A framework for detecting deviations in complex event logs

Li, G.; van der Aalst, W.M.P.

Published in:
Intelligent Data Analysis

DOI:
10.13140/RG.2.1.1848.7928
10.3233/IDA-160044

Published: 19/08/2017

Document Version
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Intelligent Data Analysis 21 (2017) 759–779
DOI 10.3233/IDA-160044
IOS Press

A framework for detecting deviations in complex event logs

Guangming Li* and Wil M.P. van der Aalst
Eindhoven University of Technology, Eindhoven, The Netherlands

Abstract. Deviating behavior within an organization can lead to unexpected results. The effects of deviations are often negative, but sometimes also positive. Therefore, it is useful to detect deviations from event logs which record all the behavior of the organization. However, existing model-based and cluster-based approaches are inaccurate or slow when dealing with complex event logs, i.e. logs of less structured processes having many activities and many possible paths. This paper proposes a novel approach that is faster than cluster-based approaches because it creates a so-called *profile* which is less time-consuming than creating clusters. Furthermore, the approach is also more accurate than model-based approaches because we use an iterative approach to improve the result. Our experiments show that approach outperforms existing techniques in a variety of circumstances.

Keywords: Process mining, deviation detection, clustering, behavioral profiles

1. Introduction

Process mining is a family of techniques to extract knowledge about business processes from event logs which record process executions consisting of different business activities [1]. Process mining techniques are widely used, not only in situations where processes are structured and well-defined (e.g., procurement, finance, and e-government), but also in environments such as healthcare, customer relationship management (CRM) and product development where things are less structured [2]. Such environments often allow for a higher degree of freedom and this may lead to unexpected deviations, e.g., a patient can directly visit a doctor without an appointment in an emergency. Since deviations impact business processes, it is of the utmost importance to detect them from event logs.

Deviation detection is a significant problem which has been explored within diverse research areas and application domains [3], such as detecting failure behavior (e.g., bugs) in software systems [4], detecting fraudulent claims in insurance companies [5] and detecting intrusions in a network [6]. The lion's share of deviation detection done in context of process mining focusses on *conformance checking*. This requires a normative model and therefore knowledge of what constitutes a deviation. In this paper we focus on deviation detection *without* a normative process model and just used the event data to detect deviations.

Deviation detection techniques can be divided into two categories, i.e., model-based approaches and cluster-based approaches. Techniques in the first category basically employ conformance checking method on a discovered process model to detect deviations [7,8]. These techniques first mine an

*Corresponding author: Guangming Li, Eindhoven University of Technology, P.O. Box 513, 5600 MB, Eindhoven, The Netherlands. E-mail: g.li.3@tue.nl.

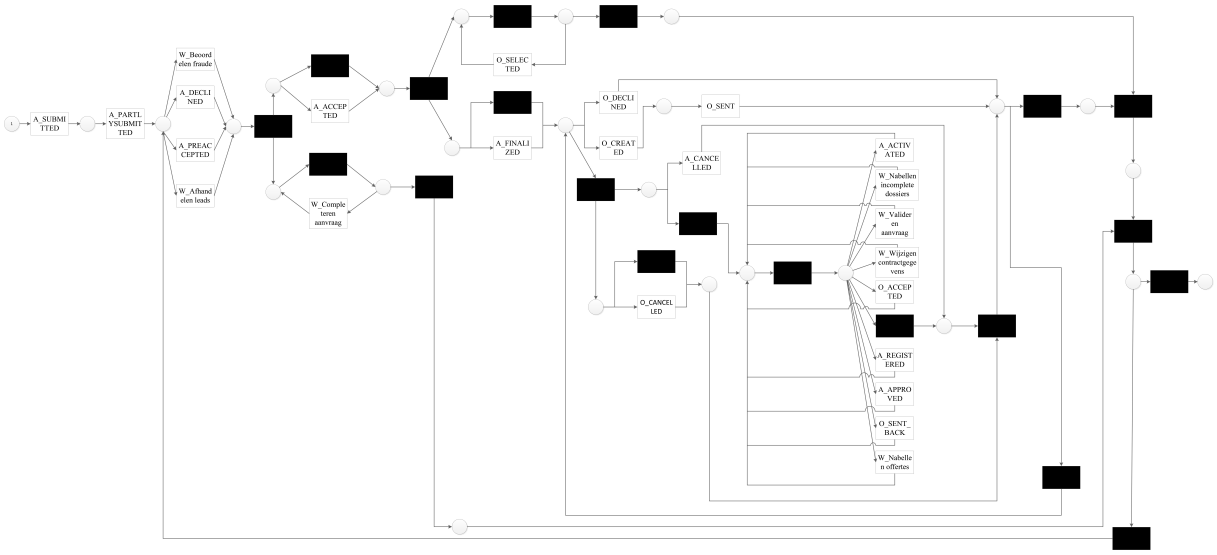


Fig. 1. A model discovered from BPI Challenge 2012.

appropriate model as a reference model and then classify cases which do not fit the model as deviations. This works well on structured processes, but has problems when dealing with less structured processes. It is difficult to specify an appropriate model in this setting due to the high variety of behavior. Within the second category, techniques use clustering [9–11] to detect deviations based on the idea of perceiving cases in small clusters as deviations. In this case there is no conformance checking on a discovered reference model: by grouping similar cases the outliers become visible. Clustering techniques are more suitable for complex processes (i.e., no need to learn a reference model), but they are more time-consuming compared to techniques in the first category. In summary, although there already exist techniques such as the ones mentioned above, deviation detection in less structured environments remains a challenge. As an example, consider a log, recording the behavior for a loan process, extracted from the BPI Challenge 2012.[1] Due to the high variability (13087 cases, 4366 traces and 36 activities), discovered models look like the model in Fig. 1 and cannot serve as a reference model. Clustering on the other hand takes to long. For example, the *clustering* algorithm (cf. Section 6.3) takes 40 minutes on this moderate sized event log. Later we will show that the approach presented in this paper can uncover deviations in a fraction of this time while avoiding the creation of a meaningless reference model.

In order to deal with these challenges, we propose a novel approach, whose basic principle is that a case from a log is a deviation if it is not similar to the collection of mainstream cases in the log.

More precisely, as shown in Fig. 2, (1) we sample the cases (C) from the input log based on a *norm* function (cf. Section 4.1) to get a set of "more normal" cases (denoted as C_S) as mainstream cases. (2) Once we have C_S, the problem of specifying a deviation has been transformed into computing how much a case is similar to C_S. In order to compute the similarity, one first has to figure out what makes a case similar to C_S. Since the case and C_S have different types of characteristics with respect to different perspectives, we create a so-called *profile* [19] to characterize them from specific perspectives. (3) Then we quantify the similarity based on the *profile* and identify normal cases (C_N) and deviating cases (C_D)

[1] http://www.win.tue.nl/bpi/2012/challenge.

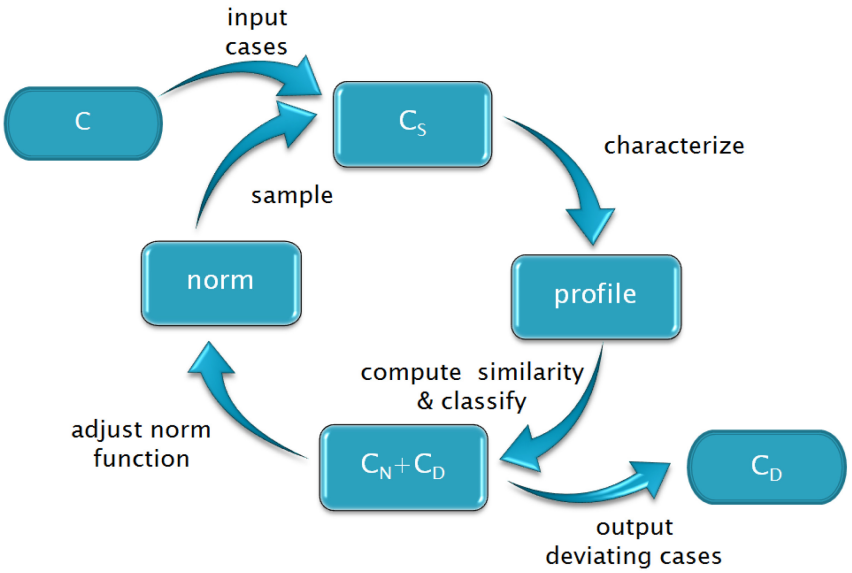


Fig. 2. The framework of detecting deviations.

according to their similarity. (4) Adjust the *norm* function to increase (decrease) the likehood that normal (deviating) cases are sampled. We improve the quality of detected class of deviating cases by iterating the above steps, and return final deviating cases in the last loop.

In summary, our contribution is based on creating a profile, rather than models or clusters, to detect deviations and improving the performance iteratively. Our approach is more accurate than model-based approaches and faster than cluster-based approaches when dealing with complex and less structured logs.

The remainder of the paper is organized as follows. The next section summarizes existing approaches for deviation detection. In Section 3, we introduce preliminaries and provide a definition for the *profile* notion. Section 4 proposes a framework for detecting deviations and we apply it to the control-flow perspective in Section 5. The approach has been implemented in ProM and in Section 6 we evaluate the approach. Finally, Section 7 concludes the paper.

2. Related work

The idea of detecting deviations from event logs is not new and many approaches have been proposed. In this section, we provide an overview of existing approaches and compare them with our approach.

Model-based Approaches. Bezerra and Wainer [12] propose three similar methods to detect deviating cases, i.e., *threshold* ([13] extends it to dynamic thresholds), *iterative* and *sampling* algorithms. Among these methods, the last one gives the best result. It first creates a sample log through sampling a given log and then considers the cases, which do not perfectly fit the model discovered from the sample log, as deviations. In [14], authors propose an approach which consists of five steps: (i) scoping, (ii) process discovery, (iii) filtering of fitting models, (iv) model selection, and (v) splitting of log. The key step in the approach is to select the most appropriate model which is structurally simple and behavioraly specific among fitting models. Similarly [15], employs the genetic algorithm [16] to discover an appropriate

model from a preprocessed log and then classifies the cases which are not instances of the model as deviations.

