Efficient Process Discovery From Event Streams Using Sequential Pattern Mining

Marwan Hassani*, Sergio Siccha†, Florian Richter‡ and Thomas Seidl§
Data Management and Data Exploration Group
RWTH Aachen University, Germany
Email: *hassani@cs.rwth-aachen.de, †siccha@cs.rwth-aachen.de, ‡florian.richter@rwth-aachen.de, §seidl@cs.rwth-aachen.de

***Abstract*—Process mining is an emerging research area that brings the well-established data mining solutions to the challenging business process modeling problems. Mining streams of business processes in the real time as they are generated is a necessity to obtain an instant knowledge from big process data. In this paper, we introduce an efficient approach for exploring and counting process fragments from a stream of events to infer a process model using the Heuristics Miner algorithm. Our novel approach, called *StrProM*, builds prefix-trees to extract sequential patterns of events from the stream. StrProM uses a batch-based approach to continuously update and prune these prefix-trees. The models are generated from those trees after applying a decaying mechanism over their statistics. The extensive experimental evaluation demonstrates the superiority of our approach over a state-of-the-art technique in terms of execution time using a real dataset, while delivering models of a comparable quality.**

I. Introduction

Process mining is an emerging research area that brings the well-established data mining solutions to the challenging business process modeling problems. These problems are provided to the data mining algorithms in the form of event logs. Typically, event logs contain detailed information about the activities that are taken within a business process. The main aim of process mining is to discover, monitor and improve real processes by extracting knowledge from event logs [1]. The achieved improvement of the extracted process models can be measured by the extent of conformance between them and the original event logs.

Although process mining is relatively a young research field, several approaches [2], [3], [4], [5], [6], [7] are already existing in its literature. These works presented interesting methods by examining different features of the event logs (cf. Section II). All of them have however assumed the existence of the complete event logs and the possibility to access it as much as needed to generate, in most of them, a single final process model. This is infeasible when considering the huge increases in the size of event logs generated from modern information systems supporting business processes [8]. The proposed approaches will face serious efficiency issues with the increase in both the size and the dimensionality of the collected events [9]. Moreover, one important recent research question in the field of process mining is the concept drift of the underlying business process [10]. Decision makers will lose important insights over such drifting process by having merely a single final model. An important additional evolving requirement in this context is the necessity to have instant knowledge about the process model in the real time of observing the event logs.

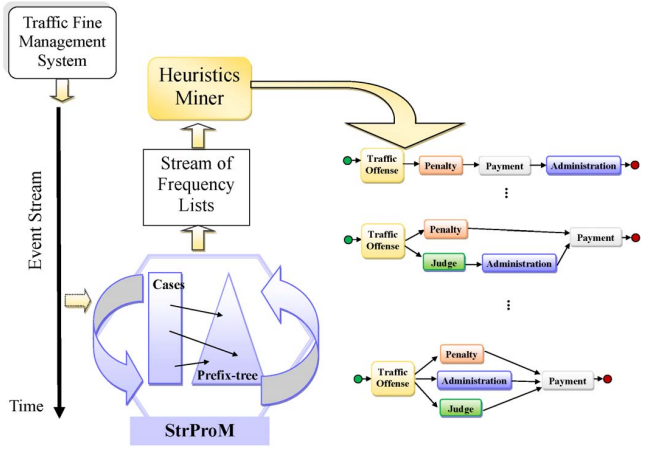

Fig. 1. An overview of our *StrProM* algorithm using an example stream of events from a Traffic Fine Management System. The streaming process models (Right) present a drift in the models (causal nets) over the time with the activities: *Traffic Offense, Penalty, Payment, Administration, Judge*.

With these new requirements, one started to speak about *event streams*, *streams of process models* and *streaming process discovery* [8]. Consider the event stream flowing from an example Traffic Fine Management System in Fig. 1. Observed events are received over the time by the streaming process discovery part in the middle column which continuously extracts and updates a stream of process models in the right side of Fig. 1. Stream mining is a well-established area with a rich literature in clustering, classification, outlier detection and frequent pattern mining [11], [12], [13], [14], [15] (cf. Section II). Efficient solutions for mining sequential patterns within data streams have been proposed recently to extract knowledge from evolving data streams with guarantees on the memory usage and the running time [16], [17].

In this paper, we present our novel streaming process discovery algorithm, called *StrProM*, that efficiently mines the flowing event stream to find streams of frequency lists. StrProM uses the novel prefix-tree structure with a decaying mechanism and builds above the Heuristics Miner [4] (cf. Fig. 1). More precisely, our contributions are:

1) Building a special prefix-tree data structure to get interesting statistics from the batches and applying a

978-1-4799-7560-0/15 $31.00 © 2015 IEEE
DOI 10.1109/SSCI.2015.195

novel pruning mechanism over the tree to guarantee an efficient processing of the events in the stream and an upper bound on memory usage,
2) Performing a unique, storage-aware pruning method to outdated cases,
3) Introducing an additional decaying of relation counts and activity counts to allow a faster adaptation to new behaviors and gradually forget older ones and
4) Achieving considerable time savings when comparing StrProM against a state-of-the-art competitor [8] using a real dataset, by keeping the quality of the models at least as good as the competitor.

The remainder of this paper is organized as follows: Section II lists some related work. In Section III we give a formulation of the problem. Section IV describes our novel method. In Section V we show our extensive experimental evaluation of StrProM against a state-of-the-art algorithm using a real dataset. Section VI concludes this paper and defines some future directions.

