"The final publication is available at Springer via http://
dx.doi.or/10.1007/978-3-319-65000-5".

Multi Instance Anomaly Detection in Business Process Executions

Kristof Böhmer and Stefanie Rinderle-Ma

University of Vienna, Faculty of Computer Science
{kristof.boehmer,stefanie.rinderle-ma}@univie.ac.at

Abstract. Processes control critical IT systems and business cases in dynamic environments. Hence, ensuring secure model executions is crucial to prevent misuse and attacks. In general, anomaly detection approaches can be employed to tackle this challenge. Existing ones analyze each process instance individually. Doing so does not consider attacks that combine multiple instances, e.g., by splitting fraudulent fund transactions into multiple instances with smaller "unsuspicious" amounts. The proposed approach aims at detecting such attacks. For this, anomalies between the temporal behavior of a set of historic instances (ex post) and the temporal behavior of running instances are identified. Here, temporal behavior refers to the temporal order between the instances and their events. The proposed approach is implemented and evaluated based on real life process logs from different domains and artificial anomalies.

Keywords: Runtime Anomaly Detection, Secure Business Processes, Multiple Instances, Temporal Anomalies

1 Introduction

Business process anomaly detection identifies *anomalous behavior* in recorded (ex post) or ongoing (real time) process executions in order to expose and prevent fraud, misuse, unknown attacks, and errors. Hence it constitutes a critical IT security defense line in today's interconnected business driven organizations [3, 5]. Existing process anomaly detection work analyzes single process instances in order to distinguish if their behavior is anomalous (i.e., *unlikely*) or not. Doing so does not provide protection against all possible attack vectors. For example, assume an attack scenario where the attacker (Trudy) strives to quickly transfer funds from an organization's bank account. Therefore, Trudy could instantiate a single transaction process and transfer all the money at once. Alternatively, Trudy could start multiple transaction processes in parallel and split up the transactions into smaller chunks. The first approach would likely be detected by existing anomaly detection approaches while the second would not.

This is, because in the first case the transferred funds are exceptionally high, i.e., they exceed previously transferred funds and are, therefore, unlikely. In this case, analyzing each process instance execution individually is sufficient to identify Trudy's attack. In the second case each individual process instance only

transfers a small amount of money. Through this, the transferred funds are, likely, comparable to transactions represented in the known historic behavior. Accordingly the executions would *not* be identified as anomalous.

Hence, an anomaly detection approach is required that is able to consider *multiple* instances – from the same or different process models. In the example scenario these are instances which take place before, during, or after one of the fraudulent transaction process executions. Through this the parallel executions are noticed as unusual and the second attack scenario is identified. The assumption behind this is that the massive parallel execution of multiple transaction processes – which was never observed before – is unlikely (i.e., anomalous).

This paper proposes a configurable and unsupervised anomaly detection heuristic for business processes that exploits the temporal dependencies between multiple instances. In detail, for each instance of interest, all temporal relations of preceding, succeeding, and simultaneous process executions are taken into account. The business process instances and executions which are taken into consideration can stem from *various models*, i.e., not all instances need to be spawned from the same process model but, e.g., from multiple models.

This work applies *design science research*, cf. [14]. Doing so multi instance process executions were identified as a problem (i.e., an unprotected attack vector). To tackle this problem artifacts are created and evaluated, here this is a prototypical implementation of the proposed multi instance anomaly detection approach. Stakeholders for the approach are organizations and security experts.

More precisely, we assume a set of process models R, and a set of execution log files L. Hereby, R could be a process repository and L holds all executions of the processes in R. The key idea is to generate an anomaly detection signature G for a process $P \in R$ so that it represents for P the behavior of P's instances and temporally related instances from multiple other models. This is achieved by mining and combining temporal relations from multiple instances, stored in L, that take place during or close to executions of P into a signature G.

Finally, behavior that should be analyzed for anomalies, from P's instances and other temporally related instances, is assumed as given. For example, such behavior can be extracted from logs, for ex post analysis, or be collected directly during model executions – from process execution engines – for real time analysis. To analyze if behavior is anomalous or not it is mapped to G, which enables to calculate its behavior likelihood. If the behavior to analyze for anomalies is found to be unlikely, when comparing it to the logged historic behavior in L, then the behavior is identified as anomalous. The artifacts generated in this work comprise multi instance process behavior mining and signature generation and matching algorithms. The presented approach is evaluated using real life process execution logs from multiple domains along with artificially generated anomalies.

This paper is organized as follows: Related work is discussed in Section 2. Prerequisites and the proposed approach are introduced in Section 3. The proposed anomaly detection approach (i.e., signature generation and matching) is, in detail, described in Section 4. Section 5 holds the evaluation. Finally, conclusions, discussions, and future work is given in Section 6.

2 Related Work

Related anomaly detection work was searched for in the *process domain* and in the *security domain*. The results found for the process domain were limited as related work focuses only on single individual process models and instances. Hence, the anomalies which this work is capable of identifying are not supported by existing process anomaly detection work. Our systematic literature review in [6] provides a more detailed analysis. The most comparable work [12] analyzes temporal behavior of individual activities to identify unlikely anomalous execution behavior. However, this work also concentrates only on single instances.

In a broader context, i.e., the security domain in general, several temporal anomaly detection approaches are suggested, cf. [10]. However, according to [10] and our own findings, those approaches are typically domain or data specific (i.e., focus on specific protocols, such as, SIP or network packages) and can, because of this, hardly be generalized, e.g., to analyze process behavior data for anomalies. It can be concluded that an anomaly detection approach specifically tailored for process behavior is a necessity to identify related attacks and anomalies.

Moreover, it was found that existing approaches show, likely, an underwhelming anomaly detection performance when dealing with unexpected behavior. Existing anomaly detection work frequently classifies unexpected behavior, even if it only slightly deviates from, e.g., a signature, as anomalous. This could potentially result in a large amount of false positives [5] in flexible and dynamic execution scenarios. Hence, this work proposes a novel approach to deal with unexpected behavior by assigning it with an artificially calculated likelihood.