The model-based approaches typically discover an appropriate model as a reference model, and then use the conformance checking technique to classify cases which do not fit the model as deviations. However, it is a challenging (and often impossible) task to discover and select the appropriate model from a complex log. In any case this requires domain-specific knowledge. In our approach, a *profile* is created to replace the appropriate model to detect deviations, which is much easier and more accurate.

Cluster-based Approaches. Ghionna et al. [17] cluster cases based on frequent patterns extracted from a log, and then treat cases in clusters whose sizes are below a threshold as deviationss [18] performs a hierarchical clustering of a log, in which each case is seen as a point of a properly identified space of features. The method was originally devoted to discovering an expressive process model from each cluster, but we can also use it to detect deviations by classifying cases in small clusters as deviations. Song et al. [19] create a *profile* (which is different from the notion used in this paper) to contain specific attributes of a case and then map a case to a vector based on the *profile*. In this way, the whole log is mapped into a vector space. According to the distances between every two vectors, clusters are generated and cases in small clusters are considered as deviations.

Cluster-based approaches are more suitable for unstructured processes than model-based approaches. However, cluster-based approaches are optimized to find clusters rather than deviations. As a consequence, they are time-consuming due to the time it takes to cluster cases. In contrast, our approach is more efficient since it detects deviations based on the similarity between each case and a sample log.

3. Preliminaries

This paper proposes a novel method based a so-called *profile* which characterizes "normal cases" from specific perspectives. In this section, we first define *event logs* because the definition of a log used in this paper is quite different from the standard notation [1] and then provide a definition for the *profile* notion.

Definition 1 (Universes). *In the remainder we assume the following universes:*

- $\mathcal{A}$ *is the set of all possible activity names,*
- $\mathcal{C}$ *is the set of all possible case (process instance) identifiers.*

Cases (process instances) are represented by a unique identifier. This allows us to refer to a specific case even if two cases have the same trace.

Definition 2 (Event logs). $L = (C, A, \rho)$ *is an* event log *if and only if:*

- $C \subseteq \mathcal{C}$ *is a set of cases (i.e., process instances),*
- $A \subseteq \mathcal{A}$ *is a set of activities, and*
- $\rho \in C \rightarrow A^*$ *maps each case onto a sequence of activities (i.e., a trace).*

$\mathcal{L}$ *is the set of all possible event logs.*

Let $C = \{c_1, c_2, \ldots, c_n\}$ denote a set of cases. $\rho(C) = [\rho(c_1), \ldots, \rho(c_n)]$ denotes the multiset of traces $\rho(c_i)$. $\mathcal{P}(C)$ is the powerset of C, i.e., $C' \in \mathcal{P}(C)$ if and only if $C' \subseteq C$.

Definition 3 (Profile). *Let* $L = (C, A, \rho) \in \mathcal{L}$ *be an event log. A* profiling function *on* L *is a function* $prof^L \in C \times \mathcal{P}(C) \rightarrow [0, 1]$ *that quantifies the similarity between a case and a set of cases (higher is more similar).*

If a profile is based on some feature f, we write $prof_f^L$. A profile can be considered as a set of characteristics extracted from some cases. A profiling function computes the similarity which describes to what extent the case contains characteristics in the set. A profile is configurable notion and is not limited to one specific feature. The user can create multiple profile. Next, we combine profiles to quantify the similarity based on a set of features.

Definition 4 (Combining profiles)**.** *Let $L = (C, A, \rho) \in \mathcal{L}$ be an event log. Let F be a set of features and $w \in F \to \mathbb{R}^+$ a weight function. ${prof}^L{}_{F,w} \in C \times \mathcal{P}(C) \to [0, 1]$ takes the weighted average, i.e.,*

$${prof}^L{}_{F,w}(c, C') = \frac{\sum_{f \in F} w(f) \times prof_f^L(c, C')}{\sum_{f \in F} w(f)}$$

The above definitions do not give concrete functions to quantity the similarity. The profile functions used vary when detecting deviations from different perspectives. In Section 6, we concrete profiles focusing on the control-flow perspective. In the remainder, we will drop the subscripts and simply assume a given profile function $prof^L$.

4. A framework for detecting deviations based on profile

In Section 3, we create a profile to quantify the similarity between a case and C_S to judge if the case is deviating. The quality of the judgement depends on C_S, i.e., it is better if C_S only contains "more normal" cases. Therefore we exploit the idea of iteratively and incrementally refining C_S. By combining this idea with the profile introduced in the previous section, a novel approach for detecting deviations is proposed. The approach is a framework since the profile is configurable by selecting or detecting a suitable profile function, i.e., users can create their own profile to detect deviations from some perspective. Next, we first illustrate the two main steps of the approach, i.e., *sampling* and *classifying*, and then present the whole framework in Section 4.3.

4.1. Sampling

Basically, we create C_S through sampling an event log. In order to make C_S contain "more normal" cases, we need to make these cases more likely to be selected in the sampling step. To this aim, first a function is created to assign a *norm* value to each case based on its normality.

Definition 5 (Norm Function)**.** *Let $L = (C, A, \rho) \in \mathcal{L}$ be an event log, and $norm \in C \to \mathbb{R}^+$ is a* norm *function that assigns a positive value to each case. The higher the value is, the "more normal" the case is. Cases with lower norm values correspond to deviating cases.*

The norm function needs to be initialized by users and we will give more details in Section 4.3. Based on norm values and a given sample size, we employ a function named *sampling* to derive C_S from a log, in which cases with higher norm values are more likely to be included.

Definition 6 (Sampling)**.** *Let $L = (C, A, \rho) \in \mathcal{L}$ be an event log, $norm \in C \to \mathbb{R}^+$ a norm function, and $ss \in \{0 \dots |C|\}$ a sample size. $C_S = sample(L, norm, ss)$ is a random set of cases sampled from C using a relative likelihood based on norm, i.e., $|C_S| = ss$, $C_S \subseteq C$, and $c_1 \in C$ is k times as likely to be included as $c_2 \in C$ if $norm(c_1) = k \times norm(c_2)$.*

Note that the same case cannot be included twice in C_S. Of course, two cases having the same trace may be included in C_S if this is mainstream behavior.

4.2. Classifying

After deriving C_S from a log, we compute the similarity by means of a profile. Then, based on the similarity values of all cases and a given number which indicates the amount of deviating cases, we partition the log into normal cases (C_N) and deviating cases (C_D), and update $norm$ based on the classification, i,e., assign higher (lower) norm values to normal (deviating) cases.

Definition 7 (Classifying). *Let $L = (C, A, \rho) \in \mathcal{L}$ be an event log, $prof^L \in C \times \mathcal{P}(C) \to [0, 1]$ a profiling function, $C_S \subseteq C$ a subset of sampled cases, $norm \in C \to \mathbb{R}^+$ a norm function, and $nd \in \{1 \dots |C|\}$ the number of cases to be marked as deviating. $classify(L, C_S, nd, norm, prof^L) = (C_N, C_D, norm')$ such that*

- *C_N and C_D partition C, i.e., $C = C_N \cup C_D$ and $C_N \cap C_D = \emptyset$,*
- *$|C_D| = nd$,*
- *for any $c_N \in C_N$ and $c_D \in C_D : prof^L(c_N, C_S) \geqslant prof^L(c_D, C_S)$,*
- *$norm' \in C \to \mathbb{R}^+$ such that $norm'(c_N) = norm(c_N) \times R_N$ for $c_N \in C_N$ and $norm'(c_D) = norm(c_D) \times R_D$ for $c_D \in C_D$.*

where $R_N > 1$ and $0 < R_D < 1$ are used to adjust the norm values.

Note that $prof^L$ refers to a generic profiling function. It can be based on any profile or combining profiles as users need, which indicates the algorithm proposed in this section is a generic framework for detecting deviations.

4.3. The algorithm CyclicSC

The classification step detects deviating cases and updates *norm* based on C_S derived in the sampling step, while the sampling step creates C_S based on *norm* updated in the classifying step. The above steps are related to each other and may need to be applied repeatedly to converge. As mentioned in Section 3.2, we want to derive C_S which contains "more normal" cases. To this aim, we propose an algorithm *cyclicCS* which combines and iterates the above two steps to refine C_S, and finally returns the deviating cases.

Definition 8 (Cycling). *Let $L = (C, A, \rho) \in \mathcal{L}$ be an event log, $norm^1 \in C \to \{1\}$ a special norm function that assigns value 1 to each case, $ss \in \{0 \dots |C|\}$ a sample size, $prof^L \in C \times \mathcal{P}(C) \to [0, 1]$ a profiling function, $lt \in \mathbb{N}^+$ a loop threshold, and $nd \in \{1 \dots |C|\}$ the number of cases to be marked as deviating. $cyclicSC(L, ss, lt, nd) = (C_N, C_D)$ based on the following procedure:*
1. Let $i = 0$ and $norm = norm^1$,
2. $C_S := sample(L, norm, ss)$,
3. $(C_N, C_D, norm') := classify(L, C_S, nd, norm, prof^L)$,
4. $norm := norm'$, $i := i + 1$, and return to step 2 if $i \leqslant lt$ or output C_N and C_D if $i > lt$.

The above definition presents the input, output and steps of the *cyclicSC* algorithm, which is also shown in Fig. 3. Initially, we set $i = 0$ to let $norm = norm^1$. In each loop, according to the given ss and the new $norm$ (i.e., generated in the previous loop), a set of cases C_S is selected from C using the function *sample*. Next, based on C_S, nd and $prof^L$, we use the function *classify* to partition C into C_N and C_D, and update *norm* as the input of the function *sample* in the next loop. We iterate the above steps until the ending condition is satisfied, i.e., the number of loops exceeds the threshold lt and then output C_N and C_D.