II. Related Work

Process Mining

Process mining originates from the fields of business process modeling and adapts techniques from data mining and various other fields. In 1998, Cook and Wolf [18] published an article in the field of software engineering on the discovery of models from event-based data, Agrawal et al. [19] published on the discovery of process models from workflow-logs and van der Aalst [3] examined the usage of Petri nets in the context of workflow management. Today, this area of process mining is called *process discovery* and over the last decade, several approaches and algorithms for the discovery of workflow models have been developed.

In 2001, the foundations for the Heuristics Miner algorithm were laid by Weijters and van der Aalst [2]. This led to the development of the algorithm called Little Thumb [20] in 2003 and Little Thumb itself lead to the development of the Heuristics Miner [4] algorithm by Weijters et al. It is a widely used process discovery algorithm, and according to [1] it is one of the few process discovery algorithms that is able to handle noise and incompleteness in event logs. In our contribution, we use it as the underlying model inferring method.

There are several other *process discovery* algorithms, and probably the most famous one is the α-Algorithm [5], developed by van der Aalst et al. in 2004. It is of a special theoretical importance, since it can be shown for a specific class of workflow nets, that the α-Algorithm is able to rediscover any model based on a generated suitable event-log [5]. Further *process discovery* algorithms are the Multi-Phase Miner [6] and the Fuzzy Miner [7]. In 2005, the framework ProM [21] was published. It includes all of the aforementioned algorithms and many more techniques from the different fields of process mining, and is since under further development [22]. We also aim to create a ProM-plugin from our contribution.

Stream Process Mining

The challenge to transfer concepts from stream mining to process mining has been mentioned in the *Process Mining Manifesto* [23]. Stream mining has been and is still studied extensively in the area of data mining [9]. Nevertheless, adopting stream data mining techniques to the field of process mining is not straightforward. We focus on the task of *streaming process discovery*, which, among other things, includes working on potentially unbounded event logs. Few algorithms have been developed, that are able to perform process discovery in a single pass over the data. Namely, Burattin et al. developed an online adaption of the Heuristics Miner to streaming event logs [24]. In [8], Burattin et al. performed an extensive analysis of an online Heuristics Miner, using Lossy Counting to keep track of the frequencies of the activities and the direct-follows relations. Another *streaming process discovery* algorithm was developed by Redlich et al. [25], and is based on their *process discovery* algorithm Constructs Competition Miner [26]

There are other approaches that use the Declare language to build a process model in real-time. This language [27] was developed to support loosely structured processes. It uses linear temporal logic to describe rules that the observed process adheres to. An algorithm to mine a declarative model from streaming event data was proposed by Maggi et al. in [28]. The Online Heuristics Miner results were modified to mine a Declare model by Burattin et al. in [29].

Stream Sequential Pattern Mining

In our contribution, we adapt stream sequential pattern mining algorithms to collect the frequencies of activities and their direct-follows relations. In 2004, Peit et al. proposed a static algorithm to mine sequential patterns: PrefixSpan [30]. This algorithm was developed into a stream sequential pattern mining algorithm called SS-BE [16] by Mendes et al. Using a sliding window approach, Hassani and Seidl adapted SS-BE to handle multiple input-streams [17]. This algorithm can only find so called consecutive sequential patterns, though.

III. Preliminaries and Problem Definition

To get an overview about the main concepts of stream process mining, we will introduce them in this section.

In process mining we usually consider static logs of events. Each *event* e consists of at least a case identifier c, an activity label a and a timestamp t. For simplification we will only consider events of this minimal shape $e = (c, a, t)$, but in real-world applications it is possible to add arbitrarily more attributes. For convenience we define projection functions for each attribute: $c(e) = c, a(e) = a, t(e) = t$.

Each event in the log reflects an occurred action in a business process and all events sharing the same case identifier form a sequence of actions that we typically call a *trace*.

While a log can be defined as a multiset of events, we want to deal with a sequence of events. We define the sequence of events $S : \mathbb{N} \to \mathbb{E}$ as an *event stream*. To simplify the notation we use its string representation.

$$S = \langle e_1, e_2, e_3, \ldots \rangle$$

Different to sequential pattern mining, where we aim at finding frequent consecutive subsequences, stream process mining tries to find pairs of events in common cases and collect

their frequencies. This knowledge is then used to infer a model, which allows for interpretation of causalities between different activities. As many cases are usually handled at the same time, it is necessary to observe all of them in parallel to find these pairs.

Therefore it can be useful to differentiate between *open* and already *closed cases* during the runtime of a stream. There is of course also the third option, that a particular case has not been seen until now. Let e_k be the most recently observed item in the stream and c a case such that there was an event e_j with $0 \leq j \leq k$ and $c(e_j) = c$. We call c an open case, if we expect to observe an event e_m with $m > k$ and $c(e_m) = c$. If we can guarantee that we will not see another occurrence of c, we call c closed. It should be noted that it is usually possible to introduce start and end activities to the event universe. This helps to safely close an open case after we observed an event which is marked with an end-activity. On the other hand there are a lot of reasons why the detection of the case closure can fail due to technical problems or faulty human interaction. If we only rely on observing end markings, we will unnecessarily keep cases open which will finally congest the memory. To deal with this problem it is a good idea to forget cases which have not been seen for some time.