Existing approaches which are comparable to the presented artificial likelihood calculation are so called *soft matching* techniques. Soft matching generalizes expected behavior patterns by constructing multiple slightly deviating, but still presumably "correct" patterns (e.g., based on expert knowledge) [2]. Hereby the area of data or behavior which is identified as non-anomalous is widened. Unfortunately, it frequently requires expert knowledge to soften the patterns and through this soft matching lacks in flexibility, compared to the presented automatic approach. Moreover, the presented approach enables to "aggregate" multiple occurrences of slightly unlikely behavior to identify collective anomalies [5] which could be missed by less sensitive detection approaches. This is because this work does not flag each observed process execution behavior solely as anomalous or non-anomalous. Instead a more flexible likelihood is calculated and aggregated over multiple successive process execution events and instances.

3 Prerequisites and General Approach

This paper proposes a multi instance anomaly detection approach that enables to distinguish process execution behavior as anomalous (i.e., unlikely) or not. For this the behavior is compared with a signature generated from given process execution logs L which represent historic process executions. Generating signatures from logs is beneficiary as logs are frequently generated automatically by

today's process execution engines, contain real behavior and executions, and include manual adaptations. Moreover, exploiting execution logs enables to become independent from abstracted and potentially outdated documentation [11].

Let each execution log $l \in L$ hold the associated process model **n**ame and a bag of execution **E**vents, i.e., $l := (n, E)$. Each execution event $e \in l.E$, i.e., $e := (s, c)$ represents an activity execution by its **s**tart and **c**ompletion timestamp s and c respectively, with $s, c \in \mathbb{N}_{>0}$. An exemplary log for model A with two activity executions could be defined as $l_A := (A, \{(s_1, c_1), (s_2, c_2)\})$. We assume that each individual model execution (i.e., each instance) is held by an individual execution log $l \in L$ and that timestamps are defined in a range of $\mathbb{N}_{>0}$.

Such a brief definition is sufficient because the presented approach mainly analyzes temporal relations between models and their instances. For this the following auxiliary functions, inspired by a subset of Allen's interval algebra [1], are defined. The start timestamp of an instance execution is found by $\texttt{min}(E) := \{e.s | e \in E; \forall e' \in E, e.s \leq e'.s\}^0$. Here, $\{\cdots\}^0$ returns the only element held by a set or bag if it is a *singleton* or a random set/bag element if it is not. A similar definition is applied for $\texttt{max}(E)$ to determined the end timestamp of an instance execution. Through this the duration of an instance is $\texttt{dur}(E) := \texttt{max}(E) - \texttt{min}(E)$. Moreover, $\texttt{execP}(t, L)$ extracts a bag of process model names that are executed at point t (i.e., a timestamp) based on the logs in L, i.e., $\texttt{execP}(t, L) := \{l.n | \exists l \in L, \exists e \in l.E; e.s \leq t \wedge e.c \geq t\}$. Similarly, $\texttt{act}(t_1, t_2, L)$ counts the activities that are executed in a specific interval given by t_1, t_2, i.e., $\texttt{act}(t_1, t_2, L) := |\{e | l \in L, e \in l.E; e.s \geq t_1 \wedge e.c \leq t_2\}|$ where $t_1 \leq t_2$. Function $\texttt{next}(t, L)$ determines the process instance start or end timestamp that occurs as close after t as possible, i.e., $\texttt{next}(t, L) := \{t_1 | t_1, t_2 \in T; \nexists t_2 < t_1; t_1 > t \wedge t_2 > t\}^0$ where $T := \{e.s | l \in L, e \in l.E\} \cup \{e.c | l \in L, e \in l.E\}$. Further, $\texttt{mid}(t_1, t_2) := t_1 + ((t_2 - t_1)/2)$ calculates the average of two timestamps where $t_1 < t_2$.

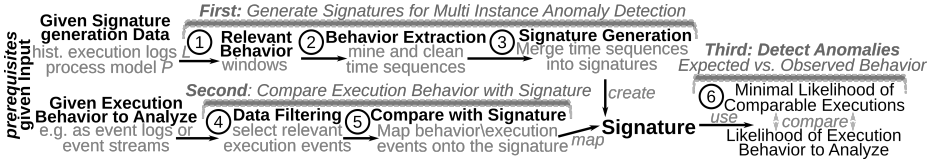

Fig. 1. Proposed multi instance anomaly detection approach – overview

Fig. 1 provides an overview on the proposed anomaly detection heuristic. The related algorithms are presented in Sect. 4. Firstly, a signature is generated for a process $P \in R$ based on a set L of historic instance executions – both are assumed as given input. The first idea is to extract process execution events in L that precede, succeed, or occur simultaneously to executions of P's instances, cf. Fig. 2. The figure depicts three processes – A, B, and C – along with a number of instances (i.e., the rectangles, e.g., $AI1$ to $AI4$) and activity execution events (i.e., the vertical bars in the instance rectangles, e.g., $AI3$ holds 4 activity executions). For the sake of brevity Fig. 2 depicts only a snapshot of all instance executions, i.e., additional instances are stored in L but not depicted.

Assume a signature is generated for process B in Fig. 2. Then the signature generation starts by identifying *relevant* execution events in L (i.e., historic

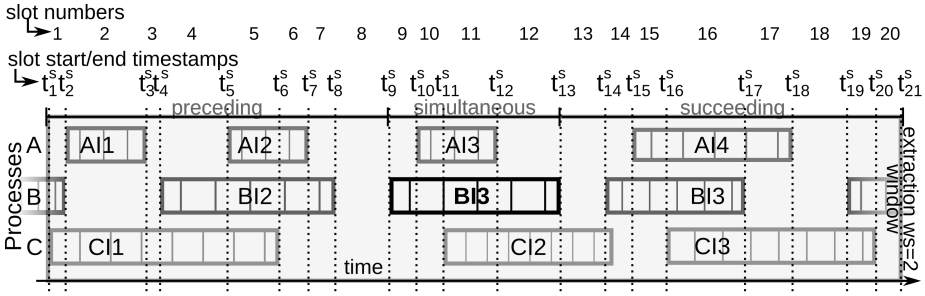

Fig. 2. Running example for window & behavior extraction and noise reduction

behavior) ①. Relevant events are events which most likely affect B's instances (i.e., events which precede B's instances) or which are affected by B's instances (i.e., events which are succeeding and simultaneous to B's instances). Extraction windows are applied to identify such events in the following.