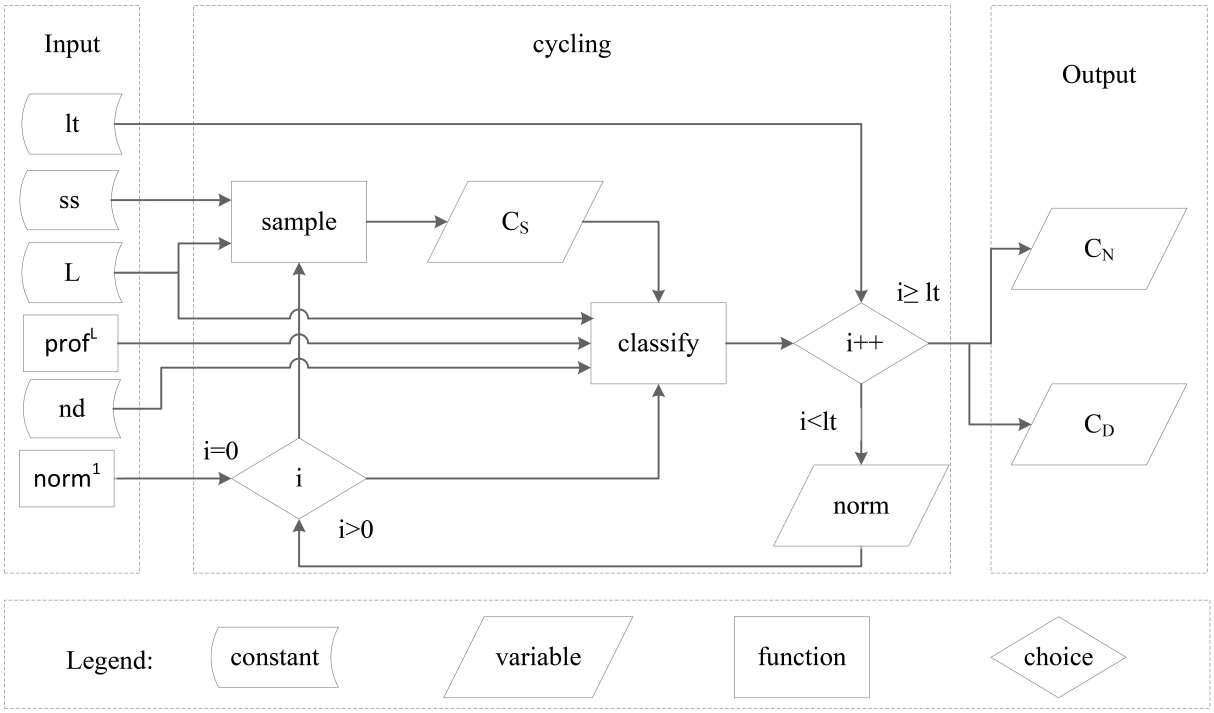


Fig. 3. The flowchart of the algorithm *cyclicSC*.

At the beginning of the algorithm, we need to configure some input constants and functions. In our experiments in Section 6, we let $ss = |C| \times (1 - dp)$ and $nd = |C| \times dp$, where dp is a given value which specifies what fraction of the cases in *L* will be detected as deviating cases. Besides, *lt* also need to be specified to control the number of loops. The algorithm is designed in such a way that each refinement leads to a better C_S in most cases. However, this is not always the case. Our experiments show that the approach tends to converge to a "better" C_s. However, we will also discuss its limitations.

If there is no a-priori or domain knowledge, function *norm* is initialized as $norm^1$ by default, i.e., we assume all the cases are on the same "normal" level. However, the sampling stage can also benefit from a-priori or domain information. For instance, if we know some case is "more normal" in advance, we can assign a "higher" norm value to it, which helps us achieve a better result, since the initial *norm* decides the quality of the first C_S and has a major influence on the final C_S and C_D. On the contrary, if we assign "lower" norm values to "more normal" cases, the detected deviating cases may not be deviating. Function $prof^L$ needs to be created according to the specific application, and in next section, we present how to create it for detecting deviations from the control-flow perspective.

5. Detecting deviations using the control-flow perspective: An application of the framework

Using the framework proposed in Section 4, we can detect deviations from different perspectives through creating different profiles. In this section, we present an application of the framework to detect deviations from the control-flow perspective. Specifically, we create a combined profile $prof^L_{\{df,de\},w^1}$ based on the directly follows relation (*df*, cf. Section 5.1) and the dependency relation (*de*, cf. Section 5.2) using weight function $w^1 \in F \to \{1\}$, i.e., $w^1(df) = w^1(de) = 1$. The results are discussed in

next section. Next to the above two relations, we can use many other relations (e.g., for detecting deviations from other perspectives) where the basic idea of the algorithm still holds. Next, we demonstrate how to create profiles based on the above two relations, respectively.

5.1. Creating a profile based on the directly follows relation

The directly follows relation is a key relation in process mining used in many model discovery methods, such as the α-algorithm. It extracts the characteristics of successive relations between events.

The definition of the directly follows relation in this paper is different from the one in [1], where the relation is in the context of a log and the frequency is not taken into consideration. Here the directly follows relation is in the context of a case (rather than a log) and the frequency is used to quantify the directly follows relation. Specifically, given a set of cases C', we use, for instance, $\#_{C'}(a, b)$ to count the frequency that a is directly followed by b in the multiset of traces $\rho(C')$. Note that a directly follows relation can occur multiple times in the same trace. Next, we create a profile based on the frequency to compute the similarity between a case c and a set of cases C'. The idea is that the similarity is higher if the directly follows relations in $\rho(c)$ have a higher frequency in $\rho(C')$.

Definition 9 (Profile Based On Directly Follows Relation). *Let $L = (C, A, \rho) \in \mathcal{L}$ be an event log. $prof^L{}_{df} \in C \times \mathcal{P}(C) \to [0, 1]$ is the profiling function based on the directly follows relation. With $\#_{C'}(a, b) = \sum_{c' \in C'} |\{1 \leqslant i < |\rho(c')| \mid (\rho(c')_i, \rho(c')_{i+1}) = (a, b)\}|$ and $maxfreq(C') = \max_{a,b \in A} \#_{C'}(a, b)$,*

$$prof^L{}_{df}(c, C') = \begin{cases} \dfrac{\sum_{1 \leqslant i < |\rho(c)|} \#_{C'}(\rho(c)_i, \rho(c)_{i+1})}{(|\rho(c)| - 1) \times maxfreq(C')} & , if\ |\rho(c)| \geqslant 2 \\ 0 & , otherwise \end{cases}$$

In order to better understand the above definition, we use a small example. Consider the set of cases is C_1 with $\rho(C_1) = [\langle a, c, d, f\rangle^{10}, \langle a, b, d, f\rangle^5, \langle a, c, d, e, b, d, f\rangle^5]$, and an example case is c_1 with $\rho(c_1) = \langle a, b, d, f\rangle$. Now: $\#_{C_1}(a, b) = 5$, $\#_{C_1}(b, d) = 10$, $\#_{C_1}(d, f) = 20$ and $maxfreq(C_1) = 20$. The similarity value can be computed as follows: $prof^L_{df}(c_1, C_1) = (5 + 10 + 20)/(3 \times 20) = 0.58$.

5.2. Creating a profile based on the dependency relation

The profile based on the directly follow relation only abstracts characteristics between two successive events. Therefore, it is not enough to adequately represent the control-flow perspective. In order to derive more features, we create another profile based on a so-called *dependency relation* to abstract features between two disconnected events.

Definition 10 (Co-Occurrence Relation). *Let $A \subseteq \mathcal{A}$ be a set of activities. $\sigma = \langle t_1, t_2, t_3, \ldots, t_n\rangle \in A^*$ is a trace. $a \backsim_\sigma b$ if and only if there are $i, j \in \{1, \ldots, n\}$ and $i \neq j$ such that $t_i = a$ and $t_j = b$.*

The co-occurrence relation is commutative. For instance, $a \backsim_\sigma b$ is the same as $b \backsim_\sigma a$ which means both a and b occur in the trace σ. For an example trace $\sigma_1 = \langle a, b, d, f\rangle$, the following trace-based co-occurrence relations can be found:

$$\backsim_{\sigma_1} = \{(a, b), (b, a), (a, d), (d, a), (a, f), (f, a), (b, d), (d, b), (b, f), (f, b), (d, f), (f, d)\}.$$

Based on the co-occurrence relation, we define the dependency relation to reflect the non-local dependency between events.

Definition 11 (Dependency Relation). *Let $L = (C, A, \rho) \in \mathcal{L}$ be an event log and $C' \in \mathcal{P}(C)$.* minConf *and* minSupp *are two threshold values. $a \Rightarrow_{C'} b$ if and only if $freq(a,b)/freq(a) \geqslant$* minConf *and $freq(a,b)/|C'| \geqslant$* minSupp *with $freq(a) = |\{c|c \in C' \wedge a \in \rho(c)\}|$ and $freq(a,b) = |\{c|c \in C' \wedge a \backsim_{\rho(c)} b\}|$.*

For a dependency relation $a \Rightarrow_C b$, a and b are called *antecedent* and *consequent*, respectively. Note that the arrow does not mean b occurs after a in a trace. It states that if we see a, then likely b occurs in the same trace as well. The arrow means the dependency relation is directional, i.e., the reverse of a dependency relation does not always hold. For instance, $a \Rightarrow_C b$ does not imply $b \Rightarrow_C a$.

Next, we create a profile based on the dependency relation to compute the similarity between a case c and a set of cases C'. The idea is that, for all dependency relations which are derived from C' and whose antecedents occur in $\rho(c)$, we map the similarity to 1 if all their consequents also occur in $\rho(c)$. Otherwise, we map the similarity to 0. For convenience, we use $\{x \in \rho(c)\}$ to represent the set of all the activities in $\rho(c)$. For instance, $\{x \in \langle a, b, a, f\rangle\} = \{a, b, f\}$.

