Log-Based Understanding of Business Processes through Temporal Logic Query Checking

Margus Räim[1], Claudio Di Ciccio[2], Fabrizio Maria Maggi[1], Massimo Mecella[3], and Jan Mendling[2,⋆]

[1] University of Tartu, Estonia
f.m.maggi@ut.ee
[2] Vienna University of Economics and Business, Austria
{claudio.di.ciccio,jan.mendling}@wu.ac.at
[3] Sapienza Università di Roma, Italy
massimo.mecella@uniroma1.it

Abstract. Process mining is a discipline that aims at discovering, monitoring and improving real-life processes by extracting knowledge from event logs. Process discovery and conformance checking are the two main process mining tasks. Process discovery techniques can be used to learn a process model from example traces in an event log, whereas the goal of conformance checking is to compare the observed behavior in the event log with the modeled behavior. In this paper, we propose an approach based on *temporal logic query checking*, which is in the middle between process discovery and conformance checking. It can be used to discover those LTL-based business rules that are valid in the log, by checking against the log a (user-defined) class of rules. The proposed approach is not limited to provide a boolean answer about the validity of a business rule in the log, but it rather provides valuable diagnostics in terms of traces in which the rule is satisfied (witnesses) and traces in which the rule is violated (counterexamples). We have implemented our approach as a proof of concept and conducted a wide experimentation using both synthetic and real-life logs.

Keywords: Process Discovery, Business Rules, Linear Temporal Logic, Temporal Logic Query Checking.

1 Introduction

The increasing availability of event data has stimulated the uptake of automatic process discovery in research and practice. Discovery techniques typically take event logs as an input to generate a process model as an output, for instance, in the form of a Petri net. This works particularly well for structured processes.

However, event logs might not always stem from structured processes, but from RFID readers, from GPS data of trucks, or from unstructured knowledge-intensive processes. Events from these types of processes have often in common that they yield

⋆ The work of Massimo Mecella has been partly supported by the Italian projects SUPER, RoMA and the cluster Social Museum and Smart Tourism, and by the EU projects SmartVortex and VOICE. The work of Claudio Di Ciccio and Jan Mendling has received funding from the EU Seventh Framework Programme under grant agreement 318275 (GET Service).

R. Meersman et al. (Eds.): OTM 2014, LNCS 8841, pp. 75–92, 2014.
© Springer-Verlag Berlin Heidelberg 2014

spaghetti-like Petri net models. In order to make discovery work for those event logs, declarative representations have been defined that capture behavioral constraints.

So far, several specifications of declarative behavior have been defined, e.g., Declare [23], behavioral profiles [26], or Dynamic Condition Response Graphs [16]. A standard formalism underneath these notations is Linear Temporal Logic (LTL). However, existing LTL-based discovery techniques like [12,14,20,22,19] hardly make use of the flexibility that LTL provides, but rather work with a limited set of predefined templates. On the other hand, works like [25] for conformance checking with respect to LTL-based business rules, do not support domain experts in finding classes of LTL rules that are interesting to check. In this paper, we address this research gap. Our contribution is an approach for discovering LTL rules from event logs based on temporal logic query checking using placeholders. The technique produces as outcome activities that can replace the placeholders in the query to produce rules satisfied in the log. If the activities considered for these placeholders cover the full process alphabet, the approach complements existing process discovery techniques. If a placeholder is replaced by a single activity, our approach supports conformance checking. In the middle there is the possibility of choosing the replacement for a placeholder within a (user-defined) set of activities. The proposed approach is not limited to provide a boolean answer about the validity of a business rule in the log, but it rather provides diagnostics in terms of traces in which the solution is satisfied (*witnesses*) and traces in which the solution is violated (*counterexamples*). Our concepts have been implemented and evaluated using synthetic and real-life logs. In this way, we are able to find behavioral rules that might not have been visible, e.g., to predefined Declare templates.

Against this background, the paper is structured as follows. Section 2 defines the preliminaries and the research problem. Section 3 defines our approach based on the log temporal structure and a corresponding query checking algorithm. Section 4 describes experimental results of using our implementation on synthetic and real-life logs. Section 5 reflects our contribution in the light of related work. Finally, Section 6 concludes the paper.

2 Preliminaries

The research objective of this work is to support the domain expert with a technique to discover temporal business rules and to define rule templates with placeholders. This is, on the one hand, a verification problem [3], and, on the other hand, a temporal query checking problem. We approach this from the perspective of temporal logical query checking; therefore we first introduce the notions of event logs, linear temporal logic over finite traces and respective queries, and then we discuss the research problem.

2.1 Event Logs

An event log captures the manifestation of events pertaining to the instances of a single process. A *process instance* is also referred to as a *case*. Each event in the log corresponds to a single case and can be related to an activity or a task. Events within a case need to be *ordered*. An event may also carry optional additional information like

timestamp, *transaction type*, *resource*, *costs*, etc. For analysis, we need a function that maps any event e onto its *class* $\overline{e}$. In this paper, we assume that each event is classified based on its activity. $\mathcal{A}$ denotes the set of *activities* (*log alphabet*). $\mathcal{A}^+$ is the set of all non-empty finite sequences. A case is described as a *trace* over $\mathcal{A}$, i.e., a finite sequence of activities.

Definition 1 (Trace). *Given a finite* log alphabet $\mathcal{A}$ *of events, the trace* $\boldsymbol{t} = \langle a_1, \ldots, a_n \rangle$ *is a finite sequence of events* $a_i \in \mathcal{A}$. *We denote by* $\boldsymbol{t}(i)$ *the* i*-th event in the trace, with* $i \in [1, n]$ *where* n *is the length of* $\boldsymbol{t}$*, i.e.,* $n = |\boldsymbol{t}|$. *Thus,* $\boldsymbol{t} \in \mathcal{A}^+$*, where* $\mathcal{A}^+$ *denotes the set of non-empty sequences of elements of* $\mathcal{A}$. *Two traces* $\boldsymbol{t}$ *and* $\boldsymbol{t}'$ *are defined* equivalent *when* $|\boldsymbol{t}| = |\boldsymbol{t}'| = n$ *and, for all* $i \in [1, n]$, $\boldsymbol{t}(i) = \boldsymbol{t}'(i)$.

Examples of traces are $\mathbf{t}_1 = \langle a, b, c, d \rangle$ and $\mathbf{t}_2 = \langle a, b, b, b, a, d \rangle$.

Definition 2 (Event Log). *An event log* $L \in \mathcal{P}(\mathcal{A}^+)$ *is a finite collection of traces.*

For example, $L = [\langle a, b, c, d \rangle, \langle a, b, c, d \rangle, \langle a, b, b, b, a, d \rangle]$ is a log consisting of three cases. Two cases follow the trace $\langle a, b, c, d \rangle$ and one case follows the trace $\langle a, b, b, b, a, d \rangle$.

2.2 Linear Temporal Logic over Finite Traces

Linear Temporal Logic (LTL) [24] is a language meant to express properties that hold true in systems that change their state over time. The state of the system is expressed in terms of propositional formulas. The evolution is defined by transitions between states. A typical LTL query expressing *fairness* conditions is $\Box\Diamond\Phi$, where Φ is a propositional formula indicating the condition to always ($\Box$) eventually ($\Diamond$) hold true. LTL$_f$ [10,9] is the variant of LTL interpreted over finite system executions. It adopts the syntax of LTL. Formulas of LTL (and LTL$_f$) are built from a set $\mathcal{A}$ of propositional symbols (*atomic propositions*, altogether constituting the *alphabet*) and are closed under the boolean connectives ($\neg$, unary, and $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$, binary) and the temporal operators $\circ$ (*next*), $\Diamond$ (*eventually*), $\Box$ (*always*), unary, and U (*until*), binary. The syntax is defined as follows.

$$
\begin{array}{lllllll}
\varphi, \psi = & a & \mid \rho & \mid \tau & & & \text{(with } a \in \mathcal{A}\text{)} \\
\rho = & \neg\varphi & \mid \varphi \vee \psi & \mid \varphi \wedge \psi & \mid \varphi \rightarrow \psi & \mid \varphi \leftrightarrow \psi & \\
\tau = & \circ\varphi & \mid \Diamond\varphi & \mid \Box\varphi & \mid \varphi U \psi & &
\end{array}
$$

Intuitively, $\circ\varphi$ means that φ holds true in the next instant in time, $\Diamond\varphi$ signifies that φ holds eventually before the last instant in time (included), $\Box\varphi$ expresses the fact that from the current state until the last instant in time φ holds, $\varphi U \psi$ says that ψ holds eventually in the future and φ holds until that point. The semantics of LTL$_f$ is provided in terms of finite runs,[1] i.e., finite sequences of consecutive instants in time, represented by finite words π over the alphabet $2^{\mathcal{A}}$. The instant i in run π is denoted as $\pi(i)$, with $i \in [1, |\pi|]$, $|\pi|$ being the length of the run. We remark here that the alphabet

