Fast and Accurate Business Process Drift Detection

Abderrahmane Maaradji[1,3], Marlon Dumas[2], Marcello La Rosa[3,1], and Alireza Ostovar[3]

[1] NICTA, Australia
Abderrahmane.Maaradji@nicta.com.au
[2] University of Tartu, Estonia
marlon.dumas@ut.ee
[3] Queensland University of Technology, Australia
{m.larosa,alireza.ostovar}@qut.edu.au

Abstract. Business processes are prone to continuous and unexpected changes. Process workers may start executing a process differently in order to adjust to changes in workload, season, guidelines or regulations for example. Early detection of business process changes based on their event logs – also known as business process drift detection – enables analysts to identify and act upon changes that may otherwise affect process performance. Previous methods for business process drift detection are based on an exploration of a potentially large feature space and in some cases they require users to manually identify the specific features that characterize the drift. Depending on the explored feature set, these methods may miss certain types of changes. This paper proposes a fully automated and statistically grounded method for detecting process drift. The core idea is to perform statistical tests over the distributions of runs observed in two consecutive time windows. By adaptively sizing the window, the method strikes a trade-off between classification accuracy and drift detection delay. A validation on synthetic and real-life logs shows that the method accurately detects typical change patterns and scales up to the extent that it works for online drift detection.

1 Introduction

Business processes are prone to evolution in response to various factors, including changes in the regulatory environment, competitive environment, supply, demand and technology capabilities, as well as seasonal factors. Some process changes are planned and documented, but others may occur unexpectedly and remain unnoticed by some process stakeholders. For example, this may be the case of changes undertaken by the initiative of individual process workers in order to adapt to variations in workload or in resource capacity, changes brought about by replacement of human resources, changes in the frequency of certain types of (problematic) cases, or exceptions that in some cases give rise to new workarounds that over time solidify into norms. Undocumented process changes like those described above may over time affect process performance.

In this setting, process analysts and managers require methods and tools that allow them to detect and pinpoint process changes as early as possible. *Business process drift* *detection* [1–5] is a family of process mining techniques to detect changes based on observations of business process executions recorded in *event logs* consisting of *traces*, each representing one execution of the business process.

Existing methods for business process drift detection are based on the idea of extracting *features* (e.g. patterns) from traces. One possible feature is for example that task A occurs before task B in the trace, while another type of feature is for example that B occurs more than once in the trace. To achieve a suitable level of accuracy, these techniques either explore large feature spaces automatically or they require the users themselves to identify the specific features that are likely to characterize the drift – implying that the user already has an *a priori* idea of the characteristics of drift. In all cases, these methods may miss certain types of changes that are not covered by the types of features employed. Furthermore, the scalability of these techniques is hindered by the need to extract and analyze a potentially large set of high-dimensional feature vectors. As a result, existing techniques are not suitable for real-time drift detection.

This paper proposes a fully automated and scalable method for detecting concept drift in business process event logs. The core idea is to perform statistical hypothesis testing over the distributions of runs observed in two consecutive time windows. The underpinning assumption is that if a change occurs at a given time point, the distribution of runs before and after this time point will be statistically different, provided that the number of traces in the time window is sufficiently large for statistical testing. By adaptively sizing the window, the method strikes a trade-off between classification accuracy (F-score) and drift detection delay. The proposed method has been empirically evaluated on synthetic and real-life logs in order to assess its accuracy and scalability.

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

2 Related Work

Bose et al. [1, 3] propose a method to detect process drifts based on statistical testing over feature vectors. This method is however not automated. Instead, the user is asked to identify the features to be used for drift detection, implying that the user has some knowledge of the possible nature of the drift. Furthermore, given the types of features supported, this method is unable to identify certain types of drifts such as inserting a conditional branch or a conditional move. Finally, this method requires the user 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 addressed 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 latter method requires that the user sets 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.

Accorsi et al. [5] propose a drift detection method based on trace clustering. The idea is to cluster the traces based on the average distance between each pair of activities in the traces. Similar to Bose et al. [1, 3], this method heavily depends on the choice of window size, such that a low window size leads to false positives while a high window size leads 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 changes to the distances between activity pairs, e.g. changes involving an activity being skipped.

Carmona et al. [2] propose another process drift detection method based on an abstract representation of the process 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 prefixes of traces against the constructed polyhedron. If a significant number of these prefixes do not lie in the polyhedron, a drift is declared. To find a second drift after the first one, the entire detection process has to be executed from the start, thus hindering on the scalability of the method. In experiments we conducted with the logs used in Sections 4 and 5, the implementation of this method took hours to complete. Another drawback of this method is its inability to pinpoint the exact moment of the drift.

Burattin et al [6] address the problem of online discovery of process models from event streams. The goal is to discover a process model from the log and to update the discovered process model as new events are produced. The authors adapt an automated process discovery method, namely the Heuristics Miner, so as to handle incremental updates. Our proposal is complementary 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.