Definition 12 (Profile Based On Dependency Relation). *Let $L = (C, A, \rho) \in \mathcal{L}$ be an event log. $prof^L_{de} \in C \times \mathcal{P}(C) \rightarrow \{0, 1\}$ is the profiling function based on the dependency relation, i.e.,*

$$prof^L_{de}(c, C') = \begin{cases} 1, & if\ \{x \in \rho(c)\} \supseteq \{b \in A | \exists a \in \rho(c)\ \ a \Rightarrow_{C'} b\} \\ 0, & otherwise \end{cases}$$

In order to better understand the above definition, we use the same set of cases C_1 with $\rho(C_1) = [\langle a, c, d, f\rangle^{10}, \langle a, b, d, f\rangle^5, \langle a, c, d, e, b, d, f\rangle^5]$, and the same case c_1 with $\rho(c_1) = \langle a, b, d, f\rangle$ to explain it. When both *minConf* and *minSupp* are configured as 1, the set of dependency relations derived from C_1 is $\{a \Rightarrow_{C_1} d, a \Rightarrow_{C_1} f, d \Rightarrow_{C_1} a, d \Rightarrow_{C_1} f, f \Rightarrow_{C_1} a, f \Rightarrow_{C_1} d\}$. Hence $\{a, b, d, f\} \supseteq \{a, d, f\}$ and $prof^L_{de}(c_1, C_1) = 1$.

6. Experiments and evaluation

The *cyclicSC* algorithm has been implemented as a plugin in the ProM.[2] In this section, we test how parameters of the *cyclicSC* algorithm influence its performance, and then compare it with other state-of-the-art techniques both on synthetic and real-life logs.

6.1. Creating synthetic logs and evaluation metrics

First we use synthetic data in order to do a controlled experiments where the ground-truth is known. The basic process of evaluation on synthetic logs is: (i) creating normal logs, (ii) adding artificial deviations into the logs, (iii) detecting deviations, and (iv) checking whether the detected deviations are real deviations, i.e., artificial deviations.

In order to get convincing evaluation, it is necessary to test the algorithm on a wide variety of logs. In this paper, we employ the method in [20] for the random generation of business processes and their execution logs.

Log Generation. The method has four key parameters *AND* (*A*), *XOR* (*X*), *loop* (*L*) and *deep* (*D*) to control the collection of randomly generated processes as shown in Fig. 4 (e.g., $A = 100$, $X =$

[2] https://svn.win.tue.nl/repos/prom/Packages/LiGuangming/Trunk/.

Table 1
Created deviating cases using different deviation types

Normal case	A	C		K		O	N	L	D	B
Add	A	C	D	K		O	N	L	D	B
Remove	A	C		K		O		L	D	B
Replace	A	C		K		A	N	L	D	B
Add, remove & replace	A	B		K	N	O	N		D	B

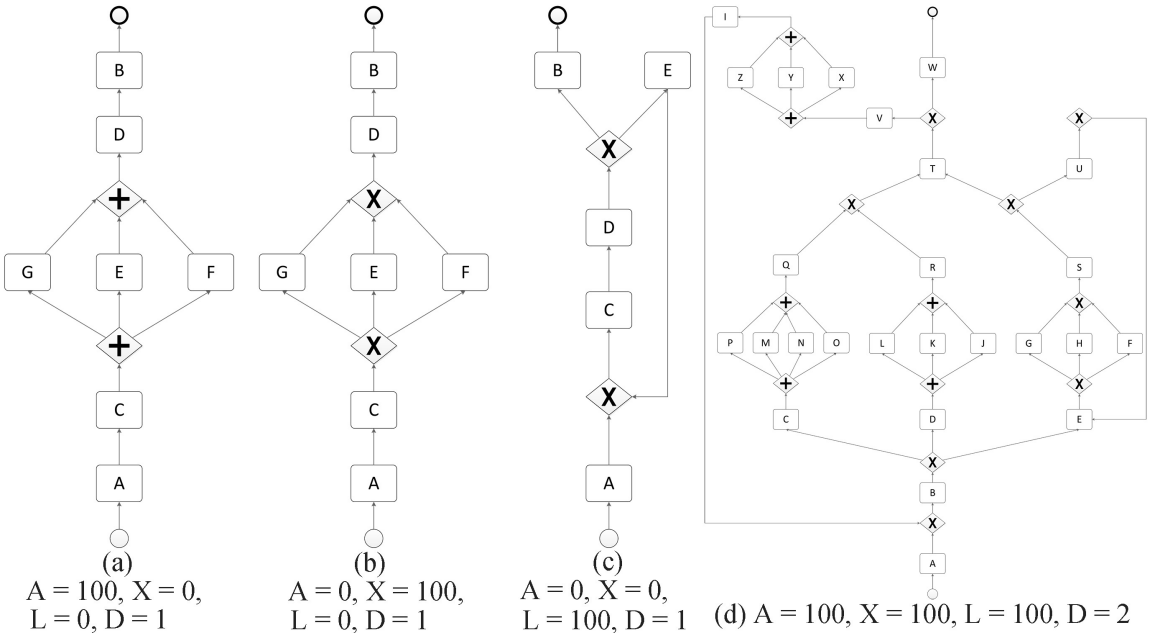


Fig. 4. Processes with different parameters.

0, $L = 0$ and $D = 1$ indicate there only exist AND split-join patterns in the corresponding process and the complexity of the process is at level 1). In order to achieve different processes with various patterns, we set the first three parameters to either 100 or 0, which means the processes contain or do not contain the corresponding patters. Besides, we vary *deep* from 1 to 2 to create processes having different levels of complexity. For the same parameter setting, generated processes may be different. Therefore, we generate five models for each parameter setting, and then create 10 execution logs for each model. Eventually, 80 models and 800 logs are created, which cover a wide and representative set of logs. The 800 logs are considered as "normal logs" without deviants.

Once we have synthetic logs, as shown in Table 1, we inject artificial deviations using the following three *deviation types* (*dt*) :

- *add* an event at a random place in a case;
- *remove* an event at a random place from a case;
- *replace* an event at a random place in a case.

In order to control how many normal cases are transformed into deviation, we introduce a parameter *deviation percentage* (*dp*) which describes what fraction of the cases in the log are deviating after we import deviations. With different deviation types (add, remove and replace) and percentages (0.1, 0.2 and 0.3), we create 9 deviating logs for each normal log, i.e., there are 9×800 logs for next experiments.

Evaluation Metrics. In a sense, the deviation detection problem can be regarded as a classification problem with two classes of objects: deviating cases and normal cases. Therefore we can create a *confusion matrix* [1] to compute *accuracy, precision* and *recall* to show the performance of an algorithm to detect deviations. Since the experiments are on a broad collection of logs (9×800 logs), we take the average value as the final result.

In experiments, the *cyclicSC* algorithm will always detect the same number of deviations as we create, which is the reason that the precision and recall are always the same. However, for the *naive* and *clustering* algorithms (cf. Section 6.3), these numbers may be different. Sometimes the deviating clusters or variants may not only contain the right number of cases as we hope. In this case, we find out the closest number of deviations, i.e., closest to the number of real deviations.

6.2. Controlled parameter settings

Among the parameters of the *cyclicSC* algorithm, the profiling function $prof^L$ and the loop threshold lt have great influence on the performance. In this part, we experiment on 9×800 logs always using $ss = |C| \times (1 - dp)$, $nd = |C| \times dp$ (cf. Section 6.1 for dp) and the initial $norm = norm^1$, to test the influence and discuss limitations in some situations.

6.2.1. Experiments with psrofiling functions

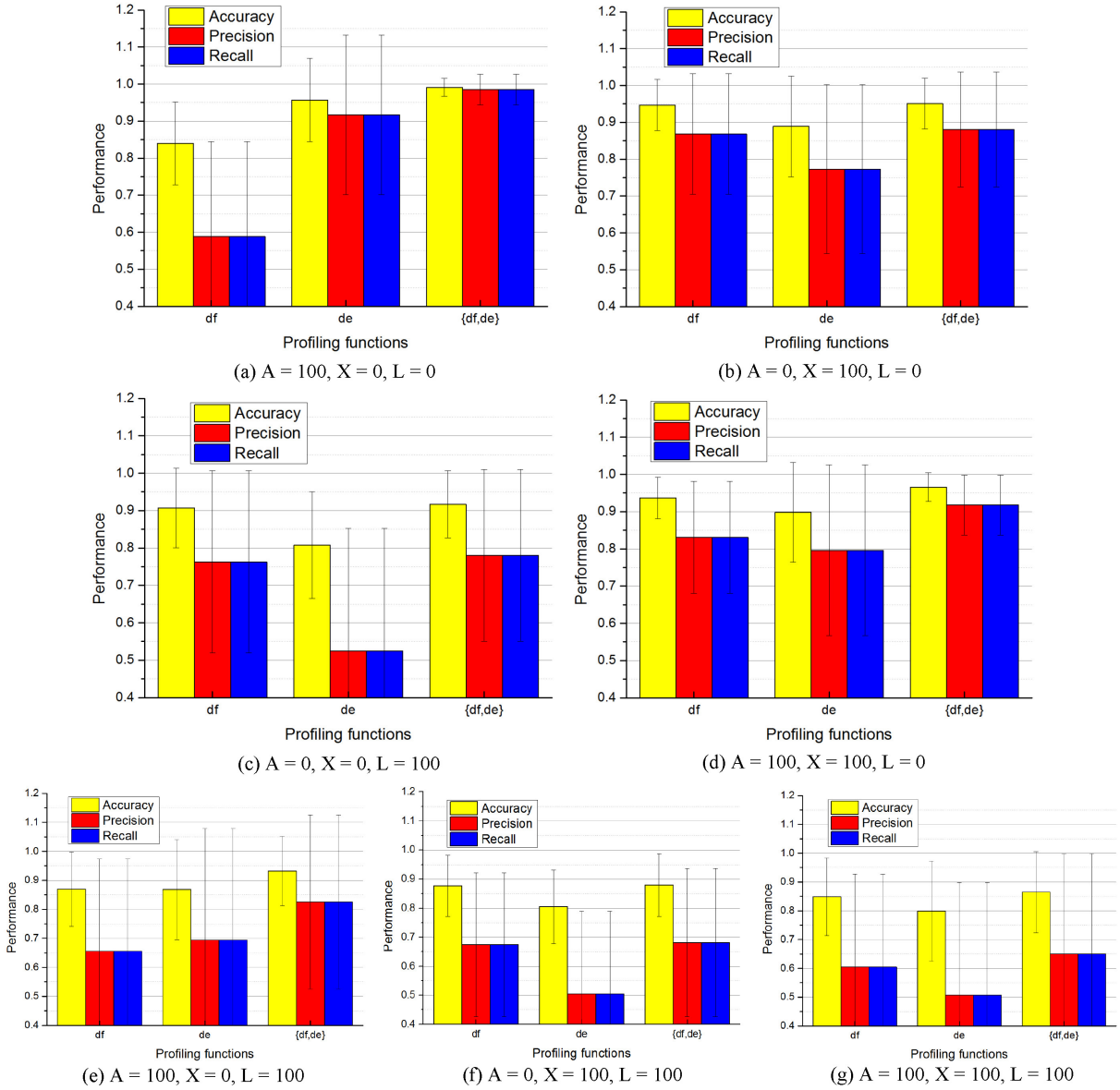
In Section 5, we create $prof^L_{df}$ and $prof^L_{de}$ based on the directly follows relation and dependency relation, respectively, and then employ the combined profiling function $prof^L_{\{df,de\},w^1}$ to detect deviations from the control-flow perspective. In the following experiments, with $lt = 1$, we present the performance of single and combined profiling functions on various logs with different patterns and different deviation types. For convenience, the three functions are denoted as df, de and $\{df, de\}$, respectively.

