.519
Stdev. processing time (ns.)	3.245	2.589	2.691
Avg. memory usage (byte)	75.392	81.014	84.696
Stdev. memory usage (byte)	75.392	1.230	1.725

As Fig. 14a 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 2). Intuitively this makes sense as a bigger window size of the Lossy Counting algorithm implies less frequent cleanup operations.

Like processing time, memory usage of the Lossy Counting data structures does not show an increasing trend (Fig. 14b). 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 less cleanup operations implies more active members within the data structures, and hence, a higher memory usage.

7 Related work

For a detailed overview of process mining, we refer to [1]. For an overview of models, techniques and algorithms in stream-based mining and analysis, e.g. frequency approximation algorithms, we refer to [40–42]. Little work has been done on the topic of stream-based process discovery and stream-based process mining in general. The notion of *streams of events* is not new, i.e. several fields study aspects related to streams of (discrete) events. Compared to the field of complex event processing (CEP) [43], the S-BAR architecture can be seen as an *event consumer*, i.e. a decoupled entity that processes the events produced by the underlying system. However, whereas the premise of CEP is towards the *design* of event- based systems and architectures, this work focuses on the *behavioural analysis* of such systems. The area of event mining [44] focuses on gaining knowledge from historical event/log data. Although the input data are similar, i.e. streams of system events, the assumptions on the data source are different. Within event mining, data mining techniques such as *pattern mining* [44, Chpt. 4] are used as opposed to techniques used within this paper, i.e. techniques discovering end-to-end process models with associated execution semantics. Also, event mining includes methods for system monitoring, whereas the S-BAR architecture can serve as an enabler for business process monitoring and prediction.

To the best of the author's knowledge, this paper is the first work that presents a generic architecture for the purpose of event stream- based process discovery. As such, the work may be regarded as a generalization and standardization effort of some of the related work mentioned within this section.

In [45], an event stream-based variant of the Heuristics Miner is presented. The algorithm uses three internal data structures using both Lossy Counting [22] and Lossy Counting with Budget [46]. The authors use these structures to approximate a causal graph based on an event stream. The authors additionally present a sliding window-based approach. Recently, an alternative data structure has been proposed based on prefix trees [47]. 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. In [48], Redlich et al. design an event stream-based variant of the CCM algorithm [49]. The authors identify the need to compute dynamic footprint information based on the event stream, which can be seen as the abstract representation used by CCM. The dynamic footprint is translated to a process model using a translation step called *Footprint Interpretation*. 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 this paper, the evaluation relies heavily on the concept of *completed traces*. In [50], Burattin et al. propose an event stream-based process discovery algorithm to discover declarative process models. The structure described to maintain events and their relation to cases is comparable with the one used in [45]. 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.

8 Discussion

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 w.r.t. the complexity of abstract representation computation and discuss the impact of the absence of trace initialization and termination information.

8.1 Complexity of abstract representation computation

There are limitations w.r.t. the algorithms we are able to adopt using abstract representations as basis. This is mainly related to the computation of the abstract representation within the conventional algorithm.

As an example, consider the α^+-algorithm [51] which extends the original α-Miner such that it is able to handle self-loops and length-1-loops. For handling self-loops, the α^+-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 abstraction. For example, if $L = [\langle a, b, c\rangle, \langle a, b, b, c\rangle]$, the algorithm will construct $L' = [\langle a, c\rangle]$ and compute directly follows metrics based on L'.

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 $(a', a) \in \mathcal{D}^{\mathcal{A}}$ and $(a, a'') \in \mathcal{D}^{\mathcal{A}}$, s.t. $a \neq a'$ and $a \neq a''$, we deduce that (a', a'') is part

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

(a)

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

(b)

Fig. 15 Two abstract representations, **a** event log, **b** event stream

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 shows that the relation is not always equal.

Imagine a process $\mathcal{P} = \{\langle a, b, b, c\rangle, \langle a, e, b, d\rangle\}$. Clearly, any noise-free event log over this process is just a multiset over the two traces in $\mathcal{P}$. In case of the conventional α^+-algorithm, removing the b-activity leads to the two traces $\langle a, c\rangle$ and $\langle a, e, d\rangle$. Consider the corresponding directly follows abstraction, depicted in Fig. 15a. Observe that all possible directly follows pairs that we are able to observe on any stream over $\mathcal{P}$ are: (a, b), (a, e), (b, b), (b, c), (b, d), (e, b). Applying the described procedure yields the abstraction depicted in Fig. 15b. 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 an abstraction-based algorithm that constructs the abstract representation in *one pass* over the event log.

8.2 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 in Sect. 5.1.1 regarding start/end activity sets used by the α-Miner and Inductive Miner, this can have a severe impact on computing the abstract 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/abstract 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 in Fig. 11 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 w.r.t. 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.