Given such an event stream we want to find the underlying process model by collecting the traces as process instances and transform them into a human-comprehensible visualization. The Heuristics Miner [4] is a well-established algorithm for transforming an event log into a graph model and allows for a good quality in terms of being able to replay many traces from the log while not allowing too much additional behavior and staying human-interpretable.
The Heuristics Miner discovers the control-flow model of a process by connecting activities which fulfill a set of constraints representing their degree of dependency. So for two activities a and b, Heuristics Miner will only establish the connection from a to b, if ab is a frequent pattern within the event log. The existence of a subsequence of two activities a and b in a certain case is typically called the *direct-follows relation* and denoted by $a > b$. By $|a > b|$ we state the number of direct-follows occurrences seen in the current batch. To reduce the size of the model, Heuristics Miner prunes the model by ignoring those connections with a support below a minimum support threshold. This threshold is user-defined and denoted by τ_{PO}. The following constraint is used to build an initial candidate set for edges in the result model.

$$|a > b| \geq \tau_{PO} \tag{1}$$

The Heuristics Miner additionally computes a *dependency measure* for the pair (a, b), such that a connection from a to b will only be established if b is dependent of a. In the literature, the dependency measure is denoted by

$$a \Rightarrow b = \frac{|a > b| - |b > a|}{|a > b| + |b > a| + 1} \in [-1, 1]. \tag{2}$$

If $a \Rightarrow b$ is close to 1, then there is a strong tendency that a is usually followed by b. If it is close to -1, then there is a high chance to observe an a after a b. In case of a value nearly 0, there is no significant causality between both activities. As with the positive observation threshold, the user has to specify which degree of dependency τ_{dep} is reasonable to declare two activities as dependent. This means that the following condition has to be fulfilled:

$$a \Rightarrow b \geq \tau_{dep} \tag{3}$$

To keep the model simple and comprehensible, the Heuristics Miner eliminates connections between a and b if there is another successor $post_{best}(a) = \text{argmax}_{b'}\, a \Rightarrow b'$, such that the difference between $a \Rightarrow b$ and $a \Rightarrow post_{best}(a)$ exceeds a third user-defined threshold τ_{best}. Analogously by defining $pre_{best}(b) = \text{argmax}_{a'}\, a' \Rightarrow b$, the Heuristics Miner checks all predecessors of b to compare $a \Rightarrow b$ with $pre_{best}(b) \Rightarrow b$. Formally all dependency connections are allowed which are accepted by these two constraints:

$$|(a \Rightarrow post_{best}(a)) - (a \Rightarrow b)| < \tau_{best} \tag{4}$$

$$|(pre_{best}(b) \Rightarrow b) - (a \Rightarrow b)| < \tau_{best} \tag{5}$$

This gives connections between activities with high dependency a huge advantage. A found relation can be frequent and can also have a high dependency measure. For many relations which fulfill these requirements, Heuristics Miner only keeps the top dependent ones. A model shows only the relations with strongest dependencies, which should be the most important ones for the process.

Similar to the dependency measure, the Heuristics Miner introduces a measure for the discriminability of the splits and joins.

$$a \Rightarrow (b \wedge c) = \frac{|b > c| + |c > b|}{|a > b| + |a > c| + 1} \in [0, 1] \tag{6}$$

$$(b \wedge c) \Rightarrow a = \frac{|b > c| + |c > b|}{|b > a| + |c > a| + 1} \in [0, 1] \tag{7}$$

By using the user-defined AND-threshold, it defines two relations (a, b) and (a, c) (analogously (b, a) and (c, a)) as an AND-split (join), if

$$a \Rightarrow (b \wedge c) \geq \tau_{AND}. \tag{8}$$

On the contrary, if

$$a \Rightarrow (b \wedge c) < \tau_{AND}. \tag{9}$$

holds, then we mark this split (join) with XOR.

As the Heuristics Miner uses these constraints to determine the existence of a causal connection between two activities, it is sufficient to know the frequencies of all existing relations to infer the different measures and therefore infer a causal net. This is a rather important detail as it allows to transform the problem of mining a model into mining for sequential patterns of length 2 per case.

In [8] the introduced approaches share the idea of collecting items from a stream and then use the Heuristics Miner on this static data. A very well performing algorithm in this work is the Heuristics Miner with Lossy Counting algorithm (LC) and its improvement, the Heuristics Miner with Lossy Counting with Budget algorithm (LCB). Both approaches utilize three data structures to collect information about the activities, relations and open cases. Inspired by the lossy

counting method from the field of mining frequent items from streams, it suggests to mine activity counts and to extend the lossy counting approach to also count relations. It has to use the third structure for cases to allow for mining relations within the same case.

In the following we want to adapt a method of another mining field as well. The SS-BE method [16] itself is also inspired by the Lossy Counting algorithm but mines sequential patterns in streams. It uses a prefix-tree to store each sequence and its frequency. After a predefined period of observed items, the tree will be pruned generating a bounded error while reducing its size to keep all the frequent sequences only. The pruning cuts branches with a support below a minimum support threshold.

IV. *StrProM:* A Stream Process mining algorithm

In [8] several approaches are presented to collect counts of activities and case-related direct-follows relations by observing an event stream and to use this data as an input for the Heuristics Miner algorithm. The miner algorithm will then infer a control-flow model in form of a causal net as it would do with static data. Considering the introduced approaches we have selected the Lossy Counting with Budget algorithm since it is most suitable for comparing with our novel approach considering its very fast event processing time. LCB utilizes three data structures for activities, relations and open cases. All of them have to be updated after a new stream event appeared. Our approach aims at reducing the process time by decreasing the effort for updating three structures and just uses one data structure. This allows a much faster process time but demands a more extensive processing of this structure to yield the counts on activities and relations. Since we usually do not need this information as often as the stream progresses with new events, we can delay this step to decrease average event process times significantly.

