Semi-Supervised Log Pattern Detection and Exploration Using Event Concurrence and Contextual Information (Extended Version)

Xixi Lu[1], Dirk Fahland[1], Robert Andrews[2], Suriadi Suriadi[2], Moe T. Wynn[2], Arthur H.M. ter Hofstede[2], Wil M.P. van der Aalst[1]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
[2] Queensland University of Technology, Brisbane, Australia
(March, 2017)

Abstract. Process mining offers a larger variety of techniques for analyzing process execution event logs. Although end-to-end process models can be discovered automatically, often surprisingly many insights can only be obtained and verified by carefully investigating the actual event data. Uncovering such insights is an explorative and iterative process as witnessed in many case studies. Automated pattern detection on event logs can support this. Here, unsupervised techniques generate a multitude of patterns based on statistical properties of the log only (lacking domain context), whereas supervised pattern detection requires domain experts to specify patterns to detect by hand (lacking the event log context). In this paper, we reconcile supervised and unsupervised pattern detection. We let users both, build basic patterns by hand and automatically obtain patterns through unsupervised learning. We visualize pattern matches in the context of the event log (also showing concurrency and other contextual information) and let the user modify earlier patterns based on the domain insights. This enables an iterative approach for identifying more complex patterns than those obtainable by existing techniques. We implemented our approach in the ProM framework and evaluated the tool using both the BPI Challenge 2012 log of a loan application process and an insurance claims log from a major Australian insurance company.

1 Introduction

Real-life business processes and correspondingly recorded event logs tend to be complex and unstructured. To derive useful insights for improving business processes, process mining offers a large variety of techniques for analyzing these event logs. Although discovering end-to-end process models from such complex event logs is useful, experience shows that many important insights can only be obtained by carefully investigating the actual event data, see for example the reports for the BPI challenges [1].

Pattern detection has been proven to be a powerful mechanism for dealing with complex event logs in process mining and helping to gain valuable insight into common behavior in process executions [2–8]. More specifically, pattern detection has been applied in various contexts: (i) to specify patterns as constraints and detect non-compliant cases [6]; (ii) to automatically detect frequent common behavior [8]; (iii) to find low level behavior patterns for log simplification [3, 4]; (iv) to model re-occurring quality

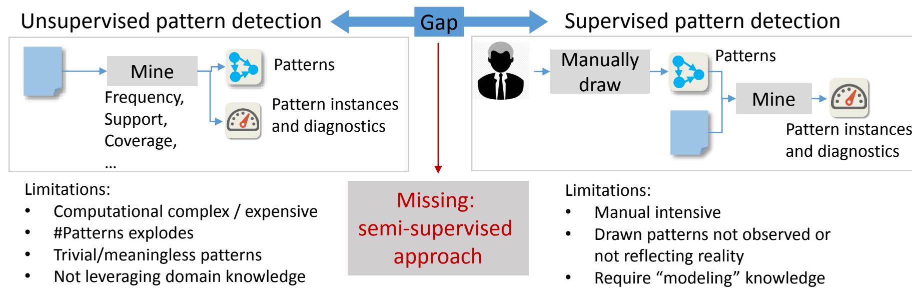

Fig. 1: Problem analysis: there is a gap between unsupervised and supervised pattern detection on event logs.

issues in event logs [9]. However, existing approaches to pattern detection are either exclusively *unsupervised learning* or exclusively *supervised learning*, with each having their own limitations, as depicted by Fig. 1.

Unsupervised learning [3, 7] generates patterns based on statistical properties of the log, such as frequency, support and confidence, and suffers from a problem known as "pattern explosion", resulting in many uninteresting or trivial patterns. Patterns that are less frequent are much more difficult to detect. In contrast, most *supervised approaches* [5,6] assume the patterns to be given in a formal language such as LTL, (Colored) Petri nets or Declare models, e.g., manually drawn by the experts. This modeling task is often laborious and may miss unexpected but relevant patterns from the log.

In this paper, we propose a semi-supervised approach to detect *log patterns*. An overview of the approach is shown in Fig. 2. We first *convert* traces in the log into partial orders of events, deriving concurrent events and contextual information using e.g. time information. We then *visualize* these partial orders allowing the user to explore traces, select events, and extract patterns of interest. Extracted patterns are expressed as simple graphs that specify direct and indirect succession or unknown (partial) ordering of activities. We show how such patterns can be extracted automatically by converting the output of existing unsupervised pattern detection algorithms, or by our proposed *semi-supervised detectors*. By highlighting in the log all occurrences of patterns chosen by a user, we enable the user to explore the patterns in their context (where and how frequently they occur). We provide operations to let a user modify a pattern based on an occurrence in a particular context (by extending them with additional activities, different ordering relations, etc.) or create new patterns based on existing ones. This facility to explore and modify patterns iteratively helps the user to balance between unsupervised and supervised learning.

The contributions are the following. We formally define a pattern as a partial order of activities in terms of concurrence (or independence), directly-cause or eventually-cause relations. We distinguish one event as a *core-event*, which is the focal point of a pattern. We discuss a matching algorithm using the notion of core-event to retrieve pattern instances and calculate the support and confidence of a pattern based on the number of core-events that match the pattern. We propose semi-supervised pattern detectors using the core-event of interest, and we provide operations to extract and modify patterns based on matches in an event log. Based on the literature study (see Sect. 2),

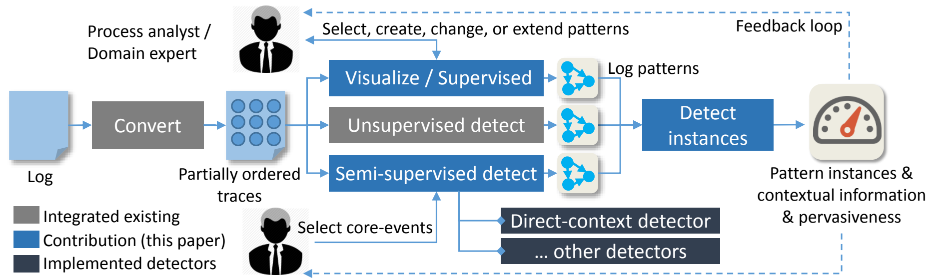

Fig. 2: An overview of the proposed approach.

8 out of 11 investigated pattern detection approaches do not allow the user to explore instances of patterns. We implemented the approach as a plug-in for the ProM process mining framework[3]. The tool supports both supervised and semi-supervised detection. Moreover, we integrate two unsupervised approaches to show that our approach does not substitute but *complements* existing pattern detection approaches. We evaluated the approach in two case studies conducted using real-life event data sets.

The remainder of this paper is organized as follows. Sect. 2 discusses related work. Sect. 3 discusses the preliminaries and the input for our approach. In Sect. 4, we formally define the patterns and pattern instances. In Sect. 5, we explain our approach to detect patterns and find pattern instances. Sect. 6 reports results for the two case studies conducted. Sect. 7 concludes the paper and discusses future work.

2 Related work

We use Table 1 to discuss and structure related work. Here we discuss related work on log pattern detection on a conceptual level; an evaluation of the techniques for which an implementation is available is available in Sect. 6.2.

Log Pattern Definition. Patterns in the process mining literature have been defined differently. Early work focuses on clustering frequently co-occurring activities; such a cluster of activities is considered as a (low-level) pattern and mapped into a high level activity [4, 10]. Later, more specialized definitions are used. Bose et al. [3] defined patterns as repeated sequences. [7] and [8] defined patterns as partial orders of activities in which the edges represent only eventual-cause relations. The work in [5, 11] considered patterns as Petri nets. In our case, we use partial orders, distinguishing concurrence, directly-cause and eventually-cause.

Unsupervised Log Pattern Detection. Unsupervised log pattern detection approaches take an event log as input and generate patterns based on predefined measures [3, 7, 8]. Some limitations are known. Firstly, such unsupervised approaches are computationally complex and expensive, generating a massive amount of possible patterns based on their frequencies or other measures. If one sets the values for the measures too high, then only very frequent, trivial patterns are returned, thereby missing many interesting

[3] http://www.promtools.org/

Table 1: Comparison of related pattern detection approaches.

	S/U/M/V*	Exact instances?	provide unsuper. support	Change/ define patterns?	Explore pattern instances?
Bose et al., Pattern abstraction [3]	U	Exact	+ (beh.)†	-	-
Günther et al., Fuzzy mining [4]	U	Exact	+ (act.)	-	-
Mannhardt et al., Log abstraction [5]	S	Approx.	- (act.)	+	+/-
Maggi et al., LTL checker [6]	S	Exact	- (com.)	+/-	+/-
Leemans et al., Episodes miner [7]	U	Exact	+ (beh.)	-	-
Diamantini et al., Pattern discovery [8]	U	Exact	+ (beh.)	-	-
Baier et al., Activity maching [12]	M	Exact	- (act.)	-	-
Ferreira et al., Label abstraction [10]	U	Exact	+ (act.)	-	-
Tax et al., Local models [11]	U	Approx.	+ (beh.)	-	-
Song et al., Dotted chart [13]	V	NA	- (beh.)	-	-
Shneiderman et al., EventFlow [2]	S.V.	Exact	- (beh.)	+	+
Lu et al., Pattern explorer	M.S.V.U.	Exact	+ (beh.)	+	+

* S. for supervised; U. for unsupervised; M. for Semi-Supervised; V. for visualization.
† In parentheses, the aim of the technique is abbreviated: beh. for behavior analysis; act. for low level activity abstraction; com. for compliance checking.

results. By setting the values too low, too many patterns are returned (so called 'pattern explosion') [3, 7, 8]. Secondly, as a result of not leveraging domain knowledge, many of the patterns generated by unsupervised learning are not of interest or are meaningless. Finally, most unsupervised approaches do not return pattern instances or additional contextual or diagnostic information of the detected patterns, thus obstructing the user from analyzing the patterns [7].

