On the Relevance of a Business Constraint to an Event Log

Claudio Di Ciccio[a,*], Fabrizio Maria Maggi[b], Marco Montali[c], Jan Mendling[a]

[a] *Vienna University of Economics and Business, Austria*
[b] *University of Tartu, Estonia*
[c] *Free University of Bozen-Bolzano, Italy*

Abstract

Declarative process modeling languages such as DECLARE describe the behavior of processes by means of constraints. Such constraints exert rules on the execution of tasks upon the execution of other tasks called activations. The constraint is thus fulfilled both if it is activated and the consequent rule is respected, or if it is not activated at all. The latter case, named vacuous satisfaction, is clearly less interesting than the former. Such a distinction becomes of utmost importance in the context of declarative process mining techniques, where processes are analyzed based on the identification of the most relevant constraints valid in an event log. 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 techniques inapplicable when the target constraint language is extensible, and may contain formulae that go beyond the pre-defined patterns. In this paper, we tackle this 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

*Corresponding author.
E-mail address: claudio.di.ciccio@wu.ac.at
Postal address: Vienna University of Economics and Business, Institute for Information Business (Building D2, Entrance C) – Welthandelsplatz 1, A-1020 Vienna, Austria
Phone number: +43 1 31336 5222
Email addresses: claudio.di.ciccio@wu.ac.at (Claudio Di Ciccio), f.m.maggi@ut.ee (Fabrizio Maria Maggi), montali@inf.unibz.it (Marco Montali), jan.mendling@wu.ac.at (Jan Mendling)

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

traces. Our solution relies on the annotation of finite state automata to incorporate relevance-related information. We discuss the formal grounding of our approach and describe the implementation thereof. We finally report on experimental results gathered from the application of our approach to real-life data, which show the advantages and feasibility of our solution.

Keywords: Vacuity Detection, Declarative Process Mining, Constraint Activation, Linear Temporal Logic, Finite State Automata

1. Introduction

The increasing availability of event data recorded by information systems, electronic devices, web services, and sensor networks provides detailed information about the execution of actual processes within and across organizations. Process mining techniques can use such event data to discover and enhance process models, as well as to check whether the actual behavior of a process conforms to the expected behavior [1]. When a process works in a variable and knowledge-intensive setting, it is crucial to achieve a trade-off between flexibility and control [2, 3]. In this situation, declarative approaches provide a suitable target for process mining, since classical, imperative process models tend to become too detailed and intricate [4].

The common denominator of declarative process modeling approaches is to avoid the explicit, exhaustive enumeration of the acceptable sequences of tasks in a process: the allowed behaviors are implicitly obtained by considering all sequences of tasks that satisfy a given set of constraints, declaratively specifying what needs to be accomplished, without stating how. In this way, process models offer the so-called *flexibility by design* [5], while remaining compact. Among the several proposals for declarative process modeling, DECLARE [3, 6] and DCR graphs [7] employ temporal logics (respectively over finite and infinite traces) to formalize constraints, and build on the well-established, automata-theoretic techniques for such logics to carry out consistency checking, enactment, monitoring, and mining of declarative process models.

There is, however, a fundamental issue when applying such logic-based approaches: although they provide a clear, formal definition of whether an execution trace satisfies a constraint, they do not give a precise and generally applicable means to state whether the satisfaction is relevant. Let

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

us consider, for example, a constraint imposing that *if* a request occurs *then* it is eventually followed by a grant. In case a request occurs in a trace and, later on, a grant occurs too, the constraint is satisfied in the trace. If the request never occurs in the trace the constraint is also satisfied. However, such a satisfaction is arguably irrelevant, as the trace does not "actively interact" with the constraint. This is an example of what is called in the literature *vacuous satisfaction* [8, 9]. It is evident that for the practical development of process mining and operational support techniques, such as monitoring, conformance checking, and discovery, providing a yes/no judgement about constraint satisfaction is too coarse-grained, and answering the following two questions becomes essential:

- *Relevance*: does a trace *non-vacuously* satisfy a constraint?
- *Activation counting*: if so, *how much relevant* is the constraint to the trace?

In the context of DECLARE discovery, relevance and activation counting are key to reducing the (potentially large) number of constraints that may be extracted from an event log, pruning away those that satisfy the traces contained in the log, but in an irrelevant way [10, 11, 12, 13]. In conformance checking, activation counting is crucial to compute the so-called *health indicators* that measure the "degree of adherence" between a constraint and an execution trace [14, 15, 16, 17]. Since temporal logic-based formalisms do not provide a principled way to answer such questions, existing works adopt an ad-hoc approach, which fixes a predefined set of constraint patterns, and requires to explicitly spell out the meaning of relevance for each single pattern. This makes it difficult to understand the suitability and correctness of the proposed solutions; at the same time it makes them inapplicable when new types of constraints, going beyond the predefined set of patterns, are considered.