A. Indexed Prefix-Trees

Inspired by the SS-BE method [16], we imported a subsequence mining method to the field of stream process mining. This algorithm uses lexicographic trees to store subsequences of items. We adapted this idea in the sense that a currently open case is just a subsequence of the finally closed ones. So while we are collecting events of a case by observing the stream, we fill a prefix-tree T_{traces} with new arriving items. This refers to Lines 11-26 in Algorithm 1. For each open case we store a pointer in a map structure D_{cases} to the corresponding node in the tree (Lines 11-12) for a fast access when we observe a new event for this case. If a case is closed, we can delete the pointer and still keep the collected information in the tree. The usage of case pointers leads to a noticeable decrease of event processing time, as for each arriving stream event, the algorithm only has to insert it at the correct position in the tree and update the pointer. The updates of the relation and activity frequency lists (F_A and F_R) do not need to be done every time a new item is observed.

Consider a small example of only three cases here. Let us assume the following traces: $c_1 = (ab)$, $c_2 = (bc)$ and $c_3 = (bab)$. Imagine that we observe all events in the following order defining a stream S:

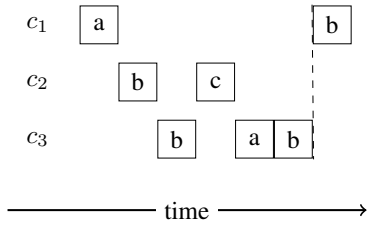

Fig. 2. Example of an event stream S over three cases $c_1 = ab, c_2 = bc$ and $c_3 = bab$. The dashed line indicates that we have observed only 6 of 7 events yet.

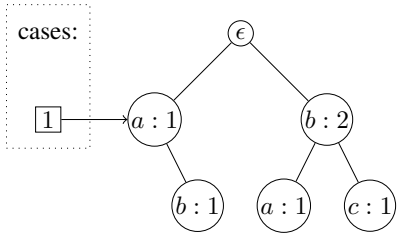

Fig. 3. Example of a prefix-tree after 6 of 7 events have been observed.

$$S = \langle (1, a, t_0)(2, b, t_1)(3, b, t_3)(2, c, t_4) \\ (3, a, t_5)(3, b, t_6)(1, b, t_7) \rangle$$

For a more illustrative understanding of the stream and its temporal ordering, take a look at Fig. 2. After six events have been observed, the *prefix-tree* in Fig. 3 contains the whole information we have collected so far. For the cases c_2 and c_3 we observed their last activities, so these two cases are closed. As only one item of case c_1 is remaining, we end up with this case still open. The corresponding case pointer is still active in the cases list.

The tree will grow continuously. We have to keep in mind that considering stream processing typically means a huge amount of data, so we will finally store more items than our capacity allows. As a theoretical boundary, the maximum node degree is limited by the number of different activities. The height of the tree is bounded by the case length. For instance we take a number of 20 activities with an average case length of 20. This would finally end up in a tree consisting of roughly 10^{25} nodes or 10^{13} terabytes of data. Depending on the application we could easily have more different activities or longer process instances, which will increase the huge amount of data even more.

Even if we are able to store all the data, the time needed to infer a model grows with the size of the tree. On the other hand, a complete prefix-tree of observed events offers far more information than we actually need for a heuristics net. Knowing that we only need counts on direct-follows instead of the positional information of each relation leads to the following pruning strategy.

B. Tree Pruning

We again take a look at the SS-BE [16] approach. After a certain period, it applies a pruning to the tree eliminating

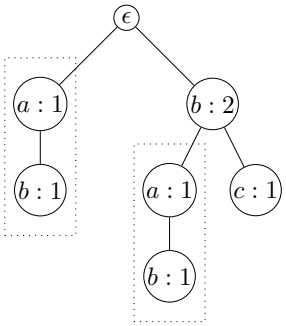

Fig. 4. Prefix-tree with redundancy after processing the complete stream S.

branches with a support below a particular minimum support threshold. Considering our aims of mining the process, we are also interested in branches with a low support. Many 2-patterns in the depth of the tree with a low individual observation frequency can add up to large numbers. These information make a significant difference in the resulting model. On the other hand, and in contrary to SS-BE, we do not need to keep all the structural information of the tree. Regarding the process mining task we have a lot of redundant information which can also be stored in a more compressed structure. For instance, consider Fig. 4, where the stream has been observed completely. The pattern ab occurs two times in the tree. For our needs it is sufficient to know that $|a > b| = 2$.

So instead of pruning branches of the tree we will do almost the inverse of the SS-BE algorithm. We store the information given by the tree in two frequency maps for activities $F_A : A \to \mathbb{N}$ and relations $F_R : (A \times A) \to \mathbb{N}$. To collect all counts, we have to traverse the tree only once. For each node n with label a and its child m with label b, we retrieve the value stored in m. This value is then added to the relation frequency list for $a > b$. In addition, we can sum up the activity frequencies in the same run. This is done in Algorithm 1, Lines 32-42.

After we have extracted all the tree information, we can remove most of the tree. For each open case, we store the last activity and attach it directly to the root node. The case pointers for the fast access will point to these nodes on the second tree level. They are initialized with a value of 0 as we already transferred the old values into the frequency lists before. After the tree pruning has been finished, the algorithm continues as before with a tree of height 1 and we fill it again. In Fig. 5 we used the pruning on our example. Using a period length of 5 and starting at the beginning, we have to prune after we observe the a in case c_3. First, we extract the contained information:

$$F_A = \{a : 2, b : 2, c : 1\}$$
$$F_R = \{(b, a) : 1, (b, c) : 1\}$$

The new tree only consists of the root and one node labeled with a as both open cases c_1 and c_3 have shown an a most recently. Case c_2 is already closed and therefore does not need a pointer.