Supervised Log Pattern Detection. Supervised pattern detection approaches in process mining take patterns and logs as input and detect pattern instances as results [5]. Such supervised approaches require the user to model patterns in a formal language (e.g. as (Colored) Petri nets, or LTL constraints), which relies on the expertise of the user. This may potentially require formalizing a large set of pattern descriptions. Moreover, the user may miss potentially important patterns through incomplete specification or model idealized patterns that are not observed in reality.

Log Explorer and Visualization. Advanced log visualization analytics have also been proposed as a way to help the user observe patterns. The dotted chart [13] is a simple way of visualizing event logs and helping the user spot and interpret patterns such as batch processing. However, no pattern extraction approaches are supported, nor is it possible to query for all instances of the observed patterns. EventFlow [2] has been proposed as a more advanced tool for visualizing event sequences. It allows for advanced querying. However, EventFlow also requires the user to create patterns (queries) and does not support generating patterns in an unsupervised or semi-unsupervised way. In our case, we allow semi-supervised pattern detection; the detected patterns are suggested to the user and can be used as queries. Moreover, we support partially ordered events and help the user detect and explore causal dependencies between events [14].

3 Preliminaries

In this section, we recall a few basic concepts such as partial orders, event logs, and partially ordered traces. These are used later in the paper.

3.1 Basic Notations: Partial Order, DAG, Projection, Event logs

Let X be a set of elements. $X' \subseteq X$ is a subset of elements of X. $R \subseteq X \times X$ denotes a set of relations between X. R is a *partial order* over X if and only if R is irreflexive, anti-symmetric, and transitive. $R : X \to Y$ is a function that maps an element in X to an element in Y, we use $Dom(R)$ to denote the domain of R and $Rng(R)$ the range of R, i.e., $Dom(R) \subseteq X$ and $Rng(R) \subseteq Y$. Let $G = (N, <)$ be a *directed acyclic graph (DAG)* with $< \subseteq N \times N$. We use $<^-$ to denote transitive reduction of $<$ and $<^+$ to denote the transitive closure of $<$. The relation $<^+$ will be used to denote reachability and is also known as a partial order. In this paper, for all nodes $n, n' \in N$, $n \not<^+ n'$ and $n' \not<^+ n$, we say n and n' are *concurrent* and use $n \,||_<\, n'$ to denote this. We use $\downarrow$ to denote a *projection function*, i.e., $X \downarrow_{X'} = X \cap X'$, and $R \downarrow_{X'} = R \cap (X' \times X')$. Let $G = (N, <)$ be a DAG and $N' \subseteq N$. We overload the projection function and define the projection for a graph, $G \downarrow_{N'} = (N \downarrow_{N'}, <\downarrow_{N'})$, also known as *induced subgraph*.

An *event log* represents the observed behavior of a process. Each case going through the process results in a trace of events in the event log.

Definition 1 (Event, trace, event log). *Let $\mathcal{E}$ be the universe of unique* events*, i.e., the set of all possible event identifiers. A* trace $\sigma \in \mathcal{E}^*$ *is a finite sequence of events. An* event log $L = \{\sigma_1, \sigma_2, \cdots, \sigma_n\} \subseteq \mathcal{E}^*$ *is a set of traces.*

Here we assume no event appears twice in the same trace nor in the same log. We use E_σ for the set of events in trace σ and E_L for the set of events in log L. Let U be a set of attribute names. For each event $e \in \mathcal{E}$ and name $d \in U$, $\pi_d(e)$ returns the value of attribute d for event e, or $\pi_d(e) = \perp$ otherwise. For example, $\pi_{case}(e)$ denotes the trace of e; $\pi_{act}(e)$ is the activity associated with e; $\pi_{time}(e)$ denotes the timestamp of e; $\pi_{resource}(e)$ denotes the resource that executed e. Fig. 3(a) shows an example of a trace. The trace $\sigma = \langle e_1, e_2, ..., e_5 \rangle$ contains five events, totally ordered as is. Event e_1 has activity $\pi_{act}(e_1) = Injury$ and is executed on $\pi_{time}(e_1) =$ 08/09/2016-00:30:00.

3.2 Partially Ordered Traces

We explain the use of partial orders for analyzing events, define partially ordered traces, and discuss how to obtain them from an event log (Def. 1). Many recent papers consider partial orders of events [14–16], instead of totally ordered event sequences. One reason for this consideration is that a particular total order of events may be unreliable or unknown. For example, if events a and b are recorded only on day granularity (not seconds), then the totally ordered log may contain the sequence $\langle a, b \rangle$ whereas in reality $\langle b, a \rangle$ occurred. On the other hand, ordering events totally based on time may be misleading, i.e. two events may be causally unrelated but just happen to occur in a particular order. Representing events as a partial order (where a and b can occur "unordered" or "concurrent") alleviates this problem and allows us to represent more accurate contextual information of events [14, 16].

Definition 2 (Partially Ordered Trace). *A* partially ordered trace $\varphi = (E, \prec)$ *is a Directed Acyclic Graph (DAG), in which* $\prec$ *denotes the inferred* "cause" relations[4] *over events E. If* $e \prec e'$*, we say* e caused e'*. We use* $\prec^-$ *to denote* directly-cause*,* $\prec^+$ *to denote* eventually-cause*, and* $||_\prec$ *to denote the* concurrent *relation.*

Note that the *eventually-cause* relations $\prec^+$ is equivalent to the reachability relation of the DAG $(E, \prec)$ and forms therefore a partial order. Fig. 3(b) shows a partially ordered trace $(E, \prec)$, in which $E = \{e_1, e_2, \cdots, e_5\}$ and $\prec = \{e_1 \prec e_2, e_1 \prec e_3, e_2 \prec e_4, e_3 \prec e_4, e_4 \prec e_5\}$. In this particular case, $\prec^- = \prec$ and $\prec^+ = \prec \cup \{e_1 \prec e_4, e_1 \prec e_5, e_2 \prec e_5, e_3 \prec e_5\}$. $||_\prec = \{e_2 \, ||_\prec \, e_3\}$. Note that $\prec$, $\prec^-$, and $\prec^+$ are irreflexive and acyclic.

Partial orders of events may be obtained from totally ordered traces. A few works [15, 17, 18] assume to have an oracle that indicates the set of activities that are concurrent or unordered and use this oracle to convert totally ordered events into partial orders. Such an oracle could be obtained by interviewing domain experts or be computed from event logs [14].

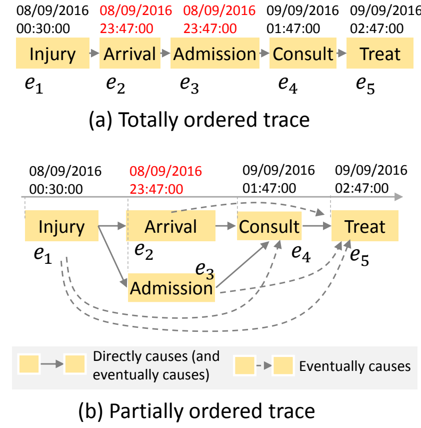

Fig. 3: A sequential trace and its converted partially ordered trace.

In this paper, we overload the symbol φ to denote an conversion oracle function that, for a trace σ, returns the partially ordered trace $\varphi(\sigma) = (E_\sigma, \prec_\sigma)$. We deploy the oracle that considers the events occurring within a short time to be concurrent [14] as our default oracle. Let $\sigma = \langle e_1, \cdots, e_n \rangle$ be a trace. $\varphi_{time}(\sigma, dt) = (E_\sigma, \prec_\sigma)$ in which $\prec_\sigma = \{e_i \prec_\sigma e_j \mid 1 \leq i < j \leq n \wedge \exists_{i \leq k < j} \pi_{time}(e_{k+1}) - \pi_{time}(e_k) > dt\}$. In addition, we also overload the φ function to handle an event log $L = \{\sigma_1, \cdots, \sigma_n\}$ and return a set of partially ordered traces, one for each trace in L, i.e., $\varphi(L) = \{\varphi(\sigma_1), \cdots, \varphi(\sigma_n)\}$. An example of such a conversion based on the timestamps is shown in Fig. 3, with $dt = 0$ sec.

In this paper, we use π_{act} as our default labeling function for events and assume it is universally available. Note that it is possible to consider other functions, for example $\pi_{resource}$, to explore resource related patterns such as hand-over of work.

4 Patterns and Pattern Instances

Having defined partially ordered traces of an event log, we now present the concepts of log patterns. In Sect. 4.1 we first motivate and then define the log patterns and pattern instances. Next, in Sect. 4.2, we discuss how pervasiveness measures for the patterns, such as support, confidence and coverage, are computed. In Sect. 5, we discuss our three approaches to pattern detection.

[4] We use the term *"cause" (causality)* only to distinguish the relations in a partially ordered trace from the *follow* relations (i.e., *directly-follow* and *eventually-follow*) in totally ordered traces.