An individual *extraction window* $w := [wt_1; wt_2]$ is created for each of B's instances ①. Extraction windows enable to determine which of the behavior held by L is relevant (i.e., preceding, succeeding, and simultaneous events) for a specific instance and model and should, therefore, be contained in the generated signature. The beginning and end of the window (i.e., wt_1 and wt_2) is calculated by multiplying the duration of the respective instance (this example uses $BI3$ and $l_{BI3} \in L$) with a user chosen **w**indow **s**ize modifier $ws \in \mathbb{R}_{>0}$. So $wt_1 := \texttt{min}(l.E) - (ws \cdot \texttt{dur}(l.E))$ and $wt_2 := \texttt{max}(l.E) + (ws \cdot \texttt{dur}(l.E))$. Hence, when assuming $ws = 2$ and $\texttt{min}(l_{BI3}.E) = t^s_9$, $\texttt{max}(l_{BI3}.E) = t^s_{13}$ then $w_{BI3} = [t^s_1; t^s_{21}]$.

The size of an extraction window is defined in a direct relation to the duration of the corresponding instance. Moreover, the parameter ws enables to adapt the extraction window size to the density of the analyzed event logs. For example, if a log is very dense (i.e., it holds a large amount of events in a short timespan) then applying extraction windows with a fixed size could result in an overly detailed signature (i.e., overfitting occurs) which could, subsequently, lead to flawed anomaly detection results, cf. [7]. In comparison a sparse log combined with a fixed size window could result in a signature that contains insufficient historic behavior to identify anomalies (i.e., underfitting occurs).

Subsequently step ② is applied to mine all the behavior that occurs in a chosen window in a *time sequence*. Therefore, the window is split into multiple *slots* based on the start and end of the process instances covered by the window (i.e, the dotted lines and slot timestamps in Fig. 2). Each slot is defined as $o := (N, t^s_s, t^s_e)$ where t^s_s and t^s_e represent the **s**tart and **e**nd timestamps of the slot and the bag $N := \texttt{execP}(\texttt{mid}(t^s_s, t^s_e), L)$ holds the names of models whose instances occur between t^s_s and t^s_e. For example, slot 5 in Fig. 2 would be defined as $o_5 = (\{A, B, C\}, t^s_5, t^s_6)$. If multiple instances from the same process model are executed in parallel then the related model's name occurs in $o.N$ multiple times (e.g., two parallel executions of model A would result in $o.N = \{A, A\}$). Subsequently, all slots are combined into a time sequence, i.e., an ordered list of slots $ts := \langle o_1, o_2, \cdots, o_n \rangle$, e.g., $ts_{BI3} = \langle o_1, o_2, \cdots, o_{20} \rangle$ for instance $BI3$, cf.

Fig. 2. Finally, noise in the mined time sequences is addressed, for example, by removing slots which do not cover any instance execution (e.g., slot 8 in Fig. 2).

The signature generation ends by merging the resulting time sequences from all windows (one window for each instance is generated) into one signature ③. The signatures are represented as *likelihood graphs*, which were also already successfully applied in [5] for this purpose. Here, likelihood graphs enable to calculate the likelihood of instance behavior to determine unlikely (i.e., anomalous) ones. For this a likelihood graph encodes which and how instances and models are typically temporally related to each other during their execution (e.g., how instances succeed or precede each other). Moreover, it encodes the likelihood and order of such relations based on the mined time sequences, cf. Fig. 4.

Secondly, the signature is utilized to assess if a given process instance execution behavior, for P's instances, is anomalous or not. Hence, given behavior is filtered ④, and mapped ⑤ to the signatures (i.e., for each process model an individual signature is generated) to determine the likelihood of given instance execution behavior. Of course, some of the instance behavior could be unexpected because it never occurred before and is, accordingly, also not represented by the signatures (i.e., it cannot be mapped to a signature), cf. [5]. In such cases a configurable artificial behavior likelihood is calculated to flexibly deal with noise and slight – likely harmless – deviations from the historic behavior.

Thirdly, the likelihood of the given instance execution behavior is compared to a reference likelihood generated from P's historic instance executions stored in the historic execution log files L, ⑥. If a deviation between both likelihoods (reference likelihood and likelihood of the given instance execution to analyze) is observed then the analyzed given instance execution is identified as anomalous.

4 Multi Instance Anomaly Detection

This section presents the algorithms for the approach set out in Fig. 1.

4.1 Temporal Behavior Mining from Execution Logs

The proposed anomaly detection approach starts with a process model $P \in R$ (i.e., a signature is generated for P) and logs containing historic process execution behavior L. Subsequently, extraction windows are constructed for P's instances, as described before. This enables to mine historic execution behavior that takes place before, during, and after P's executions as *time sequences*.

Mining Time Sequences Each time sequence is a sequence of slots $o := (N, t_s^s, t_e^s)$ which are ordered based on their end timestamp, i.e., t_e^s. In the following a signature is generated by merging multiple time sequences. The presented mining approach generates a time sequence (i.e., a sequence of slots) for each extraction window – and through this for each instance. Therefore, the start and end timestamps of each process instance covered by the window are exploited, i.e., the dotted vertical lines in Fig. 2. Alg. 1 formalizes the mining of a single

Algorithm mineTS(*extraction window* $w := [wt_1; wt_2]$, *execution logs* L)
Result: mined time sequence ts
$ts := \langle\rangle$; $first := w.wt_1$ // initially ts is empty
// extract the interval between instance start and end timestamps as slots
while $second := next(first, L) \wedge second < w.wt_2$ **do**
$ts := ts \oplus (\texttt{execP}(\texttt{mid}(first, second), L), first, second)$
$first := second$ // preserve for next iteration
// interval from the last instance start or end till the end of the window
$ts := ts \oplus (\texttt{execP}(\texttt{mid}(first, w.wt_2), L), first, w.wt_2))$
return ts // the mined time sequence for the window w and a given log L

Algorithm 1: Mines a time sequence for a given window w and logs L

time sequence for a given window w and the historic execution logs in L. In the following the symbol $\oplus$ denotes the appending of a slot to the end of a sequence.

The time sequence generated for the execution scenario depicted in Fig. 2 is shown at the left side of Fig. 3. For the sake of brevity Fig. 3 only depicts the number of the respective slot and the covered process model names while start and end timestamps are omitted. Only the first eight slots are depicted.

