Detecting Drift from Event Streams of Unpredictable Business Processes

Alireza Ostovar[1(✉)], Abderrahmane Maaradji[1], Marcello La Rosa[1], Arthur H.M. ter Hofstede[1,2], and Boudewijn F.V. van Dongen[2]

[1] Queensland University of Technology, Brisbane, Australia
{alireza.ostovar,abderrahmane.maaradji,m.larosa,
a.terhofstede}@qut.edu.au
[2] Eindhoven University of Technology, Eindhoven, The Netherlands
b.f.v.dongen@tue.nl

Abstract. Existing business process drift detection methods do not work with event streams. As such, they are designed to detect inter-trace drifts only, i.e. drifts that occur between complete process executions (traces), as recorded in event logs. However, process drift may also occur *during* the execution of a process, and may impact ongoing executions. Existing methods either do not detect such *intra-trace* drifts, or detect them with a long delay. Moreover, they do not perform well with *unpredictable* processes, i.e. processes whose logs exhibit a high number of distinct executions to the total number of executions. We address these two issues by proposing a fully automated and scalable method for online detection of process drift from event streams. We perform statistical tests over distributions of behavioral relations between events, as observed in two adjacent windows of adaptive size, sliding along with the stream. An extensive evaluation on synthetic and real-life logs shows that our method is fast and accurate in the detection of typical change patterns, and performs significantly better than the state of the art.

1 Introduction

Business processes tend to evolve due to various types of changes in the business environment in which they operate. For example, these can be changes in regulations, competition, supply, demand and technological capabilities, as well as internal changes in resource capacity or workload, or simply changes in seasonal factors. Some of these changes may not be documented at all, e.g. those initiated by individual process participants due to replacement of staff, or exceptions that in some cases give rise to new workarounds that over time become common practices. In the long run, undocumented process changes may affect process performance and more in general, hamper process improvement initiatives.

This motivated academics to devise methods and tools that allow business analysts to pinpoint process changes as early as possible. *Business process drift detection* [1–5] is a family of process mining techniques which aim at detecting changes based on observations of business process executions recorded in *event logs*. Event logs consist of *traces*, each representing one execution of the

I. Comyn-Wattiau et al. (Eds.): ER 2016, LNCS 9974, pp. 330–346, 2016.
DOI: 10.1007/978-3-319-46397-1_26

business process. Accordingly, a business process drift is defined as a statistically significant change in the process behavior [5].

Still, state-of-the-art methods in this area suffer from two major limitations. First, they do not work in online settings with streams of events that incrementally record the executions of a business process. As such, they are designed to detect inter-trace drifts only, i.e. drifts that occur between complete process executions (traces), as recorded in event logs. Even if some approaches work in online settings (e.g. [5]), they still deal with streams of complete traces or abstractions thereof. However, process drift may also occur *during* the execution of a process, and may impact ongoing executions. Existing methods either do not detect such *intra-trace* drifts, or detect them with a long delay, as they need to wait for the trace to complete. A related problem is that they do not perform well with *unpredictable* processes, i.e. processes whose logs exhibit a high number of distinct traces over the total number of traces – a typical characteristic of healthcare logs. This is because they rely on statistical tests over trace distributions, which may not have sufficient data samples when the proportion of distinct traces over the total number of traces is very high, in other words, where there is high variability in the log.

To address these two limitations, we propose a fully automated, online method for detecting process drifts from event streams. We perform statistical tests over distributions of behavioral relations between events such as conflict, causality and concurrency, as observed from two adjacent windows of adjustable size, which we slide over the stream. Given that behavioral relations between events are a type of sub-trace features, the method does not suffer from low accuracy when the log is highly variable (i.e. for unpredictable processes). We extensively evaluate the accuracy and scalability of our method by simulating event streams from artificial and real-life logs. The results show that the approach is fast and highly accurate in detecting common change patterns, and significantly better than the state of the art in process drift detection.

The paper is structured as follows. Section 2 discusses related work. Section 3 introduces the proposed method while Sects. 4 and 5 present its evaluations on synthetic and real-life logs respectively. Section 6 concludes the paper.

2 Related Work

Various methods have been proposed to detect process drifts from event logs [1–5]. These methods are based on the idea of extracting *features* (e.g. patterns) from the traces of an event log. For example, Carmona et al. [1] propose to represent a log as a polyhedron. This representation is computed for prefixes in a random sample of the initial traces in the log. The method checks the fitness of subsequent trace prefixes against the constructed polyhedron. If a significant number of these prefixes does not lie in the polyhedron, a drift is declared. The method guarantees that drifts of certain types will always be detected. However, to find a second drift after the first one, the entire detection process must be restarted, thus adversely affecting on the scalability of the method.

In previous experiments we conducted [5], the execution of this implemented method took hours to complete. Another drawback is its inability to pinpoint the exact moment of the drift.

Accorsi et al. [2] propose to cluster the traces in a moving window of the log, based on the average distance between each pair of events in the traces. This method heavily depends on the choice of the window size: a low window size may lead to false positives while a high size may lead to false negatives (undetected drifts), as drifts happening inside the window go undetected. In addition the method is not designed to deal with loops, and may fail to detect types of changes that do not cause significant variations to the distances between activity pairs, e.g. changes involving an activity being skipped.

