Semantical Vacuity Detection in Declarative Process Mining

Fabrizio Maria Maggi[1], Marco Montali[2], Claudio Di Ciccio[3], and Jan Mendling[3]

[1] University of Tartu, Estonia.
f.m.maggi@ut.ee
[2] Free University of Bozen-Bolzano, Italy.
montali@inf.unibz.it
[3] Vienna University of Economics and Business, Austria.
claudio.di.ciccio@wu.ac.at; jan.mendling@wu.ac.at

Abstract. A large share of the literature on process mining based on declarative process modeling languages, like DECLARE, relies on the notion of *constraint activation* to distinguish between the case in which a process execution recorded in event data "vacuously" satisfies a constraint, or satisfies the constraint in an "interesting way". This fine-grained indicator is then used to decide whether a candidate constraint supported by the analyzed event log is indeed relevant or not. Unfortunately, this notion of relevance has never been formally defined, and all the proposals existing in the literature use ad-hoc definitions that are only applicable to a pre-defined set of constraint patterns. This makes existing declarative process mining technique inapplicable when the target constraint language is extensible and may contain formulae that go beyond pre-defined patterns. In this paper, we tackle this hot, open challenge and show how the notion of constraint activation and vacuous satisfaction can be captured semantically, in the case of constraints expressed in arbitrary temporal logics over finite traces. We then extend the standard automata-based approach so as to incorporate relevance-related information. We finally report on an implementation and experimentation of the approach that confirms the advantages and feasibility of our solution.

Keywords: Vacuity Detection, Declarative Process Mining, Constraint Activation

1 Introduction

The increasing availability of event data recorded by information systems, electronic devices, web services, and sensor networks provides detailed information about the actual processes in systems and organizations. Process mining techniques can use such event data to discover process models and check the conformance of process executions. When a process works in a flexible and knowledge-intensive setting, it is desirable to describe it in terms of a constraint-based declarative process model rather than of a detailed procedural model. Consider, for instance, a physician in a hospital confronted with a variety of patients that need to be handled in a flexible manner but, at the same time, following some general regulations and guidelines. In such cases, declarative process models are more effective than procedural models [28, 27, 1]. Instead of explicitly

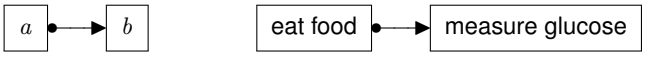


Fig. 1. Response template and a possible instantiation

specifying all possible sequences of tasks in a process, declarative models implicitly specify the allowed behavior of the process with *constraints*, i.e., rules that must be followed during the process executions.

In [1, 26], the authors introduce a declarative process modeling language called DECLARE. DECLARE is characterized by a user-friendly graphical representation with formal semantics grounded in LTL over finite traces (LTL_f [9, 8]). For example, the response constraint in Fig. 1 means that every action eat food must eventually be followed by action measure glucose, and this can be formalized with the LTL_f formula $\Box(\text{eat food} \rightarrow \Diamond\text{measure glucose})$. DECLARE has been fruitfully applied in the context of process discovery [20, 22, 18, 5, 13, 14] and compliance/conformance checking [6, 2, 16, 25, 23, 4, 19]. In both tasks, when execution traces are analyzed so as to check whether they satisfy a given DECLARE constraint, it is not sufficient to obtain a yes/no answer, but it becomes crucial to understand whether the trace is *relevant*, in the sense that it *actively interacts* with the constraint.

For this reason, most of the existing process mining techniques based on DECLARE rely on the notion of *constraint activation*. This notion is, in general, useful to assess the "degree of adherence" of a process execution with respect to a constraint. In particular, in process discovery, it becomes crucial to define interestingness metrics (like support and confidence [20, 13]) to select the most relevant constraints to be discovered among a set of candidates that can be, in some cases, extremely large. In the context of conformance checking, constraint activations are useful to define a set of "health indicators" to measure the *healthiness* of a process execution by evaluating the proportion of constraint activations that lead to a violation and the proportion of constraint activations that lead to a fulfillment [21, 4, 19, 25]. This quantitative analysis would not be possible without the notion of constraint activation.

In spite of the huge interest in this problem, the notion of constraint activation has never been formally defined, and all the papers so far have worked with ad-hoc definitions, explicitly spelled out for each of the DECLARE constraints. This poses a twofold issue. On the one hand, the lack of a general, formal approach to this problem makes it extremely difficult to understand whether ad-hoc approaches are indeed correct, when the constraints under study go beyond simple patterns like the aforementioned *response*. On the other hand, ad-hoc and pattern-based definitions of relevance make existing declarative process mining technique inapplicable when the target constraint language is extensible and may contain formulae that go beyond pre-defined patterns. The goal of this paper is to overcome these issues, by proposing for the first time a general, systematic characterization of relevance and activation for temporal constraints. Our approach is formally-grounded, and at the same time it is by and large compatible with the human intuition exploited in the previous literature.

The notion of constraint activation is related to the notion of *vacuity detection* in model checking. In the *response* example of Fig. 1, if eat food never occurs in a trace, then the constraint is "vacuously" satisfied, that is, satisfied without showing any form of interaction with the trace. However, existing techniques for vacuity detection (i.e., for

determining whether a given trace is a relevant, *interesting witness* for the formula of interest) [3, 17] present two key limitations in our context. First, they focus on temporal formulae over infinite traces (standard LTL in particular), whereas we are interested in finite traces only (as customary in BPM). Second, they suffer from *syntax sensitivity*. This implies that expressing a constraint through two semantically equivalent but syntactically different formulae could lead to a different judgement for the same trace.

By leveraging a finite-trace semantics for constraints, we move from a syntax-dependent to a fully semantical characterization of constraint activation and, in turn, vacuity detection. Our approach is grounded on the RV-LTL 4-valued semantics [2], which is adapted to the finite-trace setting in accordance with existing literature [21, 7]. RV-LTL is exploited to provide a fine-grained characterization of the "constraint activation state" in a given execution context. The relationship between activation states is then explored to identify when the execution of a given task results in a "relevant" transition. We abstractly formulate this theory of relevance at the logical level, but then we concretize it by leveraging the automata-theoretic approach for temporal logics over finite-traces. In particular, we show how the finite-state automaton characterizing the constraint of interest can be enriched with activation-related information without affecting the complexity of its construction. We finally report on implementation and experimentation of our solution, confirming its advantages and feasibility.

The paper is structured as follows. In Section 2, we introduce some preliminary notions. In Section 3, we describe the motivation behind our contribution. In Section 4, we give the definition of constraint activation. Section 5 shows how to check constraint activations using automata. In Section 7, we evaluate our approach on two real-life logs. Section 7 concludes the paper and spells out directions for future work.

2 Preliminaries

We start by introducing the necessary preliminary notions used in the rest of the paper. We fix a finite set Σ of tasks, i.e., atomic units of work in the process. This set provides the alphabet on top of which process execution traces are defined.