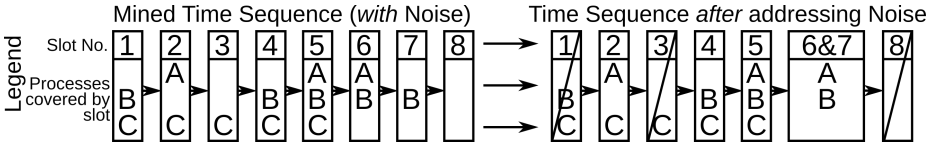

Fig. 3. Time sequence before and after addressing noise, window size is $ws = 2$

Addressing Noise in Time Sequences The mined time sequences likely contain slots which are not relevant for or even interfering with the following signature generation. Such noise in the time sequences could result in an overfitting of the generated signatures, i.e., the signatures would be "too" specific and detailed, cf. [7]. This could result in false positives, i.e., non-anomalous executions which are incorrectly identified as anomalous. So, the proposed noise reduction heuristic will deal, for a given time sequence ts, with all slots which are *empty* or *volatile*. A slot $o := (N, t_s^s, t_e^s)$ is empty if $o.N = \varnothing$, i.e., if no instance is executed at the timespan (t_s^s to t_e^s) covered by the slot, e.g., slot 8 in Fig. 2 is empty.

Moreover, a slot is volatile if it covers only a low amount of activity executions. Typically volatile slots are placed at the beginning or end of instances and cover only a short time span. Hence even a minor shift in an instance's start or end timestamp can have a large impact on the slot. For example, if instance $BI2$'s duration, cf. Fig. 2, would only be a bit shorter (e.g., when it would end at t_7^s instead of t_8^s) then slot 7 would no longer be present or be part of the mined time sequence. Formally, a slot o is identified as volatile if $\texttt{act}(o.t_s^s, o.t_e^s, L) < c$, i.e., c controls the minimum number of activities covered by the slot, cf. Alg. 2.

Empty slots are removed from a time sequence ts, using list comprehension notation, i.e., $ts := \langle o \in ts | o.N \neq \varnothing \rangle$. For volatile slots, in comparison, it is checked if they could be aggregated with *one or more* directly successive slots, which are also volatile, to become non-volatile. If this is not possible then they are also removed, cf. Fig. 3 (right side). For this Alg. 2 must identify directly connected volatile slots. Hence, the algorithm stores the start of the first volatile slot $volS$ and the end of the most recent successive volatile slot in $volE$. Based on this information $\texttt{act}(volS, volE, L) > c$ enables to determine if an aggregation

of the found successive volatile slots results in a non-volatile slot. Imagine that slot 18 ($\{C\}, t^s_{18}, t^s_{19}$) and 19 ($\{B, C\}, t^s_{19}, t^s_{20}$) in the running example Fig. 2 were found as volatile. This is because c was assumed as 3 and each of both slots covers less than three complete activity executions, i.e, $\texttt{act}(t^s_{18}, t^s_{19}, L) = 2$ for slot 18 and $\texttt{act}(t^s_{19}, t^s_{20}, L) = 2$ for slot 19. However, by aggregating both slots a new slot is created that is not volatile. Thus the aggregated slot becomes ($\{B, C\}, t^s_{18}, t^s_{20}$) (i.e, $\texttt{act}(t^s_{18}, t^s_{20}, L) = 4$) and replaces the old slots 18 and 19.

Algorithm addressVolatileSlotsInTS(*time sequence* $ts = \langle o_1, o_2, \cdots, o_n \rangle$, *execution logs* L, *slot volatile threshold* $c \in \mathbb{N}_{>0}$)
- **Result:** noise free time sequence ts_{nv} (volatile slots were aggregated or removed)
- $ts_{nv} := \langle\rangle$;$volN := \varnothing$; $volS := 0$; $volE := 0$ // store intermediate results for the following steps, $volS$ and $volE$ are timestamps while $volN$ holds model names
- **foreach** $o \in ts$ **do**
  - **if** $act(o.t^s_s, o.t^s_e, L) < c$ // check if o is a volatile slot **then**
    - $volE := o.t^s_e$; $volN := volN \cup o.N$// aggregate slots
    - **if** $volS = 0$ // i.e., first volatile slot found **then**
      - $volS := o.t^s_s$ // preserve start time of the first volatile slot found
    - **else if** $act(volS, volE, L) > c$ // aggregated slot is not volatile **then**
      - $ts_{nv} := ts_{nv} \oplus (volN, volS, volE)$ // append aggregated slot on ts_{nv}
      - $volN := \varnothing$; $volS := 0$; $volE := 0$ // purge preserved data
  - **else**
    - // a non-volatile slot was found, purge preserved data because only directly successive volatile slots are aggregated
    - $ts_{nv} := ts_{nv} \oplus o$; $volN := \varnothing$; $volS := 0$; $volE := 0$
- **return** ts_{nv}// similar to input time sequence ts but without volatile slots

Algorithm 2: Addressing volatile slots in mined time sequences

Consider the right side of Fig. 3. The original time sequence (left side, cf. the running example in Fig. 2) was adapted to remove or address noise (right side). Note, slots which are crossed out were removed because they are empty (slot 8) or volatile slots which could not be aggregated with other volatile slots (slot 1 and 3). Two volatile slots (slot 6 and 7) were replaced by an aggregated non-volatile slot, i.e., "6&7". The slot volatile threshold c was assumed as three.

4.2 Signature Generation from Time Sequences

Subsequently, signatures are generated for each individual process model P based on time sequences TS which were generated for P's historic instances in L. For this the noise free time sequences are merged and transition likelihoods are calculated. Transition likelihoods represent the likelihood that slots which cover specific instances of processes follow each other (based on all time sequences $ts \in TS$). For example, the likelihood that a slot which holds an execution of process model A, B, and C is followed by a slot with execution A and C, cf. slots 16 and 17 in Fig. 2. In the following this likelihood information is utilized to differentiate between likely and unlikely (i.e., anomalous) model instance executions.

This work proposes to represent the mined temporal behavior in three independent signatures (i.e., one for behavior that happens before, during, or after a process model's execution). For this the mined time sequences are split accordingly into three parts – based on P's associated instance starts and ends, cf. Fig. 2. This decreases the size of each signature. Also this was found to increase the anomaly detection performance of the presented approach. The latter is because during each point in time a signature can be applied that specializes on

the specific kind of behavior that is currently observed (e.g., behavior that was historically observed after or during an instance execution). Hereby, the applied signature can be more specific than one large generic signature that needs to cover all the historic instance behavior (i.e., before, during, and after) at once.