First we test profiling functions on logs with different patterns (e.g., A=100, X=0 and L=0 indicate there only exist AND split-join patterns in logs). The first three bar charts in Fig. 5 indicate the performance on logs which only contain one type of behavior (e.g., only concurrency). In comparison, $prof^L_{df}$ works better for logs with XOR split-join or loop patterns while $prof^L_{de}$ performs well for logs with AND split-join patterns. The last four charts reveal how different profiling functions work on logs with multiple patterns. Logically, when a new kind of patterns is added into logs, the performance will be better (worse) if the profiling function can (cannot) manage the patterns. For instance, if we import XOR patterns into logs which do not contain XOR split-join patterns (Fig. 5(a)), the performance of $prof^L_{df}$ on new logs (Fig. 5(d)) improves as $prof^L_{df}$ can well deal with XOR split-join patterns.

Next we test the algorithm on logs with different deviations, i.e., different deviation types (dt) and percentages (dp). As shown in Fig. 6, in contrast, $prof^L_{df}$ suits the *add* deviation type while $prof^L_{de}$ fits the *remove* deviation type. They perform almost the same for the *replace* deviation type. For the first three charts, the deviating cases only contain one corresponding deviation. If we allow for different types of deviations in a deviating case, e.g., always three types (Fig. 6(d)), performance improves. If there exist more deviations in a case, the deviating case is easier to detect.

As we can see in the last three charts, along with the increase of the deviation percentage, the performance of $prof^L_{de}$ drops apparently while the accuracy of $prof^L_{df}$ declines slightly. In contrast, $prof^L_{df}$ outperforms $prof^L_{de}$ and can achieve good recall and precision on logs with high deviation percentages.

In summary: the above experiments show that, $prof^L_{df}$ and $prof^L_{de}$ benefit from particular settings, while the combined profiling function can achieve the same or a better result than each individual profiling function. This suggests that we can improve performance through extracting more useful features from a log.



(a) A = 100, X = 0, L = 0

(b) A = 0, X = 100, L = 0

(c) A = 0, X = 0, L = 100

(d) A = 100, X = 100, L = 0

(e) A = 100, X = 0, L = 100

(f) A = 0, X = 100, L = 100

(g) A = 100, X = 100, L = 100

Fig. 5. The performance of the *cyclicSC* algorithm configured with different profiling functions for logs with different patterns.

6.2.2. Experiments with loop thresholds

In the algorithm, we try to get a log containing "more normal" cases by integrating the sampling and classifying steps. However, the actual identification from "normal" cases may fail. Therefore, we test the algorithm on various logs to reveal this limitation in particular situations.

Since we know which cases are real deviating ones, we filter them to create a normal log. Then, we replace the sample log with the normal log to detect deviations. In this case, the performance should be the best and we call it "perfect performance". In the experiments $prof^L = prof^L_{\{df,de\},w^1}$ and we compare the performance of the *cyclicSC* algorithm at different loops with the perfect performance. If the former performance can approach the latter one, it means the algorithm succeeds in getting a

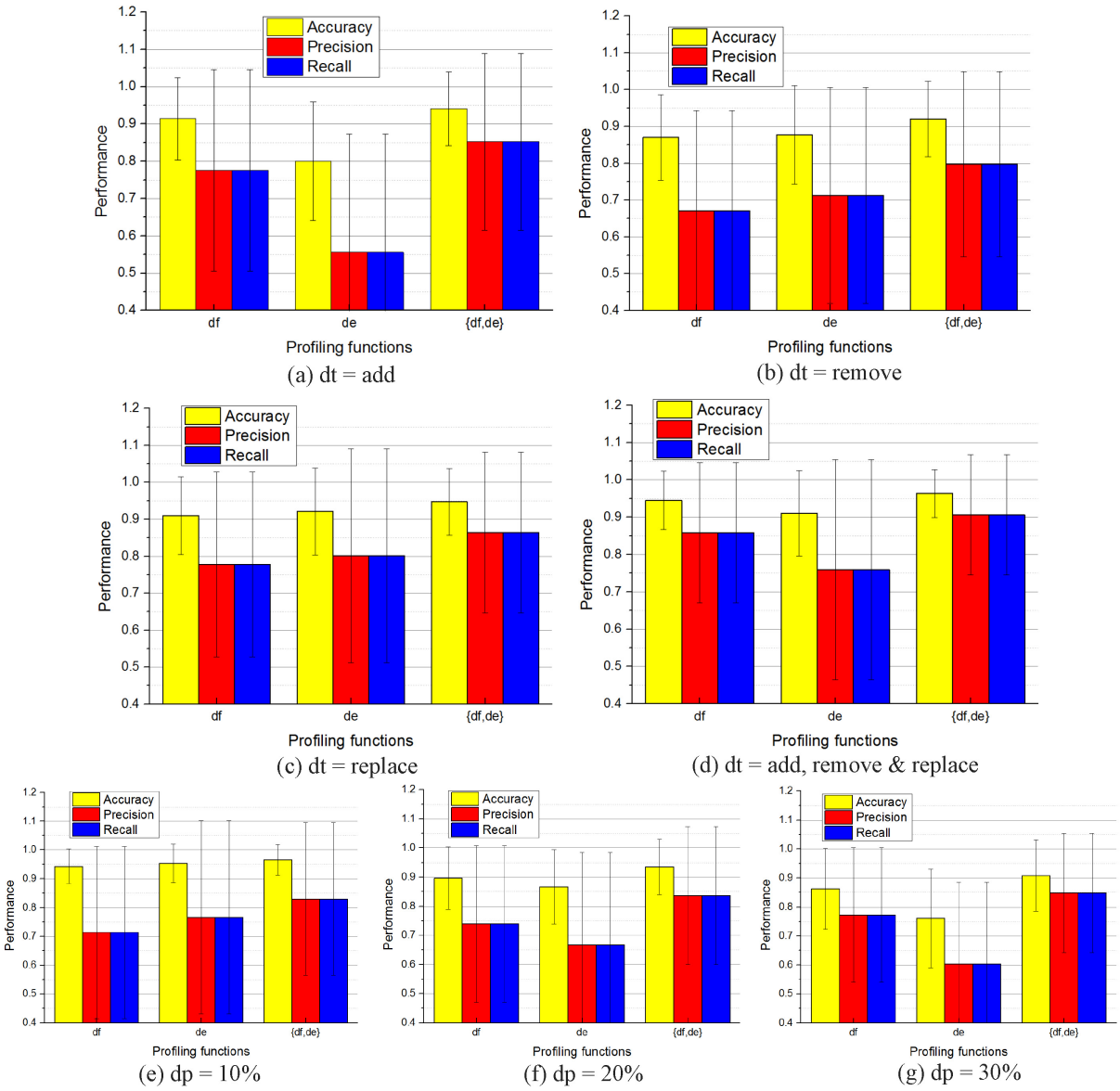


Fig. 6. The performance of the $cyclicSC$ algorithm for logs with different deviations.

"normal" log. More precisely, we compare the former performance at the 1st loop (T1), the 6th loop (T6), and the 15th loop (T15) with the perfect performance.

Among all the logs, we choose four typical logs, Log1 ($dp = 10\%, dt = add, deep = 1, A = 0, X = 0, L = 0$), Log2 ($dp = 20\%, dt = add, deep = 1, A = 0, X = 100, L = 0$), Log3 ($dp = 20\%, dt = add, deep = 1, A = 0, X = 0, L = 0$) and Log4 ($dp = 10\%, dt = add, deep = 1, A = 0, X = 100, L = 0$) to show performance with respect to the number of loops under different circumstances. As shown in Fig. 7, the performance of the algorithm for Log1 reaches "perfect performance" after iterating while the performance for Log2 rises along with loops, but does not reach "perfect performance". In contrast, the performance for Log3 and Log4 remains the same when the number of loops increases.

