Online Process Discovery to Detect Concept Drifts in LTL-Based Declarative Process Models

Fabrizio Maria Maggi[1], Andrea Burattin[2], Marta Cimitile[3], and
Alessandro Sperduti[2]

[1] University of Tartu, Estonia
f.m.maggi@ut.ee
[2] University of Padova, Italy
{burattin,sperduti}@math.unipd.it
[3] Unitelma Sapienza University, Italy
marta.cimitile@unitelma.it

Abstract. Today's business processes are often controlled and supported by information systems. These systems record real-time information about business processes during their executions. This enables the analysis at runtime of the process behavior. However, many modern systems produce "big data", i.e., collections of data sets so large and complex that it becomes impossible to store and process all of them. Moreover, few processes are in steady-state and due to changing circumstances processes evolve and systems need to adapt continuously. In this paper, we present a novel framework for the discovery of LTL-based declarative process models from streaming event data in settings where it is impossible to store all events over an extended period or where processes evolve while being analyzed. The framework continuously updates a set of valid business constraints based on the events occurred in the event stream. In addition, our approach is able to provide meaningful information about the most significant concept drifts, i.e., changes occurring in a process during its execution. We report about experimental results obtained using logs pertaining the health insurance claims handling in a travel agency.

Keywords: Process Discovery, Event Stream Analysis, Operational Support, Concept Drift, Linear Temporal Logic, Business Constraints, Declare

1 Introduction

Together with conformance checking and process model enhancement, process discovery is one of the three basic forms of process mining [25]. In particular, process discovery aims at producing a process model based on example executions in an event log and without using any a-priori information. When using event logs recorded by information systems supporting changeable and highly dynamic processes (e.g., healthcare processes), traditional process discovery techniques (mainly based on procedural process modeling languages) often produce the so called spaghetti-like models. In these models, too many execution paths are explicitly represented so that they become completely unreadable.

In the last years, several works have been focused on the discovery of declarative process models, i.e., on the discovery of a set of business constraints that hold during the process execution [12, 8, 10, 17, 15, 16]. These techniques are very suitable for processes characterized by high complexity and variability due to the turbulence and the changeability of their execution environments. One of the challenges in process mining listed in the Process Mining Manifesto[4] is *to find a suitable representational bias to visualize the resulting models but also to be used internally when searching for a model*. In this sense, the dichotomy procedural versus declarative can be seen as a guideline when choosing the most suitable language to represent models resulting from process discovery algorithms: process mining techniques based on procedural languages can be used for predictable processes working in stable environments (e.g., a process for handling travel requests), whereas techniques based on declarative languages can be used for unpredictable, variable processes working in turbulent environments (see also [19]).

In this paper, business constraints are represented using the Declare notation [26]. Declare is a declarative language that combines a formal semantics grounded in Linear Temporal Logic (LTL) on finite traces,[5] with a graphical representation. In essence, a Declare model is a collection of LTL rules, each capturing a control flow dependency between two activities.

The Process Mining Manifesto also states that *process mining should not be restricted to off-line analysis and can also be used for online operational support*. In particular, process mining techniques should be able to mine an event stream, i.e., a *real-time, continuous, ordered sequence of items* [13]. Algorithms that are supposed to interact with event streams must respect some requirements, such as: a) it is impossible to store the complete stream; b) backtracking over an event stream is not feasible, so algorithms are required to make only one pass over data; c) it is important to quickly adapt the model to cope with unusual data values; d) the approach must deal with variable system conditions, such as fluctuating stream rates. Currently, only few algorithms are able to mine an event stream [7, 6]. In addition, this is the first paper that presents algorithms for discovering declarative process models based on streaming event data. In particular, our technique is able to show a snapshot of the business constraints valid at a certain point in time during the process execution, thus providing the user with a continuously updated picture of the process behavior.

Another challenge mentioned in the Process Mining Manifesto is related to the fact that *process mining must deal with concept drifts*, i.e., with the situation in which the process is changing while being analyzed. Processes may change due to periodic/seasonal changes (e.g., "in December there is more demand" or "on Friday afternoon there are fewer employees available") or due to changing

[4] The Process Mining Manifesto is authored by the IEEE Task Force on Process Mining (`http://www.win.tue.nl/ieeetfpm/`).

[5] For compactness, we will use the LTL acronym to denote LTL on finite traces.

conditions (e.g., "the market is getting more competitive"). Such changes impact processes and it is vital to detect and analyze them. In this paper, we identify concept drifts in a Declare model in terms of constraint activations. Activations for Declare constraints are defined in [5]. For example, for a business constraint like "every request is eventually acknowledged" each request is an activation. An activation becomes a fulfillment or a violation depending on whether the request is followed by an acknowledgement or not. Concept drifts are discovered by analyzing how the percentages of activations, fulfillments and violations for a set of Declare constraints vary over the time.

To assess the applicability of our approach, we conducted an experimentation with a set of synthetic logs. In particular, we used two variants of a process pertaining the health insurance claims handling in a travel agency.

The paper is structured as follows. Section 2 introduces the characteristics of event stream mining as well as some basic notation about Declare. Next, Section 3 introduces the proposed algorithms for the online discovery of Declare models. Section 4 illustrates the two approaches for stream mining we use in this paper based on sliding window and lossy counting respectively. Section 5 presents and discusses the experimental results. Finally, Section 6 concludes and spells out directions for future work.

2 Preliminaries

In this section, we introduce some preliminary notions. In particular, in Section 2.1, we summarize some basic information about event stream mining needed to understand the paper. In Section 2.2, we give an overview of the Declare language.

2.1 Event Stream Mining

In the data mining literature, there are few definitions of event stream. In this work, we consider an event stream as an unbounded, sequence of data items, observed at a high speed [2, 3]. Stream mining approaches can be divided into two main categories: *data-based* and *task-based* [11]. Data-based mining algorithms reduce the stream into finite datasets, which are supposed to be representatives of the complete stream; task-based algorithms are modified (or new) approaches, specifically designed for streams, in order to minimize time and space complexity. The framework presented here falls into this latter category.