The problem of drift detection has also been studied in a broader context in the field of data mining [7], 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 tasks (concurrency, conflict, 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 when there is a statistically significant difference between the observed process behavior before and after this point*. A key design choice to turn this formulation into a decision procedure is to define what we mean by a *difference in the observed process behavior*. If we turn around this problem, the question becomes *when are two processes the same?* [8]. A number of equivalence notions have been proposed to address this question, borrowed from the field of concurrency theory [9]. One widely accepted notion of process equivalence is *trace equivalence*: two processes are the same if they have the same set of traces, thus they are different if their set of traces exhibits a (statistically significant) difference. However, this trace-based representation can be over-sensitive in our context because it does not capture concurrency. Indeed, any significant variation in the frequency of relative ordering of two activities that are anyways in parallel is treated as a drift. For example, if two activities b and c are in parallel, any significant variation in the frequency of occurrence of b followed by c vs. c followed by b gives rise to a drift, even though the parallel relation between these activities still holds. From this perspective, a more suitable approach is to reason in terms of runs (a.k.a. configurations) of a process, where concurrency is explicitly captured. For example, the two traces abcd and acbd characterize the process where a is followed by b and c in parallel and these are followed by d. In a run-based representation, only one run is needed to represent both traces: the run where a is followed by b and c in parallel and these are followed by d. As business processes typically contain concurrent activities, we opt for a run-based representation of logs and thus a notion of run-equivalence, known as *configuration equivalence* or *pomset equivalence* [9].

Given the above, we map the problem of process drift detection to that of finding a time point such that the set of runs before this point is statistically different from the set of runs after (for a given time window size). This formulation leads to a two-staged approach. First, we calculate a set of runs from a given sub-log, and then we apply statistical testing to find significant differences between the adjacent sets of runs. The next two sub-sections discuss these two stages in turn, while the third sub-section discusses the window size.

3.1 From event logs to partial order runs

An event log consists of a set of traces, each capturing the sequence of events for a given case of the process ordered by timestamp. For example, $L = [\sigma_1^2, \sigma_2^3]$, where $\sigma_1 = \langle a, b, c, d \rangle$ and $\sigma_2 = \langle a, c, b, d \rangle$, defines a log containing 5 traces and a total of 20 events (for simplicity we used the simple event log representation [10]). It is formally defined as follow:

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

While a trace defines a *total order* of events, encoding the concurrency relationship in it results into a partial order run. For simplicity, in the following we formalize the concurrency relationship using the *Alpha concurrency* from [11]. It is possible however to use a more accurate definition of concurrency such as the *Alpha*+ [12] or *Alpha*++ [13], or the one proposed in [14]. These alternative definitions do not suffer from the issue of confusing concurrency with short loops [12].

Definition 2 (Alpha concurrency). *Let L be an event log over the set of event labels $\mathcal{L}$ and $\sigma \in L$ be a log trace. A pair of tasks with labels $a, b \in \mathcal{L}$ are said to be in* alpha directly precedes relation*, denoted $A \prec_{\alpha(L)} B$, iff there exists a trace $\sigma =$*

$\langle \lambda(e_0), \lambda(e_1), \ldots, \lambda(e_{n-1}) \rangle$ in L, such that $A = \lambda(e_i)$ and $B = \lambda(e_{i+1})$. We say that a pair of tasks $A, B \in \mathcal{L}$ are alpha concurrent*, denoted $A \parallel_{\alpha(L)} B$, iff $A \prec_{\alpha(L)} B \wedge B \prec_{\alpha(L)} A$.*

Note that the *Alpha concurrency* is a symmetric relation, and is applied over labels and not over event occurrences. For instance, we can identify that b $\prec_\alpha$ c from trace $\sigma_1 = \langle a, b, c, d \rangle$, and c $\prec_\alpha$ b from trace $\sigma_2 = \langle a, c, b, d \rangle$. Therefore, b and c are considered to be parallel, noted $b \parallel_\alpha c$.

In the following, we assume there exists an oracle χ which provides the concurrency relation $\parallel_\chi$. We will consider that $\parallel_\chi = \{(e, e') \mid \lambda(e) \parallel_{\alpha(L)} \lambda(e')\}$ for an event log L and its alpha concurrency relation $\parallel_{\alpha(L)}$.

As mentioned before, based on the concurrency relationship a trace is (losslessly) transformed to a *partial order* representation of its events. Definition 3 describes formally how, given a relation $\parallel_\chi$, a trace can be transformed into a partially ordered run.

Definition 3 (Transformation of a trace into a run). *Let L be an event log over the set of event labels $\mathcal{L}$ and $\parallel_\chi$ be the concurrency relation provided by an oracle χ. Moreover, let E be a set of event occurrences, $\lambda : E \to \mathcal{L}$ a labelling function. We say that event e_i* directly precedes *event e_{i+1}, denoted $e_i \lessdot e_{i+1}$, iff there exists a trace $\sigma = \langle \lambda(e_0), \ldots, \lambda(e_1), \ldots, \lambda(e_{n-1}) \rangle$ in L with an order $i \in [0, n-1]$. Therefore, the tuple $\pi = \langle E_\pi, \leq_\pi, \lambda_\pi \rangle$ is the* partially ordered run *corresponding to trace σ, induced by the concurrency relation $\parallel_\chi$ and the directly precedes relation $\lessdot$, where:*
- *E_π is the set of events occurring in σ,*
- *$\leq_\pi$ is the causality relation defined as $\leq_\pi = E_\pi^2 \cap (\lessdot^+ \setminus \parallel_\chi)^*$,*[4] *and*
- *$\lambda_\pi : E_\pi \to \mathcal{L}$ is a labelling function, i.e. $\lambda_\pi = \lambda|_{E_\pi}$.*

We write $\Pi_\chi(L)$ to denote the set of all partially ordered runs induced by $\parallel_\chi$ over the set of traces in L.