After a pruning has been done, we have collected the activity and relation frequencies. These are additive and the old data can just be updated. A causal net can then be constructed using the Heuristics Miner constraints. The pruning period and the interval between model inferring are not depending on each other and we do not have to construct a model after each pruning period.

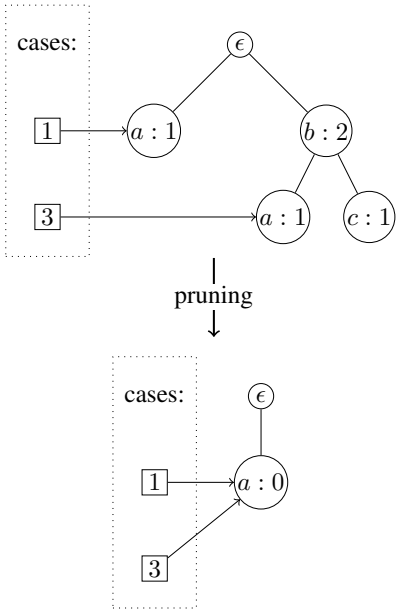

Fig. 5. Tree pruning at $t = 5$ while observing S with pruning period of 5.

C. Decaying

Non-stationary processes obviously need different models during their existence. Information in the stream of events get less important the older this data becomes. To grant more importance to newer process instances or cases, it is useful to forget older relations and activities. When we prune the tree and add the newly collected counts on activities and relations, we modify the old counts using a decaying factor $\lambda \in [0, 1]$. This is done at Algorithm 1, Lines 36-37.

$$F_A^{new} = (1 - \lambda)F_A^{old} + F_A^{extracted}$$
$$F_R^{new} = (1 - \lambda)F_R^{old} + F_R^{extracted}$$

Choosing $\lambda = 1$ the algorithm will ignore all data collected in the previous batches, allowing for an exclusive focus on the recent data. If we do not want any decaying at all, we can choose $\lambda = 0$ such that all data is treated with the same importance. This is for example useful for the processing of a static but very large database as an event stream.

D. Case Pruning

A last thing to consider is keeping the number of observed concurrent cases limited. In the worst case there can be a point in time when all cases are open simultaneously. In real-world applications this case is very unlikely, although in the case of an error-prone stream in a technical sense or a comparable influence there can be missing end events. But the number of simultaneously open cases tends to grow very large for many real-world datasets, while the number of activities and relations is usually very small. This leads to the fact that monitoring the open cases is the dominating factor considering the memory consumption.

Algorithm 1 The *StrProM* Algorithm

Input: S: event stream;
1: $\lambda \in [0, 1]$: decay factor;
2: p: pruning period;
3: c_{max}: case observation limit

4: Initialize the data structure T_{traces}, D_{cases}
5: Initialize frequency maps F_A and F_R

6: **loop**
7: /* pruning period p */
8: **for** $0, \ldots, p$ **do**
9: $(c, a, t) \leftarrow observe(S)$

10: /* Update T_{traces} with observed event */
11: **if** $\exists c \in D_{cases}$ **then**
12: $(n, t) \leftarrow D_{cases}(c)$
13: **if** $\exists T_{traces}(n.a)$ **then**
14: $T_{traces}(n.a) \leftarrow T_{traces}(n.a) + 1$
15: **else**
16: $T_{traces}(n.a) \leftarrow 1$
17: **end if**
18: $D_{cases}(c) \leftarrow (n.a, t)$
19: **else**
20: **if** $\exists T_{traces}(a)$ **then**
21: $T_{traces}(a) \leftarrow T_{traces}(a) + 1$
22: **else**
23: $T_{traces}(a) \leftarrow 1$
24: **end if**
25: $D_{cases}(c) \leftarrow (a, t)$
26: **end if**

27: /* remove least recent cases from observation */
28: **if** D_{cases} reach size of c_{max} **then**
29: delete oldest cases from observation
30: **end if**
31: **end for**

32: /* after pruning period, collect data and prune tree */
33: **for all** $n.a, n.a.b \in T_{traces}$ **do**
34: /* collect pairs of nodes and their children first
35: consider decay factor d here */
36: $F_A(b) \leftarrow (1 - \lambda)F_A(b) + T_{traces}(n.a.b)$
37: $F_R(a, b) \leftarrow (1 - \lambda)F_R(a, b) + T_{traces}(n.a.b)$
38: **end for**
39: **for all** $a \in T_{traces}$ **do**
40: /* children of root are still missing */
41: $F_A(a) \leftarrow (1 - \lambda)F_A(a) + T_{traces}(a)$
42: **end for**

43: /* create new tree for next period */
44: initialize T^{new}_{traces}
45: **for all** $c \in D_{cases}$ **do**
46: /* update open case pointers */
47: $(n.a, t) \leftarrow D_{cases}(c)$
48: $T^{new}_{traces}(a) \leftarrow 0$
49: $D_{cases}(c) = (a, t)$
50: **end for**
51: /* next period's tree has height 1 */
52: $T_{traces} \leftarrow T^{new}_{traces}$
53: **end loop**

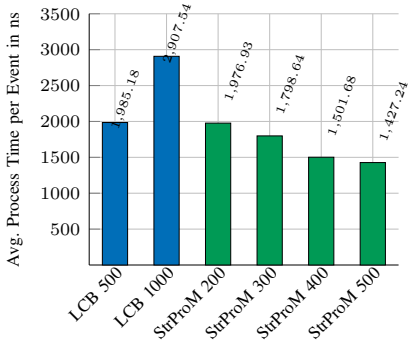

Fig. 6. Comparison of average process times per event for the Lossy Counting with Budget ($b = 500$ and $b = 1000$) approach and our StrProM algorithm. We used a decaying factor of $d = 0.1$ and a case observation limit of 800. The pruning period ranges from 200 to 500 events per period.