Def. 1 (Execution trace). *An* (execution) trace *over Σ is a possibly empty, finite sequence of tasks $\langle \mathsf{t}_1, \ldots, \mathsf{t}_n \rangle$ belonging to the set Σ^* of finite sequences over Σ. We use ε to denote the empty trace.*

We use the standard *concatenation* operator over traces: given two traces $\tau_1 = \langle \mathsf{t}^1_1, \ldots, \mathsf{t}^1_m \rangle$ and $\tau_2 = \langle \mathsf{t}^2_1, \ldots, \mathsf{t}^2_n \rangle$, we have that a trace τ_3 is the *concatenation* of τ_1 and τ_2, written $\tau_1 \cdot \tau_2$, if $\tau_3 = \langle \mathsf{t}^1_1, \ldots, \mathsf{t}^1_m, \mathsf{t}^2_1, \ldots, \mathsf{t}^2_n \rangle$. We also use notation $\tau \cdot \mathsf{t}$ as a shortcut for $\tau \cdot \langle \mathsf{t} \rangle$.

Intuitively, constraints are used to declaratively describe which traces are considered *compliant*, and which instead are forbidden. Typically, this intuitive notion of conformance is formally expressed using the notion of logical consequence over temporal logics, whose models are indeed traces [24]. The most widely used logic for declarative process modeling is LTL over finite traces (LTL_f). This logic is at the basis of concrete constraint modeling languages such as DECLARE. As pointed out in [7], the most widely adopted approach to reason about and execute declarative process models is to leverage the automata-theoretic approach for temporal logics, exploiting the

well-known result that every LTL_f formula can be captured by a corresponding (deterministic) finite-state automaton (FSA). However, FSAs are actually richer than LTL_f, and in fact capture sophisticated constraints expressed in monadic second-order logic over finite traces (MSO_f), a logic that is expressively equivalent to regular expressions, and also to linear-dynamic logic over finite traces (LDL_f). Interestingly, LDL_f has been recently applied to monitor business constraints going beyond the typical DECLARE patterns [7]. To abstract away from the specific logic of interest, in this paper we employ the generic term *(business) constraint* as a way to refer to a (closed) formula in any of the logics mentioned above. We use LTL_f in our examples just for presentation purposes. As pointed out above, all such logics can be characterized using DFAs. We call *constraint automaton* the DFA corresponding to a constraint of interest.

Def. 2 (Constraint Automaton). *Let φ be a constraint over Σ. The* constraint automaton *$\mathcal{A}_\varphi$ of φ is a* DFA *$\langle \Sigma, S, s_0, \delta, F \rangle$, where: (i) Σ is the input alphabet (which corresponds to the set of tasks); (ii) S is a finite set of states; (iii) $s_0 \in S$ is the initial state; (iv) $\delta : S \times \Sigma \to S$ is the (task-labeled) state-transition function; (v) $F \subseteq S$ is the set of accepting states. $\mathcal{A}_\varphi$ has the property of precisely accepting those traces $\sigma \in \Sigma^*$ that satisfy φ. Without loss of generality we assume that $\mathcal{A}_\varphi$ is* not trimmed*, i.e., for every state $s \in S$ and every task $\mathsf{t} \in \Sigma$, $\delta(s, \mathsf{t})$ is defined.*

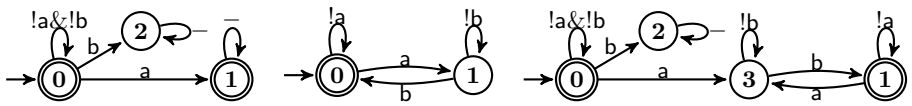
Examples of algorithms that produce the constraint automaton given a constraint expressed in LDL_f or LTL_f can be found in [15, 9, 7].

Given a constraint automaton $\mathcal{A} = \langle \Sigma, S, s_0, \delta, F \rangle$ and two states $s_1, s_2 \in S$, we say that s_2 *is reachable from* s_1 *in* $\mathcal{A}$, written $\delta^*(s_1, s_2)$, if $s_1 = s_2$ or there exists a trace that leads from s_1 to s_2 according to δ. We say that $\mathcal{A}$ *accepts a trace* τ, or equivalently that τ *complies with* $\mathcal{A}$, if there exists a path that reaches an accepting state starting from the initial state, such that for $i \in \{0, \ldots, |\tau|\}$, the i-th transition in the path matches with the i-th task in τ.

Fig. 2 shows the constraint automata representing the following LTL_f DECLARE templates, grounded on two tasks a and b:

- *Precedence* ($\varphi_p = \neg \mathsf{b}\, \mathcal{U}\, \mathsf{a}$) - each b must be preceded by a;
- *Response* ($\varphi_r = \Box(\mathsf{a} \to \Diamond \mathsf{b})$ - each a must be eventually followed by b;
- *Succession* ($\varphi_p \wedge \varphi_r$) - combination of precedence and response.

For compactness, in the figure, we graphically employ sophisticated labels as a shortcut for multiple transitions connecting two states with different task-labels. For example, a transition labeled with !a is a shortcut for a set of transitions between the same two states, each one labeled with a task taken from $\Sigma \setminus \mathsf{a}$. A transition labeled with "$-$" is a shortcut for a set of transitions between the same two states, one per task in Σ. Notably, this compact notation allows us to use the same automaton regardless of Σ (assuming just that Σ contains the tasks mentioned by the constraint, plus at least one additional, "other" task). Following Def. 2, in Fig. 2, we do not *trim* the automata, i.e., we explicitly maintain all states, even the trap states that cannot reach any accepting state (like state 2 in Fig. 2(a) and 2(c)). Our approach seamlessly works for trimmed automata as well.



(a) *precedence* (b) *response* (c) *succession*

Fig. 2. Automata for the *precedence*, *response* and *succession* DECLARE constraints

3 Background and Motivation

When checking whether a process execution complies with a constraint, one among two outcomes arises: the execution may *violate* the constraint, or it may *satisfy* it. In the latter case, however, the reason for satisfaction may be twofold. On the one hand, it could be the case that the trace interacts with the constraint, ensuring that the constraint is satisfied at the time the trace is completed. On the other hand, it could be the case that the constraint is trivially satisfied because there is no interaction with the trace. Consider again the *response* constraint in Fig. 1. This constraint is satisfied when food is eaten and then the glucose is eventually measured; this is in fact an interesting situation. However, this constraint is also satisfied by those traces where no food is ever eaten. In this latter case, we say that the constraint is *vacuously satisfied*. Traces where a constraint is *non*-vacuously satisfied are called *interesting witnesses* for that constraint. As pointed out in the introduction, discriminating between these two situations is crucial in a variety of (declarative) process mining tasks, such as conformance checking and declarative process discovery. In this section, we deepen the discussion provided in the introduction, by considering the two main limitations of existing approaches when it comes to interesting witnesses: syntax-dependence and ad-hoc definitions.