Bose et al. [3] propose a method to detect process drifts based on statistical testing over feature vectors. The method is not fully automated, as the user is asked to identify the features to be used for drift detection, implying that they have some a-priori knowledge of the possible nature of the drift. Further, this method is unable to identify certain types of drifts such as inserting a conditional branch or a conditional move, even if the relevant process activities are selected as features. Finally, similar to Accorsi et al. [2], the user is required to set a window size for drift detection. Depending on how this parameter is set, some drifts may be missed. This latter limitation is partially lifted in a subsequent extension [4], which introduces a notion of *adaptive window*. The idea is to increase the window size until it reaches a maximum size or until a drift is detected. However, this technique requires the user to set a minimum and a maximum window size. If the minimum window size is too small, minor variations (e.g. noise) may be misinterpreted as drifts (false positives). Conversely, if the maximum window size is too large, the execution time is affected and some drifts may go undetected.

All these methods may miss certain types of changes that are not covered by the types of features used. Moreover, their scalability is constrained by the need to extract and analyze a feature space that is potentially very large. Hence, they are not suitable for online settings. This motivated us to propose a new method [5] for detecting process drifts determined by a wide range of typical process change patterns [6]. The method is based on statistical tests over the distribution of *runs* (an abstraction of complete traces), as observed in two consecutive time windows. The size of these windows is adjusted based on changes in log variability. This method outperforms all the above methods in terms of detection accuracy and scalability. As such, we selected it as a baseline for the experiments in this paper. As shown in the experiments, this method also does not cater for highly variable event logs. In such logs each distinct run occurs only a few times, leading to a less reliable statistical test, to hence to many false negatives. Further, the method cannot detect intra-trace drifts from event streams.

To the best of our knowledge, the only method that deals with event streams has been proposed by Burattin et al. [7]. However, this work mainly focuses on the online discovery of process models captured as a set of business constraints (formulated in Linear Temporal Logic) between events. Any change in the extracted constraints over time may be considered as a drift. Nonetheless, there is no statistical support for detecting whether changes are in fact

significant, and the exact positions of the identified drifts are not reported. As such, drift detection accuracy is not evaluated. In another study, Burattin et al. [8] adapt an automated process discovery method, namely the Heuristics Miner, to handle incremental updates as new events are produced. Our proposal is complementary to this as it allows drifts to be detected accurately and efficiently, and can be used as an oracle to identify points in time when the process model should be updated.

Drift detection has been studied in the field of data mining [9], where a widely studied challenge is that of designing efficient learning algorithms that can adapt to data that evolves over time (a.k.a. *concept drift*). This includes for example changes in the distributions of numerical or categorical variables. However, the methods developed in this context deal with simple structures (e.g. numerical or categorical variables and vectors thereof), while in business process drift detection we seek to detect changes in more complex structures, specifically behavioral relations between process events (e.g. concurrency, conflicts, loops). Thus, methods from the field of concept drift detection in data mining cannot be readily transposed to business process drift detection.

3 Drift Detection Method

From a statistical viewpoint, the problem of business process drift detection can be formulated as follows: *identify a time point before and after which there is a statistically significant difference between the observed process behaviors*. Therefore, to detect a drift we need features that properly capture the behavior of a process. By monitoring and analyzing the feature vectors over time, we can identify the time points where the feature vectors exhibit statistically significant changes. We explored a few different features including *Direct Follow relations* (direct succession), *Follow relations* (succesion), *Block Structures* (extracted from *process trees* produced by the Inductive Miner [10]) and α^+ *Relations* [11]. We found that while the direct follow and follow relations are over-fitting features, block structures were under-fitting features. However, α^+ relations proved to be the suitable level of abstraction for capturing the behavior of unpredictable processes represented in an event stream.

To detect a process drift we perform a statistical test, namely the G-test of independence,[1] over distributions of α^+ relations observed in two adjacent time windows of adaptive size, sliding along with a stream of events. Basically, the most recent events are equally divided into reference window (less recent events), and detection window (more recent events). Each time a new event enters the event stream, the two windows shift forward so that the new event is in the detection window. The set of events within each window is used to build a corresponding sub-log. This sub-log represents the process behavior observed

[1] The G-test is a non-parametric hypothesis statistical test which assumes no a-priori knowledge of the statistical distributions. The G-test is a better approximation to the theoretical chi-squared distribution than the chi-squared test [12].

within the respective window. The sliding window is a well-established technique in the concept drift community [9].

Then the α^+ relations and their frequencies are extracted from each sub-log, and used to populate a $2 \times n$ matrix, the so-called *contingency matrix*, where n is the number of distinct relations. Each column in the contingency matrix corresponds to a category of a statistical variable, here an α^+ relation. The first row in the contingency matrix contains the frequencies of the relations in the detection window, i.e. the *observed* frequencies, while the second row contains the frequencies of the relations in the reference window, i.e. the *expected* frequencies.

The result of applying the G-test of independence on the contingency matrix is the significance probability (*P-value*) that the populations of α^+ relations over the two windows come from the same distribution. A *P-value* above a predefined threshold[2] accepts the null hypothesis, i.e. the frequency distributions of the α^+ relations in the two windows are similar. However, a *P-value* below the threshold rejects the null hypothesis, meaning that the α^+ relations in the two windows come from different distributions. In other words, they reflect different process behaviors (process drift).

3.1 Intra-trace vs. Inter-trace

A drift may occur between complete executions of a process. We call this an *inter-trace* drift. For example, a new legislation requires an insurance company to perform a more stringent verification on new claims, while old claims are exempted. These however are not the only type of drift. In reality, a drift may also occur *during* the execution of a process and may impact ongoing process instances [6]. We call these *intra-trace* drifts. For example, an insurance check may need to be removed altogether due to a contingency plan triggered by severe weather conditions (e.g. a flood). Such a change may impact new process instances as well as the instances that have already started, but that have not yet gone through the check to be removed.