[1] Called "traces" in the literature [10], here renamed in order to avoid confusions with the sequences of events in logs.

of finite words for LTL$_f$ runs do not correspond to $\mathcal{A}$, but to any possible propositional interpretation of the propositional symbols in $\mathcal{A}$ ($2^{\mathcal{A}}$). In the following, we indicate that, e.g., a is interpreted as true ($\top$) at instant i in π by $a \in \pi(i)$. Conversely, if $a \notin \pi(i)$, a is interpreted as false ($\bot$). Given a finite run π, we inductively define when an LTL$_f$ formula φ (resp. ψ) is true at an instant i, denoted as $\pi, i \models \varphi$ (resp., $\pi, i \models \psi$), as:
$\pi, i \models a$ for $a \in \mathcal{A}$, iff $a \in \pi(i)$ (a is interpreted as true in $\pi(i)$);
$\pi, i \models \neg\varphi$ iff $\varphi, i \not\models \pi(i)$;
$\pi, i \models \varphi \wedge \psi$ iff $\pi, i \models \varphi$ and $\pi, i \models \psi$;
$\pi, i \models \varphi \vee \psi$ iff $\pi, i \models \varphi$ or $\pi, i \models \psi$;
$\pi, i \models \circ\varphi$ iff $\pi, i+1 \models \varphi$, having $i < |\pi|$;
$\pi, i \models \varphi U \psi$ iff for some $j \in [i, |\pi|]$, we have that $\pi, j \models \psi$, and for all $k \in [i, j-1]$, we have that $\pi, k \models \varphi$.
The semantics of remaining operators can be derived by recalling that:

$$\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi); \qquad \Diamond\varphi \equiv \top U \varphi;$$
$$\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi); \qquad \Box\varphi \equiv \neg\Diamond\neg\varphi.$$

2.3 Temporal Logic Query Checking

Model checking [8] was originally proposed as a verification technique: given a model of a software system and a specification, formal methods are adopted to prove the compliance of the model to the specification. Temporal model checking focuses on the automated verification of temporal logic formulas over temporal structures, modeled as Kripke structures. Kripke structures are finite state automata having sets of states, a labeling function mapping states to atomic propositions interpreted as true, and a left-total transition relation among states, i.e., every state leads to a transition. Initial states are those states from which runs of the model enactment can start.

Temporal logic query checking [3] aims at discovering properties of the model which are not known *a priori*. Therefore, the extra input element as compared to model checking is a *query*, i.e., a temporal logic formula containing so-called *placeholders*, typically denoted as $?_x$. The output of the query checking technique is a set of temporal logic formulas, which derive from the input query. Every output formula stems from the replacement of placeholders by propositional formulas, which make the overall temporal logic formula satisfied in the given Kripke structure.

The seminal work of Chan considered Computation Tree Logic (CTL) [7] as the language for expressing queries. Unlike LTL, which adopts a linear time assumption, CTL is a branching-time logic: the evaluation of the formula is based on a tree-like structure, where different evolutions of the system over time are simultaneously considered in different branches of the computation model. Consider as an example the CTL query $AG(?_x \vee p)$. This query states that for every evolution of the system (A), it is globally true (G) that p or $?_x$ hold. In order to drive the verification by limiting the search space of possible satisfying replacements for placeholders, Chan introduced the opportunity to limit the propositional symbols involved. For instance, a CTL query like $AG(?_x)\{a, b\}$ would restrict the possible solutions returned to the ones including only a and b as proposition letters.

3 Proposed Approach

In this section, we describe the proposed approach in detail. The starting point is an event log and an LTL$_f$ query. The outcome is a set of formulas derived from the query by placeholder replacement, along with diagnostics about the validity of each formula in the input log.

3.1 From the Event Log to a Log Temporal Structure

Here, we illustrate the steps to obtain from the log a behaviorally equivalent temporal structure, for evaluating LTL$_f$ formulas over it. The first step consists in creating a trace temporal structure for each trace in the log. The only allowed run for such a temporal structure is the trace itself. Then, we combine the collection of trace temporal structures into a single entity (community). Thereupon, we derive a log temporal structure, bisimilar to such a community, by joining the common suffixes of traces. This translates to joining the states of trace temporal structures for which all next successor states are pairwise interpreted as the same atomic propositions. The resulting polytree is the input for the query checking algorithm described later in Section 3.2.

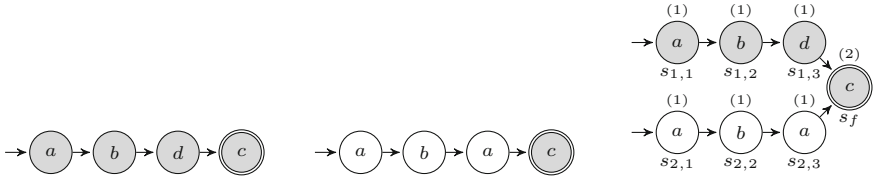
Definition 3 (Trace temporal structure). *A trace temporal structure is a tuple* $\mathcal{T} = \langle S, \triangleright, s_1, s_f, L \rangle$ *over alphabet* $\mathcal{A}$ *where:*

S *is a finite non-empty set of* states.
$s_1, s_f \in S$ *are the initial and final state of* $\mathcal{T}$.
$L : S \to (\mathcal{A} \to \{\top, \bot\})$ *is the labeling function, associating each state* $s \in S$ *to an interpretation of each propositional literal* $a \in \mathcal{A}$*, either assigned as true,* $\top$*, or false,* $\bot$*. If, e.g.,* a *is interpreted as true in state* s*, we indicate it as follows:* $L(s, a) \mapsto \top$.
$\triangleright : S \setminus \{s_f\} \to S$ *is the bijective* successor state function*, associating each state* $s \in S \setminus \{s_f\}$ *to one following state* $s' \in S$*. For the sake of conciseness, when* $(s, s') \in \triangleright$*, we also refer to* s' *as* $s\triangleright$.

The successor state function $\triangleright$ constitutes the transition relation for the temporal structure. It is *bijective* over the domain of $S \setminus \{s_f\}$ and the codomain of S. As such, an inverse function exists, which we refer to as $\triangleleft$ (*predecessor state function*). When $(s, s') \in \triangleright$, we also refer to s as $\triangleleft s'$. With a slight abuse of notation, we define predecessors and successors for *sets* of states too: given $\hat{S} \subseteq S$, we have $\triangleleft\hat{S} \doteq \left\{ s \in S | s = \triangleleft\hat{s}, \hat{s} \in \hat{S} \right\}$ and $\hat{S}\triangleright \doteq \left\{ s \in S | s = \hat{s}\triangleright, \hat{s} \in \hat{S} \right\}$. We denote by $\mathcal{A}^{\mathcal{T}}$ the universe of possible trace temporal structures over alphabet $\mathcal{A}$.

Figs. 1a and 1b show two trace temporal structures. They stem from two traces, i.e., $\langle a, b, d, c \rangle$ and $\langle a, b, a, c \rangle$. Indeed, in order to check LTL$_f$ formulas on traces, we apply a mapping function $\mathcal{M}$, defined as follows. The $\mathcal{M}$-mapping of an element of $\mathbf{t}$ to an element of $\mathcal{T}$ is denoted by the infix notation $\overset{\mathcal{M}}{\longmapsto}$.

Definition 4 ($\mathcal{M}$-mapping from a trace to a trace temporal structure). *Let* $\boldsymbol{t} = \langle a_1, \ldots, a_n \rangle \in \mathcal{A}^+$ *be a trace, and* $\mathcal{T} = \langle S, \triangleright, s_1, s_f, L \rangle \in \mathcal{A}^{\mathcal{T}}$ *a trace temporal structure, both defined over the common alphabet* $\mathcal{A}$.
A mapping $\mathcal{M} : \mathcal{A}^+ \to \mathcal{A}^{\mathcal{T}}$ *is such that:*



(a) States satisfying $\square(a \rightarrow \Diamond b)$ in trace temporal structure $\mathcal{T}_1$, derived from $\mathbf{t}_1 = \langle a, b, d, c\rangle$.

(b) States satisfying $\square(a \rightarrow \Diamond b)$ in trace temporal structure $\mathcal{T}_2$, derived from $\mathbf{t}_2 = \langle a, b, a, c\rangle$.