3.1 Syntax-Dependent Vacuity Detection

In [17], the authors introduce an approach for vacuity detection in temporal model checking for LTL (over infinite traces), so as to determine whether a given trace is an interesting witness for an LTL formula; they provide a method for extending an LTL formula φ to a new formula $witness(\varphi)$ that, when satisfied, ensures that the original formula φ is non-vacuously satisfied. In particular, $witness(\varphi)$ is generated by considering that a path π satisfies φ non-vacuously (and then is an interesting witness for φ), if π satisfies φ and π satisfies a set of additional conditions that guarantee that every subformula of φ does really affect the truth value of φ in π. We call these conditions *vacuity detection conditions* of φ. They correspond to the formulae $\neg\varphi[\psi \leftarrow \bot]$ where, for all the subformulae ψ of φ, $\varphi[\psi \leftarrow \bot]$ is obtained from φ by replacing ψ by false or true, depending on whether ψ is in the scope of an even or an odd number of negations. Then, $witness(\varphi)$ is the conjunction of φ and all the formulae $\neg\varphi[\psi \leftarrow \bot]$ with ψ subformula of φ:

$$witness(\varphi) = \varphi \wedge \bigwedge \neg\varphi[\psi \leftarrow \bot]. \tag{1}$$

Consider, e.g., the response constraint $\Box(\mathsf{eat\ food} \rightarrow \Diamond\mathsf{measure\ glucose})$. The vacuity detection condition is $\Diamond\mathsf{eat\ food}$, so that the interesting witnesses for this constraint are all traces where $\Box(\mathsf{eat\ food} \rightarrow \Diamond\mathsf{measure\ glucose}) \wedge \Diamond\mathsf{eat\ food}$ is satisfied.

This approach was applied to DECLARE in [22] for vacuity detection in the context of process discovery. However, the algorithm introduced in [17] can generate different results for equivalent LTL$_f$ formulae. Consider, for instance, the following equivalent formulae (expressing a DECLARE *alternate response* constraint):

$$\varphi = \Box(\mathsf{a} \rightarrow \Diamond \mathsf{b}) \wedge \Box(\mathsf{a} \rightarrow \bigcirc((\neg \mathsf{a}\, \mathcal{U}\, \mathsf{b}) \vee \Box(\neg \mathsf{b})))$$
$$\varphi' = \Box(\mathsf{a} \rightarrow \bigcirc(\neg \mathsf{a}\, \mathcal{U}\, \mathsf{b}))$$

When we apply (1) to φ and φ', we obtain that $witness(\varphi) \neq witness(\varphi')$.

We focus on φ. Since $\varphi = \Box(\neg \mathsf{a} \vee \Diamond \mathsf{b}) \wedge \Box(\neg \mathsf{a} \vee \bigcirc((\neg \mathsf{a}\, \mathcal{U}\, \mathsf{b}) \vee \Box(\neg \mathsf{b})))$, one of the subformulae of φ is $\psi = \Box(\neg \mathsf{b})$. Since ψ is in the scope of an even number of negations, the corresponding vacuity detection condition is

$$\neg(\Box(\neg \mathsf{a} \vee \Diamond \mathsf{b}) \wedge \Box(\neg \mathsf{a} \vee \bigcirc((\neg \mathsf{a}\, \mathcal{U}\, \mathsf{b}) \vee \mathit{false}))) \equiv \neg(\Box(\neg \mathsf{a} \vee \Diamond \mathsf{b}) \vee \Diamond(\mathsf{a} \wedge \neg \bigcirc(\neg \mathsf{a}\, \mathcal{U}\, \mathsf{b}))$$

Considering that $\neg(\Box(\neg \mathsf{a} \vee \Diamond \mathsf{b})$ and $\Diamond(\mathsf{a} \wedge \neg \bigcirc(\neg \mathsf{a}\, \mathcal{U}\, \mathsf{b}))$ are always false in conjunction with φ, this vacuity detection condition is always false in conjunction with φ. This is sufficient to conclude that $witness(\varphi) = \mathit{false}$.

We now focus on φ'. Since $\varphi' = \Box(\neg \mathsf{a} \vee \bigcirc(\neg \mathsf{a}\, \mathcal{U}\, \mathsf{b}))$, its subformulae are

$$\psi'_1 = \varphi' \qquad \psi'_2 = \neg \mathsf{a} \vee \bigcirc(\neg \mathsf{a}\, \mathcal{U}\, \mathsf{b}) \qquad \psi'_3 = \mathsf{a}(1)$$
$$\psi'_4 = \bigcirc(\neg \mathsf{a}\, \mathcal{U}\, \mathsf{b}) \qquad \psi'_5 = \neg \mathsf{a}\, \mathcal{U}\, \mathsf{b} \qquad \psi'_6 = \mathsf{a}(2) \qquad \psi'_7 = \mathsf{b}.$$

The corresponding vacuity detection conditions are: *(i)* true for ψ'_1 and ψ'_2; *(ii)* $\neg(\Box(\bigcirc(\neg \mathsf{a}\, \mathcal{U}\, \mathsf{b}))) \equiv \Diamond(\neg \bigcirc(\neg \mathsf{a}\, \mathcal{U}\, \mathsf{b}))$ for ψ'_3; *(iii)* $\neg(\Box(\neg \mathsf{a} \vee \mathit{false})) \equiv \Diamond \mathsf{a}$ for ψ'_4 and ψ'_5; *(iv)* $\neg(\Box(\neg \mathsf{a} \vee \bigcirc(\mathit{false}\, \mathcal{U}\, \mathsf{b}))) \equiv \Diamond(\mathsf{a} \wedge \neg \bigcirc(\mathsf{b}))$ for ψ'_6.

Constraint-based declarative languages like DECLARE are used to describe requirements to the process behavior. In this case, each LTL$_f$ rule describes a specific constraint with clear semantics. Therefore, we need a *univocal*, syntax-independent and intuitive way to diagnose vacuously compliant behavior in constraint-based processes.

3.2 Ad-Hoc Approaches

An alternative approach to the syntax-dependent vacuity detection recalled in Section 3.1 is to restrict the constraint language, considering a pre-defined family of constraint patterns rather than a full-fledged temporal logic. This is the case, e.g., of DECLARE. [20, 11] take advantage from this feature of DECLARE, and provide an ad-hoc definition of constraint activation and vacuity, explicitly handling each templates. However, this approach fails when DECLARE is extended with new templates, a feature that has been deemed essential since the very first seminal papers on this approach [26]. The following example introduces a quite interesting template that cannot be expressed by using the core templates of DECLARE.

Example 1. We call *progression* of a tuple of tasks $\langle \mathsf{t}_1, \ldots, \mathsf{t}_n \rangle$ a trace that contains $\mathsf{t}_1, \ldots, \mathsf{t}_n$ in the proper order (possibly with other tasks in between), and ends with t_n. We use this notion to introduce a *progression response* constraint that extends the DECLARE *response* as follows: given two tuples $U = \langle \mathsf{u}_1, \ldots, \mathsf{u}_k \rangle$ and $V = \langle \mathsf{v}_1, \ldots, \mathsf{v}_m \rangle$ of source and target task tuples, the progression response constraint states that, whenever a progression of the source U is observed, then a progression of the target V must

be observed in the future; if this happens, the constraint goes back checking whether a new progression of the source is observed. This constraint can be used, e.g., to specify that whenever an order is finalized *and then* paid, the future course of execution must contain an order delivery *followed by* the emission of a receipt. The LTL_f formalization of this constraint is overly complex. Given a tuple $T = \langle \mathsf{t}_1, \ldots, \mathsf{t}_n \rangle$, we call *progression formula* the LTL_f formula $\Phi^T_{prog} = \Diamond(\mathsf{t}_1 \wedge \Diamond(\mathsf{t}_2 \wedge (\cdots \wedge \Diamond \mathsf{t}_n)))$. With this notion at hand, in the general case, the *progression response from U to V* can be formally captured in LTL_f as $\Box\left(\neg\Phi^U_{prog} \vee \Phi^{\langle U,V\rangle}_{prog}\right)$, where $\langle U, V\rangle$ is the tuple of tasks that appends V after U. For example, by using tasks fin, pay, del, rec to respectively denote the order finalization, its payment, its delivery, and the emission of a receipt, the aforementioned progression response is formalized in LTL_f as:

$$\Box(\neg\Diamond(\mathsf{fin} \wedge \Diamond\mathsf{pay}) \vee \Diamond(\mathsf{fin} \wedge \Diamond(\mathsf{pay} \wedge \Diamond(\mathsf{del} \wedge \Diamond\mathsf{rec}))))$$ ■