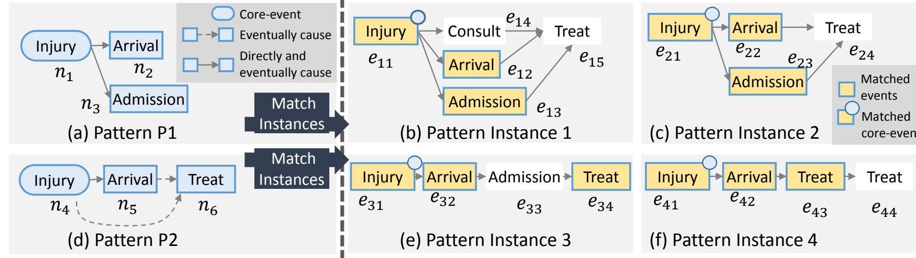

Fig. 4: Two patterns and four highlighted pattern instances, two for each pattern.

4.1 Core-Event, Pattern and Pattern Instance

To support process analysts in expressing and modifying log patterns with ease, the patterns should be simple. Moreover, if such a pattern resembles the event structure of our traces, it should be easier for the user to observe and recognize these patterns. We therefore define a pattern too as a labeled DAG, which allows for expressing common causal dependencies that occur in a partially ordered trace, namely *directly-cause* and *eventually-cause*. The *concurrence* relation is then deduced for any two events that are not eventually causing one another. Fig. 4(a) and (d) show two examples of patterns.

In addition to considering patterns as labeled DAGs, we, for two reasons, explicitly include a notion called a *core-event* in the patterns. Firstly, the *core-event* notion allows for unambiguous notions of pattern matches and unambiguous computation of measures such as frequency. Without a core-event, having a pattern P that says "a eventually-caused b" and a trace $\sigma_1 = \langle a, b, b, a\rangle$, should one count this as one instance or two? Given another trace $\sigma_2 = \langle a, b, a, b, c\rangle$, if one counts the number of combinations, such an approach results in three pattern instances in σ_2 while there is only two a's and two b's, causing confusion. In our case, the core-event anchors the pattern in a particular perspective, we simply count the number of distinct events that match the core-event and satisfy the pattern; we ignore whether the pattern for the same core-event occurs several times. Take the same example, if a is the core-event of P, then we have one a for σ_1 and two a's for σ_2. Similarly, if the pattern P has b as the core-event, then we have two b's for σ_1 and also two b's for σ_2. Moreover, having a core-event also allows us to unambiguous find anti-instances (events) that do not satisfy a pattern on the event level and not just on the case level. For example, having pattern P with a as the core-event, the second a in σ_1 do not satisfy the pattern. Without the core-event, σ_1 may be consider as compliant or not, depending on the interpretation of the pattern.

Definition 3 (Log Pattern). *A log pattern $P = (N, \mapsto, \rightsquigarrow, \alpha, c)$ is a directed acyclic graph, of which:*
- *N is a set of nodes,*
- *$\mapsto$ is a set of edges among N and denotes the directly-cause relation[4],*
- *$\rightsquigarrow$ is a set of edges among N and denotes the eventually-cause relation[4],*
- *$\alpha : N \to A$ is a partial function that assigns a label $\alpha(n)$ to any node $n \in N$,*
- *$c \in N$ is the core-event of the pattern*

and satisfies the following constraints:

1. *$\mapsto \subseteq \rightsquigarrow$, i.e., the directly-cause relation is a subset of the eventually-cause relation;*

2. $(N, \rightsquigarrow)$ *is a partial order, from which the concurrence* $||_{\rightsquigarrow}$ *can be deduced;*
3. *for all* $n, n' \in N$*, if there is* $n'' \in N$ *such that* $n \rightsquigarrow n'' \rightsquigarrow n'$*, then there is no* $n \mapsto n'$*.*
   We also say c has context P and call $N \backslash \{c\}$ *the* context-nodes *of c.*

By regarding a pattern as a core-event (activity) that occurred in a particular context, a pattern instance is an occurrence of the core-event in the log in the same context.

Definition 4 (Pattern Instance). *Let* $P = (N, \mapsto, \rightsquigarrow, \alpha, c)$ *be a log pattern,* $(E_\varphi, \prec)$ *a partially ordered trace,* $E' \subseteq E_\varphi$ *a subset of events, and* $e_c \in E'$ *an event.* (e_c, E') *is an* instance *of pattern P if and only if there is a bijective function* $I : E' \to N$ *such that*
- e_c *is mapped to the core-event, i.e.,* $I(e_c) = c$*.*
- *for each event* $e \in E'$*, event e and the corresponding node* $I(e)$ *have the same label, i.e.,* $\pi_{act}(e) = \alpha(I(e))$*;*
- *for all events* $e, e' \in E'$*, the relations between e and e' satisfy the relations between* $I(e)$ *and* $I(e')$*, i.e.,* (1) $I(e) \mapsto I(e') \Rightarrow e \prec^{-} e'$*,* (2) $I(e) \rightsquigarrow I(e') \Rightarrow e \prec^{+} e'$*, and* (3) $I(e') \, ||_{\rightsquigarrow} \, I(e) \Rightarrow e' \, ||_{\prec} \, e$

If (e_c, E') *is an instance of pattern P, we also say* e_c satisfies *P.*

The behavior specified by a pattern are preserved in the instances of the pattern. Fig. 4(b), (c), (e) and (f) exemplify four pattern instances highlighted in their partially ordered traces: highlighted e_{11} and e_{21} satisfy P1; e_{31} and e_{41} satisfy P2. It is important to note that changing the core-event of a pattern does not change the behavioral relations of the pattern, but does change the instances that match the pattern.

Definition 5 (A Maximal Set of Pattern Instances). *Let* $P = (N, \mapsto, \rightsquigarrow, \alpha, c)$ *be a pattern, L an event log,* φ *the conversion oracle. A maximal set of pattern instances* $PI(P, \varphi(\sigma))$ *of trace* $\sigma \in L$ *is defined as a largest set of all instances of P in* $\varphi(\sigma)$ *that differ in their core-event, i.e., for any instance* (e', E') *of P, if* $(e', E') \notin PI(P, \varphi(\sigma))$ *then there exist* $(e', E'') \in PI(P, \varphi(\sigma))$*. We write* $PI(P, L, \varphi) = \bigcup_{\sigma \in L} PI(P, \varphi(\sigma))$ *for the union of a* maximal set of pattern instances *of all traces in log L. We write* $PIC(P, L, \varphi)$ *to denote* the set of all core-events *that satisfy P, i.e.,* $PIC(P, L, \varphi) = \{e \mid (e, E') \in PI(P, L, \varphi)\}$*.*

We also define the set of *anti-pattern instances* $AntiPIC(P, L, \varphi)$, which is the set of core-events that do not satisfy P, i.e., $AntiPIC(P, L, \varphi) = \{e \in E_L \mid \pi_{act}(e) = \alpha(c)\} \backslash PIC(P, L, \varphi)$. Note that independent of which maximal set of pattern instances is returned, the set of all core-events and the set of anti-pattern instances of a pattern remain the same. Consider for example pattern P2 in Fig. 4(d) and the four partially ordered traces shown on the right-hand side to be the set of all partially ordered traces in $\varphi(L)$. A maximal set of pattern instances of P2 in $\varphi(L)$ always contains four pattern instances with e_{11}, e_{21}, e_{31} and e_{41} as the core-events that satisfy P2. As e_{41} (*Injury*) in Fig. 4(f) already satisfies P2 with context-event e_{43} (*Treat*), e_{44} (*Treat*) is not considered as a context event. However, if n_6 (*Treat*) was considered as core-event of P2, we would obtain five instances with the core-events $e_{15}, e_{24}, e_{34}, e_{43}$, and e_{44}. Furthermore, e_{31} and e_{41} are anti-pattern instances of pattern P1.

4.2 Pattern Support, Confidence and Coverage

To help the user assess the pervasiveness of a pattern, we define the following five measures of a pattern based on the set of all pattern instances. Let $P = (N, \mapsto, \rightsquigarrow, \alpha, c)$ be a pattern. Given a log L and the partially ordered traces $\varphi(L)$, we have the set of all core-events $PIC(P, L, \varphi) = \{e_1, e_2, \cdots, e_n\}$ that satisfy P.

- *Pattern support* indicates how many distinct events satisfy P. i.e., $P\text{-}supp(P, L, \varphi) = \mid PIC(P, L, \varphi) \mid$.
- *Pattern confidence* is the number of events that satisfy P divided by the total number of events that have the same label as the core-event; this measure indicates how often is the occurrence of the core-event a predictor for the occurrence of the entire pattern. i.e., $P\text{-}conf(P, L, \varphi) = \frac{\text{P-supp}(P,L,\varphi)}{|\{e \in E_L | \pi_{act}(e) = \alpha(c)\}|}$.
- *Case support* is the number of traces that have at least one pattern instance satisfying P, i.e., $C\text{-}supp(P, L, \varphi) = \mid \{\sigma \in L \mid \exists\, e \in PIC(P, L, \varphi), e \in E_\sigma\} \mid$.
- *Case confidence* is the case support of P divided by the number of cases that have an event with the same label as c, i.e., $C\text{-}conf(P, L, \varphi) = \frac{C\text{-}supp(P,L,\varphi)}{|\{\sigma \in L | \exists\, e \in \sigma, \pi_{act}(e) = \alpha(c)\}|}$.
- *Case coverage* is the case support of P divided by the number of cases in the log, i.e., $C\text{-}cover(P, L, \varphi) = \frac{C\text{-}supp(P,L,\varphi)}{|L|}$.