In addition, in order to detect a drift using a stream of traces, we have to wait until each trace completes before we can use it. This delays the detection of the drift. On the other hand, working on a stream of events allows us to instantly use each observed event, thereby detecting a drift as soon as possible during the execution of the process.

3.2 Event Stream and α^+ Relations

An event log is a set of traces, each capturing the sequence of events originated from a given process instance. Each event represents an occurrence of an activity. The configuration where these events are read individually from an online source is known as event streaming. An event stream is a potentially infinite sequence of events, where events are ordered by time and indexed. Events of the same trace do not need to be consecutive in the event stream, i.e. traces can be "overlapping". Formally:

[2] The typical value of the threshold, i.e. significance level, for the G-test is 0.05 [13].

Definition 1 (Event log, Trace, Event stream). Let L be an *event log* over the set of labels $\mathcal{L}$, i.e. $L \in \mathbb{P}(\mathcal{L}^*)$. Let $\mathcal{E}$ be the set of event occurrences and $\lambda : \mathcal{E} \to \mathcal{L}$ a labelling function. An *event trace* $\sigma \in L$ is defined in terms of an order $i \in [0, n-1]$ and a set of events $\mathcal{E}_\sigma \subseteq \mathcal{E}$ with $|\mathcal{E}_\sigma| = n$ such that $\sigma = \langle \lambda(e_0), \lambda(e_1), \ldots, \lambda(e_{n-1}) \rangle$. An *event stream* is a function $S : \mathbb{N}^+ \to \mathcal{E}$ that maps every element from the index $\mathbb{N}^+$ to $\mathcal{E}$.

In this paper, we use the α^+ relations, as an extension of the α relations, to capture the behavior of a process. The α-algorithm defines three exclusive relations: *conflict*, *concurrency* and *causality*. The α^+-algorithm adds two more relations: *length-two loop* and *length-one loop*. The α^+ relations are formally defined as follows:

Definition 2 (α^+ relations from [11]). Let L be an event log over $\mathcal{L}$. Let a,$b \in \mathcal{L}$:

- $a \triangle_L b$ if and only if there is a trace $\sigma = l_1 l_2 l_3 \ldots l_n$ and $i \in 1, \ldots, n-2$ such that $\sigma \in L$ and $l_i = l_{i+2} = a$ and $l_{i+1} = b$,
- $a \diamond_L b$ if and only if $a \triangle_L b$ and $b \triangle_L a$,
- $a >_L b$ if and only if there is a trace $\sigma = l_1 l_2 l_3 \ldots l_n$ and $i \in 1, \ldots, n-2$ such that $\sigma \in L$ and $l_i = a$ and $l_{i+1} = b$,
- $a \rightarrow_L b$ if and only if $a >_L b$ and ($b \not>_L a$ or $a \diamond_L b$),
- $a \#_L b$ if and only if $a \not>_L b$ and $b \not>_L a$, and
- $a \parallel_L b$ if and only if $a >_L b$ and $b >_L a$, and $a \not\diamond_L b$.

A length-two loop relation, including a and b, is denoted with $a \triangle_L b$. The frequency of this relation in a log is the number of occurrences of the substring aba. A causality relation from a to b is denoted with $a \rightarrow_L b$. The frequency of this relation in a log is the number of occurrences of the substring ab. A parallel relation between a and b is denoted with $a \parallel_L b$. The frequency of this relation in a log is the minimum of the frequencies of the two substrings, ab and ba. A conflict relation between a and b is denoted with $a \#_L b$, and indicates that there is no trace with the substring ab or ba. The frequency of this relation in a log is the number of occurrences of a and b. The α^+-algorithm also discovers length-one loop relations as a pre-processing operation. For example, there is a length-one loop including the activity a in a log if there is a trace with the substring aa. The frequency of this relation in a log is the number of occurrences of the substring aa.

3.3 Statistical Testing over Event Streams

This section describes our online drift detection algorithm as presented in Algorithm 1. The drift detection algorithm has three parameters: 1. *eventStream*: a stream of events. 2. *initWinSize*: initial size of the detection and reference windows. 3. *maxBufSize*: maximum available memory for the event buffer storing the incoming events, namely *eventBuf*. Since the algorithm works online the size of this buffer must not exceed *maxBufSize*. Therefore, each time a new event e arrives we first check if the buffer has reached its maximum size, and if so we shift the events in the buffer and discard the least recent event (lines 10–11). We then insert the new event into the buffer (line 12).

The first statistical test should be performed when the number of events in the buffer is $2 \times initWinSize$ (line 14). Before each statistical test we adapt the size of the two windows to improve the accuracy of the approach (line 15). The notion of adaptive window is explained in Sect. 3.4. The method *updateSublogs* updates the sub-logs related to the detection and reference windows, namely *detSubLog* and *refSubLog*, respectively, using the events within their corresponding windows (line 16). The first time this method is called the sub-logs are built from scratch. The α^+ relations and their frequencies are extracted from the two sub-logs and populated in a contingency matrix (line 18). We then perform the G-test of independence on this contingency matrix and obtain the *P-value* (line 19). The value of the G-test threshold, *GtestThreshold*, is set to the typical value of the G-test, which is 0.05.