Each signature is represented as a likelihood graph $G = (V, D)$, cf. [5]. A likelihood graph is a directed cyclic graph that consists of a set of vertexes $v \in V$ and a set of edges $d \in D$ with $D \subseteq V \times V \times [0;1]$. Each vertex v represents processes covered by a specific $o.N$ for a given slot o. In comparison each edge $d = (v_s, v_e, tl)$ represents the *transition likelihood* $tl \in [0;1]$ from one "slot" v_s (i.e., a vertex holding the process model names covered by a slot) to another vertex v_e based on the mined time sequences $ts \in TS$.

Alg. 3 creates a likelihood graph (i.e., a signature) by merging multiple time sequences. For this, the algorithm extracts from each slot, covered by the merged time sequences, the processes covered by that slot and stores them in the set V. Moreover, the set VC is populated with triplets $vc = (v_1, v_2, tc)$ that indicate, based on the analyzed time sequences, that a slot v_2 is preceded by a slot v_1, $tc \in \mathbb{N}_{>0}$ times (i.e., tc denotes the *transition count*). Subsequently these absolute numbers (i.e., tc) are converted into relative transition likelihoods tl and stored into D as edges. V is initialized with a dummy entry v_d that is used as a general entry point for the signature and the following mapping of behavior to it.

Algorithm `mergeTimeSequencesIntoSignature(`*time sequences* TS*, dummy vertex, i.e., the entry point for the signature* v_d`)`
- **Result:** likelihood graph (i.e., a signature) $G = (V, D)$ from the behavior in TS
- $V := \{v_d\}$; $D := \varnothing$; $VC := \varnothing$
- **foreach** $ts \in TS$ *where* $|ts| > 0$ **do**
  - $VC := VC \cup \{(v_d, ts_0, 1)\}$ `// add dummy vertex,` ts_0 `identifies the first slot in` ts`, i.e.,` ts_i `with` $0 \geq i < |ts|$ `identifies the slot with the index` i
  - **for** $i := 0; i < (|ts| - 1); i := i + 1$ **do**
    - $v_1 := ts_i.N$; $v_2 := ts_{i+1}.N$ `//` ts_i `and its successor` ts_{i+1} `in` ts
    - $V := V \cup \{v_1, v_2\}$; `// add slots to signature graph vertex set` V
    - $tcount := 1$ `// holds how frequently` v_1 `is followed by` v_2 `in all sequences`
    - **if** $(v_1, v_2, \cdot) \in VC$ `// previous` ts `contained the same transition` **then**
      - $tcount := tcount + \{vc.tc | vc \in VC, vc.v_1 = v_1 \wedge vc.v_2 = v_2\}^0$
    - `// purge old information for` v_1/v_2`, then add updated or new information`
    - $VC := \{vc \in VC | vc.v_1 \neq v_1 \wedge vc.v_2 \neq v_2\} \cup \{(v_1, v_2, tcount)\}$
- **foreach** $vc \in VC$ `// convert absolute numbers into likelihoods` **do**
  - $s := vc.tc; TC := \{vc'.tc | vc' \in VC \wedge vc'.v_1 = vc.v_1\}$
  - `// create and add edges to` D `that connect the signature vertexes in` V
  - $D := D \cup \{(vc.v_1, vc.v_2, \frac{s}{\sum_{tc_s \in TC} tc_s})\}$ `// fraction` $\mapsto$ `transaction likelihood`
- **return** $G = (V, D)$ `// return signature, it was created for sequences in` TS

Algorithm 3: Merge time sequences TS for P into a signature G

Fig. 4 depicts an example for the proposed signature generation. Two time sequences ($XI1$ and $XI2$, mined for the process "X" – left side) are merged into a likelihood graph signature representation (right side). The depicted time sequences and represented behavior occurred after X's instances (i.e., succeeding behavior). So process X is placed at the start of the time sequences and signature.

4.3 Signature Matching for Execution Event Streams and Logs

The signatures are applied to calculate the likelihood of process execution behavior based on *given execution events*. Today's execution engines store (ex post

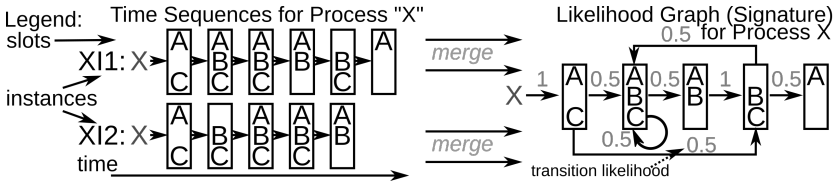

Fig. 4. Merging two mined time sequences into a signature

analysis) or stream (real time analysis) various events. This work is mainly interested in process instance and activity start and end events. Hence, all perceived events are filtered accordingly and mined into time sequences by applying the presented approach. Finally the resulting time sequences are mapped to signatures which were generated for the executed processes based on historic behavior.

To determine if a given process instance execution is unlikely (i.e., anomalous) or not its likelihood is calculated (i.e., *execution likelihood* $l_e \in \mathbb{R}_{>0}$) by mapping it to the signatures. Moreover, comparable executions (i.e., that show a similar temporal execution behavior to the given execution) are identified in the historic logs L and mapped to the same signature as the given execution to generate a *reference likelihood* $l_r \in \mathbb{R}_{>0}$. Finally, both likelihoods are compared. If the execution likelihood is *below* the smallest found reference likelihood then the analyzed execution is identified as unlikely and, because of this, as anomalous.

The *execution likelihood* is calculated by Alg. 4. Therefore, a signature (i.e., a likelihood graph $G = (V, D)$) is utilized along with a time sequence ts that is mapped to the signature (i.e., the sequence ts, here, represents given instance execution behavior that should be analyzed for anomalies). To calculate the likelihood, the slots held by the time sequence are mapped to the signature one after another while aggregating the transition likelihoods encoded in the edges $d \in D$ that connect all signature vertexes. Finally, when all recorded behavior (i.e., the time sequence ts) was mapped the likelihood is returned.