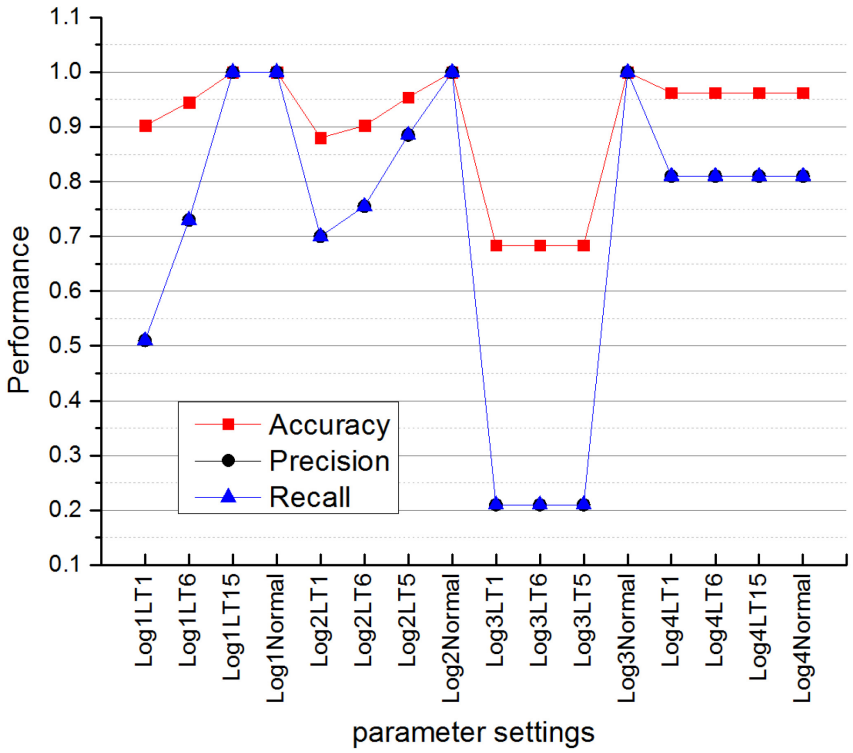


Fig. 7. Performance of the *cyclicSC* algorithm with respect to the number of loops.

To sum up, iteration does not refine the result when the performance at the first loop is too bad or as good as "perfect performance". Otherwise, through iteration, we can make the performance approaching "perfect performance", but cannot guarantee it can reach "perfect performance".

6.3. Comparison with existing methods

In the following, we compare the *cyclicSC* algorithm with two other algorithms, i.e., the *sampling* algorithm [12] which is a representative model-based method and the *clustering* algorithm [21] which is a leading cluster-based method. Besides, we import a baseline method, the *naive* method which considers the low frequent variants created by Disco as deviations (Disco[3] is a famous software in process mining field and has a function to abstract variants from a given log).

Except for the *naive* method, the other three algorithms have parameters to set for good performance. We have discussed the parameters of the *cyclicSC* algorithm in previous experiments. For the *sampling* algorithm, one needs to choose the *mining algorithm* (HM, IM or ILP) and the *sampling size* ($0 \backsim 1.0$). The *clustering* algorithm needs us to configure the *target fitness* ($0 \backsim 1.0$) and the *maximal cluster amount* ($1 \backsim 100$). Actually, there are more parameters than what is mentioned above. Here we only consider the most significant parameters for each algorithm, i.e., the parameters which have a significant influence on the results. For each parameter setting, we compute the average performance over all the logs created in Section 5.1. Then we take accuracy, precision, recall and time into consideration and select the best result

[3] http://www.fluxicon.com/products/.

Algorithms	Parameter settings
CyclicSC	$prof^L$=$prof^L_{\{df,de\},w^1}$ lt=1
Clustering	fitness=1 amount=100
Sampling	size=0.2 algorithm=HM

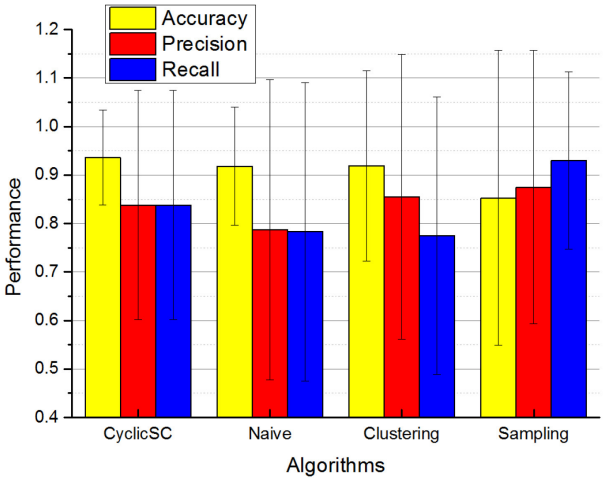


Fig. 8. The best performance of four algorithms with their parameter settings.

for each algorithm. By selecting the parameter setting which performs best, we avoid unfair comparisons due to poor settings. The accuracy, precision, recall and the corresponding parameter setting are shown in Fig. 8 while the average time is shown in Table 2.

Overall, as shown in Fig. 8, the *naive* algorithm performs worst while the other three have their own advantages. The *cyclicSC* algorithm provides results with better accuracy than others, while the *clustering* algorithm has high accuracy and precision, but low recall. The *sampling* algorithm has the best recall but the accuracy is not good.

Besides the overall performance, we also look into how each algorithm performs in different situations. First we see how each algorithm works on logs of different complexity. More precisely, we classify the logs into two categories in terms of the parameter *deep* (*deep* indicates the complexity of the models which generate the synthetic logs), and the performance is shown separately in Figs 9(a) and (b).

As shown in Fig. 9(a), when the underlying model (i.e., the model to generate logs) is easy, the *sampling* algorithm performs best since it can discover a model which is almost the same as the underlying model. The *cyclicSC* and *naive* algorithms also work well, while the *clustering* algorithm achieves bad performance since it often classifies normal and deviating cases into one cluster. When the underlying model is complex, the *sampling* algorithm tends to obtain an easy model and considers normal cases as deviating ones by mistake, which leads to high recall and low accuracy as shown in Fig. 9(b). The *naive* algorithm achieves bad performance since in this case, there exist many normal unique traces such that cases having these traces may be classified as deviating cases. In contrast, the *clustering* algorithm performs best while the *cyclicSC* algorithm achieves a medium performance.

Next, we compare the performance of four algorithms on logs based on processes employing different patterns. Similarly, we split logs into two categories in terms of parameter A, X and L, respectively. For instance, Fig. 9(c) shows the performance on half of all the logs, i.e., on logs which do not contain AND split-join patterns.

In comparison, the *cyclicSC* algorithm performs better than others on logs with AND patterns and logs without XOR or loop patterns, while the *clustering* algorithm has very different characteristics, i.e., logs with XOR and loop patterns are handled well, but does not perform well on logs with AND patterns. The *naive* approach works well when $L = 0$, but performs poorly in other situations. The *sampling* algorithm

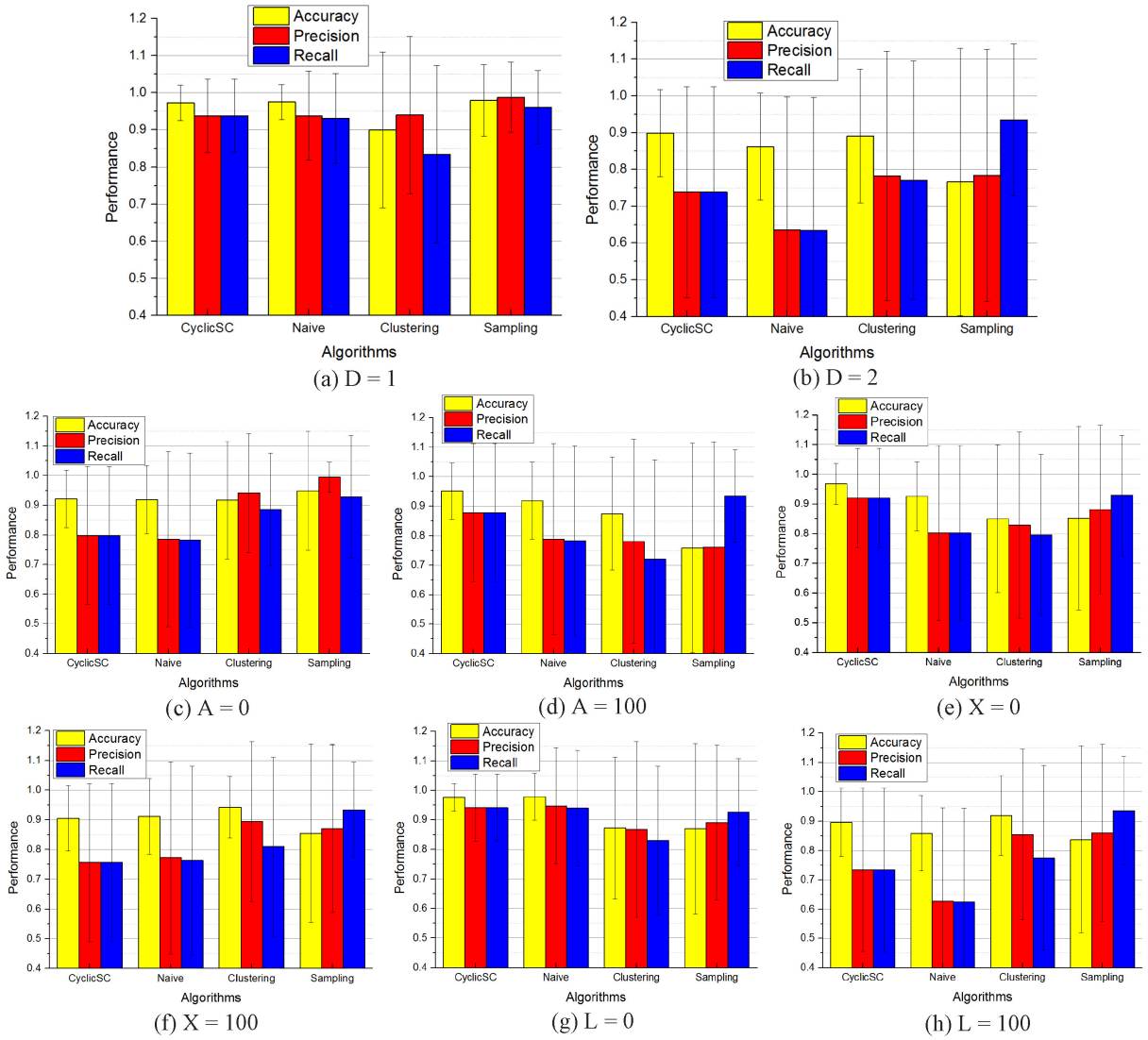


Fig. 9. The performance of four algorithms on logs with different structure.

is impacted greatly by the AND patterns, i.e., it achieves the best performance among the algorithms on logs without AND patterns, but the other way around on logs with AND patterns. Relatively, the XOR and loop patterns have little influence on the *sampling* algorithm.

