Detecting Deviating Behaviors without Models

Xixi Lu[1], Dirk Fahland[1], Frank J.H.M. van den Biggelaar[2], Wil M.P. van der Aalst[1]

[1] Eindhoven University of Technology, The Netherlands
[2] Maastricht University Medical Center
{x.lu,d.fahland,w.m.p.v.d.aalst}@tue.nl
{f.vanden.biggelaar}@mumc.nl

Abstract. *Deviation detection* is a set of techniques that identify deviations from normative processes in real process executions. These diagnostics are used to derive recommendations for improving business processes. Existing detection techniques identify deviations either only on the process instance level or rely on a normative process model to locate deviating behavior on the event level. However, when normative models are not available, these techniques detect deviations against a less accurate model discovered from the actual behavior, resulting in incorrect diagnostics. In this paper, we propose a novel approach to detect *deviation on the event level* by identifying *frequent common behavior* and *uncommon behavior* among executed process instances, without discovering any normative model. The approach is implemented in ProM and was evaluated in a controlled setting with artificial logs and real-life logs. We compare our approach to existing approaches to investigate its possibilities and limitations. We show that in some cases, it is possible to detect deviating events without a model as accurately as against a given precise normative model.

1 Introduction

Immense amounts of event data have been recorded across different application domains, reflecting executions of manifold business processes. The recorded data, also called *event logs* or *observed behavior*, show that real-life executions of process instances often deviate from normative processes [12]. Deviation detection, in the context of *conformance checking*, is a set of techniques that *check process conformance* of recorded executions against a normative process and identify where observed behavior does not fit in and thus deviates from the normative process model [1]. Accurately detecting deviating behavior at the event level is important for finding root causes and providing diagnostic information. The diagnosis can be used to derive recommendations for improving process compliance and performance [13].

Existing techniques for detecting deviations, such as alignment-based techniques [1], require a normative process in the form of a process model. However, normative models are often not available, especially in flexible environments. For instance, in healthcare, each patient often follows a unique path through the process with one-of-a-kind deviations [11]. A solution is to discover a model from an event log. The discovered model is assumed to describe the normative behavior, and then conformance checking techniques discern where the log deviates. However, the quality of deviation

detection depends heavily on the discovered model, which again depends on the discovery algorithm used and the design decisions made in the algorithm. When an event log shows high variety (for example, containing multiple process variants), discovering one normative process almost always results in underfitting models, rendering them useless for detecting deviations.

In this paper, we consider the problem of detecting deviations without discovering a normative process model. We limit our scope to only detecting *deviating events*; we define deviations as *additional* behavior observed in an event log but not allowed in the normative process; other deviations, such as steps of the normative process that are skipped, are not considered in this paper. We present a new technique to detect deviating events by computing *mappings* between events, which specify similar and dissimilar behavior between process instances. The more they that *agree* on a certain behavior, the less such a behavior is a deviation. We use this information to classify deviations.

The approach has been implemented as ProM plugin and was evaluated using artificial logs and real life logs. We compared our approach to existing approaches to investigate the possibility and the limitations of detecting deviations without a model. We show that the approach helps identify deviations without using a normative process model. In cases where dependencies between events can be discovered precisely, it is possible to detect deviating events as accurately as when using a given precise normative model. In other cases, when deviating events happen frequently and in patterns, it is more difficult to distinguish them from the conforming behavior without a normative model. We discuss ideas to overcome these problems in our approach.

In the remainder, we first discuss related work in Sect. 2, including input for our approach. Sects. 3-5 explain our method in more depth: in Sect. 3, we define and explain the relevant concepts, e.g. similar and dissimilar behavior, mapping, and cost function; Sect. 4 presents two algorithms to compute mappings; Sect. 5 discusses how to use mappings for detecting deviations. The evaluation results are presented in Sect. 6, and Sect. 7 discusses the limitations and concludes the paper.

2 Related Work

We consider an event log as input for our approach for detecting deviations. In addition, we discuss related work more in detail in this section.

Event logs and Partial Orders. An *event log* is a collection of *traces*, each of which is *a sequence of events* describing the observed execution for a case. Most process mining techniques use an event log as input. Recently, research has been conducted to obtain partial orders over events, called partially ordered traces, and use them instead to improve process mining [6, 9]. The work in [9] discussed various ways to convert sequential traces into partially ordered traces and has shown that such a conduct improves the quality of conformance checking when the as-is total ordering of events is unreliable. The approach proposed in this paper can handle partial orders as inputs, which we refer to as *execution graphs*. Two types of partial order [9] are used in this paper: data based partial order over events, i.e. two events are dependent if they access the same data attributes; and time based partial order over events, i.e. two events are dependent if they have different time stamps.