In order to illustrate the operation of building a run from a trace, let us consider the example event log L and apply the definition step-by-step. We first compute the directly precedes relationship $\lessdot$ by representing the sequencing captured by the event traces, resulting in the set $\{(a_{\sigma_1}, b_{\sigma_1}), (b_{\sigma_1}, c_{\sigma_1}), (c_{\sigma_1}, d_{\sigma_1}), (a_{\sigma_2}, c_{\sigma_2}), (c_{\sigma_2}, b_{\sigma_2}), (b_{\sigma_2}, d_{\sigma_2})\}$. Second, we compute the (irreflexive) transitive closure $\lessdot^+$ by adding to the previous set the following new relations: $\{(a_{\sigma_1}, c_{\sigma_1}), (b_{\sigma_1}, d_{\sigma_1}), (a_{\sigma_1}, d_{\sigma_1}), (a_{\sigma_2}, b_{\sigma_2}), (c_{\sigma_2}, d_{\sigma_2}), (a_{\sigma_2}, d_{\sigma_2})\}$. Third, we compute the concurrency relation $\parallel_\chi$, and obtain $\{(b, c)\}$. Forth, we compute the causality relation $\leq_{\pi_1}$ for the run π_1 corresponding to trace σ_1 by computing set $\lessdot^+ \setminus \parallel_\chi$, which leads to removing the relation $(b_{\sigma_1}, c_{\sigma_1})$. Similarly, we remove $(c_{\sigma_2}, b_{\sigma_2})$ for $\leq_{\pi_2}$ for run π_2 from σ_2. Finally, we remove the unnecessary transitive relations $(a_{\sigma_1}, d_{\sigma_1})$ for π_1 and $(a_{\sigma_2}, d_{\sigma_2})$ for π_2 by applying the transitive reduction of the causality relation.

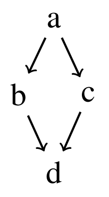

Fig. 1: Example of a run (π_1)

The result of this transformation applied on $\sigma_1 = \langle a, c, b, d \rangle$ is the run π_1 defined by the following causality relation $\leq_{\pi_1} = \{(a, b), (a, c), (b, d), (c, d)\}$ implicitly inferring

[4] $+$ indicates the transitive closure and $*$ indicates the transitive reduction of a relation.

that $b \parallel_{\chi} c$ (cf. Figure 1). Since each transformation of a trace in L results in the exact same run represented by π_1, we obtain that $\Pi(L) = \left[\pi_1^5\right]$.

Armed with this definition of run, we treat an event log as a continuous stream of traces. For each new trace we transform it to a run based on the alpha relationship that is dynamically computed on the basis of the traces observed until that point. Thus, the stream of traces is transformed into a stream of runs.

3.2 Statistical testing over runs

In order to detect a drift in a stream of runs, we monitor any statistically significant change in the distribution of the most recent runs. This test is done on two populations of the same size built from the most recent runs in the stream. Basically the most recent runs are divided into a *reference* (less recent) and a *detection* (more recent) populations. Then, we evaluate the statistical hypothesis of whether or not the reference and detection populations are similar.

In this regard, we define two juxtaposed sliding windows, namely the reference and detection windows of length w, forming together the composite window of $2w$ most recent runs. Figure 2 depicts the two sliding windows over a stream of runs with a drift point. For every new run is observed in the stream, we slide both the reference and detection windows to the right in order to read the new run and perform a new statistical test. We keep iterating this process as long there are new runs observed in the stream.

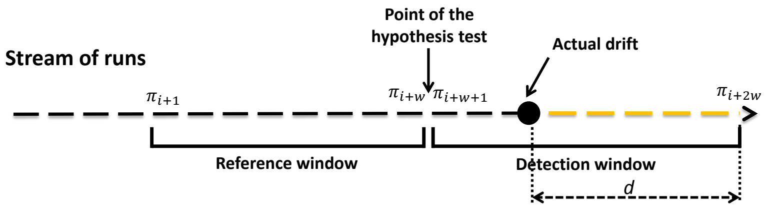

Fig. 2: Statistical test over two sliding windows

Since there is no a-priori knowledge of the run distributions (and their parameters) within the reference and detection windows population, we apply a non-parametric hypothesis statistical test. Moreover, given that an observation of the statistical variable is a run, the statistical test has to be applicable to a categorical variable. For these reasons, we selected the Chi-square test of independence between two variables.

The goal of a two-variable Chi-square test is to determine whether the reference variable and detection variable are similar. The reference variable (resp., the detection variable) is represented by the observations from the reference window (resp., the detection window). A contingency matrix is built to report the frequencies of each distinct run in each window. The Chi-square test is performed on this contingency matrix. The result of the test is the significance probability (the $P{-}value$). A drift is detected when the $P{-}value$ is less than the significance level α (the threshold), and localized at the point of juxtaposition of the reference and detection windows. The value of α is set to the typical value of the Chi-square statistical test, which is 0.05 [15].

The *delay* d shown in Figure 2 is a notion from concept drift in data mining [16]. It is not the distance between the actual drift and the location where the drift is detected. Rather, it indicates how long it takes for the statistical test to detect the drift after it has occurred, and is measured as the number of runs between the drift and the end of the detection window.

Since any statistical test is subject to sporadic stochastic oscillations, we introduced an additional filter to discard abrupt drops in the $P\text{–}value$. An abrupt stochastic oscillation is caused by the noise present in the event log, e.g. in the form of infrequent events or data gaps. Accordingly, we detect a drift only if a given number ϕ of successive statistical tests have a $P\text{–}value < \alpha$. In other terms, a persistent $P\text{–}value$ under the threshold is much more reliable than a sparse value happening abruptly. Our tests showed that a value of ϕ equal to $w/3$ provides the best results in terms of accuracy. More sophisticated approaches to filter out stochastic oscillations are however available, e.g. from the financial domain [17], and could be used instead.