As typically reported in the literature, we assume that: *i*) the data that constitutes the stream (i.e., the *events*) have a small and fixed amount of attributes; *ii*) a mining approach should be able to analyze an infinite amount of data; *iii*) a mining approach is allowed to use only a finite amount of memory and, typically, such amount is considerably smaller with respect to the data observed in a reasonable span of time; *iv*) there is an upper bound on the time allowed to analyze an event, typically the mining approach is required to linearly scale with the number of processed items (e.g., the algorithm works with one pass of

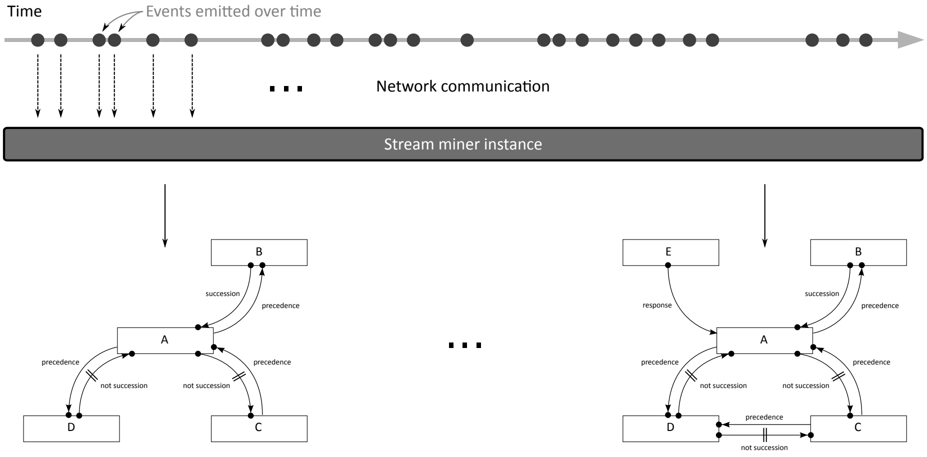


Fig. 1. General idea of event stream mining: the stream miner continuously receives events and, using the latest observations, updates the process model.

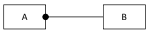
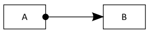
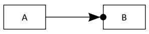
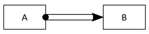
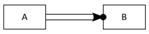
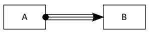
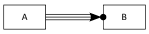
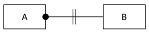
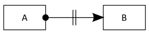
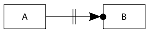
the data [21]); *v*) the "concepts" generating the event stream may be *stationary* or *evolving* [28, 30]. We assume that each event in an event stream is associated with an activity (i.e., a well-defined step in the process), is related to a particular case (i.e., a process instance) and is executed at a certain point in time specified through a timestamp.

In this work, our aim is to reconstruct a declarative model of the process generating the stream, while storing a minimal amount of information. Figure 1 proposes a simple representation of our approach: one or more sources emit events (represented, in the picture, as solid dots), which are collected by our miner instance. The miner elaborates these events and keeps the process model updated. In addition, we want to characterize the portions of the event stream in which the process behavior changes in terms of business constraints. Therefore, it is important to evaluate whether our approach is able to correctly detect concept drifts.

2.2 Declare: Some Basic Notions

Declare is a declarative process modeling language introduced by Pesic and van der Aalst in [26]. A Declare model consists of a set of constraints which, in turn, are based on templates. Templates are abstract entities that define parameterized classes of properties and constraints are their concrete instantiations. Here, we indicate template parameters with capital letters (see Table 1) and real activities in their instantiations with lower case letters (e.g., constraint $\Box(a \rightarrow \Diamond b)$). Templates have a user-friendly graphical representation understandable to the user and their semantics are specified through LTL formulas. Each constraint inherits the graphical representation and semantics from its template. The most frequently used Declare templates are shown in Table 1. However, the language

Table 1. Graphical notation and textual description of some Declare constraints

Template	**Meaning**	**LTL semantics**	**Graphical notation**
responded existence(A,B)	if A occurs then B occurs before or after A	$\Diamond A \rightarrow \Diamond B$	
response(A,B)	if A occurs then eventually B occurs after A	$\Box(A \rightarrow \Diamond B)$	
precedence(A,B)	if B occurs then A occurs before B	$(\neg B \sqcup A) \vee \Box(\neg B)$	
alternate response(A,B)	if A occurs then eventually B occurs after A without other occurrences of A in between	$\Box(A \rightarrow \bigcirc(\neg A \sqcup B))$	
alternate precedence(A,B)	if B occurs then A occurs before B without other occurrences of B in between	$((\neg B \sqcup A) \vee \Box(\neg B))\wedge$ $\Box(B \rightarrow \bigcirc((\neg B \sqcup A) \vee \Box(\neg B)))$	
chain response(A,B)	if A occurs then B occurs in the next position after A	$\Box(A \rightarrow \bigcirc B)$	
chain precedence(A,B)	if B occurs then A occurs in the next position before B	$\Box(\bigcirc B \rightarrow A)$	
not responded existence(A,B)	if A occurs then B cannot occur before or after A	$\Diamond A \rightarrow \neg(\Diamond B)$	
not response(A,B)	if A occurs then B cannot eventually occur after A	$\Box(A \rightarrow \neg(\Diamond B))$	
not precedence(A,B)	if B occurs then A cannot occur before B	$\Box(\Diamond B \rightarrow \neg A)$	

is extensible and new templates can be defined with their own graphical representations and LTL semantics.

Consider, for example, the *response* constraint $\Box(a \rightarrow \Diamond b)$. This constraint indicates that if *a occurs*, b must eventually *follow*. Therefore, this constraint is satisfied for traces such as $\mathbf{t}_1 = \langle a, a, b, c \rangle$, $\mathbf{t}_2 = \langle b, b, c, d \rangle$, and $\mathbf{t}_3 = \langle a, b, c, b \rangle$, but not for $\mathbf{t}_4 = \langle a, b, a, c \rangle$ because, in this case, the second instance of a is not followed by a b. Note that, in $\mathbf{t}_2$, the considered response constraint is satisfied in a trivial way because a never occurs. In this case, we say that the constraint is *vacuously satisfied* [14]. In [5], the authors introduce the notion of *behavioral vacuity detection* according to which a constraint is non-vacuously satisfied in a trace when it is activated in that trace. An *activation* of a constraint in a trace is an event whose occurrence imposes, because of that constraint, some obligations on other events in the same trace. For example, a is an activation for the *response* constraint $\Box(a \rightarrow \Diamond b)$, because the execution of a forces b to be executed eventually.