(c) States satisfying $\square(a \rightarrow \Diamond b)$ in log temporal structure $\mathcal{L}$ for log L, consisting of $\mathbf{t}_1$ and $\mathbf{t}_2$.

Fig. 1. Trace temporal structures and log temporal structure for log $L = [\langle a, b, d, c\rangle, \langle a, b, a, c\rangle]$ and evaluation of formula $\square(a \rightarrow \Diamond b)$ over them. In Fig. 1c, weights are specified among parentheses above states. State identifiers are below states.

- *for all* $i \in [1, n]$, $\boldsymbol{t}(i) \overset{\mathcal{M}}{\longmapsto} s_i$, *with* $s_i \in S$;
  for all $s_i \in S$, *there exists* $i \in [1, n]$ *such that* $\boldsymbol{t}(i) \overset{\mathcal{M}}{\longmapsto} s_i$;
  given $i, j \in [1, n]$, *if* $i \neq j$ *then* $s_i \neq s_j$ *($\mathcal{M}$ preserves bijection over events and states);*
- *given* $a \in \mathcal{A}$ *s.t.* $a = \boldsymbol{t}(i)$, *then* $L(s, a) \mapsto \top$, *and for all* $a' \in \mathcal{A}$ *s.t.* $a' \neq a$, $L(s, a) \mapsto \bot$;
- $\boldsymbol{t}(1) \overset{\mathcal{M}}{\longmapsto} s_1$; $\boldsymbol{t}(n) \overset{\mathcal{M}}{\longmapsto} s_f$.

Fig. 1a and Fig. 1b show the trace temporal structures $\mathcal{T}_1$ and $\mathcal{T}_2$ to which $\mathbf{t}_1 = \langle a, b, d, c\rangle$ and $\mathbf{t}_2 = \langle a, b, a, c\rangle$ $\mathcal{M}$-map respectively. The reader can notice that each trace corresponds to the only allowed *run* of the $\mathcal{M}$-mapped temporal structure (see Section 2.2). As a log L is a collection of traces (see Section 2.1), we denote a collection of mapped temporal structures by $\mathcal{M}(L)$, with a slight abuse of notation. By definition, $\mathcal{M}(L) \subseteq \mathcal{A}^{\mathcal{T}}$. In order to evaluate the LTL$_f$ formula over a representation of the entire log, seen as a whole, we introduce the following temporal structure.

Definition 5 (Log weighted temporal structure). *A log weighted temporal structure (or log temporal structure, for short) is a tuple* $\mathcal{L} = \langle S, \triangleright, S_1, S_f, L, W\rangle$ *over alphabet* $\mathcal{A}$ *where:*

S *is a finite non-empty set of* states.
$S_1, S_f \in S$ *are the initial and final sets of states in* $\mathcal{L}$.
$L : S \rightarrow (\mathcal{A} \rightarrow \{\top, \bot\})$ *is the labeling function, associating each state* $s \in S$ *to an interpretation of each propositional literal* $a \in \mathcal{A}$.
$\triangleright : S \setminus S_f \rightarrow S$ *is the surjective* successor state function*, which associates each state* $s \in S \setminus S_f$ *to one following state* $s' \in S$.
$W : S \rightarrow \mathbb{N}^+$ *is the* weighing function*, associating each state* $s \in S$ *to a positive integer* $W(s) > 0$ *(*weight*).*

The successor state function $\triangleright$ constitutes the transition relation for the temporal structure. Unlike the successor state function of trace temporal structures, the successor state function of log temporal structures is only surjective and not injective: in other

words, it can happen that $s \neq s'$ but $s\triangleright = s'\triangleright$, i.e., different states share the same successor. Therefore, no inverse function can be derived. Nonetheless, with a slight abuse of notation, we denote as $\triangleleft s'$ the set of states whose successor is $s' \in S$: $\triangleleft s' \doteq \{s \in S | s' = s\triangleright\}$. Fig. 1c shows the log temporal structure derived from the collections of temporal structures depicted in Figs. 1a and 1b. We denote the universe of possible log temporal structures defined over $\mathcal{A}$ as $\mathcal{A}^{\mathcal{L}}$.

We introduce now the *community of trace temporal structures* $\mathcal{C}$, which is needed to pass from a collection of tuples to a single-tuple equivalent representation. It consists of the union of trace temporal structures. For the sake of clarity, we adopt the "." symbol in the remainder to specify to which temporal structure a given element belongs to. For instance, the set of states of community $\mathcal{C}$ is denoted as $\mathcal{C}.S$, whereas the set of states of trace temporal structure $\mathcal{T}_i$ is denoted as $\mathcal{T}_i.S$. We define the community of trace temporal structures $\mathcal{C} = \langle S, \triangleright, S_1, S_f, L\rangle$, derived from a collection of trace temporal structures $\{\mathcal{T}_1, \ldots, \mathcal{T}_m\}$, as follows:

$$\mathcal{C}.S = \bigcup_{i=1}^{m} \mathcal{T}_i.S \qquad \mathcal{C}.S_f = \bigcup_{i=1}^{m} \{\mathcal{T}_i.s_f\} \qquad \mathcal{C}.\triangleright = \bigcup_{i=1}^{m} \mathcal{T}_i.\triangleright$$
$$\mathcal{C}.S_1 = \bigcup_{i=1}^{m} \{\mathcal{T}_i.s_1\} \qquad \mathcal{C}.L = \bigcup_{i=1}^{m} \mathcal{T}_i.L$$

We denote the universe of possible communities of trace temporal structures over $\mathcal{A}$ as $\mathcal{A}^{\mathcal{C}}$. Based on the concept of community $\mathcal{C}$ that jointly represents single traces $\mathcal{T}_i, \ldots, \mathcal{T}_m \in \mathcal{M}(L)$, we can define the $\mathcal{R}$-mapping. The $\mathcal{R}$-mapping of an element of $\mathcal{C}$ to an element of $\mathcal{L}$ is denoted by the infix notation $\overset{\mathcal{R}}{\longmapsto}$.

Definition 6 ($\mathcal{R}$-mapping of a community of trace temporal structures to a log temporal stucture). *Let $\mathcal{C} = \langle S, \triangleright, S_1, S_f, L\rangle \in \mathcal{A}^{\mathcal{C}}$ be a community of trace temporal structures and $\mathcal{L} = \langle S, \triangleright, S_1, S_f, L, W\rangle \in \mathcal{A}^{\mathcal{L}}$ be a log temporal structure, defined over the common alphabet $\mathcal{A}$.*
$\mathcal{R} : \mathcal{A}^{\mathcal{C}} \to \mathcal{A}^{\mathcal{L}}$ is such that:

- *given a state $s_{\mathcal{C}} \in \mathcal{C}.S$, and a state $s_{\mathcal{L}} \in \mathcal{L}.S$,*
  - *$s_{\mathcal{C}} \overset{\mathcal{R}}{\longmapsto} s_{\mathcal{L}}$ iff*
    - *$\mathcal{C}.L(s_{\mathcal{C}}) = \mathcal{L}.L(s_{\mathcal{L}})$, i.e., the two states are interpreted in the same way over alphabet $\mathcal{A}$, and*
    - *either $s_{\mathcal{C}} \in \mathcal{C}.S_f$ ($s_{\mathcal{C}}$ is a final state of $\mathcal{C}$) or $(s_{\mathcal{C}}\triangleright) \overset{\mathcal{R}}{\longmapsto} (s_{\mathcal{L}}\triangleright)$, i.e., the successor state of $s_{\mathcal{C}}$ maps to the successor state of $s_{\mathcal{L}}$;*
  - *$W(s_{\mathcal{L}}) = \left|\left\{s_{\mathcal{C}} \in \mathcal{C}.S | s_{\mathcal{C}} \overset{\mathcal{R}}{\longmapsto} s_{\mathcal{L}}\right\}\right|$, i.e., the weight of a state $s_{\mathcal{L}}$ in the log temporal structure $\mathcal{L}$ is equal to the number of states in $\mathcal{C}$ that map to $s_{\mathcal{L}}$;*
- *given states $s_{\mathcal{L}}, s'_{\mathcal{L}} \in \mathcal{L}.S$ and $s_{\mathcal{C}}, s'_{\mathcal{C}} \in \mathcal{C}.S$, $(s_{\mathcal{L}}, s'_{\mathcal{L}}) \in \mathcal{L}.\triangleright$ iff $(s_{\mathcal{C}}, s'_{\mathcal{C}}) \in \mathcal{C}.\triangleright$;*
- *$s_{\mathcal{L}_f} \in \mathcal{L}.S_f$ iff $s_{\mathcal{C}_f} \in \mathcal{C}.S_f$ and $s_{\mathcal{C}_f} \overset{\mathcal{R}}{\longmapsto} s_{\mathcal{L}_f}$, i.e., final states in $\mathcal{C}$ correspond to final states in $\mathcal{L}$;*
- *$s_{\mathcal{L}_1} \in \mathcal{L}.S_1$ iff $s_{\mathcal{C}_1} \in \mathcal{C}.S_1$ and $s_{\mathcal{C}_1} \overset{\mathcal{R}}{\longmapsto} s_{\mathcal{L}_1}$, i.e., initial states in $\mathcal{C}$ correspond to initial states in $\mathcal{L}$.*

$\mathcal{R}$ establishes a biunivocal correspondence for every element of $\mathcal{C}$ and $\mathcal{L}$, except states. The technique that we adopt to obtain the $\mathcal{R}$-mapping log temporal structure $\mathcal{L}$ from a

collection of trace temporal structures $\mathcal{M}(L)$, represented as a community $\mathcal{C}$, is depicted in Fig. 3. All final states that are associated by the labeling function to the same literal in $\mathcal{A}$ are joint in one single final state. It is associated with the same shared literal and has a weight amounting to the sum of joined states. The state successor function is modified by linking the predecessors to the new joint state. Those operations are repeated step by step along the predecessors of the final states in the trace temporal structures, up to the initial ones.