Algorithm matchSig(*signature* $G = (V, D)$, *dummy vertex* v_d *representing the process which* G *was generated for, time sequence* ts *holding behavior to map, punishment factors* $pNDC, pDP, pOS \in (0; 1]$)
  Result: calculated likelihood $lh \in \mathbb{R}_{>0}$ for ts
  $lh := 1$; $v_l := v_d$ // behavior likelihood lh and most recent signature vertex v_l
  foreach $o \in ts$ // individually for each slot **do**
    $lh' := \{d.tl \in D | d.v_1 = v_l \wedge d.v_2 = o.N\}$
    $P := \{(v', sim) | v' \in V; sim := \frac{|v' \triangle o.N|}{|v'|+|o.N|}, \forall a \in o.N, \exists b \in v'; a = b\}$
    $MS := \{(v', sim) | v' \in V; sim := \frac{|v' \triangle o.N|}{|v'|+|o.N|}\}$// $\triangle$ notates a symmetric difference
    if $lh' = \varnothing$ // unexpected behavior was found **then**
      if $o.N \in V$ // stage one: exact behavior is present in the signature **then**
        $v_l := o.N$; $lh := lh \cdot pNDC$
      else if $P \neq \varnothing$ // stage two: present but different parallelism **then**
        $min := \{p_1 | p_1 \in P; \forall p_2 \in P, p_1.sim \leq p_2.sim\}^0$
        $v_l := min.v$; $lh := lh \cdot (1 - min.sim) \cdot pDP$
      else
        // stage three: fallback if one and two are not applicable
        $min := \{ms_1 | ms_1 \in MS; \forall ms_2 \in MS, ms_1.sim \leq ms_2.sim\}^0$
        $v_l := min.v$; $lh := lh \cdot (1 - min.sim) \cdot pOS$
    else
      $v_l := o.N$; $lh := lh \cdot lh'^0$ // if the behavior is expected
  return lh// return likelihood lh of the behavior in ts

Algorithm 4: Likelihood for a time sequence ts based on a signature G

Of course, it is possible that some behavior cannot be mapped successfully. For example, this is the case if behavior occurs in unexpected orders, e.g., instance A is succeed by B but it was expected (i.e., specified in the signature) the other way around. Another reason could be that the parallelism of observed and the expected behavior deviates, e.g., it was expected that a single instance of A is executed, but two concurrent executions of A were observed.

Existing process anomaly detection work typically classifies any unexpected behavior, such as the preceding examples, as anomalous. However, as argued in [5] this is not always beneficial. Process models and model executions are known to occur in flexible dynamic environments, struggling with ad-hoc changes, and the need to cope with multiple frequently changing requirements [9]. Hence, we assume that existing anomaly detection approaches are too strict to be successfully applied in today's flexible and dynamic process execution environments. So, the proposed anomaly detection approach provides the flexibility to deal with unexpected behavior by calculating an artificial likelihood for it.

The flexibility that should be granted by the proposed anomaly detection approach varies between different organizations, processes, and use cases. Hence, the flexibility can be configured in Alg. 4 based on three punishment factor variables, i.e., $pNDC$, pDP, and pOS. Those enable to punish unexpected behavior by reducing its calculated artificial likelihood. Hence, while the scenario in the initial motivating example is identified as anomalous – because significantly more parallelism is observed than expected based on historic executions – minor, probably harmless, deviations from the historic behavior are "granted" until, e.g., a combination of multiple minor deviations becomes too unlikely.

So when calculating the *execution likelihood* it is checked if the current slot $o \in ts$ (i.e., the behavior to map next) is a direct successor of the last mapped slot v_l. If it is, then the likelihood is extracted from the related transition likelihood hold by the signature in D. Hence, when mapping the short example timesequence $ts := \{(\{X\}), (\{A, C\}), (\{A, B, C\})\}$ (timestamps are omitted) on the signature depicted in Fig. 4 then a likelihood of $1 \cdot 0.5 = 0.5$ is calculated. However, if o is not a successor of the last mapped slot then unexpected behavior was found. For this, it becomes necessary to calculate an artificial likelihood by applying a three staged approach which is discussed in the following.

Stage *one*: It is checked if the unexpected behavior (i.e., slots) is represented in the signature but occurred in an unexpected order. If this is the case then the punishment factor $pNDC$ it utilized as the artificial likelihood. Stage two and three calculate the artificial likelihood based on the similarity of the given instance behavior and the behavior represented in the signature. Stage *two* is applied if the expected processes are executed but with an unexpected parallelism. For example, A, A, B was observed but expected was A, B, i.e., two parallel executions of process A were found but only one was expected. This stage utilizes the punishment factor pDP. Stage *three*, which utilizes the punishment factor pOS, can always be applied and is, because of this, used as a fallback. It is similar to stage two but more relaxed, i.e., it does not enforce that the behavior to map and the related signature behavior must only consist of the same processes.

Imagine that the slot o with $o.N := \{A, A\}$ should be mapped to a signature which only consist of A as the expected behavior. Because, the observed $\{A, A\}$ and the expected $\{A\}$ behavior is different the slot cannot be found in the signature. Hence the proposed approach falls back to stage two of the artificial likelihood calculation. So the likelihood is calculated as $\frac{|\{A\} \triangle \{A,A\}|}{|\{A\}|+|\{A,A\}|} \mapsto \frac{1}{3} = 0.\bar{3}$ so that the final artificial likelihood becomes $(1 - 0.\bar{3}) \cdot 0.8 = 0.5\bar{3}$ when a punishment factor pDP of 0.8 is used.

The *reference likelihood* l_r is calculated based on logged historic executions in L that show comparable behavior to the given behavior (i.e., given instance execution behavior to analyze for anomalies). In this case comparable means that the time sequence describing the given behavior and the time sequences describing the historic behavior hold similar slots. For this the presented approach to measure the similarity between two slots (i.e., for artificial likelihood calculation, cf. Alg. 4) is generalized and applied on the historic time sequences which were mined from L during the signature generation. The $k \in (0, 1]$ percent most similar historic time sequences are subsequently compared with the signature G using Alg. 4. Finally the lowest likelihood found during that comparisons is utilized as the reference likelihood l_r. If $l_e < l_r$ then the given behavior (i.e., the behavior that is analyzed for anomalies) is identified as anomalous. This bears two advantages: Executions in L are never identified as anomalous and the flexibility which was historically observed for the process under analysis, and which is because of this stored in L, is taken into account during anomaly detection.

5 Evaluation

The evaluation utilizes real life process execution logs from multiple domains and artificially generated anomalies in order to assess the anomaly detection performance and feasibility of the proposed approach. It was necessary to generate artificial anomalies as information about real anomalies are not provided by the log sources. The utilized logs were taken from the BPI Challenge 2015[1] and 2017[2] (BPIC5 and BPIC7), and Higher Eduction Processes (HEP), cf. [13].