An activation of a constraint can be a *fulfillment* or a *violation* for that constraint. When a trace is perfectly compliant with respect to a constraint, every activation of the constraint in the trace leads to a fulfillment. Consider, again, the response constraint $\Box(a \rightarrow \Diamond b)$. In trace $\mathbf{t}_1$, the constraint is activated and fulfilled twice, whereas, in trace $\mathbf{t}_3$, the same constraint is activated and fulfilled only once. On the other hand, when a trace is not compliant with respect to a constraint, an activation of the constraint in the trace can lead to a fulfillment but also to a violation (at least one activation leads to a violation). In trace $\mathbf{t}_4$, for example, the response constraint $\Box(a \rightarrow \Diamond b)$ is activated twice, but the first activation leads to a fulfillment (eventually b occurs) and the second activation leads to a violation (b does not occur subsequently). An algorithm to discriminate between fulfillments and violations for a constraint in a trace is presented in [5].

In [5], the authors define two metrics to measure the conformance of an event log with respect to a constraint in terms of violations and fulfillments, called *violation ratio* and *fulfillment ratio* of the constraint in the log. These metrics are valued 0 if the log contains no activations of the considered constraint. Otherwise, they are evaluated as the percentage of violations and fulfillments of the constraint over the total number of activations.

3 Algorithms for the Online Discovery of Declare Models

In this section, we describe the proposed algorithms for the online discovery of Declare constraints. In particular, we will illustrate three different basic algorithms. Each of them can be used to discover constraints referring to different Declare templates. Every algorithm takes as input an event stream and builds a set of candidate constraints obtained by instantiating a Declare template with all the possible combinations of activity names detected so far in the event stream. The algorithms keep up-to-date the fulfillment ratio and the violation ratio of each candidate constraint at every occurrence of an event in the event stream. Then, at each point in time, it is possible to discover a Declare model by selecting, among the candidates, the ones with the highest fulfillment ratio.

In all the algorithms, we use the notion of *map*. Here, a *map* refers to the well known *hash table* data structure [9]. Given a set of keys K and a set of values V, a map is a set $M \subseteq K \times V$. We use the following operators: *(i)* $M.\text{put}(k, v)$, to add value v with key k to M; *(ii)* $M.\text{get}(x) \in V$, with $k \in K$, to retrieve from M the value associated with key k; *(iii)* $M.\text{keys} \subseteq K$ to obtain the set of keys in M.

3.1 Response and Not Response

The algorithm presented in this section can be used for the online discovery of *response* and *not response* constraints. We illustrate the algorithm (Algorithm 1) for the response template. The algorithm is identical for the not response template, since the fulfillment ratio for a response constraint corresponds to the violation ratio of the not response constraint over the same activities (and vice versa).[6] A similar algorithm (with small modifications) is able to discover *alternate response* and *chain response* constraints.

In this algorithm, L is the set of all the activity names observed in the event stream (in all the cases). $activationsCounter_c$ is a map defined for each case c and containing, for each activity name, the number of its occurrences in c. This map can be used to count the number of activations for each constraint (the number of activations can be obtained by counting how many times the

[6] A similar observation also applies to the algorithms described in the following sections.

Algorithm 1: Online algorithm for response and not response.

```
   Input: $e = (c, a, t)$ the event to be processed ($c$ is the case id of the event, $a$ is the activity
          name, $t$ is the timestamp)
1  if $L$ is not defined then define the empty set $L$
2  if $activationsCounter_c$ is not defined then define the map $activationsCounter_c$
3  if $pendingActivations_c$ is not defined then define the map $pendingActivations_c$
4  if $a \notin activationsCounter_c$.keys then
5      $activationsCounter_c$.put$(a, 1)$
6      foreach $l \in L$ do
7          $pendingActivations_c$.put$((l, a), 0)$
8          $pendingActivations_c$.put$((a, l), 1)$
9  else
10     $activationsCounter_c$.put$(a, activationsCounter_c$.get$(a) + 1)$
11     foreach $(k_1, k_2) \in pendingActivations_c$.keys do
12         if $k_2 = a$ then                    /* $a$ equals to the second activity name */
13             $pendingActivations_c$.put$((k_1, a), 0)$
14         else if $k_1 = a$ then               /* $a$ equals to the first activity name */
15             $acts \leftarrow pendingActivations_c$.get$((a, k_2))$
16             $pendingActivations_c$.put$((a, k_2), acts + 1)$
17 if $a \notin L$ then $L \leftarrow L \cup \{a\}$
```

corresponding activity name has occurred in each case of the event stream). $pendingActivations_c$ is a map defined for each case c and containing the number of pending activations in c for each response constraint activated in c (the keys in the map are pairs of activity names representing the constraint parameters).

The algorithm receives as input an event $e = (c, a, t)$, where c is the case id, a is the activity name and t is the timestamp. This event is processed by updating maps $activationsCounter_c$ and $pendingActivations_c$. If c is a new case, these maps are first defined (lines 2-3).

Then, if a has never occurred in c (i.e., if a is not in $activationsCounter_c$), a is added as a key in $activationsCounter_c$ with number of occurrences equal to 1 (a has occurred only once in c). Lines 6-8 of the algorithm are used to update map $pendingActivations_c$ when a occurs in c for the first time. In particular, all the response constraints having a as second parameter cannot have any pending activation when a occurs (all of them are fulfilled when a occurs). Therefore, line 7 of the algorithm sets to 0 the number of pending activations in c for the response constraints having a as second parameter. On the other hand, all the response constraints having a as first parameter are activated when a occurs and waiting for some other event to occur. Then, a is a pending activation for these constraints (and the only one). Therefore, line 8 of the algorithm sets to 1 the number of pending activations in c for the response constraints having a as first parameter.

If a has already occurred in c (i.e., if a is already in $activationsCounter_c$), the number of occurrences of a is incremented by 1 in $activationsCounter_c$. Line 13 of the algorithm sets to 0 the pending activations in c for the response constraints having a as second parameter. On the other hand, all the response constraints having a as first parameter are activated when a occurs and, therefore, the number of pending activations for these constraints in c is incremented by 1.