The only independent parameter that needs to be manually set is the window size w. Below we discuss a technique to automatically modify this parameter as new runs are observed at runtime.

3.3 Adaptive window

As discussed in Section 2, the choice of window size is critical in any drift detection method as a small window size may lead to false positives while a large one may lead to false negatives as well difficulty in locating the exact point of the drift. Our method strikes this trade-off by adapting the window size in order to have a more reliable statistical test. It is inspired by [18], where the authors provide rigorous guarantees on the performance of the adaptive window technique.

Our method is motivated by the fact that a low variation does not need too many data points to remain statistically representative, whereas a higher variation would need more data points to be statistically representative. In other words, if a high (resp. low) variation is captured within the composite window then we will need more (resp. less) observations to statistically express this distribution, and this is done by increasing (resp. decreasing) the window size.

The variation (named variability as well) of a statistical variable is a concept that aims to measure the dispersion of the observations. Regarding categorical data, [19] defined a set of properties and proposed a set of measures of variability (that can be alternatively used). We simply measure the variability of a given composite window by dividing the number of distinct runs (categories) by the number of the runs in this composite window (number of observations). In order to keep this rate constant from a statistical test to the next one, then the sliding composite window size needs to be adjusted if the number of distinct runs varies. Thus the evolution of the distinct number of runs over two consecutive statistical tests is captured and replicated on the window size based on a simple cross-multiplication.

Formally, given two consecutive statistics tests T_1 and T_2, the $evolution\ ratio$ between T_2 and T_1 is defined as the ratio between the numbers of distinct runs in the composite window of T_2 over the number of distinct runs in the composite window of T_1. If the $evolution\ ratio$ is equal to 1, this means that there was no evolution in the variation between T_1 and T_2. However, an $evolution\ ratio$ less than 1 means that there is less variation in the T_2 composite window as compared to T_1, whereas an $evolution\ ratio$ greater than 1 means the opposite.

The composite window size is adjusted according to the $evolution\ ratio$, specifically the new window size is equal to the current size multiplied by the ratio (cross-multiplication), i.e. $nextWindowSize = currentWindowSize \cdot evolutionRatio$. Every time that the reference and detection windows are shifted forward to incorporate a new run in the stream, the method adjusts the window size based on this formula. In order to initialize the procedure, we start with a given window size, which can be set empirically as discussed in the next section.

4 Evaluation on synthetic logs

We implemented the proposed method on top of the Apromore platform[5] and used this tool to assess the goodness of our method in terms of accuracy and scalability in a variety of settings. This tool can read a complete event log or a continuous stream of event traces. Each new trace is used to dynamically update the alpha-relationships for each pair of events, and then transformed to a partial order run, resulting in a stream of runs. This stream of runs is then used as input for the statistical test.

4.1 Setup

To assess accuracy we used two established measures in concept-drift detection in data mining [16], namely the *F-score*, measured as the harmonic mean of recall and precision, and the *mean delay*. The latter, computed as the average number of log traces after which a drift is detected, not only measures how late we detect the drift with regard to where it actually happens, but it also indicates how far in the log traces are read to be able to detect a drift.

To simulate the presence of a drift in a log, we generated a benchmark of 72 event logs by varying different parameters as follows. First, we used a textbook example of a business process for assessing loan applications [20] as the “base” model. This model, illustrated in Figure 3, has 15 activities, one start event and three end events, and exhibits different control-flow structures including loops, parallel and alternative branches.

Next, in order to assess the ability of our method to detect drifts determined by different types of control-flow changes, we systematically altered the base model by applying in turn one out of twelve *simple change patterns* described in [21].[6] These patterns, summarized in Table 1, describe different change operations commonly identified in business process models, such as adding, removing or looping a model fragment, swapping two fragments, or parallelizing two sequential fragments.

Further, in order to emulate more complex drifts, we organized the simple changes into three categories: Insertion (“I”), Resequentialization (“R”) and Optionalization (“O”) as shown in Table 1, so as to give rise to six possible *composite change patterns* by randomly applying one pattern from each category in a nested way (“IOR”,

[5] Available at `http://apromore.org/platform/tools`

[6] Non-applicable patterns such as inlining or extracting a subprocess were excluded.

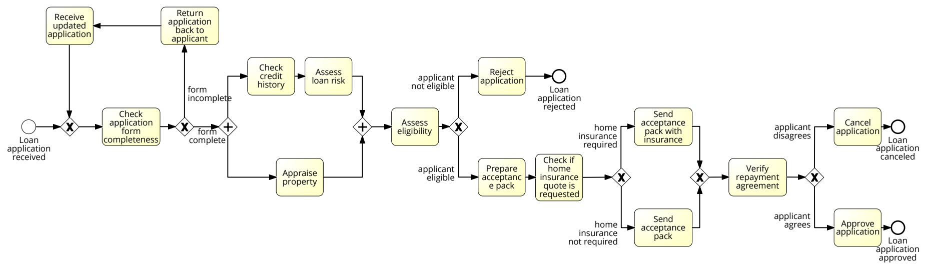
Receive updated application
Return application back to applicant
Loan application received
Check application form completeness
form incomplete
form complete
Check credit history
Assess loan risk
Appraise property
Assess eligibility
applicant not eligible
applicant eligible
Reject application
Loan application rejected
Prepare acceptanc e pack
Check if home insurance quote is requested
home insurance required
home insurance not required
Send acceptance pack with insurance
Send acceptance pack
Verify repayment agreement
applicant disagrees
applicant agrees
Cancel application
Loan application canceled
Approve application
Loan application approved

Fig. 3: Base BPMN model of the loan application process

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

Table 1: Simple control-flow change patterns

"IRO", "OIR", "ORI", "RIO", "ROI"). For example, the composite pattern "IOR" was obtained by first adding a new activity ("I"), then making this activity in parallel with an existing activity ("O") and finally by putting the whole parallel block into a loop structure ("R").