Each time the *P-value* drops below the threshold *GtestThreshold*, we store the current event and the current window size in *pbtEvent* and *pbtWinSize*, respectively (lines 23–24). Since any statistical test is subject to sporadic stochastic oscillations, we introduced an additional filter, namely *oscillation filter*. The *P-value* drops have to be consistent over many consecutive statistical tests in order to avoid reporting incidental drops in the *P-value* (oscillations). The size of the oscillation filter is calculated by function Φ which uses the window size w as input. The number of consecutive tests in which the *P-value* is below the threshold *GtestThreshold* is stored in *pbtLen*. We detect a drift only if *pbtLen* is at least equal to $\Phi(w)$ (line 25). Our experiments showed that a value of $\Phi(w) = w/2$ provides the best results in terms of accuracy (cf. Sect. 4.3). The drift is localized at the event where the *P-value* dropped consistently below the threshold, stored at *pbtEvent* (line 26). Whenever the *P-value* exceeds the threshold we reset *pbtLen*, *pbtEvent* and *pbtWinSize* (lines 28–30).

3.4 Adaptive Window

Best practices of using the G-test recommend that no more than 20 % of the expected frequencies in the contingency matrix have less than 5 occurrences, to have a reliable statistical test [12]. Thus, each time before performing the statistical test we ensure the size of the two windows is large enough to fulfil this requirement. Even though the larger the window size is the higher the chances that the requirement of the statistical test is met, a very large window size may increase the number of new events needed to detect a drift, so-called *mean delay*. Furthermore, it may also cause the detection and reference windows to span over multiple drifts, thereby letting some of the drifts go undetected. Therefore, we need to balance between improving the reliability of the statistical test, by increasing the window size, and reducing the detection delay of the method, by decreasing the window size.

The idea behind our adaptive window originates from the requirement of the statistical test mentioned above, meaning that on average we aim to have a frequency of no less than 5 for each of the α^+ relations in the contingency matrix. Given that the maximum number of possible relations over the set of labels

Algorithm 1. Drift Detection Algorithm

```
Input: eventStream; initWinSize; maxBufSize.
1  eventBuf                                                    // Event buffer
2  w ⟵ initWinSize                                            // Current window size
3  detSubLog, refSubLog          // List of sub-traces within detection and
     reference windows, respectively
4  GtestThreshold ⟵ 0.05              // Typical threshold value of G-test
5  pbtEvent ⟵ NIL                // Current event when P-value drops below
     GtestThreshold
6  pbtWinSize ⟵ −1                  // Value of w when P-value drops below
     GtestThreshold
7  pbtLen ⟵ 0       // # of consecutive tests that P-value remains below
     GtestThreshold
8  while true do
9      e ⟵ fetch(eventStream)                                  // Fetch a new event e
10     if size(eventBuf) = maxBufSize then
11         shift(eventBuf)
12     insert(eventBuf, e)
13     ebLength ⟵ length of eventBuf
14     if ebLength ≥ 2 · initWinSize then
15         newWinSize ⟵ adWin(eventBuf, w)
16         updateSublogs(eventBuf, detSubLog, refSubLog, w, newWinSize)
17         w ⟵ newWinSize
18         conMat ⟵ buildContingencyMatrix(detSubLog, refSubLog)
19         pValue ⟵ Gtest(conMat)
20         if pValue < GtestThreshold then
21             pbtLen ⟵ pbtLen + 1
22             if pbtEvent = NIL then
23                 pbtEvent ⟵ e
24                 pbtWinSize ⟵ w
25             if pbtLen = Φ(pbtWinSize) then
26                 reportDrift(pbtEvent)        // Drift detected and reported
27         else
28             pbtLen ⟵ 0
29             pbtEvent ⟵ NIL
30             pbtWinSize ⟵ −1
```

(activity names) $\mathcal{L}$ is $|\mathcal{L}|^2$, we calculate $|\mathcal{L}|$ over both detection and reference windows, denoted by $|\mathcal{L}_{det}|$, $|\mathcal{L}_{ref}|$, respectively. By multiplying $max(|\mathcal{L}_{det}|, |\mathcal{L}_{ref}|)^2$ by 5 it is likely to have enough events in both windows to fulfil the requirement of the statistical test. Hence window size w is defined as $w = max(|\mathcal{L}_{det}|, |\mathcal{L}_{ref}|)^2 \cdot 5$.

The expansion and the shrinkage of the windows is performed recursively. This is because each time the windows are, for example, expanded there may be a need to expand the windows again due to changes in $|\mathcal{L}_{det}|$ and/or $|\mathcal{L}_{ref}|$. It

is worth mentioning that our adaptive window is not dependent on the initial window size, since starting from any initial value the window sizes converge to the length needed to fulfil the requirement of the statistical test. The maximum size each window could grow to is the length of the event buffer divided by two.

It is worth mentioning that in the unlikely extreme scenario where the overlapping between traces is to the extent that each event within a window comes from a distinct trace, data streaming techniques with a gradual forgetting strategy [9] should be used.

Time complexity. Each time a new event is received from the stream, we first extract the α^+ relations in each sliding window and count their frequencies, and then perform the G-test of independence. The worst-case complexity of computing the α^+ relations is quadratic in the cardinality of the label set, i.e. $O(|\mathcal{L}|^2)$. Given a contingency matrix of maximum size $2 \times |\mathcal{L}|^2$, the complexity of the G-test is $O(|\mathcal{L}|^2)$. Since the two mentioned operations have the same complexity and are executed in a sequence, the complexity of our method is $O(|\mathcal{L}|^2)$ for every new event read from the stream.