We note that the desirability of pervasiveness measures (highly pervasiveness or otherwise) is context/application dependent. For example, for patterns representing non-complaint behavior or data quality issues, we would prefer to see low pervasiveness.

5 Pattern Detection and Pattern Instance Matching

We now introduce operations to *extract* and extend patterns in the context of a log in an explorative approach. To help the reader envision this explorative approach, we first briefly introduce the Log Pattern Explorer tool, for which a screenshot is shown in Fig. 5. The right-hand side panel shows the log patterns, manually or automatically extracted. The user may create or modify a pattern. On the left-hand side, each log trace is visualized as a partial order in its own panel: each event is visualized as a square tile; tiles can be colored based on event attributes; concurrent events are stacked on top of each other; the labels and arcs are omitted for the sake of simplicity. When a user selects one or more patterns in the right-hand panel, all pattern instances are highlighted on the left (by a color-coded frame around the tiles of the satisfying events).

In Sect. 5.1, we discuss various ways to extract, detect, and modify patterns using supervised, semi-supervised or unsupervised approaches. We then discuss an approach to compute a maximal set of pattern instances of a pattern (Def. 5) in Sect. 5.2.

5.1 Pattern Detection Approaches - Partially Ordered Traces To Patterns

Definition 3 and the tool allow the user to create any pattern of interest. Nevertheless, as shown in Fig. 2, we would like to support the user in extracting these patterns from a log with ease. For example, an expert glances through the partially ordered traces visualized in Fig. 5 and may observe some events (patterns) reoccur in many traces, e.g., through the color coding. The user can then extract a pattern by marking all events (tiles) in

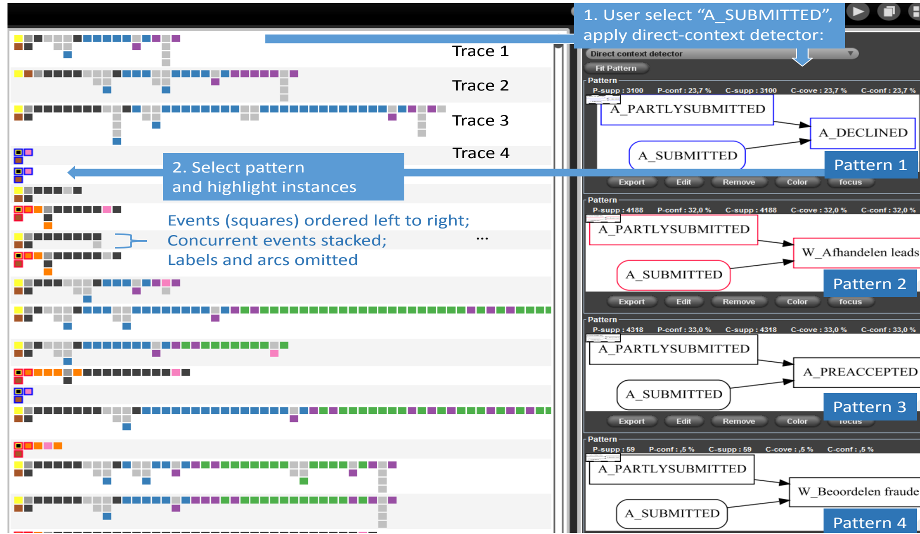

Fig. 5: The patterns found using event *A_SUBMITTED*.

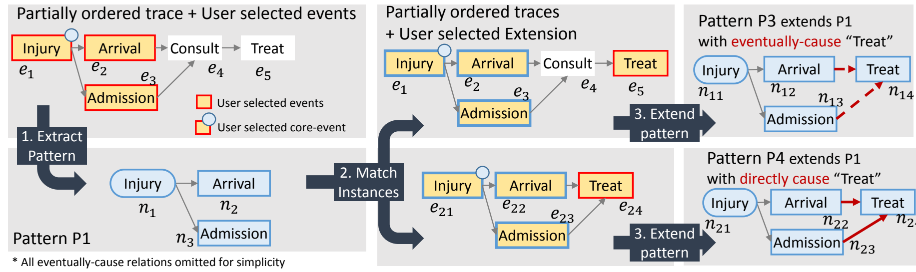

Fig. 6: Iteratively extracting and extending patterns from partially ordered traces.

a trace that make up the pattern. Next, we define an operation for also extracting the relations between the events from the trace to extract a complete pattern definition.

Supervised Pattern Extraction from Partially Ordered Traces. In essence, each partially ordered trace or any of its subgraphs could be extracted as a pattern. Let $\varphi = (E, \prec)$ be a partially ordered trace. We use $\rho(\varphi,\ e_c) = P$ to denote a conversion from a partially ordered trace into a pattern $P = (N, \mapsto, \rightsquigarrow, \alpha, c)$ with $N = E$, $\mapsto = \prec^-$, $\rightsquigarrow = \prec^+$, $\alpha = \pi_{act}$, and $c = e_c \in E$ is a chosen core-event. Similarly, one may convert any subgraph of a partially ordered trace into a pattern. Let $E' \subseteq E$ be a set of interested events selected from φ. We simply project the pattern onto E' (as defined in Sect. 3.1). The behavioral pattern induced by E' and core-event $e_c \in E'$ is defined as function $extractPattern(\varphi, e_c, E') = (\rho(\varphi, e_c))\downarrow_{E'} = (E', \prec^-\downarrow_{E'}, \prec^+\downarrow_{E'}, \pi_{act}\downarrow_{E'}, e_c) = P_{E'}$. Fig. 6 (Step 1) shows extraction of a pattern from the partially ordered trace shown in Fig. 3 by selecting events $e_1(Injury)$, $e_2(Arrival)$ and $e_3(Admission)$ and considering e_1 as the core-event. The extracted pattern $P1 = (N, \mapsto, \rightsquigarrow, \alpha, c)$ with $N = \{n_1, n_2, n_3\}$, $n_1 \mapsto n_2, n_1 \mapsto n_3$, $n_1 \rightsquigarrow n_2$, and $n_1 \rightsquigarrow n_3$; n_2 and n_3 are concurrent; for α, the labels

of events remain unchanged. In the tool shown in Fig. 5, the user can select a set of tiles (events) on the left and apply the pattern extraction function, the first tile is the core-event; the extracted pattern will appear in the panel on the right.

Creating, Extending and Changing Patterns. Seeing all instances of a pattern in their larger context, the user may change or extend the current pattern. For example, the user may want to extend a pattern with another node, or change a directly-cause into an eventually-cause. Here, we only list the following five groups of operations to change or create a pattern: (1) Change the core-event; (2) Change the labels; (3) Add (remove) a node to (from) a pattern; (4) Change directly-cause into eventually-cause, and vice versa; (5) Add (remove) a relation to (from) a pattern. Step 3 (extend pattern) in Fig. 6 exemplifies two different extensions of P1 that differ in how the added node *Treat* is relates to its predecessors.

Semi-Supervised Pattern Detection. To help the user discover interesting patterns, we propose semi-supervised pattern detectors that identify patterns for a user-chosen core-event: (1) *concurrence detector*, (2) *direct-predecessor (successor) detector*, and (3) *direct-context detector*. The concurrence detector runs as follows. Let $\mathcal{P}$ be the set of patterns we have detected so far. Let L be a log, φ a conversion oracle, and c a core-event of interest. Let $E_c = \{e \in E_L \mid \pi_{act}(e) = \alpha(c)\}$. For event $e \in E_c$, let σ be the trace containing e, and let C_e be the events that are concurrent with e in $\varphi(\sigma)$. If $(e, C_e \cup \{e\})$ is not an instance of any pattern $P \in \mathcal{P}$, then we obtain a new pattern $P' = extractPattern(\varphi(\sigma), e, C_e \cup \{e\})$ and add P' to $\mathcal{P}$. This allows to obtain a set of distinct patterns describing different sets of activities that occurred concurrently. For the *direct-predecessor (successor) detector*, C_e is the set of events that are directly-causing (that directly-caused by) core-event e. For the *direct-context detector*, C_e contains directly preceding, succeeding and all concurrent events of e. The relevance of an extracted pattern can be assessed using the measures of Sect. 4.2.

Integrating Unsupervised Pattern Detection. To also leverage on existing unsupervised pattern detection techniques, their output has to be converted to our pattern notion (Def. 3). We discuss this conversion for two techniques [3, 7]. For the technique in [3], the output patterns are sequences of activities, directly or eventually-follows. Any such pattern also satisfies our pattern definition (Def. 3). However, originally total-ordered events may now be independent (concurrent) due to the usage of partially ordered traces and may therefore no longer satisfy the converted pattern. In such cases, the user may find low *P-supp* and *P-conf*, explore anti-pattern instances and modify the pattern accordingly. The output patterns in [7] are partial orders of events in which the relations represent eventually-follow and no relations represent co-occur. We retain all eventually-follow relations and choose to consider co-occur as concurrent. Regarding choosing the core-events, the user may specify an activity (label) of interest to be automatically selected as the core-event; otherwise, a random node is selected. The user can run such an unsupervised detection in the tool shown in Fig. 5. The returned and converted patterns are shown in the right panel. The user can explore the pattern instances in the left panel. Note that the pervasiveness of a pattern (such as *P-supp* and *P-conf*) are recomputed in our case, depending on the chosen core-event.