Finally, in order to vary the distance between drifts in the log, we generated four logs of 250, 500, 750 and 1,000 traces for the "base" model as well as for each of the 18 "altered" models, using the BIMP simulator,[7] and combined each group of 5 base logs with each group of 5 altered logs by alternating base and altered logs, in order to obtain four logs of sizes 2,500, 5,000, 7,500 and 10,000 traces for each of the 18 change patterns, leading to a total of 72 logs.[8] Figure 4 depicts an application of this operation to generate a log of 5,000 traces. Each log has 9 drifts located at multiples of 10% of the log size, thus with an inter-drift distance ranging from 250 to 1,000 traces (500 in the example). Knowing the number and position of each drift in the logs provides a gold standard against which we can evaluate the accuracy of our method.

[7] `http://bimp.cs.ut.ee`

[8] All the BPMN models used for simulation, the synthetic logs and the detailed evaluation results are available with the software distribution

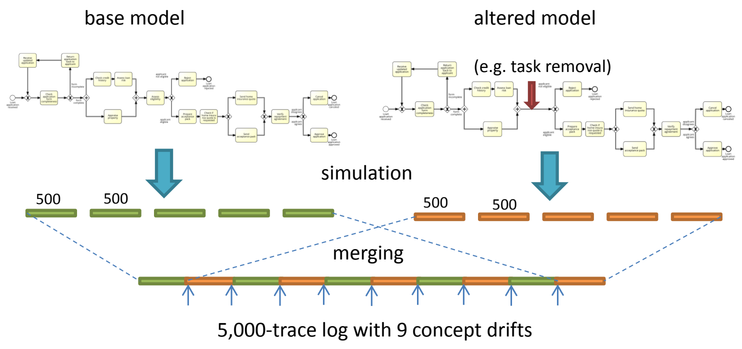

Fig. 4: Event log generation with embedded concept drift

4.2 Impact of window size on accuracy

First, we evaluated the impact of the window size on accuracy. For this, we executed our method with different fixed window sizes ranging from 25 to 150 traces in increments of 25, against each of the 72 logs. Figure 5.a reports the F-score obtained with the four log sizes (2,500 to 10,000 traces), where for each log the F-score was averaged over the logs produced by the 18 change patterns. We observe that the F-score increases as the window size grows and eventually plateaus at a window size of 150. As expected, the more data points are included in the reference and detection windows, the more reliable is the statistical distribution, and thus the more accurate is the statistical test, leading to the detection of all concept drifts (recall of 1), with few or no false positives (precision of 0.9 or above).

Not surprisingly, for a window size of 25 traces, the F-score is low (around 0.45). This is because the Chi-square does not converge if more than 20% of the data points have frequency below 5 [22], which is often the case with a window size of 25 traces, where the distinct runs might be as low as 5-10. This results in both low recall and precision. The drop in F-score at a window size of 150 for logs of 2,500 traces is not an inherent limitation of our method, but is due to having set a drift every 10% of the log, which equates to 250 traces for a log of 2,500 traces. Given that with a window size of 150 traces reference and detection windows aggregate 300 traces, in certain cases two drifts will be included within this set of traces. As a result, the method will treat the two drifts as one leading to a low recall.

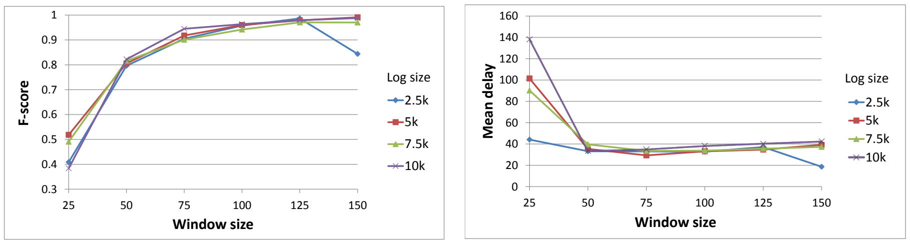

Fig. 5: F-score (a) and mean delay (b) obtained with different fixed window sizes.

Figure 5.b plots how the mean delay varies based on different window sizes, where the mean delay is averaged over the logs produced by the 18 different change patterns, according to the four log sizes. Interestingly, after an initial high mean delay, due to the unreliability of the statistical test with low numbers of data points, the mean delay grows very slowly as the window size increases. This shows that the method is very resilient in terms of mean delay to increases in windows size, having a relatively low delay of around 40 traces when the window size is 50 or above. Similar to the results for F-score, we observe a drop in the mean delay at a window size of 150, for logs of 2,500 traces. This positive effect is due to the second drift in the composite window of 300 traces being discovered before it happened with regards to the gold standard.