Algorithm 2: Online algorithm for precedence and not precedence.

```
   Input: $e = (c, a, t)$ the event to be processed ($c$ is the case id of the event, $a$ is the activity
          name, $t$ is the timestamp)
1  if $L$ is not defined then define the empty set $L$
2  if $activationsCounter_c$ is not defined then define the map $activationsCounter_c$
3  if $fulfilledActivations_c$ is not defined then define the map $fulfilledActivations_c$
4  if $a \notin activationsCounter_c$ then
5      $activationsCounter_c$.put($a, 1$)
6      foreach $l \in L$ do
7          if $l \in activationsCounter_c$ then
8              $fulfilledActivations_c$.put($(l, a), 1$)
9              $fulfilledActivations_c$.put($(a, l), 0$)
10         else
11             $fulfilledActivations_c$.put($(l, a), 0$)
12             $fulfilledActivations_c$.put($(a, l), 0$)
13 else
14     $activationsCounter_c$.put($a$, $activationsCounter_c$.get($a$) $+ 1$)
15     foreach $l \in L$ do
16         if $l \in activationsCounter_c$ then
17             $acts \leftarrow fulfilledActivations_c$.get($(l, a)$)
18             $fulfilledActivations_c$.put($(l, a)$, $acts + 1$)
19 if $a \notin L$ then $L \leftarrow L \cup \{a\}$
```

Finally (line 17), if activity name a is observed for the first time in the event stream, a is added to L.

Using the algorithm just described it is possible to count the number of activations for every candidate response constraint (using maps *activationsCounter*$_c$), and the number of violations (counting the number of activations still pending in maps *pendingActivations*$_c$). These metrics are enough for deriving, at any point in time, *fulfillment ratio* and *violation ratio* for (not) response constraints.

3.2 Precedence and Not Precedence

The algorithm presented in this section can be used for the online discovery of *precedence* and *not precedence* constraints. We illustrate the algorithm (Algorithm 2) for the precedence template. A variant of this algorithm can be easily derived for the online discovery of *alternate precedence* and *chain precedence* constraints.

L is again the set of all the activity names occurred in the event stream (in all the cases). *activationsCounter*$_c$ is a map containing, for each activity name, the number of its occurrences in c. *fulfilledActivations*$_c$ is a map defined for each case c and containing the number of fulfilled activations in c, for each precedence constraint activated in c (the keys in the map are pairs of activity names representing the constraint parameters).

The algorithm receives as input an event $e = (c, a, t)$ belonging to a case c. This event is processed by updating maps *activationsCounter*$_c$ and *fulfilledActivations*$_c$. If c is a new case, these maps are first defined (lines 2-3).

If a has never occurred in c, a is added as a key in $activationsCounter_c$ with number of occurrences equal to 1. Lines 6-12 are used to update map $fulfilledActivations_c$ when a occurs in c for the first time. In particular, for each activity l in L, the precedence constraint having a as first parameter and l as second parameter has no fulfillments. Indeed, this constraint is activated when l occurs and is fulfilled if l is preceded by a. This cannot be the case because a has occurred now for the first time. Therefore, lines 9 and 12 of the algorithm sets to 0 the number of fulfilled activations in c for the precedence constraints having l as second parameter. For each activity l in L, the precedence constraints having l as first parameter and a as second parameter are activated when a occurs. In particular, a is a fulfillment if and only if l has already occurred in c. Only in this case, the algorithm sets to 1 the number of fulfilled activations (line 8). Otherwise, the number of fulfilled activations is 0 (line 11).

If a has already occurred in c, the number of occurrences of a is incremented by 1 in $activationsCounter_c$. For each event l in L, the precedence constraints having l as first parameter and a as second parameter are activated when a occurs. Therefore, if l has already occurred in c (and only in this case), the algorithm increments by 1 the number of fulfilled activations in c for the precedence constraints having a as second parameter (line 18). If activity name a has occurred for the first time in the event stream, a is added to L.

Using the algorithm just described it is possible to count the number of activations for every candidate precedence constraint (using maps $activationsCounter_c$), and the number of fulfillments (counting the number of fulfilled activations in maps $fulfilledActivations_c$). With this information it is possible to derive, at any point in time, *fulfillment ratio* and *violation ratio* for (not) precedence constraints.

3.3 Responded Existence and Not Responded Existence

We use Algorithm 3 to discover *responded existence* and *not responded existence* constraints. We illustrate the algorithm for the responded existence template.

As in the algorithms already discussed, also for this algorithm, L is the set of all the activity names occurred in the event stream. $activationsCounter_c$ contains, for each activity name, the number of its occurrences in c. $pendingActivations_c$ contains the number of pending activations in c for each responded existence constraint activated in c.

The algorithm receives as input an event $e = (c, a, t)$ belonging to a case c. This event is processed by updating maps $activationsCounter_c$ and $pendingActivations_c$. If c is a new case, these maps are first defined (lines 2-3).

Then, if activity a has never occurred in c, a is added as a key in $activationsCounter_c$ with number of occurrences equal to 1. Lines 6-11 of the algorithm are used to update map $pendingActivations_c$ when activity a occurs in c for the first time. In particular, all the responded existence constraints having a as second parameter cannot have pending activations (all of them are fulfilled when a occurs). Therefore, line 7 of the algorithm sets to 0 the pending activations in c for the responded existence constraints having a as second parameter. On the