4 Evaluation on Synthetic Logs

We implemented the proposed method as a plug-in, namely *ProDrift 2.0*,[3] and used this tool to assess the goodness of our method in terms of accuracy and scalability in a variety of settings. The tool can read a continuous stream of events or an event log replayed as an event stream. In the rest of this section we discuss the design of the experiments, the datasets used, the impact that oscillation filter and inter-drift distance have on our method, and conclude by comparing our method with the method in [5].

4.1 Evaluation Design

To evaluate the effectiveness of our method, we created a variety of synthetic logs with different configurations, and then replayed these logs as event streams. We first modeled a base business process using CPN tools and then used this model to generate the logs.[4] The model features 28 different activities, combined with different intertwined structural patterns: three XOR structures, four AND structures, two loops of length two, and one loop of length four. We built this model in a way that the resulting log is highly variable. To produce logs that include drifts, we then injected different types of control-flow changes into the base CPN model.

We applied in turn one out of twelve *simple change patterns* [6] to the base model. These patterns, summarized in Table 1, describe different change operations commonly occurring in business process models, such as adding/removing a model fragment, putting a model fragment in a loop, swapping two fragments,

[3] Available at http://apromore.org/platform/tools.
[4] http://cpntools.org.

or parallelizing two sequential fragments. Similarly to our previous work [5], we organized the simple changes into three categories: Insertion ("I"), Resequentialization ("R") and Optionalization ("O") (cf. Table 1). These categories make six possible *composite change patterns* ("IOR", "IRO", "OIR", "ORI", "RIO", and "ROI") by nesting the simple patterns within each other. For example, the composite pattern "ROI" can be obtained by first adding a new activity ("I"), then making this activity parallel to an existing activity ("O") and finally by putting the whole parallel block into a loop structure ("R").

Each of these change patterns were applied locally on the base model in such a way that it is possible during log replay to choose between the base model execution path and the altered one. For instance, if the applied change pattern was to replace a process fragment (*rp*), the CPN model would have a branching point, called *drift toggle*, right before this fragment, that allows the execution to follow either the initial model fragment or the new process fragment. A drift is injected by switching the toggle on or off. In this way, we can generate intra-trace drifts. For instance, if the toggle is switched on when trace #500 starts, the traces that started before that trace and have not yet reached the branching point, will follow the new process behavior, thus exhibiting the change. These traces will therefore have an intra-trace drift. In the remainder, whenever we say that a drift has been injected at a given trace number (after a given number of traces) it means that the drift toggle has been switched on at the first event of that given trace number (resp. after that given number of traces have started).

Finally, in order to vary the distance between drifts, for each change pattern we generated three logs of 2,500, 5,000 and 10,000 traces, and injected drifts by switching the drift toggle on and off every 10 % of the log. This led to an inter-drift distance of 250, 500 and 1,000 traces per change pattern, with 9 drifts per log. The position of an injected drift is given by the index of the first event in the event stream, after the drift toggle has been switched on. These indexes are used as the true positives of our evaluation (the *gold standard*). Further, for each of the 6 composite change patterns, we created 3 possible combinations, by changing the type of pattern used. This led to 12 (simple patterns) + 18 (complex patterns) = 30 different variants of the CPN model times three inter-drift distances, resulting in a total of 90 logs.[5] All these logs exhibit a very high *trace*

Table 1. Simple control-flow change patterns

Code	Simple change pattern	Category
re	Add/remove fragment	I
cf	Make two fragments conditional/sequential	R
lp	Make fragment loopable/non-loopable	O
pl	Make two fragments parallel/sequential	R
cb	Make fragment skippable/non-skippable	O
cm	Move fragment into/out of conditional branch	I
cd	Synchronize two fragments	R
cp	Duplicate fragment	I
pm	Move fragment into/out of parallel branch	I
rp	Substitute fragment	I
sw	Swap two fragments	I
fr	Change branching frequency	O

[5] All the CPN models used for this simulation, the resulting synthetic logs, and the detailed evaluation results are available with the software distribution.

variability (80 % ± 2), measured as the ratio between the number of distinct traces and the number of total traces in the log. According to our analysis of real-life logs, this value is very indicative of logs of unpredictable processes, such as the one used in the second part of this evaluation.

To assess the scalability of our method for online drift detection, we measured the execution time per each new event read from the stream. To evaluate accuracy, we used F-score and mean delay. The *F-score* is computed as the harmonic mean of recall and precision, where recall measures the proportion of actual drifts that have been detected and precision measures the proportion of detected drifts that are correct. The *mean delay* [14] assesses the ability of the method to find drifts as early as possible in an event stream, and is measured as the number of events between the actual position of the drift and the end of the detection window.

4.2 Execution Times

We conducted all tests on an Intel i7 2.20 GHz with 16 GB RAM (64 bit), running Windows 7 and JVM 7 with standard heap space of 2 GB, and a stream buffer (*maxBufSize*) of 1 GB. The time required to update the α^+ relations and perform the G-test, ranges from a minimum of 10 ms to a maximum of 50 ms with an average of 14 ms. These results show that the method is suited for online drift detection, including scenarios where the inter-arrival time between events is in the order of milliseconds.