Outlier Detection and Deviance Mining. Existing outlier detection approaches have a different focus and are not applicable to our problem. These approaches first converting executions of cases to items of features and then using classification or clustering techniques [7]. However, they only identify deviating cases (thus items) and omit deviation on the event level (an analogy to classical data mining would be detecting a deviating value in a item for one feature) and are often unable to handle the situation in which a multitude of cases contain deviation. One different stream, known as *deviance mining*, classifies cases as normal or deviant, independent of their control-flow execution, but rather based on their performance (e.g. whether throughput time of a case is acceptable) [10]. Our approach is inspired by and similar to a log visualization technique known as *trace alignment* [3]. However, this visualization technique does not classify deviations but simply visualizes the mappings between traces to a user.

Conformance Checking. A state-of-art conformance checking technique is known as *(model-log) alignment* [1,9], which computes a most similar run of a given normative model with respect to each input trace. Events observed in traces that have no matching behavior in such a run are classified as deviating events, also known as log moves. However, the current cost function used by the approach is rather simple and static. For example, it is unable to distinguish consecutive events sharing the same event class. In addition, a precise model is required to identify deviations accurately, which might be unavailable and difficult to discover, whereas our approach does not require models.

Process Discovery and Trace Clustering. Process discovery algorithms aim to discover structured process models using an event log [4, 8], but still face various difficulties [5]. When event logs are highly unstructured and contain deviating behavior, discovery algorithms often fail to find the underlying structure and return *spaghetti models* due to overfitting. Some discovery algorithms aim to be noise/deviation robust but often result in returning over-generalized or underfitted models. To discover better models, one may preprocess event logs using, for example, trace clustering. Syntactic-based trace clustering [5] is a set of techniques that focus on clustering traces in such a way that structured models can be discovered as different variant of the normative model. In our evaluation, we compare our approach to [1, 2, 5, 8, 9] more in depth.

3 Mappings - Similarities and Dissimilarities between Executions

In this section, we introduce the key concepts used in this paper and explain how *similarity* and *dissimilarity* between executions of cases helps identify deviations.

Execution Graphs and Neighbors. For describing execution of a case, we use an *execution graph*. An execution graph is a directed acyclic graph $G = (E, R, l)$: the nodes E are the events recorded for the case, the edges R are the relations between the events, and the function l assigns to each event its event type. Each event is unique and has a set of attributes; one event belongs to one single execution graph. Fig. 1 shows two execution graphs. On the right of Fig. 1, e_8, e_9, e_{10} are considered concurrent because, for example, they have the same timestamps [9]. Let e be an event in an execution graph. *k-predecessors* $N_k^p(e)$ denotes the set of events from which (1) there is a path in the execution graph to e and (2) the length of the path is at least 1 and at most k; similar for

k-successors $N_k^s(e)$. In addition, we call the set of events for which there is no path to or from e the *concurrences* $N^c(e)$ of e. Moreover, for $e' \in N^c(e)$, we define the distance $dist_G(e, e') = 0$, in contrast to the traditional graph theory.

The *k-neighbors* $N_k(e)$ of e is a 3-tuple composed of the *k-predecessors*, the *concurrences* and the *k-successors* of e. For example, as shown in Fig. 1, $N_1(e_8) = (\{e_7\}, \{e_9, e_{10}\}, \{e_{11}\})$.

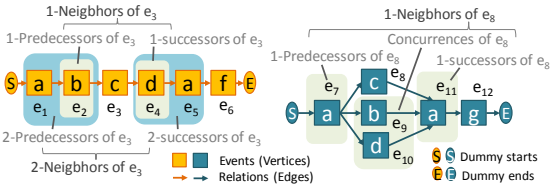

Fig. 1: Two examples of execution graphs.

Deviations, Mappings and Similarity. We consider *deviations* as non-conforming behavior that consists of observed events in an execution graph. The assumption is that such deviating events occur much less frequently and occur in a highly dissimilar context, e.g. have dissimilar neighbors and locations, since they are not specified in the normative process. In addition, it would be difficult to find the events in other cases that are similar and comparable to these deviating events. Therefore, we compute similar behavior and dissimilar behavior between each two execution graphs as a *mapping*: the *similar behavior* is formed by all pairs of events that are mapped to each other, whereas events that are not mapped are *dissimilar behavior*. Formally, a *mapping* $\lambda_{(G,G')}$ between two execution graphs is a set of binary, symmetric relations between their events, in which each event is only mapped to one other event. Fig. 2 exemplifies a mapping between the two execution graphs shown in Fig. 1. For instance, the mapping in Fig. 2 specifies that e_3 and e_8 are not mapped, and therefore, according to this particular mapping, they are dissimilar and show discrepancies between the two cases. We use $\overline{\lambda}$ to refer to the set of events that are not mapped, i.e. $\overline{\lambda}_{(G,G')} = \{e \in E \mid \neg \exists e' \in E' : (e, e') \in \lambda\} \cup \{e' \in E' \mid \neg \exists e \in E : (e, e') \in \lambda\}$[3].