In summary, our method achieves high levels of accuracy both in terms of F-score (above 0.9) and mean delay (below 40 traces) in the presence of different types of drift and for different log sizes. This happens when employing a fixed window size that is at least 75 traces long, with the best trade off between F-score and mean delay being achieved with windows of 100 traces.

We also conducted experiments using the trace-based representation of logs (instead of the run-based one). We observed that the obtained accuracy with the trace-based representation was consistently lower than the one with runs. This observation confirms the intuition discussed in Section 3.

4.3 Impact of adaptive window size on accuracy

Next, we assessed the impact of the adaptive window method on F-score and mean delay. For this, we compared the results obtained with the fixed window size shown in Figure 5, averaged over the three log sizes of 5,000, 7,500 and 10,000 traces, with the results obtained using an adaptive window. For example, we compared the results obtained with a fixed window size of 25, with those obtained with an adaptive window initialized to 25 traces. We did not use the log size of 2,500 traces to avoid the effects of the interplay between window size and number of drifts observed in logs of this size in the previous tests.

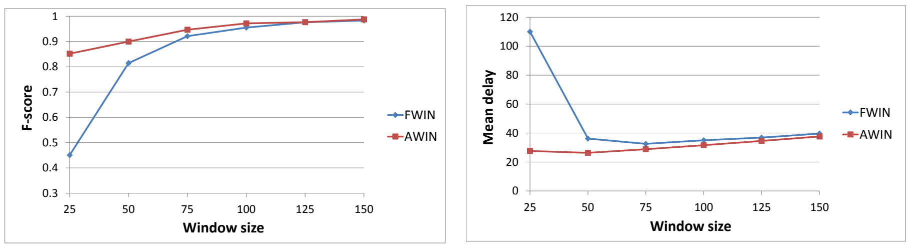

Fig. 6: F-score (a) and mean delay (b) obtained with different fixed window sizes (FWIN) vs. adaptive window sizes (AWIN).

Figure 6 reports the results of this comparison for F-score (a) and mean delay (b). The adaptive window method outperforms the fixed window method both in terms of

F-score and mean delay. Indeed, the ability to dynamically change the window size based on the variation observed in the log (measured as the ratio between number of distinct runs and total number of runs in the combined window), allows us to obtain an adequate number of runs (not too small, not too large) in the reference and detection windows to perform the statistical test. This leads to a higher F-score, since more data points are automatically added to the window when the variation is high. At the same time, it leads to a lower mean delay as the window size is shrank when the variation is low, since in these cases a low number of runs is sufficient to perform the statistical test. As an advantage, the adaptive window method overcomes the low accuracy (both in terms of F-score and mean delay) obtained when fixing the window size to values as low as 25 traces (F-score of 0.85 instead of 0.45, and mean delay of 28 instead of 110). This enables the method to be employed in those scenarios where the distance between drifts in the log is expected to be very low (i.e. in the presence of very frequent drifts) and thus keeping the mean delay as low as possible becomes essential to identify as many drifts as possible.

4.4 Accuracy per change pattern

As a further test on accuracy, we evaluated the relative levels of F-score and mean delay for each of the twelve simple change patterns and the six composite change patterns. For this we fixed the window size to 100 traces, which proved to provide the best trade off in terms of F-score and mean delay, and averaged the results obtained with the fixed window, and with the adaptive window initialized to 100 traces, over the three log sizes of 5,000, 7,500 and 10,000 traces.

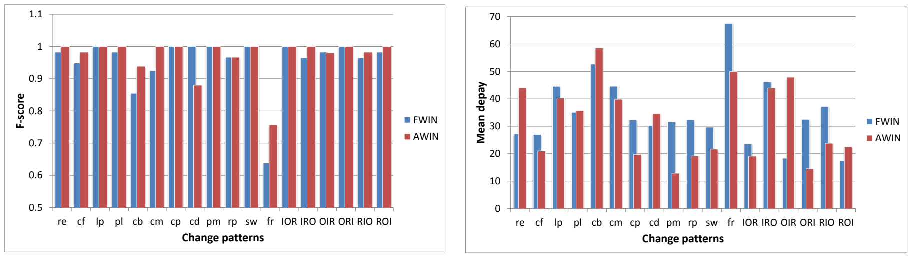

Fig. 7: F-score (a) and mean delay (b) per change pattern, obtained with fixed window size of 100 (FWIN) vs. adaptive window size initialized to 100 (AWIN).

Figure 7 shows the results. From these we can draw the following observations. First, the adaptive window method enhances F-score and mean delay for the majority of patterns (16 out of 18 for F-score and 12 out of 18 for mean delay), with the F-score often being 1. Second, the method experiences a sensibly lower F-score both for fixed and adaptive windows for the *frequency change* pattern (“fr”). This pattern modifies the frequency of certain event relations in the log. The low F-score is due to a low precision (lots of false positives). This is because our method is sensitive to frequency changes caused by the stochastic interference present in an event log. For example, even if the probabilities of taking two alternative branches in a process are observed to be 50% each in the entire log, when looking at an individual window, which is a small extract of the log, these probabilities are likely to be slightly different (e.g. they could be 40%-60% instead of 50%-50%). This interference tricks the detection of a frequency-based drift, but can be resolved by choosing a larger window size. For example, using a fixed window of 200 traces, we obtain an F-score of 0.98 (1 if using the adaptive window) for the "fr" pattern.