The topology of the resulting log temporal structure is always an *inverted polytree*. We name it inverted to highlight that every subtree of the polytree has one root and multiple leaves only if we consider the *inverse* transition relation $\triangleleft$ for the direction of arcs (see Fig. 3e).

We conclude this section with a theorem, allowing us to solve the problem of LTL$_f$ query checking over event logs by analyzing the corresponding log temporal structure. We omit its proof due to space reasons.

Theorem 1. *If a community of trace temporal structures $\mathcal{C}$ $\mathcal{R}$-maps to a log temporal structure $\mathcal{T}$, $\mathcal{C}$ and $\mathcal{T}$ are bisimilar.*

Temporal logics are bisimulation-invariant [11,8]. As a consequence, we can evaluate LTL$_f$ formulas on the log temporal structure in order to show whether the corresponding business rules were satisfied or not in the log.

Corollary 1. *Given a log L, a community of trace temporal structures $\mathcal{C}$ derived from $\mathcal{M}(L)$, and a log temporal structure $\mathcal{L}$ s.t. $\mathcal{C}$ $\mathcal{R}$-maps to it, the weight of an initial state s_1 of $\mathcal{L}$ is equal to the number of equivalent traces $\boldsymbol{t}, \boldsymbol{t}', \ldots, \boldsymbol{t}^m \in L$ s.t. $\boldsymbol{t}(1) = \boldsymbol{t}'(1) = \ldots = \boldsymbol{t}^m(1) = L(s_1)$.*

For space reasons, we omit the proof here. However, it can be verified by considering the construction of the log temporal structure. An example is given in Fig. 3, where Fig. 3a and Fig. 3f highlight the states satisfying the same formula.

3.2 Query Checking Algorithm

Algorithm 1 describes the main procedure of our approach. The *check* procedure takes as input the log temporal structure $\mathcal{L}$ (obtained from a log through the steps

Algorithm 1. $check(\mathcal{L}, \Phi(?_{x_1}\{\mathcal{A}_{x_1}\}, \ldots, ?_{x_n}\{\mathcal{A}_{x_n}\}), \mathcal{A})$, checking LTL$_f$ query $\Phi(?_{x_1}\{\mathcal{A}_{x_1}\}, \ldots, ?_{x_n}\{\mathcal{A}_{x_n}\})$ on temporal structure $\mathcal{L}$, defined over alphabet $\mathcal{A}$

Input: $\mathcal{L} = \langle S, \triangleright, S_1, S_f, L, W \rangle$, LTL$_f$ query $\Phi(?_{x_1}\{\mathcal{A}_{x_1}\}, \ldots, ?_{x_n}\{\mathcal{A}_{x_n}\})$ with placeholders $?_{x_1} \ldots ?_{x_n}$ resp. bounded to literals $\mathcal{A}_{x_1} \ \ldots, \mathcal{A}_{x_n}$, subsets of alphabet $\mathcal{A}$

Output: $\mathcal{B}_{\Phi}^{\hat{S}} = \left\{ \left\langle \hat{S}_{\Phi}, \Phi \right\rangle \right\}^*$, set of initial states of witnesses ($\hat{S}_{\Phi}$) for each LTL$_f$ formula Φ, obtained through the replacement of placeholders with the bounded literals.

1 $\mathcal{B}_{\Phi}^{\hat{S}} : \emptyset$

2 **foreach** $\{a_1, \ldots, a_n\} \in \left(\mathcal{A}_{x_1} \times \cdots \times \mathcal{A}_{x_n}\right)$ **do**

3 $\quad \Phi : \Phi(a_1, \ldots, a_n)$

4 $\quad \mathcal{B}_{\Phi}^{\hat{S}} : \mathcal{B}_{\Phi}^{\hat{S}} \cup \left\langle evalForm\left(\mathcal{L}, \Phi, S, \mathcal{A}\right), \Phi \right\rangle$

5 **return** $\mathcal{B}_{\Phi}^{\hat{S}}$

(a) $\Box(a \rightarrow \Diamond b)$ **(b)** $(\neg b\, U a) \vee \Box\neg b$

Fig. 2. Evaluation trees

described in Section 3.1) and the query expressing the business rule template, denoted as $\Phi(?_{x_1}\{\mathcal{A}_{x_1}\}, \ldots, ?_{x_n}\{\mathcal{A}_{x_n}\})$. To each placeholder $?_{x_i}$ a set of symbols $\mathcal{A}_{x_i}$ is associated, restricting the possible assignments of placeholders. The sets of symbols are contained in the log alphabet $\mathcal{A}$. The output is a set of tuples $\left\langle \hat{S}_\Phi, \Phi \right\rangle$, where $\hat{S}_\Phi$ is the set of states in $\mathcal{L}$ satisfying formula Φ. By Φ we indicate the formula stemming from $\Phi(?_{x_1}\{\mathcal{A}_{x_1}\}, \ldots, ?_{x_n}\{\mathcal{A}_{x_n}\})$ where every placeholder $?_{x_i}$ is assigned to a symbol in $\mathcal{A}_{x_i}$. Take, for example, the temporal structure $\mathcal{L}$ associated to log $L = [\langle a, b, d, c\rangle, \langle a, b, a, c\rangle]$ and the query $\Box(?_x\{a, c\} \rightarrow \Diamond(?_y\{b, d\}))$. Starting from this query, we need to evaluate the LTL$_f$ rules in set $F = \{\Box(a \rightarrow \Diamond b), \Box(a \rightarrow \Diamond d), \Box(c \rightarrow \Diamond b), \Box(c \rightarrow \Diamond d)\}$ over $\mathcal{L}$.

In the first step of our approach, we generate a set of LTL$_f$ formulas by replacing the placeholders in the input query with all the possible combination of events specified with the query (possibly all the events in the input log). Each LTL$_f$ formula is then passed to *evalForm* procedure, together with $\mathcal{L}$, the set of states of $\mathcal{L}$, namely S, and the alphabet $\mathcal{A}$.

The first call to procedure *evalForm* starts a sequence of recursive calls, aiming at decomposing query Φ into subformulas, thus building an evaluation tree. This tree is meant to unfold the (possibly nested) subformulas up to atomic propositions (the leaves). All parent nodes are therefore LTL$_f$ or propositional operators. The number of child nodes depend on the arity of the operator (e.g., 1 for $\neg$ and X, 2 for $\wedge$ and U). For example, from $\Box(a \rightarrow \Diamond b)$ (every occurrence of a is followed by at least one occurrence of b) and from $(\neg b U a) \vee \Box\neg b$ (every occurrence of b is preceded by at least one occurrence of a) the evaluation trees in Fig. 2 are generated. Once the recursive call returns the set of states in which the subformula holds true, these states are treated by the calling procedure, back to the root of the evaluation tree.

For evaluating an atomic proposition l (i.e., a leaf in the evaluation tree) in a set of states $\hat{S}$, we use the procedure *evalAtom* (Algorithm 2). In particular, if l is $\top$, the procedure returns all the states of $\hat{S}$. If l is $\bot$, the empty set is returned. If l is an atomic proposition corresponding to the occurrence of an event, the procedure returns all the states of $\hat{S}$ in which such event occurs.