Based on a mapping, we also obtain *similar neighbors* and *dissimilar neighbors* surrounding two events and are able to compare the events more accurately. A pair of events are more similar, if they share more similar neighbors. For example, using a mapping, we can derive the similar predecessors and the dissimilar predecessors of two paired events (e, e'). We refer to the dissimilar predecessors as $DN_k^p(e, e', \lambda)$, where the k indicates the *k-predecessors*. The same applies to the set of *dissimilar successors* $DN_k^s(e, e', \lambda)$ and *dissimilar concurrences* $DN^c(e, e', \lambda)$. Fig. 2 shows an example: because events e_5 and e_{11} have respectively $\{e_3, e_4\}$ and $\{e_7, e_8, e_9, e_{10}\}$ as their 2-*predecessors*, of which e_4 and e_{10} are paired, therefore $DN_2^p(e_5, e_{11}, \lambda) = \{e_3, e_7, e_8, e_9\}$. The pair (e_5, e_{11}) has two dissimilar successors e_6 and e_{12}, but no dissimilar concurrences as shown in Fig. 2. Hence, $DN_2^s(e_5, e_{11}, \lambda) = \{e_6, e_{12}\}$, and $DN^c(e_5, e_{11}, \lambda) = \varnothing$.

Cost Function and Cost Configurations. To evaluate a mapping, we define a *cost function* that assesses the similarity between paired events in the mapping. A mapping that captures more similar behavior is assigned with a lower cost. The mappings with the minimal cost are the *optimal mappings*. The cost function is shown in Equation 1 and

[3] We omit G and G' for both λ and $\overline{\lambda}$ where the context is clear.

comprises three components $cost_{Matched}$, $cost_{Struc}$ and $cost_{NoMatch}$ that assess a mapping as follows. For each pair of events (mapped to each other) in a mapping, $cost_{Matched}$ and $cost_{Struc}$ assess their *local similarity* and *global similarity*, respectively. Moreover, $cost_{NoMatch}$ assigns a penalty to the mapping for each event that is classified to be dissimilar (i.e. not mapped). For each component, we assign a weight, i.e. w_M, w_S, w_N.

$$cost(G, G', \lambda) = w_M * cost_{Matched}(G, G', \lambda) + w_S * cost_{Struc}(G, G', \lambda) + w_N * cost_{NoMatch}(G, G', \lambda) \quad (1)$$

The function $cost_{Matched}$, defined in Equation 2, helps to assess the similarity between two events regarding their properties and their local execution contexts (in this case their labels and their neighbors). The more similar, the lower the cost. Thus, a higher cost is assigned to prevent two locally dissimilar events being mapped to each other. In this paper, we only allow two events with the same label to be mapped to each other, i.e. $cost(l(e), l(e')) = 0$ if $l(e) = l(e')$, otherwise infinite.

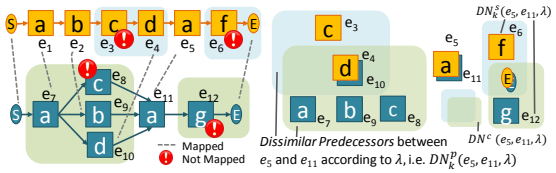

Fig. 2: An example of a mapping specifying similar and dissimilar behavior.

$$cost_{Matched}(G, G', \lambda) = \sum_{(e,e')\in\lambda} cost(l(e), l(e')) + \mid DN_k^p(e, e', \lambda) \mid + \mid DN_k^s(e, e', \lambda) \mid + \mid DN^c(e, e', \lambda) \mid \quad (2)$$

In addition, the function $cost_{Struc}(G, G', \lambda) = \sum_{(p,p'),(e,e')\in\lambda} \frac{|dist_G(p,e) - dist_{G'}(p',e')|}{2}$ helps to assess how similar two events are with respect to their positions in the global context of execution graphs. The more similar their positions in the global context, the lower cost; the cost is high if they are in very different stages of execution graphs.

Futhermore, we define the function $cost_{NoMatch}(G, G', \lambda) = \sum_{e\in\overline{\lambda}} C_N + \mid N_k(e) \mid$, which assigns a cost to events that are not mapped and helps to asses when not to map an event. For example, a higher cost is assigned to a not-mapped event if it is *important* and should be mapped. We use the number of neighbors of an event to indicate the importance in addition to a basic cost C_N of not matching an event.