4.5 Execution times

We conducted all tests on an Intel i7 2.20GHz with 16GB RAM (64 bit), running Windows 7 and JVM 8 with standard heap space of 512MB. The time required to update the alpha-relationships, extract the runs, and perform the Chi-square test, ranges from a minimum of 0.26 milliseconds to a maximum of 2.3 milliseconds with an average of 0.5 milliseconds. These results show that the method is suited for online concept drift detection, including scenarios where the inter-arrival time between completed traces is in the order of milliseconds.

4.6 Comparison with baseline

Lastly, we compared the results obtained by our adaptive window method , with those obtained by the method of Bose et al. [1, 3], since this is the most mature method for process drift detection available at the time of writing. Thus, we used the synthetic logs that we had previously generated for each of the 18 change patterns, set the window size to 100 and averaged the results over the three different log sizes of 5,000, 7,500 and 10,000 traces.

As discussed in Section 2, the method in [1, 3] has the disadvantage that it requires to manually select the order relations between event labels to be used as features to build the feature space which in turn is required to detect the drifts. Thus, knowing the specific changes made in the altered models, we manually selected the most appropriate features for each log. Figure 8 shows the results of the comparison.

Our method outperforms the method in [1, 3] both in terms of F-score and mean delay, achieving substantial F-score differences for ten change patterns, including "lp" (make fragment loopable/non-loopable), "cp" (duplicate fragment), "pm" (move fragment into/out of parallel branch) and composite patterns such as "IOR" and "RIO". This is due to the large number of false positives identified by method in [1, 3]. Further, this method fails to identify drifts based on the following changes: "cb" (make fragment skippable/not skippable) and "cm" (move fragment into/out of conditional branch), even if appropriate features are chosen.

As a final test, we selected all features available from each log in order to simulate a fully-automated application of this method. However in this case the method fails to identify any drift due to a high level of false negatives, and construction of the feature space becomes an expensive task (over 15 minutes with window size of 100 traces).

5 Evaluation on real-life log

We employed our method to detect concept drifts in an event log originated from the claims management system of a large Australian insurance company. The log consists

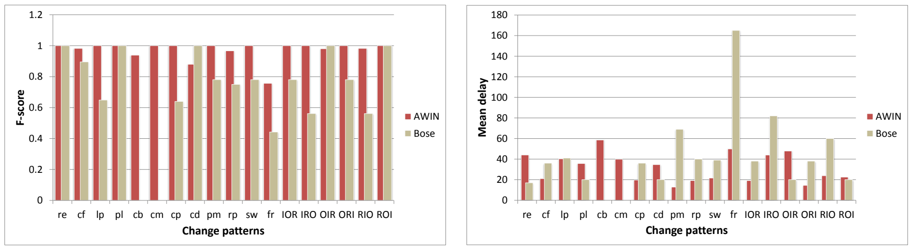

Fig. 8: F-score (a) and mean delay (b) per change pattern, obtained with our adaptive window method with size initialized to 100 (AWIN) vs. [1, 3] with fixed window size of 100 (BOSE).

of 4,509 traces with 29,108 total events of which 12 are distinct events. It records claim handling processes for motor insurance that were performed over a period of 13 months between 2011 and 2012.

We initialized the adaptive window to 100 traces. The method took 4.51 seconds to check the whole log and returned three drifts at 1,769, 1,911 and 3,763 traces, as shown by the results of the Chi-square test in Figure 9.a. In this plot we can also see a number of stochastic oscillations that were automatically filtered out by our method, as described in Section 3.2.

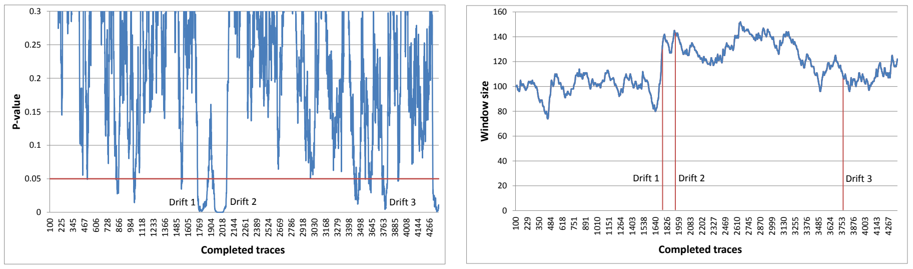

Fig. 9: Plot of the Chi-square test results (a) and adaptive window size (b).

We then validated the results with a business analyst from the insurance company, who confirmed that the three drifts correspond to a new major release (Drift 1) and two minor releases (Drifts 2 and 3) of the claims management system. These releases led to various changes in the claim handling process supported by the system, e.g. the removal of a manual task for reviewing the claim correspondence and the replacement of a manual task for checking the invoice with an automated one, with the purpose of reducing the total number of open claims. The effects of these changes are confirmed by the distribution of the number of active cases over the log timeline, shown in Figure 10, which we have annotated with the position of the drifts identified by our method and the delays in reporting these drifts. We can see that each drift is associated with a drop in the number of active cases, which confirms the effectiveness of the new releases on process performance.