Figure 9 shows the comparison among four algorithms for logs with different structures. We also tested how sensitive the four algorithms are with respect to different deviation percentages and deviation types. As shown in the top three charts in Fig. 10, along with the increase of deviation percentage, the performance of *clustering* and *sampling* algorithms degrades while the performance of the *cyclicSC* algorithm remains stable. The *naive* algorithm has bad precision and recall when the deviation percentage is low. In comparison, the *cyclicSC* algorithm is more suitable for high deviation percentage. For different deviation types as shown in the bottom three charts, the *clustering* and *sampling* algorithms perform poorly for the *remove* deviation type while the other two algorithms are hardly influenced by

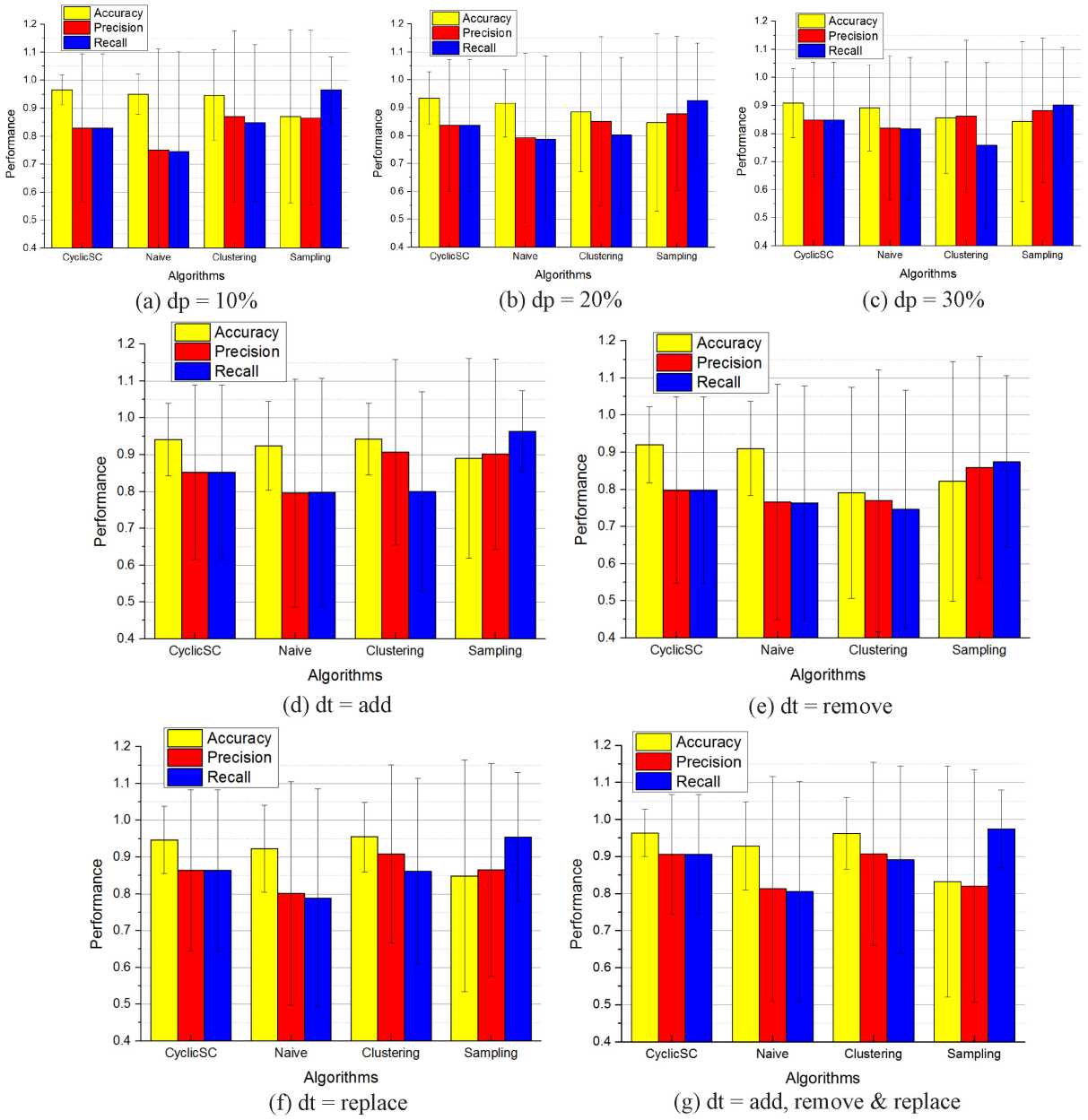


Fig. 10. The performance of four algorithms for different deviations.

the different deviation types. When we mix the three different deviation types together, the *cyclicSC* algorithm and the *clustering* algorithm outperform the other ones. Overall, the *cyclicSC* algorithm has better performance and stability than the others considering a range of different situations. This does not imply that our approach always works best, however, it is more robust for the 9×800 event logs considered.

Next, we compare the running time of four algorithms over various logs as shown in Table 2. More specifically, we classify all logs into two or three categories based on each parameter and then compute

Table 2
The running time of four algorithms over various logs

Log settings	Time(ms)			
	CyclicSC	Naive	Clustering	Sampling
X = 0	346 (± 309)	141 (± 233)	2,917 (± 7,021)	698 (± 1,035)
X = 100	408 (± 337)	198 (± 326)	5,451 (± 12,475)	836 (± 833)
A = 0	404 (± 370)	154 (± 238)	1,004 (± 1,517)	722 (± 696)
A = 100	350 (± 269)	185 (± 324)	7,364 (± 13,620)	812 (± 1135)
L = 0	219 (± 127)	56 (± 158)	2,126 (± 6,794)	514 (± 849)
L = 100	535 (± 381)	284 (± 334)	6,242 (± 12,388)	1020 (± 962)
dp = 10%	382 (± 333)	161 (± 277)	3,729 (± 10,121)	787 (± 853)
dp = 20%	376 (± 328)	170 (± 286)	4,217 (± 10,234)	778 (± 1128)
dp = 30%	374 (± 315)	178 (± 293)	4,607 (± 10,285)	736 (± 819)
dt = add	380 (± 328)	173 (± 289)	4,365 (± 10,214)	772 (± 815)
dt = remove	376 (± 328)	165 (± 279)	3,834 (± 10,186)	713 (± 802)
dt = replace	375 (± 320)	169 (± 285)	4,348 (± 10,234)	813 (± 1165)
D = 1	195 (± 54)	17 (± 12)	282 (± 218)	272 (± 117)
D = 2	559 (± 376)	324 (± 339)	8087 (± 13321)	1262 (± 1288)
average	377 (± 324)	170 (± 285)	4185 (± 10184)	767 (± 941)

the average time on logs of each category for each algorithm. For instance, we separate all logs into two parts based on parameter X (0 or 100) and the first row in Table 2 shows the average time on logs of the part whose parameter $X = 0$. As we can imagine, the *naive* algorithm takes the least time in all situations. On average, the *cyclicSC* algorithm is faster than the *sampling* and *clustering* algorithms. Specifically, the running time of the *clustering* algorithm rises dramatically when the complexity of logs increases from $D = 1$ to $D = 2$. In summary, the *clustering* algorithm takes far more time than the other three algorithms, especially for complex logs, while the *naive* and *cyclicSC* algorithms use less time in all situations.

6.4. Example with a real-life log

The final part of the experimental evaluation comprises the application of four algorithms to a real life-log which we mentioned in Section 1. The log is derived from BPI Challenge 2012 and it contains 262,200 events in 13,087 cases which record the loans in a bank for a period of five months. For the deviation detection based on such a real-life log, it is impossible to verify the accuracy, precision and recall, since we do not know which cases are real deviations. Therefore, we find out one deviating case from the log with each algorithm and compare the four cases on the activity frequency and running time perspectives.

As shown in Fig. 11, the *cyclicSC* algorithm achieves a deviating case (case ID: 197216) which mainly contains an infrequent activitiy, "W_Beoordelen fraude" which means a fraudulent activity and occurs very little (there are just 0.25% of all events in the log having this activity). The *naive* algorithm finds an approved loan (case ID: 173736) as a deviation. However, the detected deviating case is not a real deviation, but a unique trace which has some frequent activities, e.g., "W_Nabellen incomplete dossiers" (8.7%). The *sampling* algorithm detects an unfinished loan (case ID: 211182) while the *clustering* algorithm takes quite a long time to get a cancelled loan (case ID: 176894). The deviations detected by the above two approaches contain a frequent activity "W_Completeren aanvraag" (18%).