The final cost of a mapping depends on the k (defining the neighbors) and the four weights w_M, w_S, w_N and C_N. A 5-tuple composed of these five numbers is called a *cost configuration* of the cost function. The mappings with the minimal cost between two execution graphs according to a configured cost function are the *optimal mappings*.

4 Algorithms For Computing Mappings

For computing mappings between execution graphs, we propose two algorithms: one uses backtracking with a heuristic function and guarantees the return of the optimal mappings; the other provides no guarantees but runs in polynomial time.

Backtracking and Heuristic Function. The backtracking algorithm uses a *heuristic function* to prune our search space. The heuristic function is similar to the cost function and reuses $cost_{Matched}$, $cost_{Struc}$, and $cost_{NoMatch}$. The same *configuration* as the cost function is required to guarantee the lower bound property.

The algorithm starts with an empty mapping between two cases and then inductively computes the cost of the next decision, i.e. to consider two events similar or not, using the heuristic function. After making a decision to map two events, a part of the similar and dissimilar neighbors of the two events is known, according to the mapping so far, for which the heuristic function uses $cost_{Matched}$ to compute the cost. For the neighbors not yet mapped, the heuristic function estimates the cost by predicting an *optimal situation* of a future complete mapping. The optimal situation means that a maximal set of *possibly similar neighbors*, i.e. the neighbors that have the same label and are not mapped yet, becomes *similar neighbors*. Maximizing the set of *possibly similar neighbors* minimizes the set of possibly dissimilar neighbors (impossible to become similar neighbors in the future) and thus gives us a lower bound of the unknown part of the cost. Formally, we perform label multiset subtraction of not mapped neighbors to estimate the lower bound.

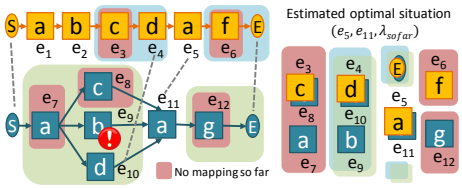

Fig. 3: An example of an incomplete mapping and the estimated lowerbound cost.

Fig. 3 illustrates an incomplete mapping that states e_4 and e_{10} are similar and e_9 is dissimilar (i.e. $\lambda_{sofar} = \{e_4 \rightarrow e_{10}, e_9 \rightarrow \perp\}$). If we decide that e_5 and e_{11} are similar (thus mapping e_5 to e_{11}), we obtain their similar neighbors e_4 and e_{10} and dissimilar neighbor e_9 according to the mapping so far. We also identify the *possibly similar neighbors* e_3 and e_8 (both labeled with c and not mapped yet), and possibly dissimilar neighbors e_7, e_6 and e_{12}. Thus, the cost returned by $cost_{Matched}$ is 1 and the estimated additional future cost is 3. The cost of structure returns 2 because the distance from S to e_5 is 5, which differs from the distance of 3 between S and e_{11}.

The running time of the back tracking algorithm is $O(2^n)$, if each graph contains n events all with unique labels, because for each event, there is a choice between mapping the event or not. In the worst case when all events have the same label, the running time is $O((n+1)!)$.

Greedy Algorithm. The second algorithm we propose is greedy and runs in polynomial time. The greedy algorithm makes the current optimal choice to map two events or not. The quality of the algorithm depends heavily on the ordering of the choice that is made. The idea is to start with *finding the "most important and unique" event e (which has the least probability to be a deviating event or to be matched to another deviating event); then, select, for e, the current most similar event, if any.* As the mapping becomes more complete, the cost returned by the heuristic function resembles more accurately the cost returned by the cost function, which helps the algorithm to make more difficult choices later.

For formalizing this "importance and uniqueness", we introduce the concept of a *k-context* and its frequency as an example. A *k-context* $C_k(e)$ of an event e consists

of the label of e, the labels of its k-predecessors, the labels of its concurrences, and the labels of its k-successors. Fig. 4 shows three 3-*contexts* with label a (on the right) based on the four execution graphs on the left. For example, $C_3(e_5) = C_3(e_{25}) = C_3(e_{35}) = (a, [b, c, d], [], [f, E])$. The *absolute frequency of a k-context* of an event e is the number of events that have the exact same k-context and is formally defined as follows. Let $\mathbb{G}$ denote a set of execution graphs. For each event e in E of $G \in \mathbb{G}$, the *absolute frequency of a k-context* is $Freq_{\mathbb{G}}(C_k(e)) = \sum_{G \in \mathbb{G}} \mid \{e' \in E \mid C_k(e) = C_k(e')\} \mid$. For example, $Freq_{\text{Fig. 4}}(a, [b, c, d], [], [f, E]) = 3$. A context having a high absolute frequency indicates that there is a large set of events sharing the same context and can be mapped to each other.