5.2 Computing a Maximal Set of Pattern Instances

We propose the following approach to compute a maximal set of instances of a pattern, which can be divided into three phases. First, all events that can be matched to core-event c of pattern P are computed; we call these events the candidates of c and use E_c to denote this set of events. In the second phase, for each candidate $e \in E_c$, we try to find a pattern instance for P with e as core-event. This is done through incremental, exhaustive construction of the mapping I (Def. 4) with backtracking and pruning. If we can complete the construction of I mapping to events E' in the trace, then (e, E') is a pattern instance and added to the maximal set. Else, e is an anti-pattern instance. The algorithm(s) for computing the pattern instances are listed below.

Algorithm *MatchingPatternInstances*(P, L, φ)
Input: Pattern P, log L, and conversion function φ.
Output: The set of all pattern instances PI.
1. $PI \leftarrow \{\}$
2. **for** partially ordered trace $(E, \prec) \in \varphi(L)$
3. **do** $Candidates(c) \leftarrow \{e \mid e \in E \wedge \pi_{act}(e) = \alpha(c)\}$
4. **for** event $e_c \in Candidates(c)$
5. **do** $I \leftarrow \{\}$, and $I(e_c) \leftarrow c$
6. isInstance $\leftarrow$*RecursivelyTryCombinations*$(P, I, (E, \prec))$
7. **if** isInstance
8. **then** $PI \leftarrow PI \cup \{(e_c, Dom(I))\}$
9. **else** $AntiPIC \leftarrow AntiPIC \cup \{e_c\}$
10. **return** PI

Algorithm *RecursivelyTryCombinations*$(P, I, (E, \prec))$
Input: Pattern P, mapping $I : E \rightarrow N$, and partially ordered trace $(E, \prec)$.
Output: Whether $(e_c, Dom(I))$ is an instance of $P\downarrow_{Rng(I)}$.
1. $(*$ Base case $*)$
2. **if** $N \backslash Rng(I)$ is empty
3. **then return** true
4. $(*$ Recursion $*)$
5. select $n \in N \backslash Rng(I)$, and $Candidates(n) \leftarrow \{e \mid e \in E \backslash Dom(I) \wedge \pi_{act}(e) = \alpha(n)\}$
6. **for** $e \in Candidates(n)$
7. **do** $(*$ make the choice $*)$
8. $I \leftarrow I \cup \{e \rightarrow n\}$
9. **if** $(e_c, Dom(i))$ is an instance of $P\downarrow_{Rng(I)}$
10. **then**
isInstance $\leftarrow$*RecursivelyTryCombinations*$(P, I, (E, \prec))$
11. **if** isInstance
12. **then return** true
13. $I \leftarrow I \backslash \{e \rightarrow n\}$ $(*$ undo the choice $*)$
14. $(*$ Has tried all candidates for n and has not found a valid solution $*)$
15. **return** false

The running-time complexity is exponential w.r.t. the size of the pattern (i.e., $|N|$), but polynomial in the size of E_c. In the best case, we try one combination and it already

is an instance, then the algorithm runs in linear time. In the worst case, one may have to try every combination, then the algorithm runs in exponential time. However, we can incrementally check the validity of the chosen candidates so far throught the projection function and efficiently prune the search space. Moreover, note that by only searching for a maximal set of pattern instances instead of all pattern instances, we evade exploring exponentially many matches for the same core-event.

6 Evaluation and Discussion

We implemented our approach as a visualizer called *Log Pattern Explorer* in the *LogPatternExplorer* package in the ProM framework[3]. We conducted two case studies to show how our semi-supervised approach supports the user in detecting complex patterns of interest and gaining important insights into a process while exploring pattern instances. In this section we present our evaluation results.

6.1 Evaluation using BPI Challenge 2012 Log

The BPI Challenge 2012 event log[5] was recorded for a loan application process in a Dutch financial institute. There are 13,087 cases in the log having in total 262,200 events and 36 activities. We used the default oracle $\varphi_{time}(L, dt)$, with $dt = 0$ sec (e.g., events are concurrent if they happened within a second), to obtain partially ordered traces. We discuss our main findings and compare our results with existing tools.

Scenario: Distinct Contexts. Using the *direct-context detector*, we first investigate the events *A_SUBMITTED*. We obtained six patterns, of which four are shown in Fig. 5. We viewed the pattern instances of each pattern and discuss three interesting observations. Firstly, the instances of Pattern 1 ended immediately after *A_DECLINED (A_DE)*. In contrast, the instances of Pattern 2 and 3 are sometimes eventually followed by *A_DE*, as shown in Fig. 7 and 8. However, glancing through the instances, there is a significant difference in the number of the events *A_DE* between these two patterns, which is our second observation. To verify this observation, we extend the two patterns with eventually-causing *A_DE*, leading to P2a and P3a in Figs. 7 and 8. The measures of the extended patterns show that there is indeed a difference: 67.8% of the instances of Pattern 2 eventually caused *A_DE* (i.e., $\frac{\textit{P-supp}(\textit{pattern}2+\textit{A_DE})}{\textit{P-supp}(\textit{pattern}2)} = \frac{2841}{4189} = 67.8\%$); whereas, only 19.3% of the instances of Pattern 3 eventually caused *A_DE* (i.e., $\frac{\textit{P-supp}(\textit{pattern}3+\textit{A_DE})}{\textit{P-supp}(\textit{pattern}3)} = \frac{835}{4320} = 19.3\%$). Thirdly, we observed the same difference between the two patterns in eventually-causing *O_ACCEPTED (O_AC)* (concurrent with three other activities as shown in P3b in Fig. 8). Following the same steps as before, we find that P3 is eventually-followed by *O_AC* in 35% of all instances which happened for P2 only in 11.8% of all instances. These observations suggest that a partly submitted application that is pre-accepted is much less likely to be declined (P3a) and more likely to be accepted (P3b) than those that have to go through *W_Afhandelen leads* (P2a).

[5] 10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f

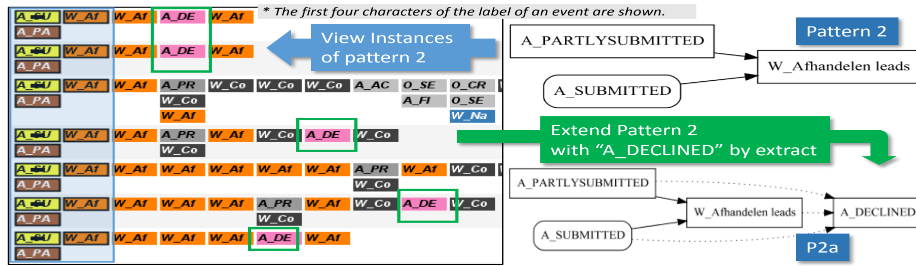

Fig. 7: Viewing the instances of Pattern 2, we observed a significant number of *A_DECLINED (A_DE)*, extended Pattern 2 with eventually-cause *A_DE*.

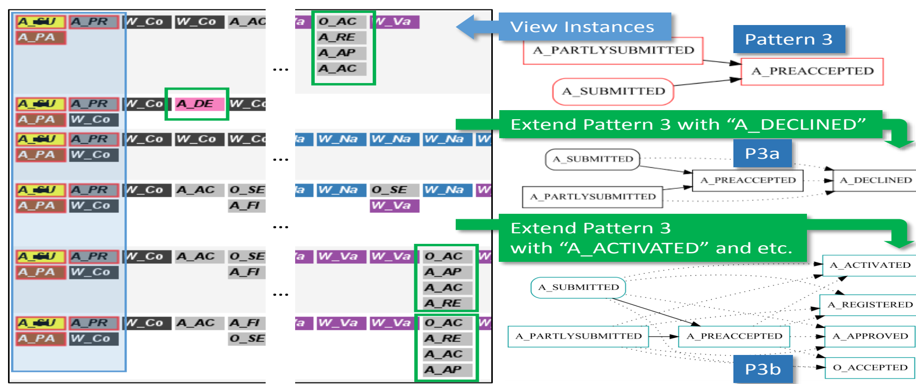

Fig. 8: Viewing instances of Pattern 3 and extended it into two different patterns.

Scenario: Directly-Cause versus Eventually-Cause. We further investigated Pattern 4 in Fig. 5 and found another interesting observation. Glancing through the instances of Pattern 4, we see only 1 of the 59 instances eventually causes *O_AC*, see Fig. 9. Observing that in some cases *W_Beoordelen fraude (Fraud)* does not directly follow, but eventually follows its core-event, we modify P4 by changing the two directly-cause relations into eventually-cause. We then obtained 92 pattern instances. Glancing through these new instances, we suddenly observed many more *O_AC*, and indeed the *P-supp* shows 26 instances eventually-cause *O_AC*. Coloring the events based on their resources showed that the directly-caused *Fraud*'s are executed by resource 112, known to be the system, whereas the eventually-caused (and not directly-caused) *Fraud*'s events are executed by human resources. This difference (i.e., $1/59$ versus $(26-1)/(92-59)$) may suggest that the system user is able to help human resources identify and filter fraudulent cases (which are eventually not accepted). Note that unsupervised approaches not distinguishing *directly-cause* and *eventually-cause* will not be able to detect these two patterns [7].