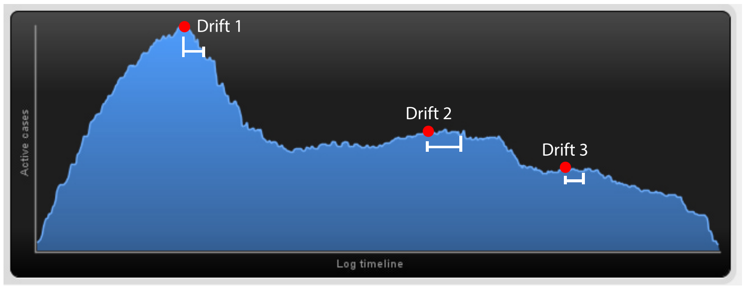

Fig. 10: The position and delay of the three drifts identified by our method, noted on active cases over log timeline

The delay in detecting the first two drifts is longer than the delay in detecting the last drift. This is due to a higher level of variation in the first part of the log (due to the more manual nature of the business process), which led our method to increase the size of the adaptive window. This is confirmed by Figure 9.b, which shows how the window size varies according to the number of completed traces. Here we can see that the detection of Drift 1 and 2 is associated with a larger window size (131 and 143) than the size used to detect Drift 3 (size 109).

6 Conclusion

The paper proposed a fully automated method for business process drift detection based on statistical testing of distributions of runs. The proposed method – especially in its "adaptive window" variant – accurately discovers typical process changes and combinations thereof, consistently outperforming a state-of-the art baseline. The evaluation results on a complex real-life log demonstrate the method's ability to detect drifts that correspond to user-recognizable process changes, as well as its scalability. The execution times in the order of milliseconds make it applicable for online drift detection.

In its present form, the proposed method treats event logs as consisting of sequences of event labels. In doing so, it does not take into account process execution data and resource allocations – usually encoded as event payloads. An avenue for future work is to make the method data-aware.

Another avenue for future research is to enhance the method in order to provide input to the user to understand the process change(s) underpinning a detected drift. One possibility to explain a drift is to present to the user the runs with the highest frequency differentials between the reference and the detection windows. This input may help the user to gain a partial and initial understanding of the process change(s), but it is unlikely to provide a comprehensive picture in the case of complex business processes. A possible direction to tackle this problem is to apply automated process discovery before and after the drift, and to use a process model comparison technique [23] in order to derive a diagnostics of the differences between the discovered pre-drift and the post-drift process models.

Acknowledgments. NICTA is funded by the Australian Government via the Department of Communications. This research is funded by the Australian Research Council Discovery Project DP150103356 and the Estonian Research Council.

References

1. Bose, R.J.C., van der Aalst, W.M., Žliobaitė, I., Pechenizkiy, M.: Handling concept drift in process mining. In: Proc. of CAiSE, Springer (2011) 391–405
2. Carmona, J., Gavalda, R.: Online techniques for dealing with concept drift in process mining. In: Advances in Intelligent Data Analysis, Springer (2012)
3. Bose, R.J.C., van der Aalst, W.M., Zliobaite, I., Pechenizkiy, M.: Dealing with concept drifts in process mining. IEEE Transactions on NNLS **25**(1) (2014) 154–171
4. Martjušev, J.: Efficient algorithms for discovering concept drift in business processes. Master's thesis, University of Tartu (2013)
5. Accorsi, R., Stocker, T.: Discovering workflow changes with time-based trace clustering. In: Data-Driven Process Discovery and Analysis. Springer (2012) 154–168
6. Burattin, A., Sperduti, A., van der Aalst, W.M.: Control-flow discovery from event streams. In: Evolutionary Computation (CEC), 2014 IEEE Congress on, IEEE (2014) 2420–2427
7. Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., Bouchachia, A.: A survey on concept drift adaptation. ACM Computing Surveys (CSUR) **46**(4) (2014)
8. Hidders, J., Dumas, M., van der Aalst, W.M., ter Hofstede, A.H., Verelst, J.: When are two workflows the same? In: Proc. of CATS, Australian Computer Society (2005) 3–11
9. van Glabbeek, R.J., Goltz, U.: Equivalence notions for concurrent systems and refinement of actions. In: Proc. of MFCS, Springer (1989)
10. van der Aalst, W.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
11. van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: discovering process models from event logs. IEEE TKDE **16**(9) (2004) 1128–1142
12. de Medeiros, A.K.A., van der Aalst, W.M., Weijters, A.: Workflow mining: Current status and future directions. In: On the move to meaningful internet systems 2003: Coopis, doa, and odbase. Springer (2003) 389–406
13. Wen, L., van der Aalst, W.M., Wang, J., Sun, J.: Mining process models with non-free-choice constructs. Data Mining and Knowledge Discovery **15**(2) (2007)
14. Cook, J.E., Wolf, A.L.: Event-based detection of concurrency. In: Proc. of FSE. (1998)
15. Nuzzo, R.: Statistical errors. Nature **506**(13) (2014) 150–152
16. Ho, S.S.: A martingale framework for concept change detection in time-varying data streams. In: Proc. of ICML, ACM (2005) 321–327
17. Murphy, J.J.: Technical analysis of the financial markets: A comprehensive guide to trading methods and applications. Penguin (1999)
18. Bifet, A., Gavalda, R.: Learning from time-changing data with adaptive windowing. In: SDM. Volume 7., SIAM (2007)
19. Wilcox, A.R.: Indices of qualitative variation. Technical report, Oak Ridge Nat. Lab. (1967)
20. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.: Fundamentals of Business Process Management. Springer (2013)
21. Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features–enhancing flexibility in process-aware information systems. DKE **66**(3) (2008) 438–466
22. Yates, D., Moore, D., Starnes, D.S.: The practice of statistics: TI-83/89 Graphing Calculator Enhanced. Macmillan (2007)
23. Armas, A., Baldan, P., Dumas, M., García-Bañuelos, L.: Behavioral Comparison of Process Models Based on Canonically Reduced Event Structures. In: Proc. BPM. (2014)