In the case of an overflow of monitored cases, which means for the algorithm that we store too many case pointers, we have to remove some of them. Therefore, we chose to collect the timestamp of the last event for each open case. When deciding which cases to remove from the observation list, we will choose the ones not seen for a long time (see Algorithm 1, Lines 28-29). It is important, with regards to the performance, not to restrain the number of tracked cases too much. If the algorithm chooses to few items, we will only achieve a very small buffer for new cases, resulting in another overflow thereafter. The case pruning will add a significant amount to the average process time if it is performed very frequently.
After we determined the mean of all collected timestamps, we retain only those cases with a timestamp above the mean value. The remaining cases are considered as obsolete and are removed. If one of the removed cases is observed again after its deletion, it will be considered as a new case.

V. Experimental Evaluation

We used a real-world dataset to evaluate our method against a state-of-the-art competitor. The dataset is called Road Traffic Fine Management Process and is publicly available[1]. As the title suggests it is a log of a system that manages road traffic fines. It contains 561470 events distributed amongst 150370 cases. Cases vary in their lengths between 2 and 20 events.

With this dataset we performed time measurements per stream event, counted the stored items and used the fitness and precision measures. We have selected the Heuristics Miner with Lossy Counting with Budget algorithm (LCB) [8] as the state-of-the-art competitor. The time needed to infer a model is not contained in the measurements because the frequency of a demand for a model update can differ very much and the construction of a new model would easily become the dominating factor here. As both approaches use similar model inferring methods, we only consider the time to process an incoming stream event and update the data structures accordingly. For measuring the size of used memory space, we put our focus on the number of observed cases and the size of the prefix-tree

[1]The dataset can be downloaded from
http://data.3tu.nl/repository/uuid:270fd440-1057-4fb9-89a9-b699b47990f5

TABLE I. PROCESS TIMES PER EVENT

Algorithm	Time	Max. Stored Items
LCB (b=500)	1985.18 ns	493 (cases)
LCB (b=1000)	2907.54 ns	993 (cases)
StrProM (c=800, tree=200)	1976.93 ns	800 (cases) + 16 (tree)
StrProM (c=800, tree=300)	1798.64 ns	800 (cases) + 18 (tree)
StrProM (c=800, tree=400)	1501.68 ns	800 (cases) + 20 (tree)
StrProM (c=800, tree=500)	1427.24 ns	800 (cases) + 24 (tree)

here. The number of activities and relations is typically very small compared to the number of concurrent cases. We tried to match the numbers of maximally stored items to compare processing times. For the quality measures, a high fitness value near 1 means that many cases found in the original log or in the stream can be replayed in the resulting model. A high precision value on the other hand, implies that there are almost no instances created by the resulting model that can not be found in the source log. In Fig. 6 we illustrated the averaged processing times per event of both algorithms. The processing times for LCB are lower when decreasing the budget size. Reducing the budget increases the time performance as there are more cleanups and therefore smaller data structures to update for each event arrival. But as already stated in [8] a lower budget is usually traded off against quality. However, when using several pruning period lengths from 200 to 500 items, the StrProM was able to process stream events in less than 2 milliseconds or > 500 events per second (see also Table I). StrProM shows a better efficiency than LCB even with higher pruning frequencies (after each 200 events). This shows that our approach is appropriate for real-world applications considering high throughput of new events.

Considering the number of stored items we reduced in our experiment the number of observed cases to 800 instead of the default 1000 cases for the LCB algorithm. This allows a buffer of 200 items for the prefix-tree. A pruning period of 500 events gives a boundary for a maximum tree size of the same value, because for each event we can only add one node or update an existing one. As the experiments have shown (Fig. 7), the tree always grows very slowly and its size never touches a value close to the pruning period length. A high concurrency of cases and much repetition in trace patterns leads to slow growth as there will be only short paths in the tree. See also the second column of Table I for the maximum number of stored items. For the StrProM approach, we listed in Table I the numbers of cases and tree nodes. The difference of the number of cases and tree nodes is very significant. After the next pruning period starts, we will again start with a tree of height 1, so the case observation list still dominates the memory usage similar to the situation in LCB.

Finally we want to talk about the quality of the resulting model. We are not able to compare it to the original process as we only have the log file without any ground truth model. However, we compared it to LCB using the same procedure already used in [8]. As Fig. 8 and Fig. 9 illustrate, StrProM yields a model with nearly the same quality that LCB can produce. This is to be expected as the algorithms use similar constraints for building their model. One important difference between their model building procedures is the step of collecting activity and relation counts. Although the collected frequency statistics will be different, the experiments show

TABLE II. AVERAGE FITNESS AND PRECISION

Algorithm	Avg. Fitness	Avg. Precision
LCB (b=1000)	0.636	0.898
StrProM (c=800, tree=300)	0.602	0.918
StrProM (c=800, tree=500)	0.621	0.941

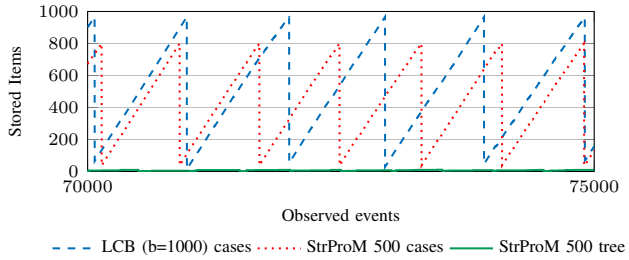

Fig. 7. Stored Items over a period of 5000 events. Considered are the number of observed open cases and in case of StrProM the size of the prefix-tree. Budget is 1000 and pruning period is 500 with a decaying factor of 0.1.