To compute a *good* mapping between two given execution graphs, the greedy algorithm first sorts the nodes (i.e. events) based on the absolute frequencies of their context, and then simply starts with the "most important" node according to the ordering, and selects the best match for this node using the heuristic function introduced in the previous section. This process of making choices is repeated, and the algorithm simply works through the nodes linearly. Therefore, the running time of the greedy algorithm is quadratic in terms of the number of events.

5 Deviation Detection Using Mappings

We use the mappings to compute *representative execution graphs (regs)* of cases and use them to locate uncommon behavior and identify deviations. A *reg* can be seen as an aggregation of a cluster of similar execution graphs and represents one variant of process execution. Each node of a *reg* represents a set of similar events; the number of events a node represent indicates the *commonness* of this behavior among cases of the *reg*. Similarly, each edge depicts a set of similar relations between the events. Fig. 5 shows three *regs*. As can be seen, a *reg* resembles a directly follows graph with unfolded duplicated labels and shows executions of its cases, but the commonness of the nodes can also be used for detecting deviations and visualizing their positions.

Fig. 5 also shows the process of aggregating execution graphs into a *reg* which we refer to as *fusion*. We compute *regs* of cases by fusing execution graphs *among which all mappings are consistent regarding all behavior*. In other words, the mappings between a set of execution graphs are consistent when all of them agree with each other about the similar behaviors. Formally, assuming a set of execution graphs is given, and Λ denotes the set of all mappings between them: Λ is *consistent* iff. Λ is transitive, i.e. for all $(e, e'), (e', e'') \in \Lambda \Rightarrow (e, e'') \in \Lambda$. The consistency of guarantees that the

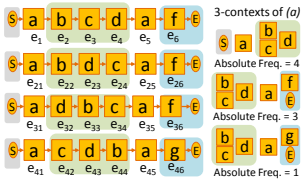

Fig. 4: 3-contexts and their absolute frequency.

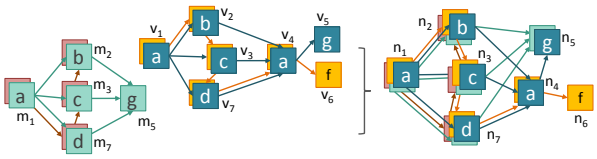

Fig. 5: Fusion process: two *regs* fused into one *reg*

ordering of fusing a set of similar events (e.g. e, e', e'') is irrelevant (thus commutative and associative). Fig. 5 illustrates a fusion of two *regs* representing four cases. The nodes m_1 and v_1 are fused into n_1, meaning that the mappings between them all agree that the four events are similar. The same holds for the rest of the nodes. Now, assume that, according to a mapping, one of the events of m_1 is actually similar to one of the events of v_4 instead of v_1, then the two *regs* will not be fused. We apply this principle incrementally by simply fusing the two most similar (groups of) cases indicated by the cost of their mappings. The algorithm returns a set of *regs* that can no further be fused.

Deviations are assumed to be uncommon behavior. If the number of events that a node n in a *reg* represents is low, it indicates that the behavior rarely occurs among the cases that are similar. If this number is below a certain threshold T relative to the maximum number of events represented by another node that has the same label in the same *reg*, we classify this node n to be uncommon and the events of n to be deviating. For example, assuming we have the *reg* on the right of Fig. 5 and T is 60%, then the events of nodes n_5 and n_6 are classified as conforming since they represent the maximum number of events with respect to their labels g and f, respectively, whereas the one of n_4 is only 50% of the maximum as 2 of 4 (represented by node n_1). Thus, the events of n_4 are classified as deviating. Another example, if the two *regs* shown on the left of Fig. 5 were not fused due to inconsistency and T is 60%, then all events are classified as normal behavior; the same for any *reg* that only represents one execution graph.

6 Evaluation and Results

The proposed deviation detection approach is implemented in the process mining toolkit ProM[4]. We conducted controlled experiments to compare our approach to existing approaches and discuss the results in this section.

Experimental Setup. We compared our approach to other techniques on how accurately deviating events are detected as shown in Fig. 6. Given a log with each event labeled as deviant or conforming, our approach and existing approaches classify each of the events as deviating or conforming. Events correctly classified as deviations (based on the labels) are considered *true positives (TP)*. Similarly, *false positives (FP)* are conforming events that are incorrectly classified as deviations; *false negatives (FN)* are deviating events that are incorrectly classified as conforming events; *true negatives (TN)* are correctly classified as conforming events. Based on this, we compute the *accuracy* score (abbreviated to *acc*)[5], i.e. $acc = (TP+TN)/(TP+TN+FP+FN)$. For example, achieving an accuracy score of 0.9 after classifying 10 events means one of the events is incorrectly classified as deviating (FP) or conforming (FN).