For evaluating an LTL$_f$ formula Φ in a set of states $\hat{S}$, we use the procedure *evalForm* (Algorithm 3). If Φ is the negation of a child expression φ, a recursive invocation of *evalForm* is used to retrieve all the states in $\hat{S}$ in which φ holds. Then, the procedure returns the complement of $\hat{S}$ with respect to this set. If Φ is the disjunction of two expressions φ and ψ, the procedure *evalForm* identifies the sets containing all

the states in $\hat{S}$ in which φ and ψ hold respectively. The procedure returns the union of these two sets. Similarly, if Φ is the conjunction of φ and ψ, the sets containing all the states in $\hat{S}$ in which φ and ψ hold are built and the procedure returns the intersection of these sets. The implication and the equivalence of two expressions are determined as combination of negation, disjunction and conjunction considering that $\varphi \rightarrow \psi$ is equivalent to $\neg(\varphi) \vee \psi$ and that $\varphi \leftrightarrow \psi$ is equivalent to $(\neg(\varphi) \vee \psi) \wedge (\neg(\psi) \vee \varphi)$.

If Φ consists of the *next* operator applied to a child expression φ, a recursive invocation of $evalForm$ is used to retrieve all the states in $\hat{S}_{\triangleright}$ (the set of the successors of $\hat{S}$) in which φ holds. The output set $\hat{S}_{\circ}$ is the set of all the predecessors of these states. If Φ is the *future* operator applied to a child expression φ, a recursive invocation of $evalForm$ is used to retrieve all the states in $\hat{S}$ in which φ holds. The output set $\hat{S}_{\diamond}$ is initialized with these states. Iteratively all the predecessors of the states in $\hat{S}_{\diamond}$ are added to $\hat{S}_{\diamond}$ up to the initial states. If Φ is the *globally* operator applied to a child expression φ, the algorithm identifies the final states in which φ is satisfied. The output set $\hat{S}_{\Box}$ is initialized with these states. Iteratively all the predecessors of the states in $\hat{S}_{\Box}$ in which φ holds are added to $\hat{S}_{\Box}$ up to the initial states. Finally, if Φ is the *until* operator applied to two expressions φ and ψ, the procedure identifies the states in $\hat{S}$ in which ψ is satisfied. The output set $\hat{S}_U$ is initialized with these states. Iteratively all the predecessors of the states in $\hat{S}_U$ in which φ holds are added to $\hat{S}_U$ up to the initial states.

Consider, for example, the evaluation tree in Fig. 2a and the log temporal structure $\mathcal{L}$ stemmed from $L = [\langle a, b, d, c\rangle, \langle a, b, a, c\rangle]$ (Fig. 1c). To check if formula $\Box(a \rightarrow \Diamond b)$ holds in L, the algorithms just described are invoked recursively from the root to the leaves of the evaluation tree of $\Box(a \rightarrow \Diamond b)$. The evaluation of the atomic proposition corresponding to the occurrence of a produces as result $\hat{S}_a = \{s_{1,1}, s_{2,1}, s_{2,3}\}$. The same algorithm applied to the node of the tree corresponding to activity b produces as result $\hat{S}_b = \{s_{1,2}, s_{2,2}\}$. Then, starting from $\hat{S}_b$, procedure $evalForm$ produces $\hat{S}_{\Diamond b} = \{s_{1,1}, s_{1,2}, s_{2,1}, s_{2,2}\}$. Starting from $\hat{S}_a$ and $\hat{S}_{\Diamond b}$, $evalForm$ produces $\hat{S}_{a \rightarrow \Diamond b} = \{s_{1,1}, s_{1,2}, s_{1,3}, s_f, s_{2,1}, s_{2,2}\}$. The final outcome of the recursive invocation of $evalForm$ is $\hat{S}_{\Box(a \rightarrow \Diamond b)} = \{s_{1,1}, s_{1,2}, s_{1,3}, s_f\}$. Since $S_{\Box(a \rightarrow \Diamond b)}$ contains the initial state $s_{1,1}$, mapped to the first element of $\mathbf{t}_1$, $\mathbf{t}_1(1)$, we can conclude that $\Box(a \rightarrow \Diamond b)$ holds in $\mathbf{t}_1$, i.e., $\mathbf{t}_1$ is a *witness* for $\Box(a \rightarrow \Diamond b)$. In the same way, $\Box(a \rightarrow \Diamond b)$ does not hold in $s_{2,1}$ (mapped to the first element of $\mathbf{t}_2$), i.e., $\mathbf{t}_2$ is a *counterexample* for $\Box(a \rightarrow \Diamond b)$.

Algorithm 2. $evalAtom(\mathcal{L}, l, \hat{S}, \mathcal{A})$, evaluating atomic propositions l in states $\hat{S}$ of temporal structure $\mathcal{T}$ defined over alphabet $\mathcal{A}$

Input: $\mathcal{L} = \langle S, \triangleright, S_1, S_f, L, W\rangle$, atomic proposition l, set of states $\hat{S} \subseteq S$, alphabet $\mathcal{A}$
Output: $\hat{S}_l \subseteq \hat{S}$ where l is evaluated as true

1 **if** $l = \top$ **then return** $\hat{S}$
2 **if** $l = \bot$ **then return** $\emptyset$
3 **if** $l = a$ with $a \in \mathcal{A}$ **then**
4 **return** $\left\{ s \in \hat{S} \mid L(s, a) \mapsto \top \right\}$

Algorithm 3. $evalForm(\mathcal{L}, \Phi, \hat{S}, \mathcal{A})$, evaluating LTL$_f$ formula Φ in states $\hat{S}$ of temporal structure $\mathcal{T}$ defined over alphabet $\mathcal{A}$

Input: Temporal Structure $\mathcal{L} = \langle S, \triangleright, S_1, S_f, L, W \rangle$, LTL$_f$ formula Φ, set of states $\hat{S} \subseteq S$, alphabet $\mathcal{A}$
Output: $\hat{S}_\Phi \subseteq \hat{S}$ where Φ holds true

if $\Phi \equiv \top$ *or* $\Phi \equiv \bot$ *or* $\Phi \equiv a$, with $a \in \mathcal{A}$ **then**
  return $evalAtom(\mathcal{T}, \Phi, \hat{S}, \mathcal{A})$
else if $\Phi \equiv \neg\varphi$, with φ LTL$_f$ formula over $\mathcal{A}$ **then**
  return $\hat{S} \setminus evalForm(\mathcal{T}, \varphi, \hat{S}, \mathcal{A})$
else if $\Phi \equiv \varphi \vee \psi$, with φ, ψ LTL$_f$ formulas over $\mathcal{A}$ **then**
  return $evalForm(\mathcal{T}, \varphi, \hat{S}, \mathcal{A}) \cup evalForm(\mathcal{T}, \psi, \hat{S}, \mathcal{A})$
else if $\Phi \equiv \varphi \wedge \psi$, with φ, ψ LTL$_f$ formulas over $\mathcal{A}$ **then**
  return $evalForm(\mathcal{T}, \varphi, \hat{S}, \mathcal{A}) \cap evalForm(\mathcal{T}, \psi, \hat{S}, \mathcal{A})$
else if $\Phi \equiv \varphi \rightarrow \psi$, with φ, ψ LTL$_f$ formulas over $\mathcal{A}$ **then**
  return $evalForm(\mathcal{T}, \neg\varphi \vee \psi, \hat{S}, \mathcal{A})$
else if $\Phi \equiv \varphi \leftrightarrow \psi$, with φ, ψ LTL$_f$ formulas over $\mathcal{A}$ **then**
  return $evalForm(\mathcal{T}, \varphi \rightarrow \psi, \hat{S}, \mathcal{A}) \cap evalForm(\mathcal{T}, \psi \rightarrow \varphi, \hat{S}, \mathcal{A})$
else if $\Phi \equiv \circ\varphi$, with φ LTL$_f$ formula over $\mathcal{A}$ **then**
  $\hat{S}_\circ^\triangleright : evalForm(\mathcal{T}, \varphi, \hat{S}\triangleright, \mathcal{A})$
  $\hat{S}_\circ : \triangleleft\hat{S}_\circ^\triangleright$
  return $\hat{S}_\circ$
else if $\Phi \equiv \Diamond\varphi$, with φ LTL$_f$ formula over $\mathcal{A}$ **then**
  $\hat{S}_\Diamond : evalForm(\mathcal{T}, \varphi, \hat{S}, \mathcal{A})$
  $\hat{S}_\Diamond^\triangleleft : \triangleleft\hat{S}_\Diamond$
  while $|\hat{S}_\Diamond^\triangleleft| > 0$ **do**
    $\hat{S}_\Diamond : \hat{S}_\Diamond \cup \hat{S}_\Diamond^\triangleleft$
    $\hat{S}_\Diamond^\triangleleft : \triangleleft\hat{S}_\Diamond^\triangleleft$
  return $\hat{S}_\Diamond$
else if $\Phi \equiv \Box\varphi$, with φ LTL$_f$ formula over $\mathcal{A}$ **then**
  $\hat{S}_\Box : evalForm(\mathcal{T}, \varphi, s_f, \mathcal{A})$
  $\hat{S}_\Box^\triangleleft : evalForm(\mathcal{T}, \varphi, \triangleleft\hat{S}_\Box, \mathcal{A})$
  while $|\hat{S}_\Box^\triangleleft| > 0$ **do**
    $\hat{S}_\Box : \hat{S}_\Box \cup \hat{S}_\Box^\triangleleft$
    $\hat{S}_\Box^\triangleleft : evalForm(\mathcal{T}, \varphi, \triangleleft\hat{S}_\Box^\triangleleft, \mathcal{A})$
  return $\hat{S}_\Box$
else if $\Phi \equiv \varphi U \psi$, with φ, ψ LTL$_f$ formulas over $\mathcal{A}$ **then**
  $\hat{S}_U : evalForm(\mathcal{T}, \psi, \hat{S}, \mathcal{A})$
  $\hat{S}_U^\triangleleft : evalForm(\mathcal{T}, \varphi, \triangleleft\hat{S}_U, \mathcal{A})$
  while $|\hat{S}_U^\triangleleft| > 0$ **do**
    $\hat{S}_U : \hat{S}_U \cup \hat{S}_U^\triangleleft$
    $\hat{S}_U^\triangleleft : evalForm(\mathcal{T}, \varphi, \triangleleft\hat{S}_U^\triangleleft, \mathcal{A})$
  return $\hat{S}_U$