Scenario: Infrequent Patterns and Anti-Pattern Instances. We then focused on the *A_ACTIVATED*, using *concurrent detector*. *A_ACTIVATED* is concurrently executed with *A_APPROVED (A_AP)* and *A_REGISTERED* in 2246 cases (*P-conf* is 1.0). In 2243 of the 2246 cases they are also concurrent with *O_AC*. Inspecting the 3 anti-

Fig. 9: Significant difference in the number of *O_ACCEPTED* between the patterns "directly-cause *W_Beoordelen fraude*" and "eventually-cause *W_Beoordelen fraude*".

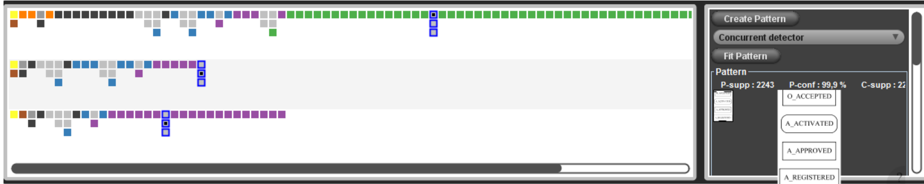

Fig. 10: Three anti-pattern instances of *A_ACTIVATED*.

pattern instances in which *A_ACTIVATED* is not concurrent with *O_AC*, no *O_AC* is found, see Fig. 10. Verified with the data-owner, we understood that these three cases could have severe financial impact. We quote the data-owner "in these cases the activity *O_ACCEPTED (O_AC)* was skipped, while *A_ACTIVATED* was executed. From a business point of view, this implies that the customer never accepted an offer on a loan application, but the money was transferred nonetheless. In total, for 63,000 euro in these three cases." We also observed in 99.7% of the 2050 cases *A_AP* directly-caused *W_Valideren aanvraag (VA)* and ended after that immediately; inspection of the 0.3% where this pattern is not observed shows that 3 additional activities were repeatedly

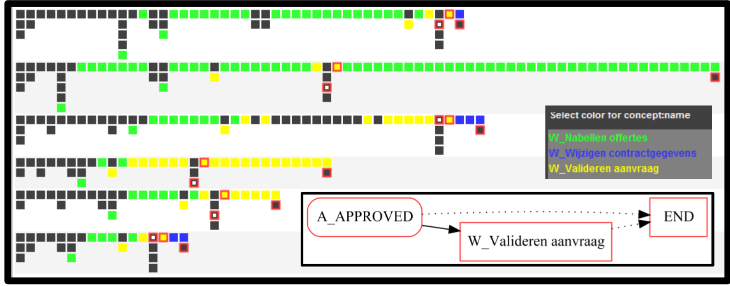

Fig. 11: Six instances of an infrequent pattern.

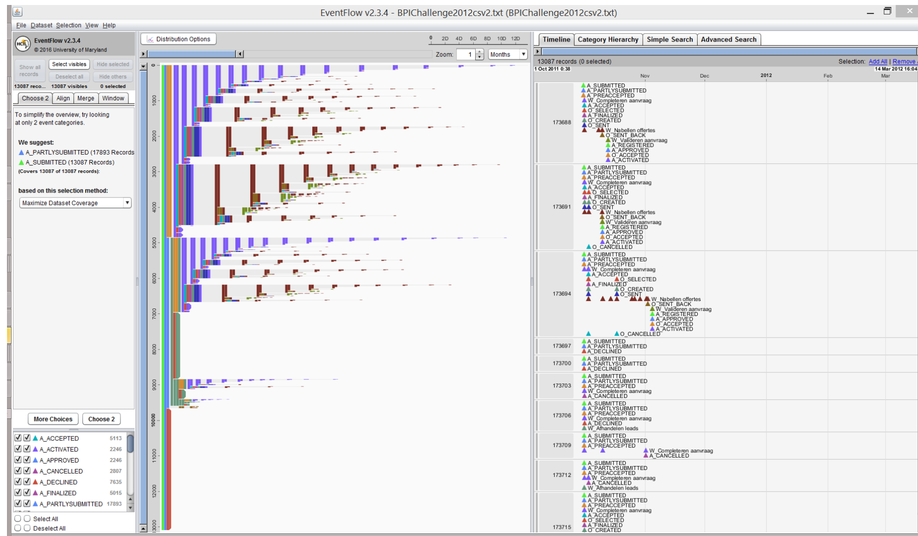

Fig. 12: The EventFlow drawn an overview of the BPI challenge 2012 log.

executed by human resources after *A_AP*, see Fig. 11. This may suggest that the case status was not up-to-date, causing more unnecessary work.

6.2 Results of Existing Approaches Using BPI Challenge Log

We applied existing techniques (for which an implementation is available in Java) on the same BPI Challenge log and discuss the results. Overall, existing approaches have difficulties with detecting patterns that contain a large set of concurrent (independent) events and retrieving their instances, such as pattern P3b. Both LTLChecker [6] and EventFlow [2] have difficulties supporting querying or detecting the instances of pattern P3b for example. Although the user observed some instances of the four concurrent activities (i.e., *A_ACTIVATED*, *A_AP*, *A_REGISTERED*, *O_AC*) in the visualization shown by EventFlow, see Fig. 12, the user has difficulties expressing this as a pattern (query) and retrieving all traces that contain the pattern. Furthermore, another difference we observed is that both techniques only retrieve (anti-)pattern instances retrieved on the case level, instead of events.

For the Dotted Chart [13], we found three versions of in the ProM framework. Applying them on the same log, the three Dotted Chart plugins have shown similar results; the visualization is competent in drawing an overview of the log and helping the analyst to observe high level patterns such as the weekend effect and seasonal patterns in the log. However, the technique is less suitable to observe concrete behavioral patterns. This is because the dots (representing events) drawn are almost indistinguishable, especially when they occurred closely to each other, see Fig. 13. Even manually assigning colors to dots have not helped, see Fig. 14.

For unsupervised approaches, the patterns that are (and can be) detected are limited to their definition of patterns. Applying Pattern Abstraction [3] on the BPI challenge

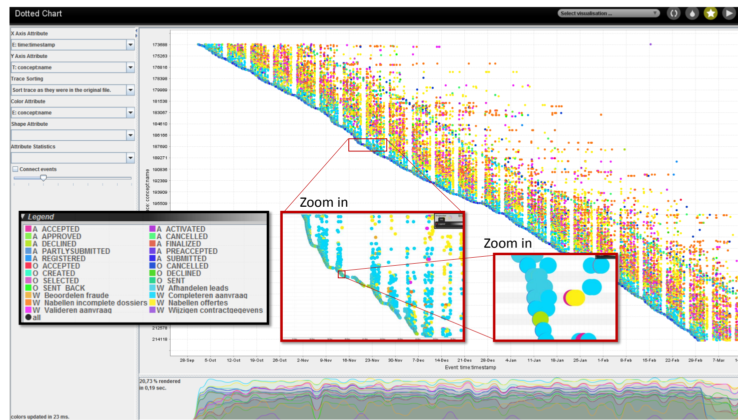

Fig. 13: The Dotted Chart 3 drawn an overview of the BPI challenge 2012 log; unable to distinguish individual dots that occurred closely after eachother.

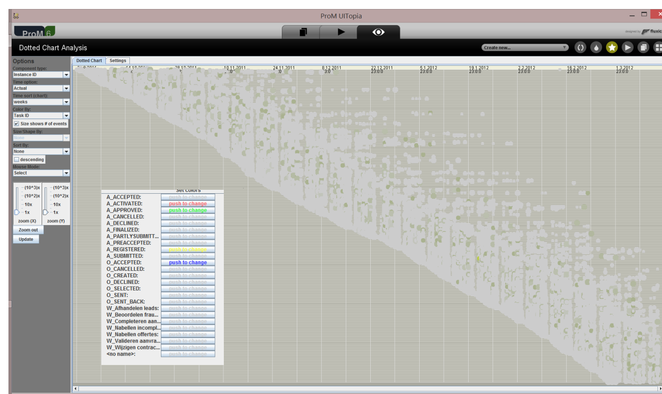

Fig. 14: The dotted chart with color manually assigned to find the concurrent pattern.

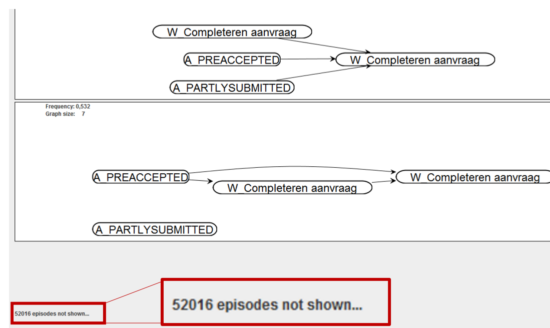

Fig. 15: 52116 patterns found by EpisodeMiner, only the first 100 most frequent are accessible.

2012 log, it detected 231 sets of activities and 5744 patterns using the default setting. However, Pattern Abstraction defines patterns as sequence of activities directly-followed each other, therefore, it cannot detect for example P2a, P3a, and P3b.

Episodes Miner [7] returned 12 frequent patterns (all of the same set of activities) using the default setting. In attempts to find infrequent patterns (e.g., P3b), we lower the frequency threshold and obtained up-to 52116 patterns; however, only the first 100 most frequent patterns are shown, which remain the same set of patterns; no infrequent pattern surfaced, see Fig. 15. As Episodes Miner do not distinguish directly-follows and eventually-follows, it is unable to distinguish for example the two patterns shown in Fig. 9.