We compared the accuracy of our approach to three existing methods shown in Fig. 6: (1) classify deviations by checking conformance [1] against the given normative model; (2) discover a normative model and then apply conformance checking using the discovered model; (3) first cluster traces to discover a more precise normative model for

[4] Both the plugins and the experiments can be found in the *TraceMatching* package of the ProM.

[5] In this paper, we only discuss the accuracy score. However, one may use the confusion matrix and compute the F1 score of event identification or swap the confusion matrix to compute the F1 score of deviation identification. We have computed all three, and they have shown similar results.

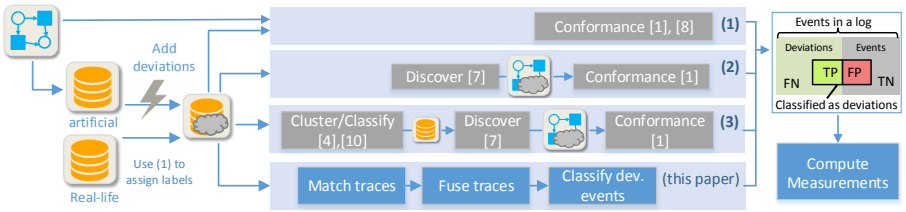

Fig. 6: Experiment design: comparing our approach to existing approaches

each process variant, and then check conformance for each cluster of traces against the corresponding variant model. For conformance checking, we use alignments [1,9]. The Inductive Miner (IMinf) [8] with filter (from 0.2 to 1.0[6]) is used for discovering models and the best result is chosen. For clustering, we used the ActiTraC (4 clusters) [5] and the Generic Edit Distance (GED with 4 and 10 clusters) [2] with standard settings.

We ran this experiment on 1 artificial and 2 real-life logs. In an artificial setting, an artificial normative model was used to generate a perfect log. For each trace in the perfect log, we then randomly add k_{dev} deviating events to derive a log with deviations labeled. The artificial hospital process model in [9] was used for generating event logs. The generated logs contain 1000 cases, 6590 events, and data-based relations between events which are used to derive the execution graphs.

For the two real-life logs, i.e. the MUMC and the Municipality (GOV) logs, we acquired their normative process model and used alignments to label deviating events (thus (1) achieves an accuracy of 1). The labeled real-life logs are then used to compare our approach to (2) and (3). The MUMC data set provided by Maastricht University Medical Center (a large academic hospital in the Netherlands) contains 2832 cases and 28163 events. The Municipality log[7] contains 1434 cases and 8577 events.

Results. In the following, we show results organized in the forms of experiments.
Experiment 1: How does our approach perform in comparison to (1), (2) and (3), and what is the effect of different configurations? Fig. 7 shows the accuracy scores (on the y-axis) of our algorithms along different configurations (on the x-axis)[8]. For other approaches, the accuracy scores remain constant (i.e. the horizontal lines) along our configurations. Interestingly, using the right configuration (highlighted by boxes), the backtracking algorithm is able to detect deviating events more accurately than sequential alignments (1) against the normative model. This is due to the situation in which two events of the same event type executed consecutively. From these two events, sequential alignments cannot find the deviating event, whereas our cost function uses the neighbors and their relative position in a global structure to distinguish them. Both backtracking and greedy have higher accuracies than (2) and (3). Another observation is that a configuration has a strong influence on the accuracy scores since the score fluctuates along the x-axis. We observe that no weight has a dominant effect on the ac-

[6] Using filter from 0.0 to 0.2, IMinf returns a flower model which is the same as classifying all events as conforming.
[7] `http://dx.doi.org/10.4121/uuid:a07386a5-7be3-4367-9535-70bc9e77dbe6`
[8] For each case, we added one deviating event resulting in a log with 13.2% deviating events. Repeating this five times, we show the average *acc* scores.

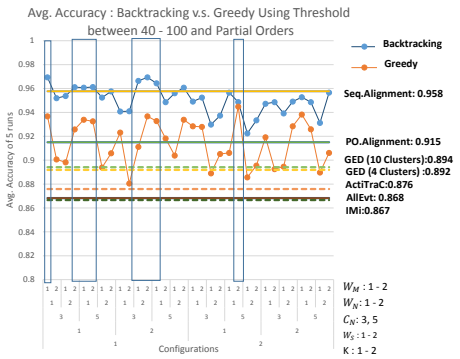

Fig. 7: Avg. *accuracy* scores using data and compared to existing approaches.