4.3 Impact of Oscillation Filter

In the first experiment, we measured the impact of the oscillation filter $\Phi(w)$ on F-score and mean delay, by varying its value from $w/4$ to w, where w is the window size. Figure 1 shows the obtained F-score and mean delay averaged over all change patterns. As expected, we observe that the F-score increases as the filter value grows and eventually plateaus when it reaches the sliding window size, by filtering out false positives. However, a larger filter value causes a much higher delay. On the other hand, while a smaller filter value leads to a smaller delay, it may induce our method to consider incidental changes as actual drifts, causing the F-score to drop, though this still remains above 0.9. As a tradeoff, for the remainder of this evaluation, we used $\Phi(w) = w/2$. With this parameter being set empirically, our method is completely automated, and no parameter setting is required from the user.

4.4 Inter-drift Distance

In the second experiment, we compared the F-score and mean delay obtained on logs of different inter-drift distances (250, 500 and 1,000), in order to assess the minimum distance that our method can handle. The results, averaged over all change patterns, indicate that the method performs similarly for the logs

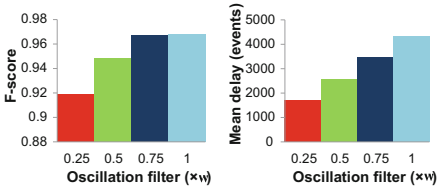

Fig. 1. F-score and mean delay using different oscillation filter values.

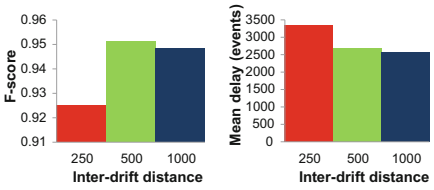

Fig. 2. F-score and mean delay using different inter-drift distances.

with 500 and 1,000 traces of inter-drift distance, achieving an F-score of about 0.95 and mean delay of about 2,500 (cf. Fig. 2). There is a slight decrease in the F-score and a notable increase in the mean delay when using a distance of 250 traces. In this case, the two sliding windows may contain two drifts as these are very close. In such cases, the method may miss one of the two drifts, leading to a lower recall. These cases however are not very common, as evidenced by the value of the F-score, which does not go below 0.92.

4.5 Comparison with Baseline per Process Change Pattern

In the third experiment, we evaluated the accuracy of our method in detecting each of the 18 change patterns. Figure 3 shows the F-score and mean delay for each change pattern in Table 1, averaged over the three log sizes, in comparison with those obtained with our previous run-based method [5] (the baseline).

Our method could find all the change patterns with a high F-score (above 0.9 in all but four cases), and a delay in the range of 2,500 events (approximately 100 traces), peaking at 4,000 events. When compared to the baseline method, our method outperforms the baseline in terms of F-score in the majority of change patterns (cf. Fig. 3(left)), while the baseline fails to detect half of the simple change patterns (*lp*, *pl*, *cb*, *cd*, *pm* and *sw*). Since in highly variable logs each distinct run is observed only a few times, the result of the statistical test is less reliable. Thus, in such logs, the run-based method can only find drift types whose occurrences replace the current set of runs with a considerably new set of runs, e.g. when removing a process fragment (pattern *re*). On the other hand, our current method considers events (as opposed to traces) and extracts fine-

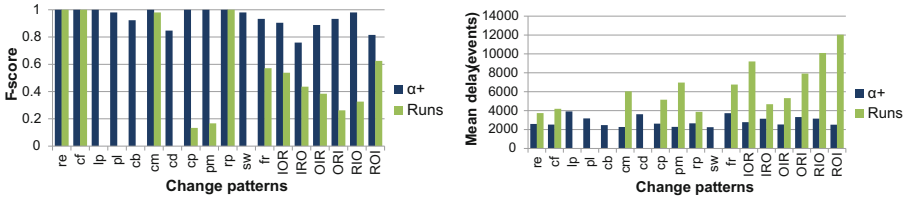

Fig. 3. F-score and mean delay per change pattern, obtained with our method vs. [5].

grained, yet abstract features that capture the process behavior into a few basic relations. Each drift type would be represented in a handful of α^+ relations, and any change in its frequency would be "echoed" through its correspondent basic relations, making it easier for the statistical test to detect such a change. Moreover, our method could always detect the drift faster than the baseline (cf. Fig. 3(right)) as it does not need to wait until a trace is completed to consider it as an input for the statistical test.

4.6 Comparison with Baseline over Different Log Variability Rates

In this last experiment with artificial logs, we evaluated our method in comparison with the baseline, when changing the variability rate of the log. As said before, the trace variability of a log is the ratio between distinct traces and the total number of traces. It varies from close to 0 %, where all traces are the same, to 100 %, where every trace is distinct. Similarly, we define the *run variability* as the ratio between distinct runs and the total number of runs. Depending on the concurrency oracle used, a high trace variability does not necessarily imply a high run variability. On the other hand, a high run variability always implies an equal or higher trace variability. For instance, a log with 50 % trace variability results in a run variability of 10 % (i.e. on average each run is repeated 10 times). This is due to the aggregation of traces into runs based on the concurrency oracle. The baseline method performs relatively well with a log with 10 % run variability. Thus, we studied how F-score and mean delay vary as we increase the run variability of a log.

For this purpose, we generated a new set of synthetic logs as described in Sect. 4.1 with different run variability rates, achieved by varying the loopback branching probability in the CPN model. For each run variability rate and change pattern, we generated logs of 10,000 traces. The results of this evaluation are reported in Fig. 4.