The BPIC5 logs hold 262,628 execution events which origin from 5,649 instances and 398 activities. The logs cover the processing of building permit applications at five (BIPC5_1 to BPIC5_5) Dutch building authorities between 2010 and 2015. In comparison the BPIC7 logs hold 1,202,267 events from 31,509 instances, recorded in 2016 and 2017, which focused on loan application management. The HEP logs contain 28,129 events, 354 execution traces (i.e., instances), and 147 activities – recorded from 2008 to 2011. Each trace holds the interactions of a student with an e-learning platform (e.g., exercise uploads). The interactions are recorded individually for each academic year $\mapsto$ HEP_1 to HEP_3. All logs

[1] `http://www.win.tue.nl/bpi/2015/challenge`—DOI: 10.4121/uuid:31a308ef-c844-48da-948c-305d167a0ec1

[2] `http://www.win.tue.nl/bpi/doku.php?id=2017`—DOI: 10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b

(i.e., BPIC and HEP) contain sufficient details to apply the proposed approach (e.g., instance execution events and relevant timestamps).

The logs were evenly and randomly separated into training (for signature generation) and test data (for the anomaly detection performance evaluation). A tenth of the test data was mutated by one (out of four) randomly chosen mutators (we regard this amount as being sufficient, cf. [4]). This enables to generate labeled non-anomalous (i.e., non-mutated) and anomalous (i.e., mutated) test data entries, i.e., to determine if both behavior "types" are correctly differentiated by the proposed approach. The applied four mutators generate multi instance anomalies that cannot be detected by existing single business process instance focused anomaly detection work, hence, a comparison with such existing work is not possible: *a*) *Parallel Executions* – a process execution is duplicated so that it occurs in parallel; and *b*) *Sequential Executions* – a process execution is duplicated so that occurs in a sequential order; and *c*) *Execution Order* – the process execution order is randomly changed; and *d*) *New or Missing Process* – new process executions are artificially added or recorded executions are removed.

The mutators were adapted and extended from our work in [4] – which was assessed by security experts as being realistic. It was chosen to combine multiple mutators to represent that real life anomalies are diverse and affect different aspects of process executions. In addition, the applied strategy also evaluates the proposed handling of unexpected execution behavior. This is, because the test data (in its mutated but also non-mutated form) contains behavior that is not represented in the training data (e.g., manual ad-hoc changes). The following results consist of the average of 100 evaluation runs – individually for each log file – to even out the randomness in the data separation and mutation.

Metrics and Evaluation The feasibility of the presented anomaly detection approach is analyzed. For this, the training data is utilized to construct signatures which are applied on the test data to differentiate between known randomly mutated (i.e., anomalous) the known non-mutated (i.e., non-anomalous) test data entries. This enables to collect four key performance indicators: *True Positive* (TP) and *True Negative* (TN) represent data entries that were correctly identified as anomalous (TP) or non-anomalous (TN). In comparison, *False Positive* (FP) and *False Negative* (FN) represent data entries which were incorrectly identified as anomalous (FP) or non-anomalous (FN). For example, FP counts non-anomalous test data entries which were incorrectly identified as being anomalous. Based on this performance indicators three standard metrics are calculated for each log file (i.e., BPIC5_1-5, BPIC7, and HEP_1-3):

a) *Precision* $P = TP/(TP + FP)$ – indicates if the identified anomalous test data entries were in fact anomalies; and *b*) *Recall* $R = TP/(TP + FN)$ – indicates if anomalies were "missed", i.e., not identified; and *c*) *Accuracy* $A = (TP + TN)/(TP + TN + FP + FN)$ – provides a general anomaly detection performance overview; $TP, TN, FP, FN \in \mathbb{N}_{>0}$; $P, R, A \in [0; 1]$.

For this paper we assume that the number of False Positives (FP) or Negatives (FN) should be low while the number of True Positives (TP) or Negatives (TN) should be high, i.e., the accuracy becomes close to one. In addition the

F_β-measure, Eq. 1, is applied because it provides a configurable harmonic mean between Precision (P) and Recall (R), cf. [8]. Hereby, β controls the balance between P and R. So, if $\beta = 1$ then a harmonic mean between P and R is calculated. In comparison a $\beta < 1$ results in a precision and a $\beta > 1$ in a recall-oriented result. $F_{0.5}$,F_1,$F_{1.5}$-measures were used to present the evaluation results.

$$F_\beta = \frac{(\beta^2 + 1) \cdot P \cdot R}{\beta^2 \cdot P + R} \tag{1}$$

Results The results were generated based on BPIC 2015/2017 and HEP process execution logs and a publicly available proof-of-concept implementation of the presented approach: `https://github.com/KristofGit/MultiInstanceAnomaly`. The implementation calculated a signature in minutes (i.e., about 2 minutes on average) and required only seconds (i.e., below 3 seconds on average) to identify a test data entry as anomalous or non-anomalous on a standard 2.6 Ghz Intel Q6300 CPU with 8 GB of RAM. Of course, the signatures can be reused, i.e., calculated once and subsequently applied in order to analyze multiple following process executions. Moreover, the presented approach can be concurrently applied to analyze multiple instances in parallel. This suggests an applicability even on larger process repositories and execution logs.

Primary tests were applied to identify appropriate configuration values for the presented approach. The *punishment factors* for unexpected behavior were set to $pNDC = 0.60$ (known behavior but unexpected order), $pDP = 0.40$ (known behavior but unexpected parallelism), $pOS = 0.30$ (unknown behavior). A lower punishment factor results in a stronger punishment. So for example, pDP is higher than pOS because the latter is only utilized if completely unknown execution behavior is observed. In comparisons the former is applied if "only" an unexpected parallel execution occurred. This is the case, for example, if three parallel executions of process A were observed but only two were expected. As a rule of thumb it can be assumed that a higher punishment improves on the TP/FN side while having a negative impact on the TN/FP performance indicators. A similar conclusion can be drawn for $k = 0.3$, i.e., the percentage of similar time sequences for reference likelihood generation purposes. When k is increased then the proposed approach becomes more relaxed because the reference likelihood typically decreases, i.e., anomalous instances are more likely "overlooked".