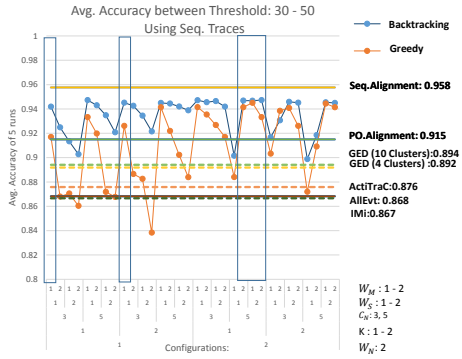

Fig. 8: Avg. *acc* scores using the sequential ordering.

curacy. Some of the configurations that achieve the highest accuracies are the following: $k = 1, c_n = 3, w_M = w_N \geq w_S$, e.g. $w_S = w_M = w_N = 1$ (we write [k1M1N1C3S1] as a shorthand).

Experiment 2: What is the effect of using sequential orders instead of partial orders on the scores? Fig. 8 (similar to Fig. 7) shows the *acc* scores of our approach using sequential ordering. The *acc* scores in Fig. 8 show a decrease in backtracking if sequential ordering is used instead of data-based partial orders. However, we still observe that our approach can perform better than partially ordered alignments [9] and (2) and (3). Interestingly, the greedy approach shows that it is less sensitive for the input format; accuracy is, for some configurations, even higher when using sequential traces.

Experiment 3: What is the effect of different deviation levels? The effects of increasing the number of deviations from 13.2% up to 43.1% (by increasing k_{dev}) on the accuracy of identifying deviating events are shown in Fig. 9. For the backtracking and the greedy approach, we used configuration $[k1M1N1S1C3]$ and configuration $[k2M2N1S1C5]$ based on the previous results. As can be seen, backtracking $[k1M1N1S1C3]$ with $T = 100$ performs as well as (1) using sequential alignments. Also, as expected, using the same configuration but with a lower threshold $T = 40$, the approach classifies fewer events as deviating and therefore is less accurate when the level of deviation increases.

Experiment 4: Performance and Scalability. We compute the average running time of the approach of 5 runs while increasing the average number of events per trace from 6.59 to 10.59. The running time of the greedy algorithm increased only by 78%, from 0.18 min (11.8 sec.) to 0.32 min (19.2 sec.), whereas the backtracking shows an exponential increase from 2.7 min to more than 3 hours, which is more than 10000%. The average running time of using ActiTraC together with discovery and alignments increased from 0.016 min to 0.172 min, showing an increase of 975%. For GED, the average running time increased by 800%, from about 0.010 min to 0.090 min.

Experiment 5: Different Models and Real-life Logs. For the two real-life logs, the results are shown in Fig. 10. For the MUMC data set, existing approaches perform better than our approach. ActiTraC achieves the best accuracy and is about 0.02 higher than

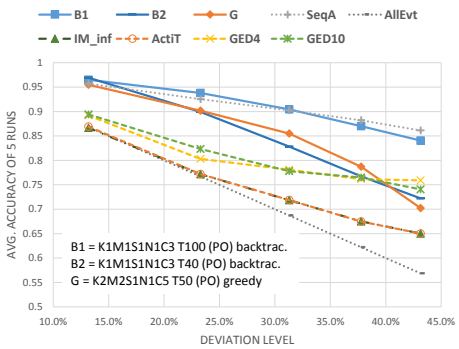

Fig. 9: Effect of deviation level on Backtrac. v.s. Greedy with selected settings.

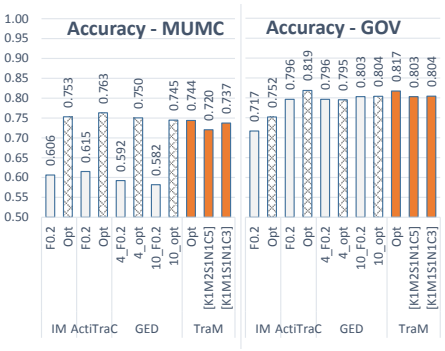

Fig. 10: *Accuracy* of different approaches on real life logs.

our approach. Surprisingly, discovering an imprecise model that allows all activities to be executed in any order was better than applying our approach. For the GOV data set, our approach achieves the second best accuracy with 0.002 lower than the ActiTraC method. Most other approaches perform worse than when classifying all events as conforming behavior. This is due to an event class which occurs frequently in the log and all occurrences are deviations. Techniques (2) based on discovery only are unable to detect these deviations.