Given $\left\langle \hat{S}_\Phi, \Phi \right\rangle \in \mathcal{B}_\Phi^{\hat{S}}$, where $\mathcal{B}_\Phi^{\hat{S}}$ is the result of the call of $eval(\mathcal{L}, \Phi(?_{x_1}\{\mathcal{A}_{x_1}\}, \ldots, ?_{x_n}\{\mathcal{A}_{x_n}\}), \mathcal{A})$, for a formula Φ, we compute:

- the number of witnesses, as $\sum_{\hat{s} \in \left(\hat{S}_\Phi \cap \mathcal{L}.S_i\right)} W(\hat{s})$, and
- the number of counterexamples, as $\sum_{\bar{s} \in \left(\mathcal{L}.S_i \setminus \hat{S}_\Phi\right)} W(\bar{s})$.

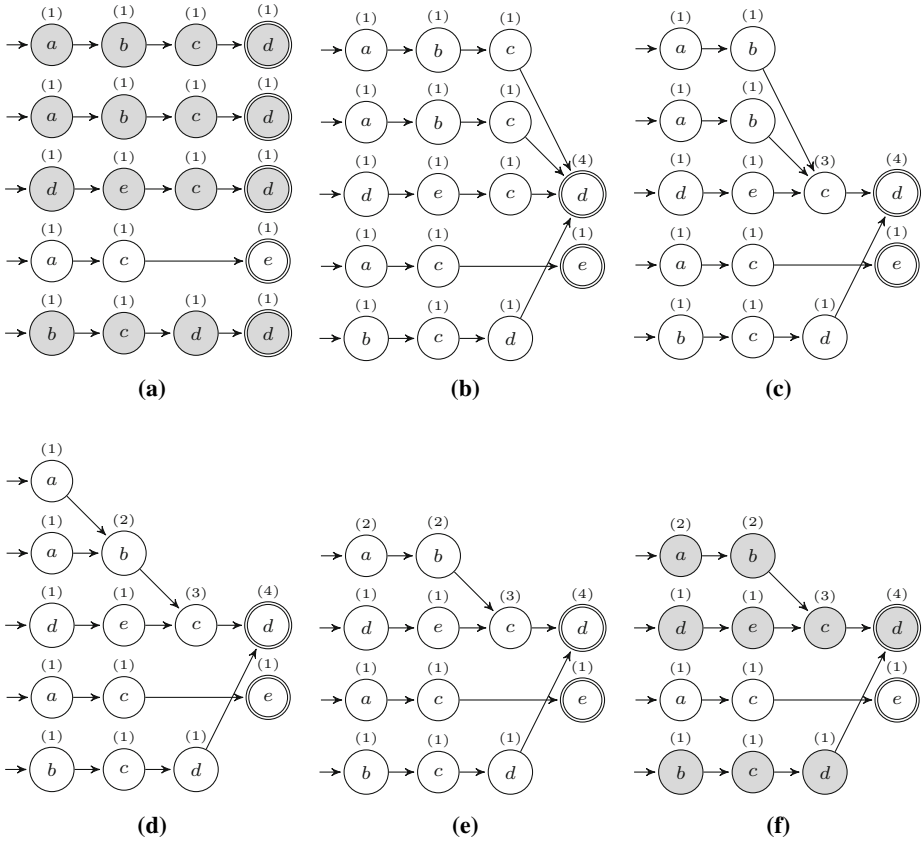


Fig. 3. Transformation into log temporal structure $\mathcal{L}$ of log $L = \{\langle a, b, c, d\rangle, \langle a, b, c, d\rangle, \langle d, e, c, d\rangle, \langle a, c, e\rangle, \langle b, c, d, d\rangle\}$, defined over log alphabet $\mathcal{A} = \{a, b, c, d, e\}$. Fig. 3a and Fig. 3f highlight the states satisfying $\square(a \rightarrow \Diamond b)$.

The number of witnesses corresponds to the sum of weights of initial states of $\mathcal{L}$ which are contained in the sets of states that satisfy Φ, $\hat{S}_\Phi$. Conversely, the number of counterexamples corresponds to the sum of weights of initial states of $\mathcal{L}$ which are not in $\hat{S}_\Phi$. Such calculation directly derives from Corollary 1.

4 Experimentation

We implemented the algorithms illustrated in the previous sections as a software encoded in `C`.[2] All tests have been executed on a machine running Windows 7 operating system on an Intel Core i7 CPU with 8GB of main memory.

[2] Source and executable files are available at `https://github.com/r2im/pickaxe`.

4.1 Benchmarks

In order to evaluate the performance of the algorithm, we have run a series of experiments aiming at showing evidence of the effect of input parameters on execution time. Logs have been artificially generated by the log generator module of MINERful [14,13], a declarative process mining tool. Traces in artificial logs were created by distributing symbols in the input log alphabet uniformly at random. The results of such tests, depicted in Fig. 4, show the trend of the computation time w.r.t. the change in the input parameters: size of the alphabet, number of traces, and length of traces. By default, parameters regarding the log and the log alphabet were initialized as follows: the log consisted of 100 traces ($|L| = 100$), each being a sequence of 10 events ($|\mathbf{t}| = 10$), drawn from a log alphabet comprising 10 activities ($|\mathcal{A}| = 10$). The queries taken into account were those corresponding to the class of 18 constraint templates defined by the declarative process modeling language Declare [9] (see Sections 1 and 5) listed in [22,14], including, e.g., $\Box(?_x \rightarrow \Diamond ?_y)$ and $(\neg ?_y \, U ?_x) \vee \Box \neg ?_y$. Placeholders have been set free to be assigned to any symbol in the log alphabet. The reported computation time is the sum of the computation times for each query. Fig. 4a, Fig. 4b and Fig. 4c show the trend of computation time when altering, resp.: *(i)* the size of the alphabet, from 5 to 50; *(ii)* the number of traces, from 400 to 4000; *(iii)* the length of traces, from 5 to 50. The first and the second graph show a linear trend, whereas the third one draws an exponential curve. The reason resides in the construction of the log temporal structure: its states tend to linearly increase with the length of traces. Therefore, the search space grows exponentially. The folding of traces into the polytree of the log temporal structure seems to limit the state space increase when the number of traces or the number of activities grow.

Table 1 summarizes the changes in the execution time when query parameters change, namely *(i)* the LTL_f operator (either temporal or propositional), and *(ii)* the number of placeholders. Results are depicted, respectively, in Table 2a and Table 2b. In both cases, the log is created using the following parameters: $|L| = 4000$, $|\mathbf{t}| = 10$, $|\mathcal{A}| = 10$. Remarkably, Table 2a shows that the possibility to visit the temporal structure only once per formula makes the evaluation of $\circ ?_x$ the fastest to be performed (see Algorithm 3). Conversely, $\Diamond ?_x$ turns out to be the most expensive in this regard. Table 2b highlights that the increase of placeholders entails an increase in the execution time approx. by an order of 10. This is due to the invocation of the $evalForm$ procedure for every combination of symbols in the alphabet, assigned to placeholders (see Algorithm 1).