On the running time perspective, the *naive* algorithm spends the least time (44 seconds) among the four algorithms. The *sampling* algorithm uses 65 seconds to finish the detecting process (i.e., conformance checking). However, if we take the process to create and find an appropriate model into consideration,

	Activity
1	A_SUBMITTED
2	A_PARTLYSUBMITTED
3	W_Beoordelen fraude
4	W_Beoordelen fraude
5	W_Beoordelen fraude
6	W_Beoordelen fraude
7	W_Beoordelen fraude
8	W_Beoordelen fraude
9	W_Afhandelen leads
10	W_Beoordelen fraude
11	W_Afhandelen leads
12	W_Beoordelen fraude
13	W_Afhandelen leads
14	W_Beoordelen fraude
15	W_Beoordelen fraude
16	W_Beoordelen fraude
17	W_Beoordelen fraude
18	W_Beoordelen fraude
19	W_Beoordelen fraude
20	W_Beoordelen fraude
21	W_Beoordelen fraude
22	W_Beoordelen fraude
23	W_Beoordelen fraude
24	W_Beoordelen fraude
25	W_Afhandelen leads
26	W_Beoordelen fraude

(a) CyclicSC

	Activity
1	A_SUBMITTED
2	A_PARTLYSUBMITTED
3	A_PREACCEPTED
4	W_Completeren aanvraag
5	W_Completeren aanvraag
6	A_ACCEPTED
7	A_FINALIZED
8	O_SELECTED
9	O_CREATED
10	O_SENT
11	W_Nabellen offertes
12	W_Completeren aanvraag
13	W_Nabellen offertes
14	W_Nabellen offertes
15	W_Nabellen offertes
16	W_Nabellen offertes
17	W_Nabellen offertes
18	O_SENT_BACK
19	W_Valideren aanvraag
20	W_Nabellen offertes
21	W_Valideren aanvraag
22	A_APPROVED
23	A_ACTIVATED
24	A_REGISTERED
25	O_ACCEPTED
26	W_Valideren aanvraag

(b) Naive

	Activity
1	A_SUBMITTED
2	A_PARTLYSUBMITTED
3	W_Afhandelen leads
4	W_Afhandelen leads
5	A_PREACCEPTED
6	W_Completeren aanvraag
7	W_Afhandelen leads
8	W_Completeren aanvraag
9	W_Completeren aanvraag
10	W_Completeren aanvraag
11	W_Completeren aanvraag
12	W_Completeren aanvraag
13	W_Completeren aanvraag
14	W_Completeren aanvraag
15	W_Completeren aanvraag
16	W_Completeren aanvraag
17	W_Completeren aanvraag
18	W_Completeren aanvraag
19	W_Completeren aanvraag
20	W_Completeren aanvraag
21	W_Completeren aanvraag
22	W_Completeren aanvraag
23	W_Completeren aanvraag
24	W_Completeren aanvraag
25	A_CANCELLED
26	W_Completeren aanvraag

(c) Clustering

	Activity
1	A_SUBMITTED
2	A_PARTLYSUBMITTED
3	W_Afhandelen leads
4	W_Afhandelen leads
5	A_PREACCEPTED
6	W_Completeren aanvraag
7	W_Afhandelen leads
8	W_Completeren aanvraag
9	W_Completeren aanvraag
10	W_Completeren aanvraag
11	W_Completeren aanvraag
12	W_Completeren aanvraag
13	W_Completeren aanvraag
14	W_Completeren aanvraag
15	W_Completeren aanvraag
16	W_Completeren aanvraag
17	W_Completeren aanvraag
18	W_Completeren aanvraag
19	W_Completeren aanvraag
20	W_Completeren aanvraag
21	W_Completeren aanvraag
22	W_Completeren aanvraag
23	W_Completeren aanvraag
24	W_Completeren aanvraag
25	W_Completeren aanvraag
26	W_Completeren aanvraag

(d) Sampling

Fig. 11. Detected deviations from a real-life log.

the practical time is much more than 65 seconds. The *clustering* algorithm takes much longer time (40 minutes) than the others. In comparison, our approach uncovers deviations in a fraction of the time used by clustering (140 seconds) while avoiding the creation of a reference model.

Through analyzing the activity frequency and running time perspectives, we figure out the *cyclicSC* algorithm can achieve a meaningful result and take limited time when applied to a real-life log.

7. Conclusion and future work

Traditional deviation detection approaches have problems in situations where event logs contain a variety of process behavior. In this paper, we propose a novel algorithm named *cyclicSC* which is faster than cluster-based approaches and more accurate than model-based approaches. Besides, experiments based on 80 models and 9×800 logs suggest that the *cyclicSC* algorithm is more robust than others. The framework is configurable and we used it to create a concrete approach for detecting deviations from control-flow perspective. One can edit the *profiling function* to detect deviations from specific perspectives. We compared the *cyclicSC* algorithm with existing methods on synthetic and real-life logs to illustrate its properties.

The *cyclicSC* algorithm also has some limitations (cf. Section 6.2.2). For instance, in some situations, the loops are not handled properly, yet overall the approach is more robust than others. In future work we would like to address these limitations and apply the approach to more real-life event logs for which we engage with end-users to establish the ground truth (as we did for the synthetic data). We would also like to test the approach in a streaming setting with concept drift. In this setting the process may gradually change and including what constitute its mainstream behavior. The iterative nature of the approach seems particulary suitable to handle this scenario.

References

[1] W.M.P. van der Aalst, Process mining: Discovery, conformance and enhancement of business processes, Springer-Verlag, Berlin, 2011.

[2] C.W. Günther, Process mining in flexible environments, PhD dissertation, Eindhoven University of Technology, 2009.

[3] V. Chandola, A. Banerjee and V. Kumar, Anomaly detection – a survey, Technical report, University of Minnesota, 2007.

[4] D. Lo et al., Classification of software behaviors for failure detection: A discriminative pattern mining approach, in: *Proceedings of the 15th ACM International Conference on Knowledge Discovery and Data Mining*, 2009, pp. 557–566.

[5] W.S. Yang and S.Y. Hwang, A process-mining framework for the detection of healthcare fraud and abuse, *Expert Systems with Applications* **31**(1) (2006), 56–68.

[6] G. Münz, S. Li and G. Carle, Traffic anomaly detection using k-means clustering, in: *GI/ITG Workshop MMBnet*, 2007.

[7] W.M.P. van der Aalst and A.K.A. de Medeiros,Process mining and security: Detecting anomalous process executions and checking process conformance, *Electronic Notes in Theoretical Computer Science* **121** (2005), 3–21.

[8] J. Swinnen, B. Depaire and M.J. Jans, A process deviation analysis – a case study, Business Process Management Workshops, LNBIP, **99** (2012), 87–98, Springer, Heidelberg.

[9] J. De Weerdt et al., Leveraging process discovery with trace clustering and text mining for intelligent analysis of incident management processes, in: IEEE Congress on Evolutionary Computation (CEC 2012), 2012, 1–8.

[10] R.P.J.C. Bose and W.M.P. van der Aalst, Trace clustering based on conserved patterns: Towards achieving better process models, in: Business Process Management Workshops, LNBIP, **43** (2009), pp. 170–181, Springer, Heidelberg.

[11] G. Greco, A. Guzzo and L. Pontieri, Discovering expressive process models by clustering log traces, *IEEE Transactions on Knowledge and Data Engineering* **18**(8) (2006), 1010–1027.

[12] F. Bezerra and J. Wainer, Algorithms for anomaly detection of traces in logs of process aware information systems, *Information Systems* **38**(1) (2013), 33–44.

[13] F. Bezerra and J. Wainer, A dynamic threshold algorithm for anomaly detection in logs of process aware systems, *Journal of Information and Data Management* **3**(3) (2012), 316–331.

[14] F. Bezerra, J. Wainer and W.M.P. van der Aalst, Anomaly detection using process mining, *Enterprise, Business-Process and Information Systems Modeling*, 2009, 149–161.

[15] H. Jalali and A. Baraani, Genetic-based anomaly detection in logs of process aware systems, *World Academy of Science Engineering and Technology* **64** (2010), 304–309.

[16] A.K. Alves de Medeiros, Genetic process mining, PhD dissertation, Eindhoven University of Technology, 2006.

[17] L. Ghionna, G. Greco and A. Guzzo, Outlier detection techniques for process mining applications, Foundations of Intelligent Systems, 2008, 150–159.

[18] G. Greco, A. Guzzo and L. Pontieri, Discovering expressive process models by clustering log traces, *IEEE Transactions on Knowledge and Data Engineering* **18**(8) (2006), 1010–1027.

[19] M. Song, C.W. Günther and W.M.P van der Aalst, Trace clustering in process mining, in: *Business Process Management Workshops*, LNBIP **17** (2009), pp. 109–120, Springer, Heidelberg.

[20] A. Burattin and A. Sperduti, PLG: A framework for the generation of business process models and their execution logs, in: *Business Process Management Workshops*, LNBIP **66** (2011), pp. 214–219. Springer, Heidelberg.

[21] J. De Weerdt, J. Vanthienen and B. Baesens, Active trace clustering for improved process discovery, *IEEE Transactions on Knowledge and Data Engineering* **25**(12) (2013), 2708–2720.

Appendix: The algorithm cyclicSC

In the paper, we describe the $CyclicSC$ algorithm by providing definitions and illustrating the main ideas using the flowchart in Fig. 3. In this appendix, we give the pseudo code of the algorithm to better explain the algorithm.

Algorithm 1 CyclicSC: discover deviating cases in an event log

Input:
An event log $L = (C, A, \rho)$, a sample size ss, a number lt, a profiling function $Prof^L$, a norm function $norm^1$, a number nd;
Output:
A set of deviating cases C_D;
$norm = norm^1$
# $norm$ is a function which assigns a value to each case based on its normality; it is initialized as $norm^1$ to assign a value 1 to each case
for $i = 1$ to lt **do**
  $C' = C$
  $C_S = \emptyset$
  $C_D = \emptyset$
Phase 1: Sampling
  for $i = 1$ to ss **do**
    $c = rouletteWheelSelection(norm, C')$
    # $rouletteWheelSelection$ is a function which randomly selects a case c from C' using a relative likelihood based on $norm$, i.e., $c_1 \in C'$ is k times as likely to be selected as $c_2 \in C'$ if $norm(c_1) = k \times norm(c_2)$
    $C' = C' \backslash \{c\}$
    $C_S = C_S \cup \{c\}$
  end for
Phase 2: Classifying
  for each $c \in C$ **do**
    $similarity(c) = Prof^L(c, C_S)$
    # $similarity$ is a function which assigns a value to each case to indicate how much it is similar to C_S
  end for
  $similarityList = ranking(similarity, C)$
  # $ranking$ is a function which returns a list in which all cases of C are ranked based on their similarity values (from small to large), i.e., $c_1 \in C$ is before $c_2 \in C$ in $similarityList$ if $similarity(c_1) \leqslant similarity(c_2)$
  for each $c \in C$ **do**
    if $similarityList.IndexOf(c) \geqslant nd$ **then**
      $norm(c) = norm(c) \times R_N$
    else
      $norm(c) = norm(c) \times R_D$
      $C_D = C_D \cup \{c\}$
    end if
    # $similarityList.IndexOf(c)$ indicates the index of c in $similarityList$; $RN > 1$ and $0 < RD < 1$ are used to adjust the norm values.
  end for
end for