9 Conclusion

In this paper, 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 some abstract representation and subsequently use this representation to discover a process model. Thus, in an event stream-based setting, it suffices to approximate the abstract representation using 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 abstract representation.

Several instantiations of the architecture have been implemented in the process mining toolkits ProM and RapidProM. We primarily focused on abstract representation approximations using algorithms designed for the purpose of frequent item mining on data streams. We structurally evaluated and compared five different instantiations of the framework. From a behavioural perspective, we focused on the Inductive Miner as it grantees 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 instantiation. The experiments show that both processing time of new events and memory usage are non-increasing as more data are received.

Future work

Within the experiments, we chose to limit the use of internal data structure to the Lossy Counting-based approach. However, more instantiations, i.e. Frequent/Space Saving, are presented and implemented. We plan to investigate the impact of several different designs of the internal data structures w.r.t. both behaviour and performance.

The architecture presented in this work focuses on approximating abstract representations and exploiting existing algorithms to discover a process model. However, bulk of the work might be performed multiple times, i.e. several new events emitted to the stream might not change the abstract representation. We therefore plan to conduct a study towards a completely incremental instantiation of the architecture, i.e. can we immediately identify whether new data changes the abstraction or even the resulting model?

Another interesting direction for future work is to go beyond control-flow discovery, i.e. can we lift conformance checking, performance analysis, etc. to the domain of event streams? Moreover, in such cases we might need to store more information, i.e. store all attributes related to events within cases seen so far. We plan to investigate the application of lossless/lossy compression of the data seen so far, i.e. using frequency distributions of activities/attributes to encode sequences in a compact manner.

References

1. van der Aalst WMP (2016) Process mining—data science in action, 2nd edn. Springer, Berlin. doi:10.1007/978-3-662-49851-4
2. van der Aalst WMP, Rubin V, Verbeek HMW, van Dongen BF, Kindler E, Günther CW (2010) Process mining: a two-step approach to balance between underfitting and overfitting. Softw Syst Model 9(1):87–111. doi:10.1007/s10270-008-0106-z
3. van der Aalst WMP, Weijters T, Maruster L (2004) Workflow mining: discovering process models from event logs. IEEE Trans Knowl Data Eng 16(9):1128–1142. doi:10.1109/TKDE.2004.47
4. Günther CW, van der Aalst WMP (2007) Fuzzy mining-adaptive process simplification based on multi-perspective metrics. In: Business process management, 5th international conference, BPM 2007, Brisbane, September 24–28, 2007, proceedings, pp 328–343. doi:10.1007/978-3-540-75183-0_24
5. Leemans SJJ, Fahland D, van der Aalst WMP (2013) Discovering block-structured process models from event logs—a constructive approach. In: Application and theory of Petri nets and concurrency—34th international conference, PETRI NETS 2013, Milan, June 24–28, 2013, Proceedings, pp 311–329
6. Weijters AJMM, van der Aalst WMP (2003) Rediscovering workflow models from event-based data using little thumb. Integr Comput Aided Eng 10(2):151–162
7. van der Werf JMEM, van Dongen BF, Hurkens CAJ, Serebrenik A (2009) Process discovery using integer linear programming. Fundam Inf 94(3–4):387–412
8. van Dongen BF, de Medeiros AKA, Verbeek HMW, Weijters AJMM, van der Aalst WMP (2005) The ProM framework: a new era in process mining tool support. In: Applications and theory of Petri nets 2005, 26th international conference, ICATPN 2005, Miami, June 20–25, 2005, Proceedings, pp 444–454. doi:10.1007/11494744_25
9. van der Aalst WMP, Bolt A, van Zelst SJ (2017) RapidProM: mine your processes and not just your data. CoRR abs/1703.03740
10. Bolt A, de Leoni M, van der Aalst WMP (2015) Scientific workflows for process mining: building blocks, scenarios, and implementation. Int J Softw Tools Technol Transf. doi:10.1007/s10009-015-0399-5
11. Dumas M, La Rosa M, Mendling J, Reijers HA (2013) Fundamentals of business process management. Springer, Berlin. doi:10.1007/978-3-642-33143-5
12. Murata T (1989) Petri Nets: properties, analysis and applications. Proc IEEE 77(4):541–580
13. Object Management Group (2011) Business process model and notation (BPMN). Formal specification formal/2011-01-03, Object Management Group
14. van der Aalst WMP (1998) The application of Petri nets to workflow management. J Circuits Syst Comput 8(1):21–66. doi:10.1142/S0218126698000043
15. van Dongen BF, de Medeiros AKA, Wen L (2009) Process mining: overview and outlook of Petri net discovery algorithms. Trans Petri Nets Other Models Concurr 2:225–242
16. de Weerdt J, de Backer M, Vanthienen J, Baesens B (2012) A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. Inf Syst 37(7):654–676. doi:10.1016/j.is.2012.02.004
17. Buijs JCAM, van Dongen BF, van der Aalst WMP (2014) Quality dimensions in process discovery: the importance of fitness, precision, generalization and simplicity. Int J Cooper Inf Syst. doi:10.1142/S0218843014400012
18. Cormode G, Hadjieleftheriou M (2009) Methods for finding frequent items in data streams. VLDB J 19(1):3–20. doi:10.1007/s00778-009-0172-z
19. Aggarwal CC (2006) On biased reservoir sampling in the presence of stream evolution. In: Proceedings of the 32nd international conference on very large data bases, VLDB'06. VLDB Endowment, pp 607–618
20. Vitter JS (1985) Random sampling with a reservoir. ACM Trans Math Softw 11(1):37–57. doi:10.1145/3147.3165
21. Cormode G, Shkapenyuk V, Srivastava D, Xu B (2009) Forward decay: a practical time decay model for streaming systems. In: 2009 IEEE 25th international conference on data engineering, pp 138–149. doi:10.1109/ICDE.2009.65
22. Manku GS, Motwani R (2002) Approximate frequency counts over data streams. In: Proceedings of the 28th international conference on very large data bases, VLDB'02. VLDB Endowment, pp 346–357
23. Weijters AJMM, Ribeiro JTS (2011) Flexible heuristics miner (FHM). In: 2011 IEEE symposium on computational intelligence and data mining (CIDM), pp 310–317. doi:10.1109/CIDM.2011.5949453