As the variability of the log increases, the baseline method's accuracy drops significantly. This is because the statistical test adopted by this method is inadequate when the number of distinct runs is large, as their frequency will be low. In contrast, capturing the process behavior at a lower level of abstraction, as done by the $\alpha+$ relations, as opposed to runs, leads to much higher frequencies in the contingency table of the statistical test, ensuring its reliability.

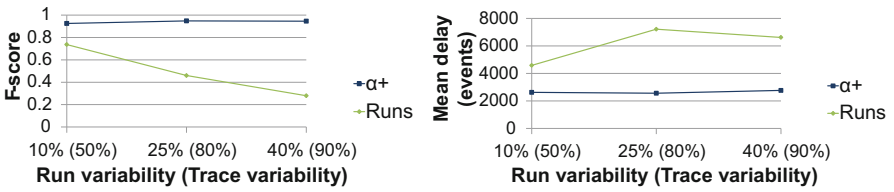

Fig. 4. F-score and mean delay per log variability, obtained with our method vs. [5].

This property is valid regardless of the variability of the log which explains the steady performance of our method.

5 Evaluation on Real-Life Log

In addition to the experiments with artificial logs, we evaluated out method on the BPI Challenge (BPIC) 2011 log, and compared the results with those obtained by the baseline.[6] This log records patient treatments in the Gynaecology department of a Dutch academic hospital. It contains 150,291 events in over 1,143 traces, of which 981 are distinct, and 623 labels. We first filtered the noise from this event log, using an offline noise filter [15], which basically removes infrequent activities.[7] This operation reduced the number of traces to 1,121, of which 798 are distinct, and the number of labels to 42, resulting in the same trace and run variability of 71 %.

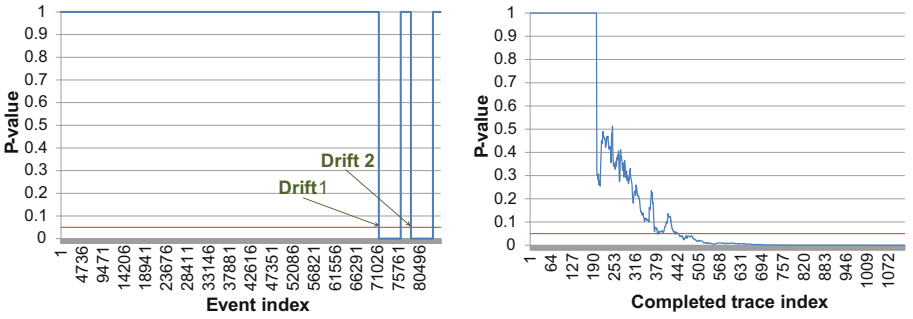

Fig. 5. P-value in our method (left) and in the baseline (right) for the BPIC 2011 log.

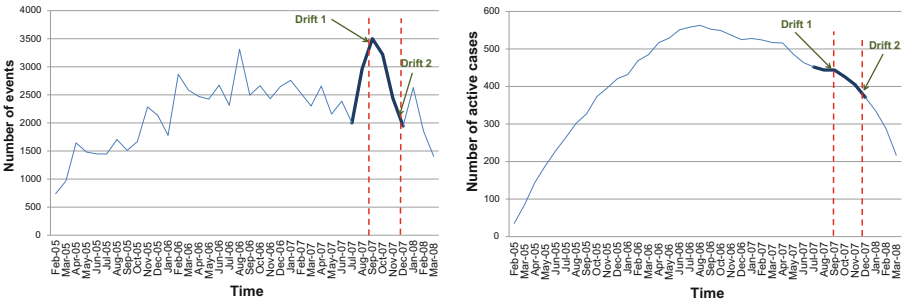

Fig. 6. Number of events (left) and active cases per month (right) in the BPIC 2011 log.

[6] http://dx.doi.org/10.4121/uuid:d9769f3d-0ab0-4fb8-803b-0d1120ffcf54.

[7] In streaming settings, online noise filters such as the Kalman filter [16] could be used instead.

We applied our method on the stream of events obtained by replaying the filtered log. The average execution time for each new event in the stream was 44 ms. As shown in Fig. 5(left), two drifts were detected at the event indexes of 71,321 and 78,541, corresponding to the dates 6/9/2007 and 29/11/2007 respectively. The baseline could not detect any drift as the p-value quickly dropped and remained under the threshold, as shown in Fig. 5(right).

In order to validate the results, we profiled the number of events per month, shown in Fig. 6(left). The plot exhibits a sharp and consistent increase in the number of events between July and Sept. 2007 followed by a sharp and consistent decrease between Sept. and Dec. 2007. We investigated the log and found that the frequencies of five activities do increase and then decrease notably over the period in question. Moreover, the number of active cases per month (cf. Fig. 6(right)) decreases gradually after August 2006. Thus, this variation in the number of events cannot be explained because of new cases. Rather, this phenomenon could be the result of some rework in the business process. A rework may manifest itself with looping behavior and/or duplicate activities, which are change patterns our method is able to detect.

In conclusion, while these observations support the hypothesis of the presence of two drifts in the log, the results should be validated with domain experts.

6 Conclusion

We presented a fully automated method for online detection of business process drifts from event streams. The method relies on a statistical test over distributions of behavioral relations observed in two adjacent windows sliding along the event stream. We proposed an adaptive window technique in order to automatically adjust the sliding windows size, striking a good tradeoff between accuracy and detection delay.