Algorithm 3: Algorithm for responded existence and not resp. existence.

```
   Input: $e = (c, a, t)$ the event to be processed ($c$ is the case id of the event, $a$ is the activity
          name, $t$ is the timestamp)
1  if $L$ is not defined then define the empty set $L$
2  if $activationsCounter_c$ is not defined then define the map $activationsCounter_c$
3  if $pendingActivations_c$ is not defined then define the map $pendingActivations_c$
4  if $a \notin activationsCounter_c$ then
5      $activationsCounter_c$.put$(a, 1)$
6      foreach $l \in L$ do
7          $pendingActivations_c$.put$((l, a), 0)$
8          if $l \in activationsCounter_c$ then
9              $pendingActivations_c$.put$((a, l), 0)$
10         else
11             $pendingActivations_c$.put$((a, l), 1)$
12 else
13     $activationsCounter_c$.put$(a, activationsCounter_c$.get$(a) + 1)$
14     foreach $(k_1, k_2) \in pendingActivations_c$.keys do
15         if $k_2 = a$ then                                  /* $a$ equals to the second event */
16             $pendingActivations_c$.put$((k_1, a), 0)$
17         else if $k_1 = a$ then                              /* $a$ equals to the first event */
18             if $k_2 \in activationsCounter_c$ then
19                 $pendingActivations_c$.get$((a, k_2))$
20             else
21                 $acts \leftarrow pendingActivations_c$.get$((a, k_2))$
22                 $pendingActivations_c$.put$((a, k_2), acts + 1)$
23 if $a \notin L$ then $L \leftarrow L \cup \{a\}$
```

other hand, for each event l in L, the responded existence constraints having a as first parameter and l as second parameter are activated when a occurs. In particular, a is a fulfillment if and only if l has already occurred in c. In this case, the algorithm sets to 0 the number of pending activations in c (line 9). Otherwise, if l has not occurred in c yet, a is a pending activation and the algorithm sets to 1 the number of pending activations (line 11).

If activity name a has already occurred in c, the number of occurrences of a is incremented by 1 in $activationsCounter_c$. Line 16 of the algorithm sets to 0 the pending activations in c for the responded existence constraints having a as second parameter (there are no longer pending activations for these constraints when a occurs). All the responded existence constraints having a as first parameter are activated when a occurs and, therefore, the number of pending activations for these constraints in c is set to 0 if the second parameter has already occurred and is incremented by 1 if the second parameter has not occurred yet. Finally (line 23), if activity name a has occurred for the first time in the event stream, a is added to L.

4 Stream Mining Algorithms

As mentioned earlier in this paper, an event stream is an infinite sequence of events. Due to the infinite amount of cases we need to cope with and due to the finite memory size available, we need approaches to remove less significant cases

Algorithm 4: Stream Mining with Sliding Window.

```
  Input: $S$: event stream; $max_M$: maximum size of the memory
1 Let $M$ be the available memory
2 forever do
3     $e \leftarrow observe(S)$                     /* Observe a new event $e = (c, a, t)$ */
      /* Check if event $e$ has to be used */
4     if $analyze(e)$ then
          /* Memory update */
5         if $size(M) = max_M$ then
6             $shift(M)$
7         $insert(M, e)$
          /* Mining update */
8         if $perform\ mining$ then
9             $DeclareMiner(M)$
```

and keep only the most representative ones. In this work, we use two approaches to deal with infinite streams: sliding window and an adaptation of lossy counting [18]. The first approach is very simple and effective but, as shown in our experimentation, not very efficient; the second approach is more complex but guarantees better performance and, at the same time, good quality of the results. In general, both these approaches give us strategies that guide the decision about which information is not useful anymore and, therefore, can be forgotten.

4.1 Sliding Window

One of the simplest way to tackle the stream mining problem is to collect events for a certain observation period and apply the "off-line version" of a discovery algorithm on the collected data. This idea is presented in Algorithm 4 and the approach is called *sliding window*. An event $e = (c, a, t)$ observed in a stream S (line 3) is analyzed (line 4) to decide whether it will be considered for mining.[7] If the *analyze* function returns true, the algorithm inserts the new event into the memory M (line 7). When M reaches its maximum size (line 5), it is necessary to delete the oldest event (line 6). Periodically (e.g., after the observation of a certain number of events), it is possible to perform the discovery. In our case, we can run the Declare Miner [15], but any other discovery algorithm can be applied.

The main drawback of the sliding window approach is that the time required for processing an event is completely unbalanced and strongly depends on the event processed. In particular, when a new event is observed, most of the times, only inexpensive operations are performed (i.e., $insert(M, e)$). However, when the model must be updated, the log retained in memory is mined from scratch. This implies that, in this case, the event is handled at least twice: the first time to store it in the memory M and the second time by the discovery algorithm.

[7] In general, all the incoming events are analyzed. Only in extreme cases in which it is not possible to store further events in memory, the new events are discarded.

Algorithm 5: Stream Mining with Lossy Counting.

```
Input: $S$ event stream; $\epsilon$ maximum allowed error; $\mathcal{T}$ template.
$N \leftarrow 1$
$w \leftarrow \lceil \frac{1}{\epsilon} \rceil$                                   /* Define the bucket width */
$\mathcal{D}_{\mathcal{T}} = \emptyset$                               /* Main data structure, its type is
    $\mathcal{D}$ : case id $\times$ replayer for template $\mathcal{T}$ $\times$ frequency $\times$ maximum error */
forever do
    $b_{current} = \lceil \frac{N}{w} \rceil$                          /* Define the current bucket id */
    $e \leftarrow observe(S)$                       /* Observe a new event, where $e = (c, a, t)$ */
    /* Update data structure $\mathcal{D}_{\mathcal{T}}$                                              */
    if $\exists$ $(c_{candidate}, r, f, \Delta) \in \mathcal{D}_{\mathcal{T}}$ such that $c_{candidate} = c$ then
        Process event $e$ on replayer $r$
        Update the $(c, r, f, \Delta)$ tuple of $\mathcal{D}_{\mathcal{T}}$, by incrementing $f$ by 1
    else
        $r_{\mathcal{T}} \leftarrow$ new replayer for template $\mathcal{T}$
        $\mathcal{D}_{\mathcal{T}} \leftarrow \mathcal{D}_{\mathcal{T}} \cup \{(c, r_{\mathcal{T}}, 1, b_{current} - 1)\}$
    /* Periodic cleanup                                                      */
    if $N = 0 \mod w$ then
        foreach $(c_{iter}, r, f, \Delta) \in \mathcal{D}_{\mathcal{T}}$ such that $f + \Delta \leq b_{current}$ do
            Remove $(c_{iter}, r, f, \Delta)$ from $\mathcal{D}_{\mathcal{T}}$
    /* Generate model                                                        */
    if model then
        Extract constraints from $\mathcal{D}_{\mathcal{T}}$
    $N \leftarrow N + 1$                                /* Increment the buckets counter */
```

Generally speaking, however, an off-line algorithm may iterate several times on a log. This issue is critical since, in online settings, for performance reasons, it is desirable a procedure that analyze each event no more than once.