A window size ws of 4 (BPIC) and 20 (HEP) was utilized. Hereby, the different ws values compensate that the log sources (e.g., BPIC or HEP) store events with a different density (i.e., the BPIC logs cover more events at the same timespan than the HEP logs). The log dependent ws-value ensures that the generated signatures represent a roughly comparable amount of process execution events for all log sources. Finally, a noise prevention value of $c = 8$ was utilized, i.e., a slot – either original or aggregated – has to cover at least 8 activity executions to not be recognized as volatile or noise and being removed. The chosen values were successfully applied on different processes and domains. Hence, we assume that they can be applied as a valid starting point for future optimizations in scenarios and domains which were not covered by the presented evaluation.

The average evaluation results are shown in Tab. 1. The accuracy metric reached an average result of 78% but also the other metrics show promising results (83% for recall and 77% for precision). Hence, it was found that the proposed approach could successfully identify the constructed anomalies in the analyzed complex multi instance execution evaluation data. It was observed that the proposed approach could more easily identify anomalous behavior for the HEP log based evaluation than during the BPIC based evaluation. This most likely origins from the different complexity of the logs (i.e., the BPIC logs hold substantially more and more complex behavior than the HEP logs). So, it was concluded that the more complex and diverse the signature generation behavior becomes, the harder it is to distinguish correct from anomalous behavior. Nevertheless, even for the challenging BPIC log based evaluation the performance of the presented work achieved an average of 70% anomaly detection accuracy.

	BPIC5_1	BPIC5_2	BPIC5_3	BPIC5_4	BPIC5_5	HEP_1	HEP_2	HEP_3	BPIC7
Accuracy	0.70	0.71	0.73	0.71	0.71	0.94	0.92	0.92	0.66
Precision	0.69	0.68	0.70	0.67	0.67	0.96	0.97	0.96	0.63
Recall	0.78	0.83	0.84	0.84	0.82	0.92	0.88	0.89	0.70
$F_{0.5}$-measure	0.71	0.70	0.72	0.70	0.72	0.95	0.95	0.94	0.64
F_1-measure	0.73	0.75	0.76	0.75	0.75	0.94	0.93	0.92	0.66
$F_{1.5}$-measure	0.75	0.78	0.79	0.78	0.77	0.93	0.91	0.91	0.68

Table 1. Anomaly detection performance of the presented approach

6 Discussion and Outlook

This work applies the common assumption that an anomaly is some kind of unlikely behavior that never or hardly occurs during a business process execution, cf. [4, 5]. Accordingly the proposed approach compares given multi instance executions with given recorded historic executions in L to calculate their likelihood in relation to the behavior in L, such that, unlikely process instance executions are identified as anomalous. Hence, this work applies an unsupervised approach as it neither assumes the historic behavior as anomalous or not. It is hard to propose a rule of thumb for predicting the required training data size (i.e., the amount of historic behavior in L). This is because the size of the required training data heavily depends on the amount of execution variety that can be observed for the analyzed instances – which is unique for each organization.

The evaluation showed an average anomaly detection accuracy of 78%, which suggests an applicability in additional scenarios. In addition a detailed analysis of the evaluation results revealed that the proposed behavior likelihood assessment based anomaly detection approach substantially improved the detection results for the analyzed complex real life execution behavior. This is, because the utilized real life evaluation data showed a substantial amount of behavior drift in the analyzed multi instance behavior data, caused, e.g., by fluctuating instance durations or varying parallel instance execution behavior. Hence, not taking these dynamics, by design, into account would have resulted in substantially worse evaluation results, e.g., by causing a high number of false positives.

Note, a large amount of false positives could harm an organization's performance, e.g., through process executions which are unnecessarily halted or terminated.

The presented approach determines process executions as anomalous based on their relations to other preceding, succeeding, or simultaneous instances. In comparison to existing work this is a rather big picture focused approach which, by purpose, ignores more fine granular details (e.g., which resource has executed an activity or what data was exchanged between two activities). Hence, in future work we will strive to combine both worlds. Hereby, multiple views on the instance behavior can be taken into consideration to identify diverse and complex anomalous behavior. We assume this as necessary to identify inside threats that actively hide their malicious intentions. Moreover, we will assess the applicability of the proposed approach to analyze complex dynamic parallel activity executions. Finally, we will strive to integrate correlation features to respect contextual aspects, e.g., by adding support for filters to analyze only process instances that meet specific conditions (e.g., based on the involved resources).

References

1. Allen, J.F.: Maintaining knowledge about temporal intervals. ACM 26(11), 832–843 (1983)
2. Atallah, M., Szpankowski, W., Gwadera, R.: Detection of significant sets of episodes in event sequences. In: Data Mining. pp. 3–10. IEEE (2004)
3. Bezerra, F., Wainer, J., van der Aalst, W.M.: Anomaly detection using process mining. In: Enterprise, Business-Process and Information Systems Modeling, pp. 149–161. Springer (2009)
4. Böhmer, K., Rinderle-Ma, S.: Automatic signature generation for anomaly detection in business process instance data. In: Business Process Modeling, Development and Support. pp. 196–211. Springer (2016)
5. Böhmer, K., Rinderle-Ma, S.: Multi-perspective anomaly detection in business process execution events. In: COOPIS. pp. 80–98. Springer (2016)
6. Böhmer, K., Rinderle-Ma, S.: Anomaly detection in business process runtime behavior – challenges and limitations. arXiv (2017)
7. Chaoji, V., Rastogi, R., Roy, G.: Machine learning in the real world. VLDB Endowment 9(13), 1597–1600 (2016)
8. Chinchor, N., Sundheim, B.: Muc-5 evaluation metrics. In: Message understanding. pp. 69–78. Computational Linguistics (1993)
9. Fdhila, W., Rinderle-Ma, S., Knuplesch, D., Reichert, M.: Change and compliance in collaborative processes. In: Services Computing. pp. 162–169. IEEE (2015)
10. Gupta, M., Gao, J., Aggarwal, C.C., Han, J.: Outlier detection for temporal data: A survey. Knowledge and Data Engineering 26(9), 2250–2267 (2014)
11. de Leoni, M., van der Aalst, W.M., Dees, M.: A general process mining framework for correlating, predicting and clustering dynamic behavior based on event logs. Information Systems 56, 235–257 (2016)
12. Rogge-Solti, A., Kasneci, G.: Temporal anomaly detection in business processes. In: Business Process Management. pp. 234–249. Springer (2014)
13. Vogelgesang, T., et al.: Multidimensional process mining: Questions, requirements, and limitations. In: CAISE Forum. pp. 169–176. Springer (2016)
14. Wieringa, R.J.: Design science methodology for information systems and software engineering. Springer (2014)