6.3 Evaluation using an Insurance Log

We also performed a second evaluation using a claims log from a leading Queensland (Australia) insurance provider. The log for this evaluation was extracted from the claims processing system of a leading Queensland (Australia) provider of Compulsory Third Party (CTP) insurance. The log included 863,828 events comprising 2,584 cases (claims finalised between January 2012 and July 2015 where the claimants‘ injury severity was minimal). To facilitate investigation, the log was filtered to include a random sample of 285 cases, with 144 distinct activities from 50,566 events representing activity completions. Unlike the previous evaluation, this case study exploited a stand-out feature of the Log Pattern Explorer tool, which is its ability to visually highlight concurrent events, to detect concurrency-related data quality (DQ) issues.

Scenario: Form-based Event Capture. A *form-based* DQ issue [9] refers to a set of events, within the same case, that were recorded with the same timestamp. This is a problem as these events did not all **occur** at the same time in reality, but were **recorded** simultaneously due to certain actions that a user may perform on a form. For example, a user may tick the ‘check all’ checkbox to indicate the completion of a set of tasks, triggering the system to record the completion of all these tasks in the log at the time the user clicked ‘save’; rather than the actual times the user completed the individual tasks.

To demonstrate the advantage of the tool, this component of the evaluation was conceived as a double-blind test in which two different researchers used different methods to independently identify frequently occurring sets of concurrent events in the log: researcher *A* used the Log Pattern Explorer and researcher *B* used RapidMiner’s[6] FP-Growth and Create Association Rules operators). Researcher *A* had neither domain knowledge about the insurer’s processes nor prior exposure to the log, while researcher *B* had substantial prior knowledge. Yet, researcher *A* managed to detect and replicate the concurrency-related issues in a quicker manner than researcher B (due to the overheads of data preparation and results interpretation imposed by the data mining tools). Below, we detail how we successfully detected *form-based* DQ issue [9] using the tool.

In preparation for mining association rules, researcher *B* filtered a set of all events $E_C \in E_L$ having timestamps within 1 second of another event in the same case (i.e. deemed to be concurrent with another event in the same case) from the insurance claims

6 https://rapidminer.com/

log, E_L. Let $T = \{(t_1, e_1), ..., (t_m, e_m)\}$ be the set of timestamp, event pairs of events $e \in E_C$ where $t_i = \pi_{time}(e_i)$, $A = \{a_1, ..., a_n\}$ be the set of activity labels of events $e \in E_C$ and $\gamma(a, \pi_{act}(e))$ be a function that returns *true* if the value of the activity label attribute of event e, $\pi_{act}(e)$, is a or *false* otherwise. E_C was 'pivoted' to form tuples of the form $(t_i, \gamma(a_1, \pi_{act}(e_i)), ..., \gamma(a_n, \pi_{act}(e_i))$ and used as input to RapidMiner's FP-Growth and Create Association Rules operators. The association rules identified by using the RapidMiner operators are shown in table 2 from which the manifestations of the Form-based DQ issue shown in table 3 may be distilled. (Note, we include columns representing the pattern support and case support metrics to indicate pervasiveness.)

On the other hand, researcher A used the Log Pattern Explorer tool to discover this DQ issue. With the default oracle $\varphi_{time}(L, dt)$ (where $dt = 0$ sec) a clue to the existence of this DQ issue is the presence of recurring stacked tiles across cases. Through colouring of events (based on their activity label) in a stack, we can easily observe frequently occurring groups of activities. For example, in Figure 16 (left), three concurrent activities (light blue, deep blue and purple), a manifestation of form-based DQ issue, can be seen to occur frequently. Through an iterative process of tile colouring, pattern editing and assessment of pattern pervasiveness, researcher A was able to distill 4 out of the top 6 manifestations of the form-based DQ issues that were independently-discovered by researcher B.

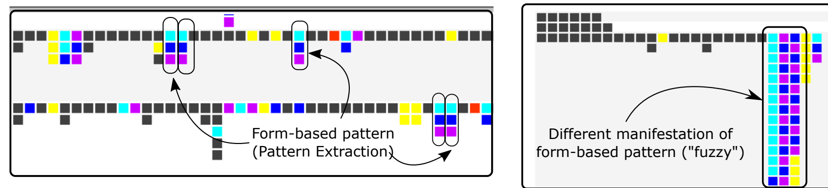

Fig. 16: Example of form-based DQ issues discovered through Log Pattern Explorer

This experiment also illustrates a 'fuzzy' way to detect variations of DQ patterns of interest. Using the 'pattern fitting' approach (described above) can still be rather limiting in finding patterns when one is in the exploration stage. For example, in Figure 16 (right), we see another instance of this DQ issue: the three activities of interest happened repeatedly and could occur in 'close proximity' to each other (either concurrently, directly-followed, or directly-succeeding one another), thus they did not fit the 'same timestamp' definition that was originally defined in [9] for the form-based DQ issue. These variations are unlikely to be known by process analysts in the early stage of analysis. The Log Pattern Explorer tool thus allows one to discover the possible variations of a particular event log quality issue, such that a more comprehensive approach can be taken in the subsequent cleaning of the log.

Scenario: Collateral Events. The *collateral events* DQ issue [9] is an event log quality issue that manifests itself when the occurrence of one event triggered the firing of other events within a short amount of time (for example, within seconds or minutes). These subsequent events being fired may not be meaningful or important from the perspective of the process being analysed (e.g. automated notification emails being sent to various parties upon the receipt of an insurance claim).

Premise	Conclusion	Confidence	Lift
Complete Initial Claim Estimate	Request Initial Claim Evidence	0.985	79.259
Request Initial Claim Evidence	Complete Initial Claim Estimate	0.942	79.259
EST	Estimate submitted for Approval	0.994	31.914
Estimate submitted for Approval	EST	1.0	31.914
Review Outstanding Accounts	Commence Settlement Assessment	0.721	26.689
Commence Settlement Assessment	Review Outstanding Accounts	0.620	26.689
Pay Settlement Assessment	Commence Settlement Assessment	0.610	2.573
QUANT	Quantum submitted for Rationale Review and Quantum Approval	0.576	11.545
Quantum submitted for Rationale Review and Quantum Approval	QUANT	1.0	11.545
Review and Action new Discharge document	Review and Action new Statutory Bodies document	0.560	7.936
Review and Action Uploaded Rehabilitation Document Consider Referral to RSA, Action Rehab Treatment Plan	Review and Action ERP	0.733	5.096
Review and Action ERP	Action Rehab Treatment Plan	0.683	5.080
Action Rehab Treatment Plan	Review and Action ERP	0.731	5.078
Review and Action Uploaded Rehabilitation Document Consider Referral to RSA, Review and Action ERP	Action Rehab Treatment Plan	0.619	4.600
Review and Action Uploaded Rehabilitation Document Consider Referral to RSA	Review and Action ERP	0.562	3.908
Review and Action ERP	Review and Action Uploaded Rehabilitation Document Consider Referral to RSA	0.706	3.908
Review and Action new Employment document	Review and Action new Correspondence document	0.746	3.792
Review and Action new Legal document	Review and Action new Correspondence document	0.701	3.565
Review and Action ERP, Action Rehab Treatment Plan	Review and Action Uploaded Rehabilitation Document Consider Referral to RSA	0.639	3.539
Action Rehab Treatment Plan	Review and Action Uploaded Rehabilitation Document Consider Referral to RSA	0.637	3.528
Review Correspondence	Review and Action new Correspondence document	0.667	3.390
Review and Action Scanned Rehabilitation Document Consider Referral to RSA	Review and Action Uploaded Rehabilitation Document Consider Referral to RSA	0.569	3.151
Review and Action new Statutory Bodies document	Review and Action new Correspondence document	0.614	3.122
Review and Action new Discharge document	Review and Action new Correspondence document	0.596	3.031

Table 2: Association rules mined using RapidMiner

Frequently occurring concurrent activities	Pattern support	Case support
QUANT, Quantum submitted for Rationale Review and Quantum Approval	303	185
Review and Action Uploaded Rehabilitation Document Consider Referral to RSA, Review and Action ERP, Action Rehab Treatment Plan	297	71
EST, Estimate submitted for Approval	184	131
Review Outstanding Accounts, Commence Settlement Assessment	126	126
Pay Settlement Asessment, Commence Settlement Assessment	98	98
Complete Initial Claim Estimate, Request Initial Claim Evidence	66	66

Table 3: Manifestations of the Form-based DQ issue distilled from association rules

The existence of *collateral events* DQ issue can be easily detected using the Log Pattern Explorer tool. In particular, using the 'same timestamp' layout, one can adjust the time window for two events to be considered concurrent and observe the changes in the layout of the events, if any, with any small increase in the concurrent time window.

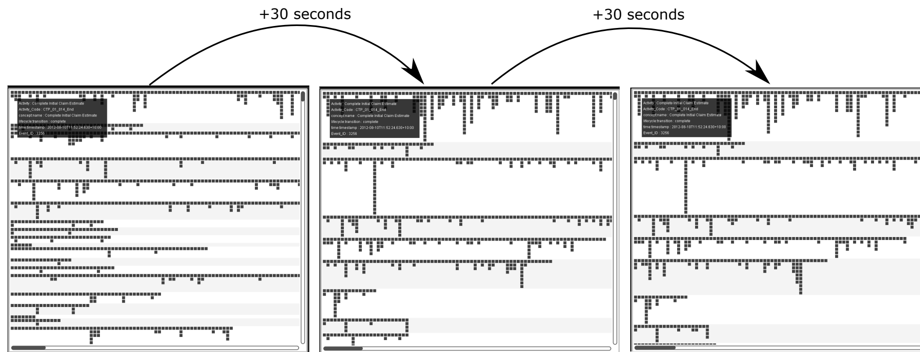

Fig. 17: Changes in the layout of events, each with a 30-second increase in the concurrency time window.