4.2 Lossy Counting

One of the strongest approach that can be used to solve the approximate frequency counting problem is called *lossy counting*. In this section, we present a modified version of lossy counting that can be used for the discovery of Declare models. The entire procedure is presented in Algorithm 5.

The basic idea of lossy counting is to divide the stream into ideal buckets, each of them with size $w = \lceil \frac{1}{\epsilon} \rceil$, where $\epsilon \in (0, 1)$ is a parameter indicating the maximum allowed error (0 means that no data is discarded, leading to large space usage; 1 indicates that almost all historical data is discarded, leading to less reliable results). The *current* bucket (i.e., the bucket containing the last event observed) is $b_{current} = \lceil \frac{N}{w} \rceil$, where N is the number of events observed so far.

The most important data structure used by this approach is a set of entries of the form (c, r, f, Δ), where c is the identifier of a case (a case id), r is the replayer associated with a Declare template $\mathcal{T}$ and implementing one of the algorithms illustrated in Section 3, f is the estimated frequency of case c and Δ is the current maximum error. Note that there is a replayer for each case and for each template and each replayer keeps track of the activations, the fulfillments and the violations in that case for that template.

Every time a new element $e = (c, a, t)$ is observed (line 6), the algorithm checks whether the data structure contains an entry for case c (line 7). If such entry exists, then its frequency value f is incremented by one and the replayer processes the new event (lines 8-9). Otherwise a new tuple is added, $(c, r_{\mathcal{T}}, 1, b_{current} - 1)$ (where $r_{\mathcal{T}}$ is a new replayer, for case c) at line 12. Every time $N \equiv 0 \mod w$, the algorithm cleans the data structure by removing the entries that satisfy the inequality $f + \Delta \leq b_{current}$ (line 15). Such condition ensures that, every time the cleanup procedure is executed, $b_{current} \leq \epsilon N$. Periodically (e.g., after the observation of a certain number of events), it is possible to extract the set of valid constraints as described in Section 3 (line 17).

5 Case Study

We implemented the algorithms illustrated in the previous sections as a plug-in of the process mining tool ProM (`http://www.processmining.org`). To carry out our experiments,[8] we have generated two synthetic logs ($\mathcal{L}_1$ and $\mathcal{L}_2$) by modeling two variants of the insurance claim process described in [4] in CPN Tools (`http://cpntools.org`) and by simulating the models. $\mathcal{L}_1$ contains 14,840 events and $\mathcal{L}_2$ contains 16,438 events. We merged the logs (four alternations of $\mathcal{L}_1$ and $\mathcal{L}_2$) using the *Stream Package*, publicly available in the ProM repositories. The same package has been used to transform the resulting log into an event stream.

Starting from the generated stream, we have compared the effectiveness and the efficiency of the online process discovery using the lossy counting approach with respect to the approach based on sliding window. In our experimentation, we discovered a Declare model every 1000 events processed, i.e., we fixed an *evaluation point* every 1000 events.[9]

For evaluating the effectiveness of the two approaches, we have used metrics *precision* and *recall* [20]. To compute recall and precision we assumed that the discovered Declare constraints could be classified into one of four categories, i.e., *i)* true-positive (T_P: correctly discovered); *ii)* false-positive (F_P: incorrectly discovered); *iii)* true-negative (T_N: correctly missed); *iv)* false-negative (F_N: incorrectly missed). Precision and recall are defined as

$$Precision = \frac{T_P}{T_P + F_P} \qquad Recall = \frac{T_P}{T_P + F_N}. \tag{1}$$

The *gold standard* used as reference is the set of all true positive instances. In our experiments, we have used as gold standards two Declare models ($\mathcal{M}_1$ and

[8] All the experiments have been conducted on a machine with an Intel i7 processor (limiting the execution to just one core), 8 GB of RAM and the Oracle Java virtual machine installed on a GNU/Linux Ubuntu operating system.

[9] In all the experiments, we discover (not) response, (not) precedence and (not) responded existence constraints.

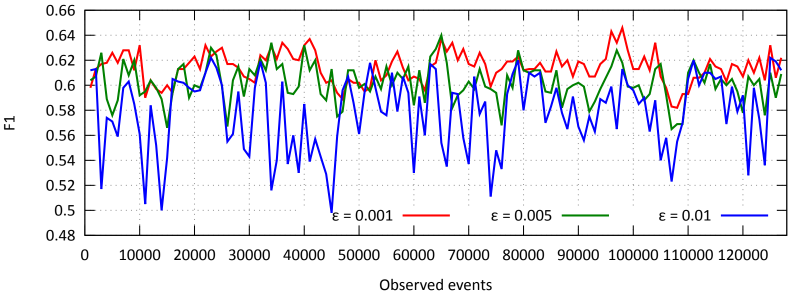


Fig. 2. F_1 trend for the lossy counting approach for different values of the maximum allowed error ϵ.

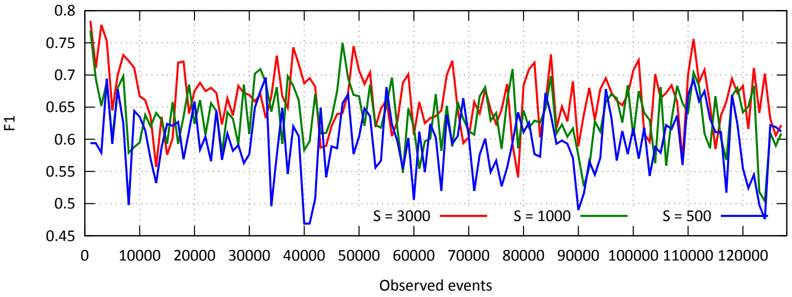


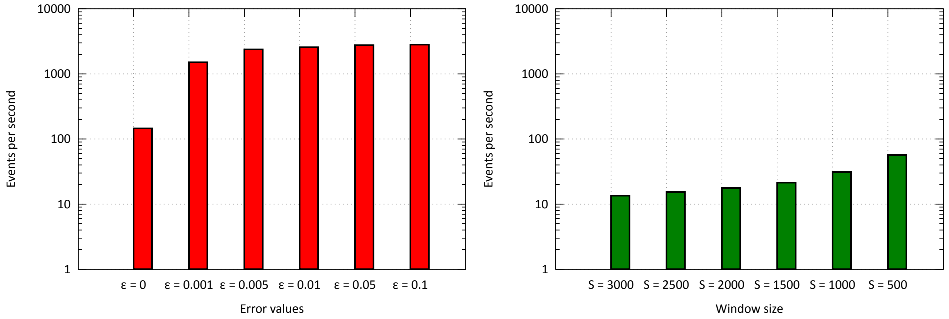
Fig. 3. F_1 trend for the sliding window approach for different values of the window size.