The definition of vacuous satisfaction for such a constraint, and the corresponding notion of constraint activation, cannot be easily hijacked from that of DECLARE patterns, nor it is easy to extract using human ingenuity. In contrast, our goal is to provide a semantical, general treatment of vacuity and activation, making it possible to seamlessly apply declarative process mining techniques also on new constraint patterns such as the progression response of Example 1, without requiring human intervention.

4 Activation of Constraints

This section discusses the core contribution of this paper, i.e., how to determine whether an execution trace *activates* a constraint or not. Our approach has three distinctive features. (1) It is *fully semantical*, in the sense that it detects when a trace is an interesting witness for a constraint, in a way that is completely independent from the specific syntactic form of the constraint. (2) It is *general*, in the sense that it does not focus on specific constraint languages such as DECLARE, but seamlessly work for all the temporal logics mentioned in Section 2, including MSO_f, LDL_f, regular expressions, and LTL_f. (3) It *seamlessly applies at run-time or a posteriori*, i.e., it can also be used to assess relevance of running, evolving traces.

Our approach consists of three steps. In the first step, we gain more details about the different states in which a constraint can be, going beyond the coarse-grained characterization of satisfied vs violated. In the second step, we leverage these additional details to semantically characterize the notion of "interesting witness", which in turn constitutes the basis for understanding whether a constraint is activated by a trace or not. In the last step, we mirror this approach into the automata-based characterization of the aforementioned logics, consequently obtaining a concrete technique to check whether a trace activates a constraint or not (this is subject of Section 5).

4.1 Activation States and Relevant Task Executions

To understand in details how a trace relates to a constraint, we leverage on the four truth values provided by RV-LTL [2], which considers LTL in the light of runtime verification. This approach has been already extensively adopted in the recent past for conformance

checking and monitoring of LTL_f and LDL_f constraints [23, 21, 7]. RV-LTL brings two main advantages in the context of this paper. On the one hand, it makes our approach working also in a monitoring setting. On the other hand, it provides the basis to check whether an execution trace actively interacts with the constraint or not.

Def. 3 (RV-LTL **truth values).** *Given a constraint φ over Σ, and an execution trace τ over Σ^*, we say that:*

- τ permanently satisfies φ, *written* $[\tau \models \varphi]_{RV} = \texttt{ps}$, *if φ the constraint is satisfied by the current trace (i.e., $\tau \models \varphi$ in the standard logical sense), and will remain satisfied for every possible continuation of the trace: for every τ' over Σ^*, we have $\tau \cdot \tau' \models \varphi$;*
- τ permanently violates φ, *written* $[\tau \models \varphi]_{RV} = \texttt{pv}$, *if φ is violated by the current trace (i.e., $\tau \not\models \varphi$ in the standard logical sense), and will remain violated for every possible continuation of the trace: for every τ' over Σ^*, we have $\tau \cdot \tau' \not\models \varphi$;*
- τ temporarily satisfies φ, *written* $[\tau \models \varphi]_{RV} = \texttt{ts}$, *if φ is satisfied by the current trace (i.e., $\tau \models \varphi$), but there exists at least one continuation of the trace leading to violation: there exists τ' over Σ^* such that $\tau \cdot \tau' \not\models \varphi$;*
- τ temporarily violates φ, *written* $[\tau \models \varphi]_{RV} = \texttt{tv}$, *if φ is violated by the current trace (i.e., $\tau \not\models \varphi$), but there exists at least one continuation of the trace leading to satisfaction: there exists τ' over Σ^* such that $\tau \cdot \tau' \models \varphi$.*

We also say that τ complies with φ *if* $[\tau \models \varphi]_{RV} = \texttt{ps}$ *or* $[\tau \models \varphi]_{RV} = \texttt{ts}$.

Why do we care about such RV-LTL *truth values?* The intuition is that once a constraint becomes permanently satisfied (ps) or permanently violated (pv), then what happens next in the trace is irrelevant for the constraint, since such truth values are indeed unmodifiable. Temporary states instead are those for which interesting task executions may still happen.

The RV-LTL truth values can be used to identify, given an execution trace, which tasks are permitted (or forbidden) next.

Def. 4 (Forbidden/permitted task). *Let φ be a constraint over Σ, and τ an execution trace over Σ^*. We say that task* t *is* forbidden by φ after τ, *if executing* t *next leads to a permanent violation state:* $[\tau \cdot \texttt{t} \models \varphi]_{RV} = \texttt{pv}$. *If this is not the case, then* t *is said to be* permitted by φ after τ.

Notice that, by definition, if a constraint is permanently satisfied (respectively, violated) by a trace, then every task is permitted (respectively, forbidden). *Why do we care about permitted tasks?* Intuitively, considering the set of permitted tasks and how it evolves over time helps when the RV-LTL characterization alone is not informative. Specifically, whenever a task execution does not trigger any change in the RV-LTL truth value of a constraint, we can assess relevance by checking whether it causes at least a relevant change in the set of permitted tasks.

We now combine the notions of RV-LTL truth value and of permitted task so as to identify when a task execution is relevant for a constraint. This combination gives rise to the notion of activation state.

Def. 5 (Activation state). *An* activation state *over Σ is a pair $\langle V, \Lambda \rangle$, where V is one of the four truth values in RV-LTL, i.e., $V \in \{\texttt{ps}, \texttt{pv}, \texttt{ts}, \texttt{tv}\}$, and $\Lambda \subseteq \Sigma$ is a set of* permitted *tasks.*

Due to Def. 3 and Def. 4, not all activation states are meaningful. For example, we know that if the current RV-LTL value is $\mathtt{pv}$, then no task is permitted. We systematize this notion by identifying those activation states that are "legal".

Def. 6 (Legal activation state). *An activation state over Σ is* legal *if it is of one of the following forms:*

- $\langle \mathtt{ps}, \Sigma \rangle$ *(every task is permitted if the constraint is permanently satisfied);*
- $\langle \mathtt{pv}, \emptyset \rangle$ *(if the constraint is permanently violated, nothing is permitted);*
- $\langle \mathtt{ts}, \Lambda \rangle$*, with $\emptyset \subset \Lambda \subseteq \Sigma$ (if the constraint is temporarily satisfied, there must be at least one permitted task that triggers a change towards violation);*
- $\langle \mathtt{tv}, \Lambda \rangle$*, with $\emptyset \subset \Lambda \subseteq \Sigma$ (if the constraint is temporarily violated, there must be at least one permitted task that triggers a change towards satisfaction).*

We denote by $\mathbb{S}_\Sigma$ the set of possible legal activation states over Σ.