4.2 Case Study

We have conducted a case study by using the BPI challenge 2011 [1] event log. This log pertains to a healthcare process and, in particular, contains the executions of a process related to the treatment of patients diagnosed with cancer in a large Dutch academic hospital. The whole event log contains $1,143$ cases and $150,291$ events distributed across 623 event classes (activities). Each case refers to the treatment of a different patient. The event log contains domain specific attributes that are both case attributes and event

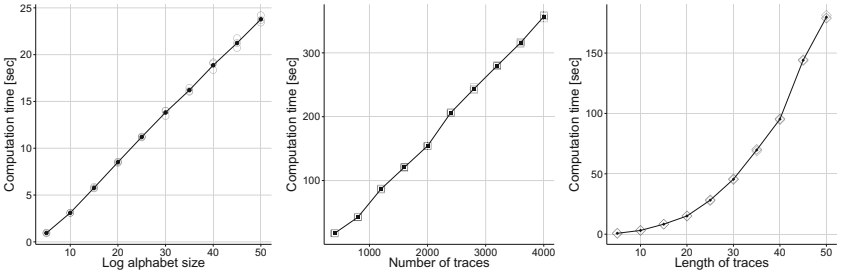
Query	Time [msec]	Query	Time [msec]
$?_x$	78	$\neg(?_x)$	109
$?_x \vee ?_y$	592	$?_x \wedge ?_y$	94
$?_x \rightarrow ?_y$	249	$?_x \leftrightarrow ?_y$	686
$\Diamond ?_x$	4 930	$\Box ?_x$	422
$\circ ?_x$	93	$?_x \, U \, ?_y$	1 357

(a) Varying query template

Query	Time [msec]
$\Box(?_x \rightarrow \Diamond(?_{y0}))$	37 628
$\Box(?_x \rightarrow \Diamond(?_{y0} \vee ?_{y1}))$	281 362
$\Box(?_x \rightarrow \Diamond(?_{y0} \vee ?_{y1} \vee ?_{y2}))$	2 027 832

(b) Varying number of placeholders

Table 1. Performance evaluation w.r.t. query properties



(a) Varying $|\mathcal{A}|$ **(b)** Varying $|L|$ **(c)** Varying $|\mathbf{t}|$

Fig. 4. Performance evaluation w.r.t. log properties and log alphabet.

attributes in addition to the standard XES attributes.[3] For example, *Age*, *Diagnosis*, and *Treatment code* are case attributes and *Activity code*, *Number of executions*, *Specialism code*, and *Group* are event attributes.

To investigate the behavior of the process as recorded in the log, we have used the following LTL$_f$ queries (in which the original Dutch language is not translated):

1. $\Box((\textit{aanname laboratoriumonderzoek}) \rightarrow \Diamond ?_x\{\textit{ca-19.9 tumormarker}, \textit{cea - tumormarker mbv meia}, \textit{ca-125 mbv meia}\})$;
2. $\diamond ?_x\{\textit{ureum}, \textit{albumine}, \textit{creatinine}\} \wedge \diamond ?_y\{\textit{calcium}, \textit{glucose}, \textit{natrium vlamfotometrisch}, \textit{kalium potentiometrisch}\}$;
3. $(\neg ?_x\{\textit{vervolgconsult poliklinisch}, \textit{1e consult poliklinisch}, \textit{telefonisch consult}\} \, U \, (\textit{aanname laboratoriumonderzoek})) \vee \Box(\neg ?_x\{\textit{vervolgconsult poliklinisch}, \textit{1e consult poliklinisch}, \textit{telefonisch consult}\})$;
4. $\neg(\diamond(\textit{aanname laboratoriumonderzoek}) \wedge \diamond ?_x\{\textit{ct abdomen}, \textit{ct thorax}, \textit{ct bovenbuik}, \textit{ct hersenen}, \textit{thorax}\})$.

In Table 2, we show the solutions of these queries. For each solution, we show the number of cases in which the solution holds (witnesses) and the number of

[3] XES (eXtensible Event Stream) is an XML-based standard for event logs proposed by the IEEE Task Force on Process Mining (`www.xes-standard.org`).

Ws	Cs	Solution
344	799	$\Box((aanname\ laboratoriumonderzoek) \rightarrow \Diamond(ca\text{-}19.9\ tumormarker))$
431	712	$\Box((aanname\ laboratoriumonderzoek) \rightarrow \Diamond(cea\ -\ tumormarker\ mbv\ meia))$
517	626	$\Box((aanname\ laboratoriumonderzoek) \rightarrow \Diamond(ca\text{-}125\ mbv\ meia))$

(a) Solutions for query $\Box((aanname\ laboratoriumonderzoek) \rightarrow \Diamond?_x\{ca\text{-}19.9\ tumormarker, cea\ -\ tumormarker\ mbv\ meia, ca\text{-}125\ mbv\ meia\})$

Ws	Cs	Solution	Ws	Cs	Solution
627	516	$\Diamond(ureum) \wedge \Diamond(calcium)$	627	516	$\Diamond(albumine) \wedge \Diamond(calcium)$
640	503	$\Diamond(creatinine) \wedge \Diamond(glucose)$	627	516	$\Diamond(ureum) \wedge \Diamond(glucose)$
623	520	$\Diamond(albumine) \wedge \Diamond(glucose)$	693	450	$\Diamond(creatinine) \wedge \Diamond(natrium\ vlamfotometrisch)$
663	480	$\Diamond(ureum) \wedge \Diamond(natrium\ vlamfotometrisch)$	640	503	$\Diamond(albumine) \wedge \Diamond(natrium\ vlamfotometrisch)$
697	446	$\Diamond(creatinine) \wedge \Diamond(kalium\ potentiometrisch)$	665	478	$\Diamond(ureum) \wedge \Diamond(kalium\ potentiometrisch)$
642	501	$\Diamond(albumine) \wedge \Diamond(kalium\ potentiometrisch)$	634	509	$\Diamond(creatinine) \wedge \Diamond(calcium)$

(b) Solutions for query $\Diamond?_x\{ureum, albumine, creatinine\} \wedge \Diamond?_y\{calcium, glucose, natrium\ vlamfotometrisch, kalium\ potentiometrisch\}$

Ws	Cs	Solution
797	346	$\neg(vervolgconsult\ poliklinisch)\ U\ (aanname\ laboratoriumonderzoek) \vee \Box(\neg(vervolgconsult\ poliklinisch))$
1046	97	$\neg(1e\ consult\ poliklinisch)\ U\ (aanname\ laboratoriumonderzoek) \vee \Box(\neg(1e\ consult\ poliklinisch))$
1041	102	$\neg(telefonisch\ consult)\ U\ (aanname\ laboratoriumonderzoek) \vee \Box(\neg(telefonisch\ consult))$

(c) Solutions for query $(\neg?_x\{vervolgconsult\ poliklinisch, 1e\ consult\ poliklinisch, telefonisch\ consult\}\ U\ (aanname\ laboratoriumonderzoek)) \vee \Box(\neg?_x\{vervolgconsult\ poliklinisch, 1e\ consult\ poliklinisch, telefonisch\ consult\})$

Ws	Cs	Solution
811	332	$\neg(\Diamond(aanname\ laboratoriumonderzoek) \wedge \Diamond(ct\ abdomen))$
1012	131	$\neg(\Diamond(aanname\ laboratoriumonderzoek) \wedge \Diamond(ct\ thorax))$
1127	16	$\neg(\Diamond(aanname\ laboratoriumonderzoek) \wedge \Diamond(ct\ bovenbuik))$
1122	21	$\neg(\Diamond(aanname\ laboratoriumonderzoek) \wedge \Diamond(ct\ hersenen))$
637	506	$\neg(\Diamond(aanname\ laboratoriumonderzoek) \wedge \Diamond(thorax))$

(d) Solutions for query $\neg(\Diamond(aanname\ laboratoriumonderzoek) \wedge \Diamond?_x\{ct\ abdomen, ct\ thorax, ct\ bovenbuik, ct\ hersenen, thorax\})$