$\mathcal{M}_2$) discovered from $\mathcal{L}_1$ and $\mathcal{L}_2$ using the Declare Miner (available in ProM) and containing constraints satisfied in all the cases. Precision and recall have been evaluated in every evaluation point. To compute them, we have used either $\mathcal{M}_1$ or $\mathcal{M}_2$ as gold standard based on whether the evaluation point corresponds to an event belonging to $\mathcal{L}_1$ or $\mathcal{L}_2$ respectively. In every evaluation point, we selected the constraints with fulfillment ratio equal to 1 among the candidate constraints generated by the lossy counting approach and discovered (with the Declare Miner) the constraints with support 100% in the sliding window approach. Then, we compared these sets of constraints with the gold standards. A discovered constraint is classified as true-positive or false-positive depending on whether it belongs to the gold standard or not. A constraint that belongs to the gold standard but that has not been discovered is a false-negative.

In Fig. 2, we show the trend of the harmonic mean of precision and recall

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \qquad (2)$$

for the lossy counting approach. The plot shows that the quality of the discovered models in every evaluation point is sufficiently high with respect to the gold standards.



(a) Time efficiency for lossy counting

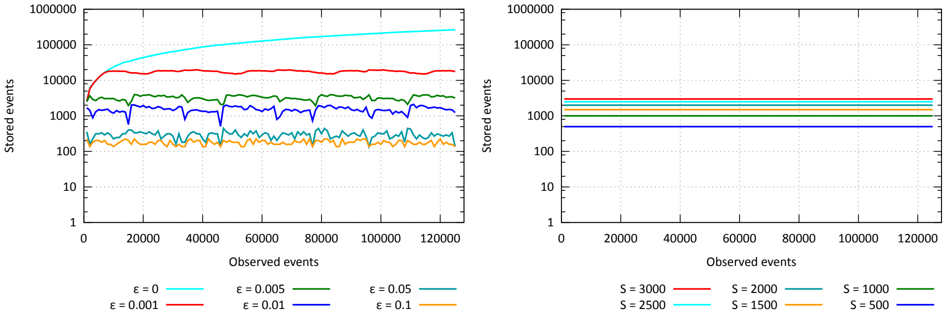
(b) Time efficiency for sliding window

Fig. 4. Time efficiency for lossy counting and sliding window approaches. Both graphs use logarithmic scales.

In Fig. 3, we show the trend of F_1 for the sliding window approach. Concerning the configuration parameters chosen for this experiment, the quality of the models generated by the sliding window approach is, on average, higher with respect to the models generated by the lossy counting approach, even if it has a higher variance. However, as shown later in this section, for the same configuration parameters, the approach based on sliding window is extremely more expensive in terms of time efficiency. Note that the values for F_1 are, in both cases, not very high. This is due to the fact that incomplete traces might not be able to provide insight into violations (see [29]). Both approaches are able to detect the concept drifts correctly, since the comparisons with both gold standards lead to good values for F_1.

The efficiency of the two approaches has been evaluated through *i*) the number of events processed per second (for evaluating the response times of the two approaches) and *ii*) the number of events stored at each evaluation point (for evaluating the memory size needed). For the lossy counting-based approach, we have evaluated these two metrics for different values of the maximum allowed error. For the sliding window-based approach, we have evaluated he metrics for different sizes of the window.

The results are shown in Fig. 4-5. Plots (a) and (b), in Fig. 4, report the number of events that each approach is able to analyze per second. As expected, the time efficiency in both cases depends on the configuration parameters. For the lossy counting approach, the lower is the allowed error, the greater is the number of replayers to be updated (and this results in a lower efficiency). For the sliding window approach, the larger is the window size, the larger is the log to be mined at every evaluation point (again, this has a negative influence over the time efficiency). In general, however, it is evident that, when trying to obtain models with good quality, the sliding approach becomes much more inefficient than the approach based on lossy counting.



(a) Space usage for lossy counting

(b) Space usage for sliding window

Fig. 5. Space usage for lossy counting and sliding window approaches. Both graphs use logarithmic scales.

Plots (a) and (b) of Fig. 5 report the space usage for the two approaches to store events. For the lossy counting approach, the larger is the allowed error, the lower is the space required (since less events must be retained). When $\epsilon = 0$, no data is discarded and, therefore, the space usage grows linearly (the plot uses a logarithmic scale). In all the other cases, given an error value, the space required is rather constant. Some variations may be due to concept drifts. Indeed, when we pass from $\mathcal{L}_1$ to $\mathcal{L}_2$ in the event stream, cases belonging to the first log are discarded because become out of date. For the sliding window case, the space usage is constant and proportional to the window size.

Table 2. Applicability to the BPI Challenges 2011, 2012 and 2013

Event stream	**No. of activity names**	**Total events processed**	**Total processing time** (secs)	**Milliseconds per event**	**Events per second**
2011 BPI Challenge	623	150 291	2977.238	19.81	50.48
2012 BPI Challenge	36	262 200	191.406	0.73	1369.86
2013 BPI Challenge	13	65 533	37.350	0.57	1754.56

Finally, we checked the applicability of the lossy counting approach to assess its scalability when the number of activities in the process is high and, then, the number of candidate constraints grows. In Table 2, we specify the execution time for the logs provided in the BPI Challenges 2011 [1], 2012 [27] and 2013 [24] (maximum allowed error 0.01). This table shows that the approach is applicable even for streams containing more than 600 activity names. Of course, this solution is more expensive in terms of time and memory. However, to improve this aspect and deal also with these extreme cases, the approach can be adapted by using approaches that try to keep track only of the most interesting candidates. This can be easily done by integrating this approach with approaches to discover frequent item sets like the one proposed in [15]. Another way to save memory in this sense is to remove redundancies as explained in [16]. For example, if a constraint is stronger than another, it is possible to monitor only the strongest one and consider the weakest only if the strongest is violated. Another approach to avoid redundancies and contradictions in the discovered constraints is explained in [22, 23].