Def. 7 (Trace activation state). *Let φ be a constraint over Σ, and τ an execution trace over Σ^*. The* trace activation state of φ in τ*, written $actState_\varphi(\tau)$, is the activation state $\langle V, \Lambda \rangle$, where: (1) $V = v$ iff $[\tau \models \varphi]_{RV} = v$ (cf. Def. 3); (2) for every $\mathtt{t} \in \Sigma$, we have $\mathtt{t} \in \Lambda$ iff $\mathtt{t}$ is permitted by φ after τ (cf. Def. 4). The* initial activation state *is the activation state computed for $\tau = \varepsilon$.*

Trace activation states enjoy the following property.

Lemma 1. *For every constraint φ over Σ and every trace τ over Σ^*, the trace activation state of φ in τ is legal, i.e., $actState_\varphi(\tau) \in \mathbb{S}_\Sigma$.*

Proof. Immediate from the definitions of trace and legal activation states. □

Example 2. Consider the DECLARE *response* constraint $\varphi_r = \Box(\mathsf{a} \rightarrow \Diamond \mathsf{b})$ over Σ. The initial activation state of φ_r is $\langle \mathtt{ts}, \Sigma \rangle$: all tasks are permitted, and φ_r is temporarily satisfied, since there are traces culminating in the violation of the constraint. Consider now the trace $\langle \mathsf{a} \rangle$: we get $actState_{\varphi_r}(\mathsf{a}) = \langle \mathtt{tv}, \Sigma \rangle$. In fact, all tasks are still permitted, but the constraint is temporarily violated because it requires the future presence of b. ■

The execution of a task induces a transition in the trace activation state. By considering the combination of the current and next trace activation states, we can understand whether the induced transition is relevant for the constraint or not. This is done by formalizing the intuitions discussed in Section 4.1.

Def. 8 (Relevant task execution). *Let φ be a constraint over Σ, $\mathtt{t} \in \Sigma$ be a task, and τ an execution trace over Σ^*. Let $\langle V, \Lambda \rangle = actState_\varphi(\tau)$ and $\langle V', \Lambda' \rangle = actState_\varphi(\tau \cdot \mathtt{t})$ respectively be the trace activation states of φ in τ and the one obtained as the result of executing $\mathtt{t}$ after τ. We say that* $\mathtt{t}$ is a relevant execution for φ after τ *(or equivalently that* $\mathtt{t}$ is a relevant execution for φ in $actState_\varphi(\tau)$*) if $V \neq V'$ or $\Lambda \neq \Lambda'$.*

4.2 Interesting Witnesses, Activation and Vacuity

Def. 8 provides the basis to assess whether a task execution is relevant to a constraint in a given execution context (characterized by the current activation state). We now lift this notion to a trace as a whole.

Def. 9 (Activation/Interesting witness). *A constraint φ over Σ is* activated *by a trace τ over Σ^* if there exists* $\mathsf{t} \in \Sigma$ *s.t.: (1)* $\tau = \tau_{pre}\ \mathsf{t}\ \tau_{suf}$*; (2)* t *is a relevant execution for φ after τ_{pre} (cf. Def. 8). If so, we also say that τ is an* interesting witness *for φ.*

Example 3. Consider the *response* constraint of Example 2, and the execution trace $\tau = \langle \mathsf{c}, \mathsf{b}, \mathsf{a}, \mathsf{b}, \mathsf{b}, \mathsf{a}, \mathsf{a}, \mathsf{b} \rangle$. By making trace activation states along τ explicit, we get:

$$\langle \mathtt{ts}, \Sigma \rangle\ \mathsf{c}\ \langle \mathtt{ts}, \Sigma \rangle\ \mathsf{b}\ \langle \mathtt{ts}, \Sigma \rangle\ \mathsf{a}\ \langle \mathtt{tv}, \Sigma \rangle\ \mathsf{b}\ \langle \mathtt{ts}, \Sigma \rangle\ \mathsf{b}\ \langle \mathtt{ts}, \Sigma \rangle\ \mathsf{a}\ \langle \mathtt{tv}, \Sigma \rangle\ \mathsf{a}\ \langle \mathtt{tv}, \Sigma \rangle\ \mathsf{b}\ \langle \mathtt{ts}, \Sigma \rangle$$

Arrows indicate the relevant task executions. In fact, the first relevant task execution is a, because it is the one that leads to switch the RV-LTL truth value of the constraint from temporarily satisfied to temporarily violated. The following task b is also relevant, because it triggers the opposite change. The second following b, instead, is irrelevant, because it keeps the activation state unchanged. A similar pattern can be recognized for the following two as: the first one is relevant, the second one is not. Notice that τ complies with φ_r. Now, consider the *not coexistence* constraint $\varphi_{nc} = \neg(\Diamond a \wedge \Diamond b)$, and the same execution trace τ as before. We obtain:

$$\langle \mathtt{ts}, \Sigma \rangle\ \mathsf{c}\ \langle \mathtt{ts}, \Sigma \rangle\ \mathsf{b}\ \langle \mathtt{ts}, \Sigma \setminus \{\mathsf{a}\} \rangle\ \mathsf{a}\ \langle \mathtt{pv}, \emptyset \rangle\ \mathsf{b}\ \langle \mathtt{pv}, \emptyset \rangle\ \mathsf{b}\ \langle \mathtt{pv}, \emptyset \rangle\ \ldots\ \langle \mathtt{pv}, \emptyset \rangle$$

The constraint is in fact initially temporarily satisfied, and remains so until one between a or b is executed. This happens in the second position of τ, where the relevant execution of b introduces a restrictive change that does not affect the truth value of the constraint, but reduces the set of permitted tasks. The consequent execution of a is also relevant, because it causes a permanent violation of the constraint. A permanent violation corresponds to an irreversible activation state, and therefore independently on how the trace continues, all consequent task executions are irrelevant. ■

In Example 3, the same trace is an interesting witness for two constraints, but for a very different reason. In one case, the trace contains relevant task executions and satisfies the constraint, whereas in the second case the trace violates the constraint. For "reasonable" constraints, i.e., constraints that admit at least one satisfying trace, every trace that violates the constraint is an interesting witness, since it necessarily contains one execution causing the trace activation state to become $\langle \mathtt{pv}, \emptyset \rangle$. In the case of satisfaction, two cases may arise: either the trace satisfies the constraint and is relevant, or the trace satisfies the constraint without ever activating it. We systematize this intuition, obtaining a *fully semantical characterization of vacuity* for temporal formulae over finite traces.