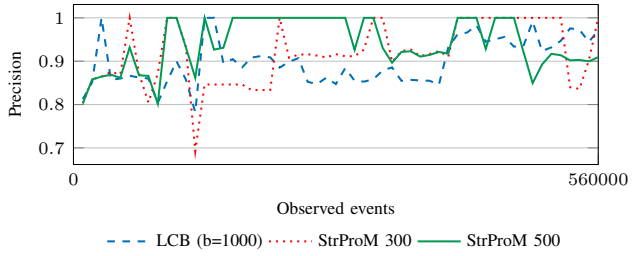

Fig. 8. The precision quality measure of LCB and two runs of StrProM, using pruning periods of 300 and 500 items. Measures are taken over the whole stream with intervals of 10000 items.

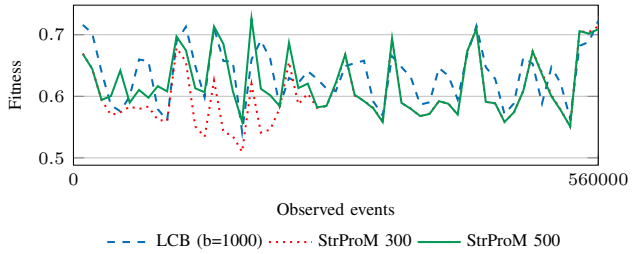

Fig. 9. The fitness quality measure of LCB and two runs of StrProM, using pruning periods of 300 and 500 items. Measures are taken over the whole stream with intervals of 10000 items.

that we do not lose quality while having a considerably better efficiency. The average fitness and precision values are very similar as it can be seen in Table II. It should be noted that StrProM tends to yield a model with a better precision and a slightly worse fitness for this dataset. This effect has to be further analyzed and clarified using more datasets in a future work.

VI. CONCLUSION AND FUTURE DIRECTIONS

In this paper, we introduced an efficient approach for exploring and counting process fragments from a stream of events to infer a process model by using the Heuristics Miner algorithm. Our novel introduced approach, called StrProM,

builds prefix-trees to extract sequential patterns of events from the stream. StrProM uses a batch-based approach to continuously update and prune these prefix-trees. The final models are generated from those trees after applying a decaying mechanism over their statistics. The Lossy Counting with Budget algorithm is used as a competitor, which is a very well performing algorithm in terms of execution time, space requirements, and quality. We showed the efficiency improvements in terms of execution time while keeping the space requirements nearly equal to that of the LCB algorithm, by evaluating both methods against a large dataset.

In the future, we plan to change the decaying factor and the pruning period to become dynamically adaptive to changes in the underlying event stream termed *concept drift*. E.g. [10] applied a statistical hypothesis testing to detect concept drifts in the process. Furthermore, choosing a good length value for the pruning period is not trivial. Small values increase the average processing times per event, while keeping the frequency maps up-to-date, but negatively affects the accuracy of the extracted model. We additionally want to address the more complicated and realistic scenario where interval-based events are considered. In the case of overlapping, more relations are considered between these temporal events (Allen relationships) [31]. In [32], an idea of a Heuristics Miner for interval-based events was discussed. Recent approaches for finding sequential patterns in a stream of such kinds of events were presented in [33], [34].

REFERENCES

[1] W. M. van der Aalst, *Process mining: discovery, conformance and enhancement of business processes*. Springer Science & Business Media, 2011.

[2] T. A. Weijters and W. M. van der Aalst, "Process mining: discovering workflow models from event-based data," in *13th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2001)*. Citeseer, 2001.

[3] W. M. van der Aalst, "The application of petri nets to workflow management," *Journal of Circuits, Systems and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

[4] T. A. Weijters, W. M. van der Aalst, and A. A. De Medeiros, "Process mining with the heuristics miner-algorithm," *Technische Universiteit Eindhoven, Tech. Rep. WP*, vol. 166, pp. 1–34, 2006.

[5] W. M. van der Aalst, T. A. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *Knowledge and Data Engineering, IEEE Trans. on*, vol. 16, no. 9, pp. 1128–1142, 2004.

[6] B. van Dongen and W. M. van der Aalst, "Multi-phase process mining: Building instance graphs," in *Conceptual Mod.–ER*, 2004, pp. 362–376.

[7] C. Günther and W. Van Der Aalst, "Fuzzy mining–adaptive process simplification based on multi-perspective metrics," in *Business Process Management*. Springer, 2007, pp. 328–343.

[8] A. Burattin, A. Sperduti, and W. M. van der Aalst, "Control-flow discovery from event streams," in *Congress on Evolutionary Computation (IEEE WCCI CEC)*, 2014.

[9] M. Hassani, "Efficient Clustering of Big Data Streams," Ph.D. dissertation, RWTH Aachen University, 2015.