Table 2. Query solutions for the proposed case study (in column headers,"Ws" stands for "Witnesses" and "Cs" for "Counterexamples")

cases in which the solution is not valid (counterexamples). For all the queries in Table 2a, there are more counterexamples than witnesses. For example, for $\Box((aanname\ laboratoriumonderzoek) \rightarrow \Diamond(ca\text{-}19.9\ tumormarker))$, there are 799 counterexamples and only 344 witnesses meaning that in 821 cases out of $1,143$ at least one occurrence of *aanname laboratoriumonderzoek* (*assumption laboratory*) is not eventually followed by *ca-19.9 tumormarker* in the same case. For queries in Ta-

bles 2b, 2c, and 2d the number of witnesses is higher than the number of counterexamples. The number of witnesses and the number of counterexamples for solutions in Table 2b are more balanced. For example, for $\Diamond(albumine) \wedge \Diamond(glucose)$, there are 623 witnesses and 520 counterexamples. This means that in 623 cases *albumine* and *glucose* coexist in the same case and in 520 cases they do not. For solutions in Tables 2c and 2d, there are more witnesses supporting the validity of the rules. From Table 2c, we can derive that very often a medical consultation is preceded by laboratory tests (*assumption laboratory*). For example, in $1,041$ cases out of $1,143$, *telefonisch consult* (*telephonic consultation*) is preceded by *aanname laboratoriumonderzoek* (*assumption laboratory*). In $1,046$ cases out of $1,143$, *1e consult poliklinisch* (*First outpatient visit*) is preceded by *aanname laboratoriumonderzoek*. From Table 2d, we can conclude that very rarely laboratory tests coexists with computed tomography (ct) tests. For example, for $\neg(\Diamond(aanname\ laboratoriumonderzoek) \wedge \Diamond(ct\ bovenbuik))$, there are $1,127$ witnesses and only 16 counterexamples meaning that in $1,127$ cases out of $1,143$ *aanname laboratoriumonderzoek* and *ct bovenbuik* (*ct upper abdomen*) do not coexist in the same case.

5 Related Work

Several approaches in the literature focus on the discovery of declarative process models [6,14,18,20,21,22,19]. In particular, the technique proposed in [12,14] is based on a two-step approach. First, the input event log is parsed to generate a knowledge base containing information useful to discover a Declare model. Then, in a second phase, the knowledge base is queried to find the set of Declare constraints that hold on the input log. The work proposed in [20] is based on an Apriori algorithm for association rule mining. In [19], the authors propose an algorithm for the online discovery of Declare rules. These works propose "ad hoc" algorithms for the discovery of a limited class of business rules. In contrast, our method can be used to discover any type of LTL_f-based constraint.

The approaches proposed in [6,18] are more general and allow for the specification of rules that go beyond the traditional Declare templates. However, these approaches can be hardly used in real-life settings since they are based on supervised learning techniques and, therefore, they require negative examples. In [21], a first-order variant of LTL is used to specify a set of data-aware patterns. Although this approach supports constraints involving data conditions, it can only be applied to discover the (limited) set of standard Declare rules.

Several approaches exist for temporal logic query checking. For example, Chan propose an approach that can cope with single placeholders in queries only appearing once as positive literals [3]. To overcome these limitations, Bruns and Godefroid propose a theoretical approach based on Extended Alternating Automata (EEA) [2,17]. Gurfinkel et al. describe their query checking tool named TLQSolver, capable of dealing with multiple placeholders, both appearing as negated or positive, and occurring several times in the query [15]. It is also proved that query checking is an instance of multi-valued model checking [5], i.e., a generalization of a classical model checking problem, where the model remains such that both transition relations and atomic propositions are two-valued, but lattice values are allowed to appear as constants in temporal logic formulas.

Indeed, their solution is based on their existing multi-valued model checker, $\mathcal{X}$Check [4]. These approaches have been implemented to deal with CTL query checking. Our approach is inspired by these works, but adapts them efficiently to LTL_f, in order to become applicable also in a real-life context.

6 Conclusion

In this paper, we have introduced an approach in the middle between process discovery and conformance checking, based on temporal logic query checking. In particular, our proposed technique produces as outcome a set of LTL-based business rules as well as diagnostics in terms of witnesses (traces of the input log in which each rule is satisfied) and counterexamples (traces of the input log in which each rule is violated). There are several directions for future work. It is possible to optimize the checking algorithms for improving performances, by further exploiting the characteristics of the contexts in which our technique is applied, such as the finiteness of the traces in an event log. More sophisticated techniques can be used to choose the best replacements for placeholders in a query. For example, the Apriori algorithm proposed in [20] can be used to automatically choose the set of activities to be assigned as replacements based on the frequency of their co-occurrence in a case.

References

1. 3TU Data Center. BPI Challenge, Event Log (2011), doi:10.4121/uuid:d9769f3d-0ab0-4fb8-803b-0d1120ffcf54
2. Bruns, G., Godefroid, P.: Temporal logic query checking. In: LICS, pp. 409–417. IEEE Computer Society (2001)
3. Chan, W.: Temporal-logic queries. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 450–463. Springer, Heidelberg (2000)
4. Chechik, M., Devereux, B., Easterbrook, S.M., Gurfinkel, A.: Multi-valued symbolic model-checking. ACM Trans. Softw. Eng. Methodol. 12(4), 371–408 (2003)
5. Chechik, M., Easterbrook, S.M., Petrovykh, V.: Model-checking over multi-valued logics. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 72–98. Springer, Heidelberg (2001)
6. Chesani, F., Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Exploiting inductive logic programming techniques for declarative process mining. In: Jensen, K., van der Aalst, W.M.P. (eds.) ToPNoC II. LNCS, vol. 5460, pp. 278–295. Springer, Heidelberg (2009)
7. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8(2), 244–263 (1986)
8. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (2001)
9. De Giacomo, G., De Masellis, R., Montali, M.: Reasoning on LTL on finite traces: Insensitivity to infiniteness. In: AAAI (2014)
10. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI (2013)
11. Deutch, D., Milo, T.: A structural/temporal query language for business processes. J. Comput. Syst. Sci. 78(2), 583–609 (2012)

12. Di Ciccio, C., Mecella, M.: Mining constraints for artful processes. In: Abramowicz, W., Kriksciuniene, D., Sakalauskas, V. (eds.) BIS 2012. LNBIP, vol. 117, pp. 11–23. Springer, Heidelberg (2012)
13. Di Ciccio, C., Mecella, M.: Studies on the discovery of declarative control flows from error-prone data. In: SIMPDA, pp. 31–45 (2013)
14. Di Ciccio, C., Mecella, M.: A two-step fast algorithm for the automated discovery of declarative workflows. In: CIDM, pp. 135–142. IEEE (2013)
15. Gurfinkel, A., Chechik, M., Devereux, B.: Temporal logic query checking: A tool for model exploration. IEEE TSE 29(10), 898–914 (2003)
16. Hildebrandt, T.T., Mukkamala, R.R., Slaats, T., Zanitti, F.: Contracts for cross-organizational workflows as timed dynamic condition response graphs. J. Log. Algebr. Program. 82(5-7), 164–185 (2013)
17. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. J. ACM 47(2), 312–360 (2000)
18. Lamma, E., Mello, P., Riguzzi, F., Storari, S.: Applying inductive logic programming to process mining. In: Blockeel, H., Ramon, J., Shavlik, J., Tadepalli, P. (eds.) ILP 2007. LNCS (LNAI), vol. 4894, pp. 132–146. Springer, Heidelberg (2008)
19. Maggi, F.M., Burattin, A., Cimitile, M., Sperduti, A.: Online process discovery to detect concept drifts in LTL-based declarative process models. In: Meersman, R., Panetto, H., Dillon, T., Eder, J., Bellahsene, Z., Ritter, N., De Leenheer, P., Dou, D. (eds.) ODBASE 2013. LNCS, vol. 8185, pp. 94–111. Springer, Heidelberg (2013)
20. Maggi, F.M., Bose, R.P.J.C., van der Aalst, W.M.P.: Efficient discovery of understandable declarative process models from event logs. In: Ralyté, J., Franch, X., Brinkkemper, S., Wrycza, S. (eds.) CAiSE 2012. LNCS, vol. 7328, pp. 270–285. Springer, Heidelberg (2012)
21. Maggi, F.M., Dumas, M., García-Bañuelos, L., Montali, M.: Discovering data-aware declarative process models from event logs. In: Daniel, F., Wang, J., Weber, B. (eds.) BPM 2013. LNCS, vol. 8094, pp. 81–96. Springer, Heidelberg (2013)
22. Maggi, F.M., Mooij, A.J., van der Aalst, W.M.P.: User-guided discovery of declarative process models. In: CIDM, pp. 192–199. IEEE (2011)
23. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: Declare: Full support for loosely-structured processes. In: EDOC, pp. 287–300. IEEE (2007)
24. Pnueli, A.: The temporal logic of programs. In: FSTTCS, pp. 46–57. IEEE (1977)
25. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M.T., van der Aalst, W.M.P.: The ProM framework: A new era in process mining tool support. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 444–454. Springer, Heidelberg (2005)
26. Weidlich, M., Mendling, J., Weske, M.: Efficient consistency measurement based on behavioral profiles of process models. IEEE Trans. Software Eng. 37(3), 410–429 (2011)