Def. 10 (Interesting/vacuous satisfaction). *Let φ be a constraint over Σ, and τ a trace over Σ^* that complies with φ (cf. Def. 3). If τ is an interesting witness for φ (cf. Def. 9), then τ* interestingly satisfies *φ, otherwise τ* vacuously satisfies *φ.*

Example 4. In Example 3, trace τ activates both the *response* (φ_r) and *not coexistence* (φ_{nc}) constraints. Now consider the execution trace $\tau_2 = \langle \mathsf{c}, \mathsf{c}, \mathsf{b}, \mathsf{c}, \mathsf{b} \rangle$. Since τ_2 contains b, it is an interesting witness for φ_{nc}: when the first occurrence of b happens, the set of permitted tasks moves from the whole Σ to $\Sigma \setminus \mathsf{a}$. Furthermore, τ_2 does not contain both a and b, and hence it complies with φ_{nc}. Consequently, we have that τ_2 interestingly satisfies φ_{nc}. As for the *response* constraint, since τ_2 does not contain occurrences of a, it does not activate the constraint. More specifically, τ_2 never changes

the initial activation state of φ_r, which corresponds to $\langle \mathtt{ts}, \Sigma \rangle$. This also shows that τ_2 complies with φ_r and, in turn, that τ_2 vacuously satisfies φ_r. ■

5 Checking Constraint Activation Using Automata

We now make the notion of activation operational, leveraging the automata-theoretic approach for constraints expressed in MSO_f or LDL_f (which, recall, are expressively equivalent and strictly subsume LTL_f). We consider in particular LDL_f, for which automata-based techniques have been extensively studied [9, 7]. Towards our goal, we exploit a combination of the automata construction technique in [7] with the notion of *colored automata* [21]. Colored automata augment FSAs with state-labels that reflect the RV-LTL truth value of the corresponding formulae. We further extend such automata in two directions. On the one hand, each automaton state is also labeled with the set of permitted tasks, thus obtaining full information about the corresponding activation states; on the other hand, relevant executions are marked in the automaton by "coloring" their corresponding transitions. We consequently obtain the following type of automaton.

Def. 11 (Activation-Aware Automaton). *The* activation-aware automaton $\mathcal{A}^{act}_{\varphi}$ *of an* LDL_f *formula* φ *over* Σ *is a tuple* $\langle \Sigma, S, s_0, \delta, F, \alpha, \rho \rangle$*, where:*

- $\langle \Sigma, S, s_0, \delta, F \rangle$ *is the constraint automaton for* φ *(cf. Def. 2 and [7]);*
- $\alpha : S \longrightarrow \mathbb{S}_{\Sigma}$ *is the function that maps each state* $s \in S$ *to the corresponding activation state* $\alpha(s) = \langle V, \Lambda \rangle$*, where:*
  - $V = \mathtt{ts}$ *iff* $s \in F$ *and there exists state* $s \in S$ *s.t.* $\delta^*(s, s')$ *and* $s' \notin F$*;*
  - $V = \mathtt{ps}$ *iff* $s \in F$ *and for every state* $s' \in S$ *s.t.* $\delta^*(s, s')$*, we have* $s' \in F$*;*
  - $V = \mathtt{tv}$ *iff* $s \notin F$ *and there exists state* $s' \in S$ *s.t.* $\delta^*(s, s')$ *and* $s' \in F$*;*
  - $V = \mathtt{pv}$ *iff* $s \notin F$ *and for every state* $s' \in S$ *s.t.* $\delta^*(s, s')$*, we have* $s' \notin F$*;*
  - Λ *contains task* $\mathtt{t} \in \Sigma$ *iff there exists* $s' \in S$ *s.t.* $s' = \delta(s, \mathtt{t})$ *and* $\alpha(s')$ *has an RV-LTL truth value different from* $\mathtt{pv}$*.*
- $\rho \subseteq Domain(\delta)$ *is the set of transitions in* δ *that are relevant for* φ*, i.e.:* $\rho = \{\langle s, \mathtt{t} \rangle \mid \langle s, \mathtt{t} \rangle \in Domain(\delta)$ *and* $\mathtt{t}$ *is a relevant execution for* φ *in* $\alpha(s)\}$

Notably, such an activation-aware automaton correctly reconstructs the notions of activation and relevance as defined in Section 4.2.

Theorem 1. *Let* φ *be an* LDL_f *formula over* Σ*, and* $\mathcal{A}^{act}_{\varphi} = \langle \Sigma, S, s_0, \delta, F, \alpha, \rho \rangle$ *the activation-aware automaton for* φ*. Let* $\tau = \langle \mathtt{t}_1 \cdots \mathtt{t}_n \rangle$ *be a non-empty, finite trace over* Σ*, and* $s_0 \cdots s_n$ *the sequence of states such that* $\delta(s_{i-1}, \mathtt{t}_i) = s_i$ *for* $i \in \{1, \ldots, n\}$*.*[4] *Then, the following holds: (1)* $\mathsf{atr}_{\varphi}(\tau) = \alpha(s_0) \cdots \alpha(s_n)$*; (2) for every* $i \in \{1, \ldots, n\}$*,* $\langle s_{i-1}, \mathtt{t}_i \rangle \in \rho$ *if and only if* $\mathtt{t}_i$ *is a relevant task execution for* φ *after* $\langle \mathtt{t}_1, \ldots, \mathtt{t}_{i-1} \rangle$*.*

Proof. From the correctness of the constraint automaton construction (cf. Def. 2 and [7]), we know that τ satisfies φ iff it is accepted by $\mathcal{A}^{act}_{\varphi}$ (i.e., iff $s_n \in F$). This corresponds to the notion of conformance in Def. 3. The proof of the first claim is then obtained by observing that all tests in Def. 11, which characterize the RV-LTL values and permitted tasks of the automaton states, perfectly mirror Def. 3 and 4. In particular, notice that the labeling of states with RV-LTL values agrees with the construction

[4] Recall that, since $\mathcal{A}^{act}_{\varphi}$ is not trimmed, then it can replay any trace from Σ^*.

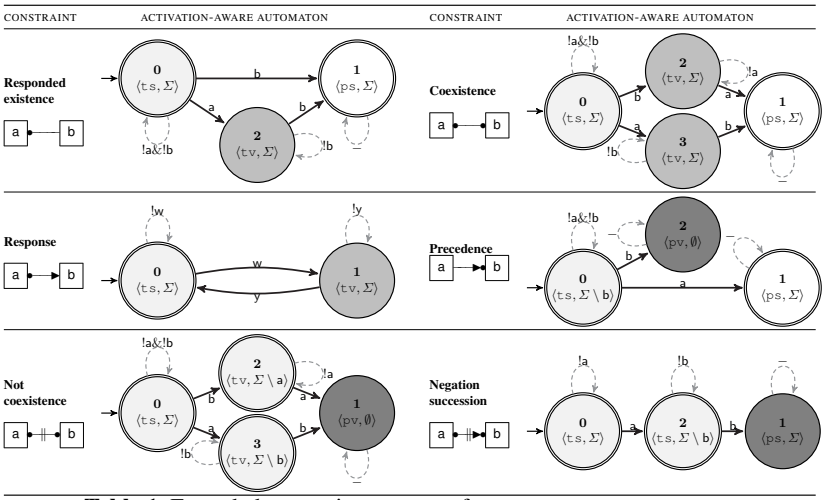


Table 1. Extended constraint automata for some DECLARE patterns

of "local colored automata" in [21], proven to be correct in [7]. The second claim immediately follows from the first one, by observing that Def. 11 define ρ by directly employing the notion of relevance in a given activation state as defined in Def. 8. □