[10] R. J. C. Bose, W. M. van der Aalst, I. Žliobaitė, and M. Pechenizkiy, "Handling concept drift in process mining," in *Advanced Information Systems Engineering*. Springer, 2011, pp. 391–405.

[11] M. Hassani, P. Spaus, M. M. Gaber, and T. Seidl, "Density-based projected clustering of data streams," in *Proc. of the 6th Intl. Conf. on Scalable Uncertainty Management*, ser. SUM '12, 2012, pp. 311–324.

[12] M. Hassani, P. Spaus, and T. Seidl, "Adaptive multiple-resolution stream clustering," in *Proc. of the Machine Learning and Data Mining in Pattern Recognition - Intl. Conf.*, ser. MLDM '14, 2014, pp. 134–148.

[13] M. Hassani, P. Spaus, A. Cuzzocrea, and T. Seidl, "Adaptive stream clustering using incremental graph maintenance," in *BigMine 2015 at KDD'15*, 2015, pp. 49–64.

[14] P. Kranen, S. Guñnemann, S. Fries, and T. Seidl, "MC-tree: Improving bayesian anytime classification," in *Proc. of the 22nd Scientific and Statistical Database Management*, ser. SSDBM '10, 2010, pp. 252–269.

[15] M. Hassani, C. Beecks, D. Töws, T. Serbina, M. Haberstroh, P. Niemietz, S. Jeschke, S. Neumann, and T. Seidl, "Sequential pattern mining of multimodal streams in the humanities," in *BTW*, 2015, pp. 683–686.

[16] L. Mendes, B. Ding, and J. Han, "Stream sequential pattern mining with precise error bounds," in *ICDM '08*, 2008, pp. 941 –946.

[17] M. Hassani and T. Seidl, "Towards a mobile health context prediction: Sequential pattern mining in multiple streams," in *Mobile Data Management (MDM), 12th IEEE Int. Conf. on*, vol. 2, 2011, pp. 55–57.

[18] J. Cook and A. Wolf, "Discovering models of software processes from event-based data," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 3, pp. 215–249, 1998.

[19] R. Agrawal, D. Gunopulos, and F. Leymann, "Mining process models from workflow logs," in *Proceedings of the 6th Int. Conf. on EDBT: Advances in Database Technology*, 1998, pp. 469–483.

[20] T. A. Weijters and W. M. van der Aalst, "Rediscovering workflow models from event-based data using little thumb." *Integrated Computer-Aided Engineering*, vol. 10, no. 2, pp. 151–162, 2003.

[21] B. van Dongen, A. K. de Medeiros, E. Verbeek, T. Weijters, and W. Van Der Aalst, "The prom framework: A new era in process mining tool support," in *App. and Th. of Petri Nets*, 2005, pp. 444–454.

[22] W. M. van der Aalst, B. F. van Dongen, C. W. Günther, R. Mans, A. A. De Medeiros, A. Rozinat, V. Rubin, M. Song, H. Verbeek, and A. Weijters, "Prom 4.0: comprehensive support for real process analysis," in *Petri Nets and Other Models of Concurrency–ICATPN*, 2007, pp. 484–494.

[23] W. M. van der Aalst, A. Adriansyah, A. K. A. de Medeiros, F. Arcieri, T. Baier, T. Blickle, J. C. Bose, P. van den Brand, R. Brandtjen, J. Buijs *et al.*, "Process mining manifesto," in *Business process management workshops*. Springer Berlin Heidelberg, 2012, pp. 169–194.

[24] A. Burattin, A. Sperduti, and W. M. van der Aalst, "Heuristics miners for streaming event data," *arXiv:1212.6383*, 2012.

[25] D. Redlich, T. Molka, W. Gilani, G. S. Blair, and A. Rashid, "Scalable dynamic business process discovery with the constructs competition miner," in *Proceedings of the 4th Int. Sym. on Data-driven Process Discovery and Analysis*, ser. SIMPDA, 2014, pp. 91–107.

[26] D. Redlich, T. Molka, W. Gilani, G. Blair, and A. Rashid, "Constructs competition miner: Process control-flow discovery of bp-domain constructs," in *Business Process Management*, 2014, pp. 134–150.

[27] M. Pesic, H. Schonenberg, and W. M. Van der Aalst, "Declare: Full support for loosely-structured processes," in *Enterprise Distributed Object Computing Conf., 11th IEEE Int.*, 2007, pp. 287–287.

[28] F. M. Maggi, A. Burattin, M. Cimitile, and A. Sperduti, "Online process discovery to detect concept drifts in ltl-based declarative process models," in *On the Move to Meaningful Internet Systems (OTM) Conferences*, 2013, pp. 94–111.

[29] A. Burattin, M. Cimitile, F. Maggi, and A. Sperduti, "Online discovery of declarative process models from event streams," *IEEE Transactions on Services Computing*, vol. Early Access Article, 2015.

[30] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, "Mining sequential patterns by pattern-growth: The prefixspan approach," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 16, pp. 1424–1440, 2004.

[31] J. F. Allen, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[32] A. Burattin and A. Sperduti, "Heuristics miner for time intervals," in *European Sym. on Artificial Neural Networks (ESANN)*, 2010.

[33] Y.-C. Chen, C.-C. Chen, W.-C. Peng, and W.-C. Lee, "Mining correlation patterns among appliances in smart home environment," in *Adv. in Knowledge Discovery and Data Mining*, 2014, pp. 222–233.

[34] D. Töws, M. Hassani, C. Beecks, and T. Seidl, "Optimizing sequential pattern mining within multiple streams," in *BTW Workshops*, 2015, pp. 223–232.