6 Conclusion

In this paper, we presented a novel framework for the online discovery of declarative process models from streaming event data. The framework is able to produce at runtime an updated picture of the process behavior in terms of LTL-based business constraints. Moreover, it gives to the user meaningful information about the most significant concept drifts occurring during the process execution. The proposed approach combines algorithms for the online discovery of Declare models and algorithms for stream mining. The framework has been implemented in ProM. Our experimentation shows the higher effectiveness of the approach based on sliding window with respect to the approach based on lossy counting at the cost of very poor performances.

As future work, we will conduct a wider experimentation of the proposed framework on several case studies also in real-life scenarios. In this way, it will be possible to understand what can be improved and how. It may be the case, for example, that some templates are more difficult to discover than others. As discussed in Section 5, optimizations are also possible in terms of time efficiency and memory usage.

Acknowledgement Andrea Burattin and Alessandro Sperduti are supported by the Eurostars-Eureka project PROMPT (E!6696). The authors would like to thank Francesca Rossi and Paolo Baldan for their advice.

References

1. 3TU Data Center. BPI Challenge 2011 Event Log, 2011. doi:10.4121/uuid:d9769f3d-0ab0-4fb8-803b-0d1120ffcf54.
2. Charu Aggarwal. *Data Streams: Models and Algorithms*, volume 31 of *Advances in Database Systems*. Springer US, Boston, MA, 2007.
3. Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernard Pfahringer. MOA: Massive Online Analysis Learning Examples. *Journal of Machine Learning Research*, 11:1601–1604, 2010.
4. R.P. Jagadeesh Chandra Bose. *Process Mining in the Large: Preprocessing, Discovery, and Diagnostics*. PhD thesis, Eindhoven University of Technology, 2012.
5. A. Burattin, F.M. Maggi, W.M.P. van der Aalst, and A. Sperduti. Techniques for a Posteriori Analysis of Declarative Processes. In *EDOC*, pages 41–50, 2012.
6. Andrea Burattin. *Applicability of Process Mining Techniques in Business Environments*. PhD Thesis, University of Bologna, 2013.
7. Andrea Burattin, Alessandro Sperduti, and Wil M. P. van der Aalst. Heuristics Miners for Streaming Event Data. *ArXiv CoRR*, December 2012.

8. F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari. Exploiting Inductive Logic Programming Techniques for Declarative Process Mining. *ToPNoC*, 5460:278–295, 2009.
9. T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. The MIT Press, 2nd edition, September 2001.
10. C. Di Ciccio and M. Mecella. Mining constraints for artful processes. In *Proc. of BIS*, LNBIP, pages 11–23. Springer, 2012.
11. Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining Data Streams: a Review. *ACM Sigmod Record*, 34(2):18–26, June 2005.
12. S. Goedertier, D. Martens, J. Vanthienen, and B. Baesens. Robust process discovery with artificial negative events. *JMLR*, 10:1305–1340, 2009.
13. Lukasz Golab and M. Tamer Özsu. Issues in Data Stream Management. *ACM SIGMOD Record*, 32(2):5–14, June 2003.
14. O. Kupferman and M.Y. Vardi. Vacuity Detection in Temporal Model Checking. *Int. Journal on Software Tools for Technology Transfer*, pages 224–233, 2003.
15. F.M. Maggi, J.C. Bose, and W.M.P. van der Aalst. Efficient discovery of understandable declarative models from event logs. In *CAiSE*, pages 270–285, 2012.
16. F.M. Maggi, R.P.J.C. Bose, and W.M.P. van der Aalst. A knowledge-based integrated approach for discovering and repairing declare maps. In *CAiSE*, 2013.
17. F.M. Maggi, A.J. Mooij, and W.M.P. van der Aalst. User-guided discovery of declarative process models. In *Proc. of CIDM*, pages 192–199. IEEE, 2011.
18. Gurmeet Singh Manku and Rajeev Motwani. Approximate Frequency Counts over Data Streams. In *VLDB*, pages 346–357, 2002.
19. Paul Pichler, Barbara Weber, Stefan Zugal, Jakob Pinggera, Jan Mendling, and Hajo A. Reijers. Imperative versus declarative process modeling languages: An empirical investigation. In *BPM Workshops*, pages 383–394, 2011.
20. A. Rozinat, A. K. Alves De Medeiros, C. W. Günther, A. J. M. M. Weijters, and W. M. P. Van Der Aalst. The need for a process mining evaluation framework in research and practice: position paper. BPM 2007, pages 84–89.
21. Nicole Schweikardt. Short-Entry on One-Pass Algorithms. In *Encyclopedia of Database Systems*, pages 1948–1949. 2009.
22. Sergey Smirnov, Matthias Weidlich, Jan Mendling, and Mathias Weske. Action patterns in business process models. In *Service-Oriented Computing*, volume 5900, pages 115–129. 2009.
23. Sergey Smirnov, Matthias Weidlich, Jan Mendling, and Mathias Weske. Action patterns in business process model repositories. *Computers in Industry*, 63(2):98–111, 2012.
24. Ward Steeman;. Bpi challenge 2013, incidents, 2013.
25. W.M.P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.
26. W.M.P. van der Aalst, M. Pesic, and H. Schonenberg. Declarative Workflows: Balancing Between Flexibility and Support. *Computer Science - R&D*, pages 99–113, 2009.
27. B.F.; van Dongen. Bpi challenge 2012, 2012.
28. Matthijs van Leeuwen and Arno Siebes. In *Machine Learning and Knowledge Discovery in Databases*, pages 672–687, 2008.
29. Matthias Weidlich, Holger Ziekow, Jan Mendling, Oliver Gnther, Mathias Weske, and Nirmit Desai. Event-based monitoring of process execution violations. In *Business Process Management*, volume 6896, pages 182–198. 2011.
30. Gerhard Widmer and Miroslav Kubat. Learning in the Presence of Concept Drift and Hidden Contexts. *Machine Learning*, 23(1):69–101, 1996.