We close this section by observing that Def. 11 can be directly implemented to build the activation-aware automaton of an LDL$_f$ formula φ. Notably, such extended information does not impact on the computational complexity of the automaton construction. This is done in three steps. (1) The constraint automaton $\mathcal{A}_\varphi$ for φ is built by applying the LDL$_f$2NFA procedure of [7], and then the standard determinization procedure for the obtained automaton (thus getting a DFA). (2) Function α is constructed in two iterations. In the first iteration, the RV-LTL truth value of each state in $\mathcal{A}_\varphi$ is computed, by iterating once through each state of the automaton, and checking whether it may reach a final state or not. This can be done in PTIME in the size of the automaton. The second iteration goes over each state of $\mathcal{A}_\varphi$, and calculates the permitted tasks by considering the RV-LTL value of the neighbor states. This can be done, again, in PTIME. (3) Function ρ is built in PTIME by considering all pairs of states in $\mathcal{A}_\varphi$, and by applying the explicit definition of relevant execution. Table 1 and Fig. 3 respectively list the activation-aware automata for some standard DECLARE patterns, and the activation-aware automaton for a progression response. State colors reflect the RV-LTL truth value they are associated to. Dashed, gray transitions are irrelevant, whereas the black, solid ones are relevant in the sense of Def. 8. Interestingly, relevant transitions for the progression response are those that "close" a proper progression of the source or target tasks. This reflect human intuition, but is obtained automatically from our semantical approach.

6 Evaluation

In order to validate our approach, we have embedded it into a prototype software codified in Java for the discovery of constraints from an event log (based on the algorithm

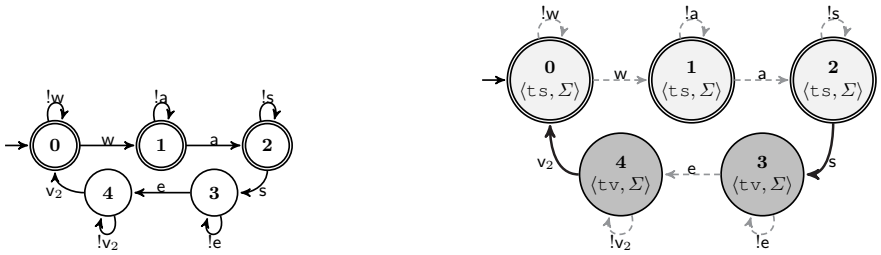


Fig. 3. Constraint automaton and activation-aware automaton for the *progression response* constraint (with three sources and two targets)

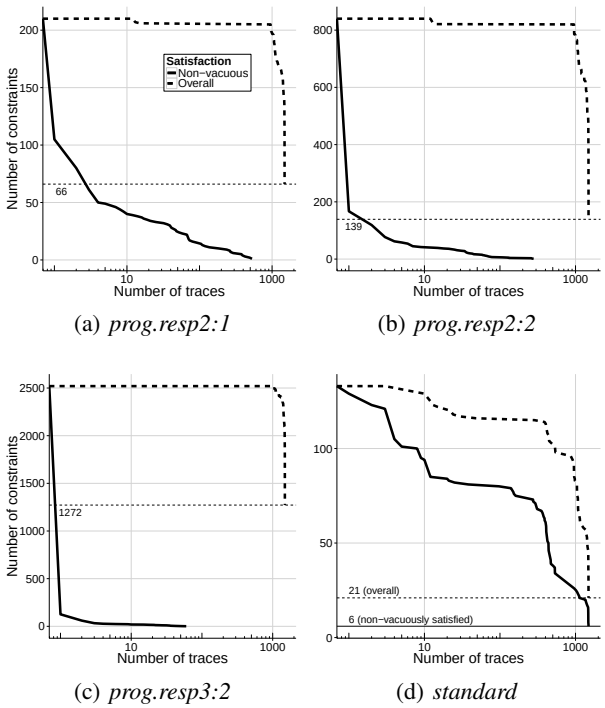
presented in [22]).[5] The approach has been run on two real-life event logs taken from the collection of the IEEE Task Force on Process Mining, i.e., the log used for the BPI challenge 2013[6] and a log pertaining to a road traffic fines management process[7]. The tests have been conducted on a machine equipped with an Intel Core processor i5-3320M, CPU at 2.60GHz, quad-core, Ubuntu Linux 12.04 operating system. In our experiments, for the discovery task, we have considered four templates belonging to the repertoire of standard DECLARE, i.e., *existence*, *alt. precedence*, *co-existence*, and *neg. chain succession*, and three variants of the *progression response* with numbers of sources and targets respectively equal to 2 and 1, 2 and 2, and 3 and 2. In the remainder, we call these templates *prog.resp2:1*, *prog.resp2:2*, and *prog.resp3:2*, respectively.

Fig. 4 shows the trends of the number of progression response constraints discovered from the BPI challenge 2013 log with respect to the number of traces (vacuously and interestingly) satisfying them. Figg. 4(a)–4(c) relate to progression response templates with an increasing number of parameters. On the abscissae of each plot lies the number of traces where the constraints are satisfied. The number of discovered constraints lies on the ordinates. The analysis of the results shows how crucial the strive for vacuity detection is, in order to avoid the business analyst to be overwhelmed by a huge number of uninteresting constraints. The discovery algorithm detected indeed that 66 *prog.resp2:1*, 139 *prog.resp2:2*, and 1,272 *prog.resp3:2* were vacuously satisfied in the entire log. The reason why the number of irrelevant returned constraints is higher for *prog.resp3:2* than for *prog.resp2:1* and *prog.resp2:2* is twofold. On the one hand, this is because the first one can only be activated when three different tasks occur sequentially, whereas the second and the third one only require two tasks to occur one after another to be activated. Another reason is that the implemented algorithm checks the validity in the event log of a set of candidate constraints obtained by instantiating each template with all the possible combinations of the tasks available in the log. Therefore, the higher number of parameters of *prog.resp3:2* leads to a higher number of candidate constraints. Fig. 4(d) shows the same trend when using the standard DECLARE templates mentioned above for the discovery. Overall, the computation took 9.442 sec, out of which 426 msec were spent to build the automata, and the remaining 9,016 msec to check the log.

[5] The tool is available at https://github.com/cdc08x/MINERful/blob/master/run-MINERful-vacuityCheck.sh

[6] DOI: 10.4121/c2c3b154-ab26-4b31-a0e8-8f2350ddac11

[7] DOI: 10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5



(a) *prog.resp2:1* (b) *prog.resp2:2*

(c) *prog.resp3:2* (d) *standard*

Fig. 4. Trends of the number of the discovered constraints with respect to the number of traces satisfying them

We show that our technique is sound, by comparing the results obtained from the road traffic fines management log using our implemented prototype with the constraints discovered by the MINERful declarative miner [14] and the DECLARE Miner [22]. The comparison has been conducted using a minimum threshold of 100% of interesting witnesses in the log. The discovered constraints are:

- Existence(Create Fine)
- Alt. precedence(Create Fine, Add penalty)
- Neg. chain succession(Create Fine, Add penalty)
- Alt. precedence(Create Fine, Appeal to Judge)
- Alt. precedence(Create Fine, Insert Date Appeal to Prefecture)
- Alt. precedence(Create Fine, Insert Fine Notification)
- Neg. chain succession(Create Fine, Insert Fine Notification)
- Alt. precedence(Create Fine, Notify Result Appeal to Offender)
- Neg. chain succession(Create Fine, Notify Result Appeal to Offender)
- Alt. precedence(Create Fine, Receive Result Appeal from Prefecture)
- Neg. chain succession(Create Fine, Receive Result Appeal from Prefecture)
- Alt. precedence(Create Fine, Send Appeal to Prefecture)
- Neg. chain succession(Create Fine, Send Appeal to Prefecture)
- Alt. precedence(Create Fine, Send Fine)
- Alt. precedence(Create Fine, Send for Credit Collection)
- Neg. chain succession(Create Fine, Send for Credit Collection)