7 Discussion and Conclusion

In this paper, we investigated the problem of detecting deviating events in event logs. We compared existing techniques, which either use or construct a normative model to detect deviations via conformance checking, with a new technique that detects deviations from event logs only. The result of our evaluation shows four interesting observations.

Firstly, when the deviations are less structured and the dependencies between events are precise, we can detect deviations as accurately as performing conformance checking using a precise normative model. This indicates that our cost function is indeed able to distinguish individual events and accurately identify similar and dissimilar behavior. However, we also observe that the accuracy of our approach depends heavily on the way the cost function is configured. Some possible solutions to ease choosing a configuration could be: (1) normalizing the cost function (e.g. one divided by each components); (2) having predefined criteria or configurations such as "matching as many events as possible"; (3) showing visual mappings between events, allowing the users to select the right ones, and ranking configurations accordingly.

Another interesting observation is that, using the cost function, the backtracking algorithm performs worse than the greedy approach for sequential traces. This may suggest that the current definition of neighbor and structure is too rigid for sequential ordering of concurrent events. One may consider the union of predecessors, concurrences and successors as the neighbor of an event, instead of distinguishing them.

We also observer that when deviations are frequent and more structured, our approach achieves slight lower accuracy than existing approaches. However, all approaches

performed rather poorly on the real life data sets. One way to improve this is to conduct "cross checking" between different process variants using the mappings between *regs* to find frequent deviations that occur in one variant but not in others. Still, all current approaches have difficulty in detecting very frequent deviations, when no normative model is available, as shown by the results for GOV data sets.

A interesting challenge is to use mappings for detecting other deviations such as missing events. Detecting some events are missing may be simple (e.g. frequent but incomplete nodes in *regs*), whereas the deduction of the exact events that are missing only from an event log appears to be much more difficult. In any cases, it is possible to implement many other deviation classifiers using *regs*, or to use the computed costs of mappings as a measure of similarity for clustering traces and detecting deviating traces instead of events. Future research will be aimed at investigating these possibilities, different cost functions, and the use of *regs* for improving process discovery.

References

1. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Replaying history on process models for conformance checking and performance analysis. Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery 2(2), 182–192 (2012)
2. Bose, R.P.J.C., van der Aalst, W.M.P.: Context aware trace clustering: Towards improving process mining results. In: Proceedings of the SIAM International Conference on Data Mining, SDM 2009, April 30 - May 2, 2009, Sparks, Nevada, USA. pp. 401–412 (2009)
3. Bose, R.P.J.C., van der Aalst, W.M.P.: Process diagnostics using trace alignment: Opportunities, issues, and challenges. Inf. Syst. 37(2), 117–141 (2012)
4. Carmona, J., Cortadella, J.: Process discovery algorithms using numerical abstract domains. IEEE Trans. Knowl. Data Eng. 26(12), 3064–3076 (2014)
5. De Weerdt, J., vanden Broucke, S.K.L.M., Vanthienen, J., Baesens, B.: Active trace clustering for improved process discovery. IEEE Trans. Knowl. Data Eng. 25(12), 2708–2720 (2013)
6. Fahland, D., van der Aalst, W.M.P.: Simplifying discovered process models in a controlled manner. Information Systems 38(4), 585–605 (2013)
7. Ghionna, L., Greco, G., Guzzo, A., Pontieri, L.: Outlier detection techniques for process mining applications. In: Foundations of Intelligent Systems, 17th International Symposium, ISMIS 2008, Toronto, Canada, May 20-23, 2008, Proceedings. pp. 150–159 (2008)
8. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs containing infrequent behaviour. In: BPM Workshops 2013, Beijing, China, August 26, 2013. pp. 66–78 (2013)
9. Lu, X., Fahland, D., van der Aalst, W.M.P.: Conformance checking based on partially ordered event data. BPM workshops 2014 (2014)
10. Nguyen, H., Dumas, M., Rosa, M.L., Maggi, F.M., Suriadi, S.: Mining business process deviance: A quest for accuracy. In: CoopIS, and ODBASE 2014, Amantea, Italy, October 27-31, 2014, Proceedings
11. Rebuge, A., Ferreira, D.: Business Process Analysis in Healthcare Environments: A Methodology Based on Process Mining. Information Systems 37(2) (2012)
12. Suriadi, S., Wynn, M.T., Ouyang, C., ter Hofstede, A.H., van Dijk, N.J.: Understanding Process Behaviours in a Large Insurance Company in Australia: A Case Study. In: Advanced Information Systems Engineering. pp. 449–464. Springer (2013)
13. Yang, W., Hwang, S.: A process-mining framework for the detection of healthcare fraud and abuse. Expert Syst. Appl. 31(1), 56–68 (2006)