24. Metwally A, Agrawal D, Abbadi A (2005) Efficient computation of frequent and top-k elements in data streams. In: Eiter T, Libkin L (eds) Proceedings of the 10th international conference on database theory, iCDT'05, pp 398–412. Springer, Berlin. doi:10.1007/978-3-540-30570-5_27
25. Demaine ED, López-Ortiz A, Munro JI (2002) Frequency estimation of internet packet streams with limited space. In: Möhring RH, Raman R (eds) Algorithms—ESA 2002, 10th annual European symposium, Rome, September 17–21, 2002, Proceedings, *Lecture Notes in Computer Science*, vol 2461, pp 348–360. Springer. doi:10.1007/3-540-45749-6_33
26. Karp RM, Shenker S, Papadimitriou CH (2003) A simple algorithm for finding frequent elements in streams and bags. ACM Trans Database Syst 28:51–55. doi:10.1145/762471.762473
27. Weijters AJMM, van der Aalst WMP, de Medeiros AKA (2006) Process mining with the heuristics miner-algorithm. BETA working paper series WP 166, Eindhoven University of Technology
28. Leemans SJJ, Fahland D, van der Aalst WMP (2013) Discovering block-structured process models from event logs containing infrequent behaviour. In: Business process management workshops-BPM 2013 international workshops, Beijing, August 26, 2013, Revised papers, pp 66–78. doi:10.1007/978-3-319-06257-0_6
29. Leemans SJJ, Fahland D, van der Aalst WMP (2015) Scalable process discovery with guarantees. In: Enterprise, business-process and information systems modeling—16th international conference, BPMDS 2015, 20th international conference, EMMSAD 2015, Held at CAiSE 2015, Stockholm, June 8–9, 2015, Proceedings, pp 85–101. doi:10.1007/978-3-319-19237-6_6
30. Bergenthum R, Desel J, Lorenz R, Mauser S (2007) Process mining based on regions of languages. In: Business process management, 5th international conference, BPM 2007, Brisbane, September 24–28, 2007, Proceedings, pp 375–383. doi:10.1007/978-3-540-75183-0_27
31. Carmona J, Cortadella J (2014) Process discovery algorithms using numerical abstract domains. IEEE Trans Knowl Data Eng 26(12):3064–3076. doi:10.1109/TKDE.2013.156
32. van Zelst SJ, van Dongen BF, van der Aalst WMP (2015) Avoiding over-fitting in ILP-based process discovery. In: Business process management—13th international conference, BPM 2015, Innsbruck, August 31–September 3, 2015, Proceedings, pp 163–171. doi:10.1007/978-3-319-23063-4_10
33. Badouel E, Bernardinello L, Darondeau P (2015) Petri net synthesis. Texts in theoretical computer science. An EATCS series. Springer. doi:10.1007/978-3-662-47967-4
34. Jensen K, Kristensen LM (2009) Coloured Petri nets—modelling and validation of concurrent systems. Springer, Berlin. doi:10.1007/b95112
35. Jensen K, Kristensen LM, Wells L (2007) Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. STTT 9(3–4):213–254. doi:10.1007/s10009-007-0038-x
36. van Zelst SJ, Burattin A, van Dongen BF, Verbeek HMW (2014) 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, September 10, 2014, p 81
37. van Zelst SJ, van Dongen BF, van der Aalst WMP (2015) 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, September 2, 2015, pp 85–89
38. Bose RPJC, van der Aalst WMP, Zliobaite I, Pechenizkiy M (2014) Dealing with concept drifts in process mining. IEEE Trans Neural Netw Learn Syst 25(1):154–171. doi:10.1109/TNNLS.2013.2278313
39. Schlimmer JC, Granger RH (1986) Beyond incremental processing: tracking concept drift. In: Proceedings of the 5th national conference on artificial intelligence. Philadelphia, August 11–15, 1986. Volume 1: Science, pp 502–507
40. Aggarwal CC (ed) (2007) Data streams, advances in database systems, vol 31. Springer, New York. doi:10.1007/978-0-387-47534-9
41. Gama J (2010) Knowledge discovery from data streams, 1st edn. Chapman and Hall, London. doi:10.1201/EBK1439826119
42. Muthukrishnan S (2005) Data streams: algorithms and applications. Found Trends Theor Comput Sci. doi:10.1561/0400000002
43. Etzion O, Niblett P (2010) Event processing in action. Manning Publications Company, Greenwich
44. Li T (2015) Event mining: algorithms and applications. Chapman and Hall, London
45. Burattin A, Sperduti A, van der Aalst WMP (2014) Control-flow discovery from event streams. In: Proceedings of the IEEE congress on evolutionary computation, CEC 2014, Beijing, July 6–11, 2014, pp 2420–2427
46. Da San Martino G, Navarin N, Sperduti A (2013) A lossy counting based approach for learning on streams of graphs on a budget. In: IJCAI 2013, proceedings of the 23rd international joint conference on artificial intelligence, Beijing, August 3–9, 2013