Such constraints are a subset of the ones returned by MINERful using the same templates, since MINERful has no vacuity detection mechanism, and coincide with the ones returned by the DECLARE Miner. The derived constraints suggest that “Create fine” occurs in every trace and precedes many other activities. In addition, some activities can-

not directly follow "Create fine". Also, we discovered that the following progression response constraints are interestingly satisfied by around 53% of traces:

- Prog.resp2:1((Create Fine, Insert Fine Notification), Add penalty)
- Prog.resp2:1((Send Fine, Insert Fine Notification), Add penalty)
- Prog.resp2:1((Create Fine, Send Fine), Add penalty)
- Prog.resp2:1((Create Fine, Send Fine), Insert Fine Notification)
- Prog.resp2:2((Create Fine, Send Fine, Insert Fine Notification), Add penalty)

Although not always activated, the first two in the list are never violated. The last three are instead violated by approximately 26% of the traces. Similar results cannot be obtained neither with MINERful that is not designed to discover non-standard DECLARE constraints nor with the DECLARE Miner that offers such facility, but only provides an ad-hoc mechanism for vacuity detection.

7 Conclusion

To the best of our knowledge, this paper presents the first semantical characterization of activation and relevance for declarative business constraints expressed with temporal logics over finite traces. As a side result, we also obtain a semantical notion of vacuous satisfaction for such logics. Our characterization comes with a concrete approach to monitor and check activation and relevance on running or complete traces, achieved by suitably extending the standard automata-theoretic approach for (finite trace) temporal logics. The carried experimental evaluation confirms the benefits of our approach, and paves the way towards a more extensive study on mining declarative constraints going (far) beyond the DECLARE patterns.

The presented solution generalizes the ad-hoc approaches previously proposed in the literature to tackle conformance checking and discovery of DECLARE constraints [22, 20, 14]. However, it is also compatible with human intuition, in the sense that it by and large agrees with such ad-hoc approaches when applied to the DECLARE patterns.

An interesting line of research is to extend our approach towards the possibility of "counting" activations. This becomes crucial when declarative process discovery is tuned so as to extract constraints that do not have full support in the log. In this case, "relevance heuristics" must be devised so as to rank candidate constraints, and these are typically based on various notions of activation counting [12]. However, providing a systematic theory of counting is far from trivial. Our intuition is that this theory can be developed only by making constraints data-aware, which in turn requires to adopt first-order variants of temporal logics for their formalization [10]. In fact, data-aware constraints can express *task correlation* [25, 10], an essential feature towards counting.

References

1. van der Aalst, W., Pesic, M., Schonenberg, H.: Declarative Workflows: Balancing Between Flexibility and Support. Computer Science - R&D (2009)
2. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. 20(4) (2011)
3. Beer, I., Eisner, C.: Efficient detection of vacuity in temporal model checking. Formal Methods in System Design 18(2) (2001)
4. Burattin, A., Maggi, F.M., van der Aalst, W.M.P., Sperduti, A.: Techniques for a posteriori analysis of declarative processes. In: Proc. of EDOC. IEEE (2012)

5. Chesani, F., Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Exploiting inductive logic programming techniques for declarative process mining. T. Petri Nets and Other Models of Concurrency 2 (2009)
6. Damaggio, E., Deutsch, A., Hull, R., Vianu, V.: Automatic verification of data-centric business processes. In: Proc. of BPM. Springer (2011)
7. De Giacomo, G., De Masellis, R., Grasso, M., Maggi, F.M., Montali, M.: Monitoring business metaconstraints based on LTL & LDL for finite traces. In: Proc. of BPM (2014)
8. De Giacomo, G., De Masellis, R., Montali, M.: Reasoning on LTL on finite traces: Insensitivity to infiniteness. In: Proc. of AAAI. AAAI (2014)
9. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Proc. of IJCAI. AAAI (2013)
10. De Masellis, R., Maggi, F.M., Montali, M.: Monitoring data-aware business constraints with finite state automata. In: Proc. of ICSSP. ACM (2014)
11. Di Ciccio, C., Maggi, F.M., Mendling, J.: Efficient discovery of target-branched declare constraints. Inf. Syst. 56, 258–283 (2016)
12. Di Ciccio, C., Maggi, F.M., Montali, M., Mendling, J.: Ensuring model consistency in declarative process discovery. In: Proc. of BPM. Springer (2015)
13. Di Ciccio, C., Mecella, M.: A two-step fast algorithm for the automated discovery of declarative workflows. In: Proc. of CIDM. IEEE (2013)
14. Di Ciccio, C., Mecella, M.: On the discovery of declarative control flows for artful processes. ACM Trans. Manage. Inf. Syst. 5(4) (2015)
15. Giannakopoulou, D., Havelund, K.: Automata-based verification of temporal properties on running programs. In: Proc. of ASE. IEEE (2001)
16. Knuplesch, D., Ly, L.T., Rinderle-Ma, S., Pfeifer, H., Dadam, P.: On enabling data-aware compliance checking of business process models. In: Proc. of ER. Springer (2010)
17. Kupferman, O., Vardi, M.Y.: Vacuity Detection in Temporal Model Checking. Int. Journal on Software Tools for Technology Transfer (2003)
18. Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Inducing declarative logic-based models from labeled traces. In: Proc. of BPM. Springer (2007)
19. de Leoni, M., Maggi, F.M., van der Aalst, W.M.P.: An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data. Inf. Syst. 47 (2015)
20. Maggi, F.M., Bose, J., van der Aalst, W.M.P.: Efficient Discovery of Understandable Declarative Models from Event Logs. In: Proc. of CAiSE. Springer (2012)
21. 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: Proc. of BPM. Springer (2011)
22. Maggi, F.M., Mooij, A.J., van der Aalst, W.M.P.: User-guided discovery of declarative process models. In: Proc. of CIDM (2011)
23. Maggi, F.M., Westergaard, M., Montali, M., van der Aalst, W.M.P.: Runtime verification of LTL-based declarative process models. In: Proc. of RV. Springer (2011)
24. Montali, M.: Specification and Verification of Declarative Open Interaction Models: a Logic-Based Approach. Springer (2010)
25. Montali, M., Maggi, F.M., Chesani, F., Mello, P., van der Aalst, W.M.P.: Monitoring business constraints with the event calculus. ACM Trans. Intell. Systems and Technology 5(1) (2013)
26. Pesic, M., Schonenberg, H., van der Aalst, W.: DECLARE: Full Support for Loosely-Structured Processes. In: Proc. of EDOC. IEEE (2007)
27. Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., Reijers, H.A.: Imperative versus declarative process modeling languages: An empirical investigation. In: BPM Workshops. Springer (2011)
28. Zugal, S., Pinggera, J., Weber, B.: The impact of testcases on the maintainability of declarative process models. In: Proc. of BPMDS/EMMSAD. Springer (2011)