We evaluated our method against different degrees of log variability and varying inter-drift distance, by injecting various change patterns into synthetic logs. The results showed that the method is able to scale up to online settings and detect drifts very accurately, while outperforming a state-of-the-art baseline for all the change patterns. A second evaluation on a healthcare log with very high variability showed that our method could detect two drifts that were supported by observations from the log.

In future we plan to empirically evaluate our technique with domain experts. Moreover, we plan to work on drift characterization in order to provide process stakeholders with relevant explanations on the detected drifts. A possible direction to tackle this problem is to apply the log delta analysis technique in [17] in order to retrieve diagnostics of the behavioral differences between the pre-drift and the post-drift sub-streams. Another avenue for future work is to study the interplay between changes in the process control flow and changes in other process perspectives, such as in the resources behavior or data involved in the execution of the process. In this respect, a starting point is to look at the work in [18], which analyses the dynamics of human resource behavior as observed from event logs.

Acknowledgments. This research is partly funded by the Australian Research Council (grant DP150103356).

References

1. Carmona, J., Gavaldà, R.: Online techniques for dealing with concept drift in process mining. In: Hollmén, J., Klawonn, F., Tucker, A. (eds.) IDA 2012. LNCS, vol. 7619, pp. 90–102. Springer, Heidelberg (2012). doi:10.1007/978-3-642-34156-4_10
2. Accorsi, R., Stocker, T.: Discovering workflow changes with time-based trace clustering. In: Aberer, K., Damiani, E., Dillon, T. (eds.) SIMPDA 2011. LNBIP, vol. 116, pp. 154–168. Springer, Heidelberg (2012)
3. Bose, R.P.J.C., van der Aalst, W.M.P., Zliobaite, I., Pechenizkiy, M.: Dealing with concept drifts in process mining. IEEE Trans. NNLS **25**(1), 154–171 (2014)
4. Martjushev, J., Bose, R.P.J.C., Aalst, W.M.P.: Change point detection and dealing with gradual and multi-order dynamics in process mining. In: Matulevičius, R., Dumas, M. (eds.) BIR 2015. LNBIP, vol. 229, pp. 161–178. Springer, Heidelberg (2015). doi:10.1007/978-3-319-21915-8_11
5. Maaradji, A., Dumas, M., Rosa, M., Ostovar, A.: Fast and accurate business process drift detection. In: Motahari-Nezhad, H.R., Recker, J., Weidlich, M. (eds.) BPM 2015. LNCS, vol. 9253, pp. 406–422. Springer, Heidelberg (2015). doi:10.1007/978-3-319-23063-4_27
6. Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features-enhancing flexibility in process-aware information systems. DKE **66**(3), 438–466 (2008)
7. Burattin, A., Cimitile, M., Maggi, F.M., Sperduti, A.: Online discovery of declarative process models from event streams. IEEE Trans. Serv. Comput. **8**, 833–846 (2015)
8. Burattin, A., Sperduti, A., van der Aalst, W.M.P.: Control-flow discovery from event streams. In: IEEE Congress on Evolutionary Computation (CEC), pp. 2420–2427. IEEE (2014)
9. Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., Bouchachia, A.: A survey on concept drift adaptation. ACM Comput. Surv. (CSUR) **46**(4), 1–37 (2014)
10. Leemans, S.J.J., Fahland, D., Aalst, W.M.P.: Discovering block-structured process models from event logs - a constructive approach. In: Colom, J.-M., Desel, J. (eds.) PETRI NETS 2013. LNCS, vol. 7927, pp. 311–329. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38697-8_17
11. de Medeiros, A.A., van Dongen, B.F., Van der Aalst, W.M.P., Weijters, A.: Process mining: extending the α-algorithm to mine short loops. Technical report, BETA Working Paper Series, WP 113, Eindhoven University of Technology, Eindhoven (2004)
12. Harremoës, P., Tusnády, G.: Information divergence is more χ^2-distributed than the χ^2-statistics. In: IEEE ISIT, pp. 533–537 (2012)
13. Nuzzo, R.: Statistical errors. Nature **506**(13), 150–152 (2014)
14. Ho, S.S.: A martingale framework for concept change detection in time-varying data streams. In: Proceedings of ICML, pp. 321–327. ACM (2005)
15. Conforti, R., La Rosa, M., ter Hofstede, A.H.: Noise filtering of process execution logs based on outliers detection (2015)

16. Bifet, A., Gavaldà, R.: Kalman filters and adaptive windows for learning in data streams. In: Todorovski, L., Lavrač, N., Jantke, K.P. (eds.) DS 2006. LNCS (LNAI), vol. 4265, pp. 29–40. Springer, Heidelberg (2006). doi:10.1007/11893318_7
17. Beest, N.R.T.P., Dumas, M., García-Bañuelos, L., Rosa, M.: Log delta analysis: interpretable differencing of business process event logs. In: Motahari-Nezhad, H.R., Recker, J., Weidlich, M. (eds.) BPM 2015. LNCS, vol. 9253, pp. 386–405. Springer, Heidelberg (2015). doi:10.1007/978-3-319-23063-4_26
18. Pika, A., Wynn, M.T., Fidge, C.J., Hofstede, A.H.M., Leyer, M., Aalst, W.M.P.: An extensible framework for analysing resource behaviour using event logs. In: Jarke, M., Mylopoulos, J., Quix, C., Rolland, C., Manolopoulos, Y., Mouratidis, H., Horkoff, J. (eds.) CAiSE 2014. LNCS, vol. 8484, pp. 564–579. Springer, Heidelberg (2014). doi:10.1007/978-3-319-07881-6_38