47. Hassani M, Siccha S, Richter F, Seidl T (2015) Efficient process discovery from event streams using sequential pattern mining. In: 2015 IEEE symposium series on computational intelligence, pp 1366–1373. doi:10.1109/SSCI.2015.195
48. Redlich D, Molka T, Gilani W, Blair G, Rashid A (2014) Scalable dynamic business process discovery with the constructs competition miner, pp 91–107
49. Redlich D, Molka T, Gilani W, Blair G, Rashid A (2014) Constructs competition miner: process control-flow discovery of BP-domain constructs, pp 134–150. doi:10.1007/978-3-319-10172-9_9
50. Burattin A, Cimitile M, Maggi FM, Sperduti A (2015) Online discovery of declarative process models from event streams. IEEE Trans Serv Comput 8(6):833–846. doi:10.1109/TSC.2015.2459703
51. de Medeiros AKA, van Dongen BF, van der Aalst WMP, Weijters AJMM (2005) Process mining for ubiquitous mobile systems: an overview and a concrete algorithm. In: Baresi L, Dustdar SM, Gall HC, Matera M (eds) Ubiquitous mobile information and collaboration systems, *Lecture notes in computer science*, vol 3272. Springer, Berlin, pp 151–165

Sebastiaan J. van Zelst is a Ph.D. candidate at the *Architecture of Information Systems* group at the Department of Mathematics and Computer Science of the Eindhoven University of Technology. His research focuses on the application and impact of stream-based analysis techniques on process mining. His personal research interests include process mining, data stream analysis and data mining.

Boudewijn F. van Dongen is an associate professor at the *Architecture of Information Systems* group at the Department of Mathematics and Computer Science of the Eindhoven University of Technology. His research focus is on Process Mining and specifically on conformance checking, and since 2003, he has been a key player in the development of the process mining tool ProM. Furthermore, he is a member of the IEEE Task Force on Process Mining and he published extensively in the process mining area, both in international conferences and in journals (e.g. DKE, EIS, IS, CAiSE, ATPN, BPM, ER, EDOC). He served in several program committees, among others for IEEE EDOC 2007, 2008, 2009, 2010, BPI 2007–2016.

Wil M. P. van der Aalst is a full professor of Information Systems at the Technische Universiteit Eindhoven (TU/e). At TU/e, he is the scientific director of the Data Science Center Eindhoven (DSC/e). Since 2003, he holds a part-time position at Queensland University of Technology (QUT). His personal research interests include process mining, Petri nets, business process management, workflow management, process modeling and process analysis. He is also a member of the Board of Governors of Tilburg University and an elected member of the Royal Netherlands Academy of Arts and Sciences, the Royal Holland Society of Sciences and Humanities, and the Academy of Europe.