Figure 17 shows three screenshots capturing the changes in the tile layout for the same event log as the concurrency time window changes from 0 seconds (left window), to 30 seconds (middle figure), and to 60 seconds (right figure). We can see that the change from 0 to 30 seconds resulted in a *substantial* change in the shape of the layout for the same log. This simple, yet powerful, information tells us that the event log contains many events within a case that were separated by fewer than 30-second difference between them - a symptom of the existence of the collateral events pattern. Equally interesting, as we increase the time window to 60 seconds, we noticed only minor changes in the layout of the tiles. An interesting observation here, therefore, is that

the 30-second (or shorter) window may be an measure of the internal system latency. This insight can be further used to determine if two events separated by less than 30 seconds are indeed sequential or a result of the delay in the logging of the events.

To conclusively detect if collateral patterns do exist, one needs to understand the make up of those events that are stacked together (that is, there needs to be an explanation as to why the occurrence of one event triggered the firing of other events within a short amount of time). In this situation, a similar approach to detecting form-based pattern (as described above) can be taken.

Scenario:Homonymous Label A *homonymous label* DQ issue [9] refers to a situation whereby two or more events within a case have the same label; however, the interpretation of those labels are different due to the changes in the context of the case. For example, in a hospital setting, an activity labeled 'Triage Patient' may be interpreted as a nurse triaging a patient as he/she arrives in an emergency department. However, in a log, we may see this activity being recorded a second time *after* the patient was discharged. This second triage activity actually refers to a nurse or a doctor reviewing the triage activity of a patient, instead of triaging the patient again. Therefore, the interpretation of the same event label between the first and second occurrences is different.

In this section, we described how we used the Log Pattern Explorer tool to discover the presence of the homonymous label DQ issue. Some domain knowledge is required as we need to know the activity label that is likely to cause this DQ issue. Through domain expert's knowledge, it was noted that the activity 'Complete Initial Estimate Workflow', which mostly happens at the beginning of a case (see Figure 18) could sometimes occur later in a case. However, when it occurs for the second or subsequent time, it does not mean that another 'initial' claim estimate was performed. Rather, it often occurs when the insurance company determines that the other party's insurer (in a multi-party accident insurance claim) is responsible for the claim. Therefore, the claim is now being transferred to the other insurance company (this is reflected by the occurrence of the 'Recover Claim and/or Management costs upon transfer of claim to another insurer' - coloured as red in the bottom part of Figure 18. The second occurrence of the 'Complete Initial Estimate Workflow', in this context, actually refers to the original insurance company estimating the cost that they have incurred thus far so that they can recover it from the insurance company to whom the claim is being transferred.

Using the Log Pattern Explorer tool, it is interesting to note that such behaviour can be easily seen by highlighting activities of interest, the recurrence of the 'Complete Initial Estimate Workflow' is clearly unfolded in the visualisation. Note that the second occurrence of this activity, in light of the changing context of the case, would not be easily observed through the analysis of a process model as most process discovery techniques would create a single node for this activity.

7 Conclusion and Future work

In this paper, we proposed a semi-supervised approach for log pattern detection. We defined our patterns as partial orders and distinguished a core-event to help the user detect patterns of interest. We use concurrency and contextual information of the core-events and support the user in extracting, modifying, and extending patterns. The two

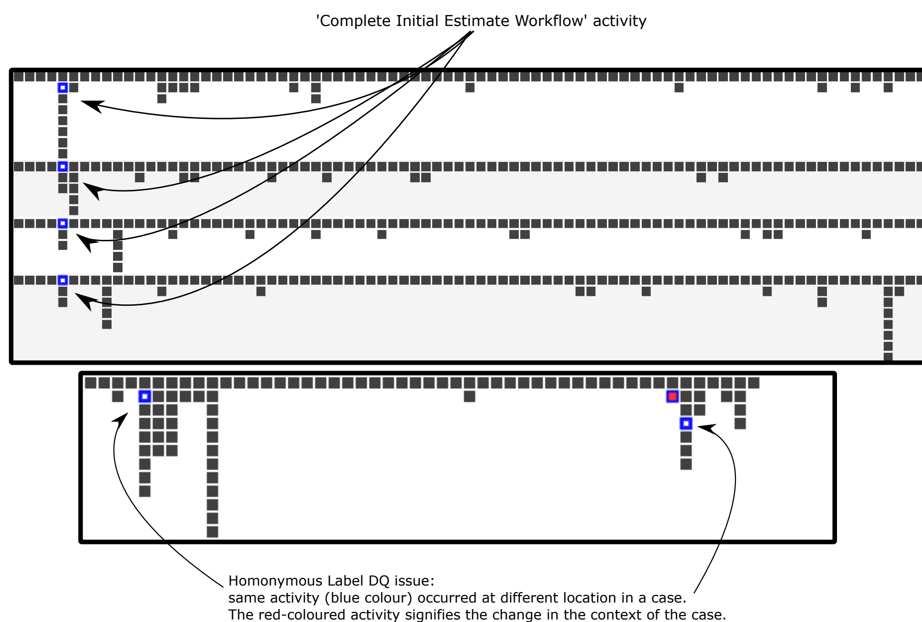

Fig. 18: The activity 'Complete Initial Estimate Workflow' (coloured as blue) often occurs early in a case (top). This activity can also be the source for a Homonymous Label DQ issue

case studies show that our approach is successful in assisting process analysts in finding complex patterns and infrequent patterns of interest. Future work aims at empirically evaluating the approach and the tool with process analysts. Moreover, we would like to integrate log cleaning operations, such as event abstraction, event relabeling, event filtering etc., and recommend such operations for the patterns detected.

References

1. Bautista, A.D., Wangikar, L., Akbar, S.M.K.: Process mining-driven optimization of a consumer loan approvals process - the BPIC 2012 challenge case study. In: BPM Workshops. (2012) 219–220
2. Monroe, M., Lan, R., Lee, H., Plaisant, C., Shneiderman, B.: Temporal event sequence simplification. IEEE Trans. Vis. Comput. Graph. **19**(12) (2013) 2227–2236
3. Bose, R.J.C., van der Aalst, W.M.: Abstractions in process mining: A taxonomy of patterns. In: BPM. Volume 5701 of LNCS., Springer (2009) 159–175
4. Günther, C., Rozinat, A., van der Aalst W.M.P.: Activity mining by global trace segmentation. In: BPM. Volume 43 of LNBIP., Springer (2009) 128–139
5. Mannhardt, F., de Leoni, M., Reijers, H.A., van der Aalst, W.M.P., Toussaint, P.J.: From low-level events to activities - A pattern-based approach. In: BPM. Volume 9850 of LNCS., Springer (2016) 125–141
6. Maggi, F.M., Montali, M., Westergaard, M., van der Aalst, W.M.P.: Monitoring business constraints with linear temporal logic: An approach based on colored automata. In: BPM 2011. (2011) 132–147
7. Leemans, M., van der Aalst, W.M.P.: Discovery of frequent episodes in event logs. In: SIMPDA. Volume 1293 of CEUR., CEUR-WS.org (2014) 31–45
8. Diamantini, C., Genga, L., Potena, D.: Behavioral process mining for unstructured processes. J. Intell. Inf. Syst. **47**(1) (2016) 5–32
9. Suriadi, S., Andrews, R., ter Hofstede, A.H., Wynn, M.T.: Event log imperfection patterns for process mining: Towards a systematic approach to cleaning event logs. Information Systems **64** (2017) 132–150
10. Ferreira, D.R., Szimanski, F., Ralha, C.G.: Improving process models by mining mappings of low-level events to high-level activities. J. Intell. Inf. Syst. **43**(2) (2014) 379–407
11. Tax, N., Sidorova, N., Haakma, R., van der Aalst, W.M.P.: Mining local process models. J. Innovation in Digital Ecosystems **3**(2) (2016) 183–196
12. Baier, T., Rogge-Solti, A., Mendling, J., Weske, M.: Matching of events and activities: an approach based on behavioral constraint satisfaction. In: SAC, ACM (2015) 1225–1230
13. Song, M., van der Aalst, W.M.: Supporting process mining by showing events at a glance. In: Proceedings of WITS. (2007) 139–145
14. Lu, X., Fahland, D., van der Aalst, W.M.: Conformance checking based on partially ordered event data. In: BPM Workshops. Volume 202 of LNBIP., Springer (2014) 75–88
15. Ponce de León, H., Rodríguez, C., Carmona, J., Heljanko, K., Haar, S.: Unfolding-based process discovery. In: ATVA. Volume 9364 of LNCS., Springer (2015) 31–47
16. Mokhov, A., Carmona, J., Beaumont, J.: Mining conditional partial order graphs from event logs. T. Petri Nets and Other Models of Concurrency **11** (2016) 114–136
17. Diamantini, C., Genga, L., Potena, D., van der Aalst, W.M.: Towards process instances building for spaghetti processes. In: Proceedings of SEBD. (2015) 256–263
18. Armas-Cervantes, A., Dumas, M., La Rosa, M.: Discovering local concurrency relations in business process event logs. (2016)