The goal of this paper is to overcome these issues by proposing for the first time a general, systematic characterization of relevance for temporal constraints in a finite-trace setting. Our approach is *formal* because it defines the notion of relevance on top of the logical semantics of constraints, and *operational*, since it suitably extends the automata-theoretic approach to handle relevance. Differently from the line of research on *vacuity detection* for linear temporal logic over infinite traces [8, 9], we leverage the finite-trace semantics to come up with a fully semantical syntax-independent characterization of relevance. We improve upon the state of the art in the context of declarative

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

process mining [18] in that our approach is independent from the repertoire of constraints under analysis: although we consider DECLARE as the language of reference, our approach can seamlessly be applied to any temporal logic-based constraint, because we fully resort on automata underlying their semantics.

We validate our approach along two directions. On the one hand, we report on the implementation and experimentation of our solution, confirming its advantages and feasibility. On the other hand, we show that our formal notion of relevance is compatible with the human intuition exploited in previous works, whereas counting activations partially diverges. We then conclude by discussing the reasons for such a divergence, arguing that the problem of counting cannot in general be tackled satisfactorily without enriching the representation of constraints with additional features.

This paper is an extended version of [19]. The three main extensions we provide here are: *(i)* a more detailed account of our approach, including a proof that the presented automata-theoretic technique is correct with respect to our semantical definition of relevance; *(ii)* an extended description of our implementation of the presented technique; *(iii)* a discussion on a new, general approach for counting activations, examining how it relates to previous approaches, and spelling out the open challenges of this problem.

The paper is structured as follows. In Section 2, we introduce the preliminary notions that we use to discuss the problem and propose our solution. In Section 3, we discuss the motivation behind our contribution. In Section 4, we describe how to determine whether a constraint is relevant to an execution trace. Section 5 shows how to identify activations using automata, and thus distinguish between vacuous and non-vacuous satisfactions. In Section 6, we describe how we implemented our approach. Section 7 reports on the results gathered by evaluating our approach on real-life logs. Section 8 discusses the problem of counting activations in a trace. Section 9 positions our research with respect to related work in the literature. Finally, Section 10 concludes the paper and spells out directions for future work.

2. Preliminaries

We start by introducing the preliminary notions used in the rest of the paper. The concepts defined hereafter constitute the background for the specification and analysis of declarative processes. Section 2.1 provides the basic terms to represent the execution of a process. Section 2.2 describes

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

the formal language with which declarative process semantics are expressed. Finally, Section 2.3 illustrates the DECLARE process modeling language.

2.1. Process Alphabet and Execution Traces

We fix a finite set Σ of tasks, i.e., atomic units of work in a process. This set provides the alphabet on top of which process execution traces are defined.

Definition 1 (Execution trace). An *(execution) trace* τ 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. □

In this work, we focus on *finite traces*, so as to reflect that each process execution is meant, sooner or later, to be completed. In the remainder of the paper, unless explicitly addressed, we will use the generic term *trace* to denote a finite execution trace.

The *length* of a trace τ, written $|\tau|$, is the number of tasks it contains: $|\langle \mathsf{t}_1, \ldots, \mathsf{t}_n \rangle| = n$. We use the standard *concatenation* operator over traces: the concatenation of trace $\tau_1 = \langle \mathsf{t}_1^1, \ldots, \mathsf{t}_m^1 \rangle$ with trace $\tau_2 = \langle \mathsf{t}_1^2, \ldots, \mathsf{t}_n^2 \rangle$, written $\tau_1 \cdot \tau_2$, is trace $\langle \mathsf{t}_1^1, \ldots, \mathsf{t}_m^1, \mathsf{t}_1^2, \ldots, \mathsf{t}_n^2 \rangle$. We define the concepts of *prefix* and *suffix* of a trace $\tau = \langle \mathsf{t}_1, \ldots, \mathsf{t}_n \rangle$ respectively as $\tau_{pre} = \langle \mathsf{t}_1, \ldots, \mathsf{t}_j \rangle$ and $\tau_{suf} = \langle \mathsf{t}_k, \ldots, \mathsf{t}_n \rangle$ for any $1 \leqslant j \leqslant n$ and $1 \leqslant k \leqslant n$. We use notation $\tau \cdot \mathsf{t}$ as a shortcut for $\tau \cdot \langle \mathsf{t} \rangle$. Finally, the i-th task of trace τ, for any $1 \leqslant i \leqslant |\tau|$, is denoted by $\tau(i)$.

2.2. Constraint-Based Process Modeling with Temporal Logics

In a declarative process model, constraints are used to express rules, best practices, and behavioral patterns that implicitly restrict the amount of accepted traces. Intuitively, an execution trace is accepted by or complies with, a declarative process model, if the trace *satisfies* all constraints contained in the model.

Usually, the formal underpinning for such intuitive notions is provided by temporal logics, whose models are indeed traces. In particular, formulae of the logic are used to capture constraints, and logical consequence to unambiguously define when a trace satisfies a constraint and is compliant with a declarative process model [20].

The most widely adopted logic for declarative process modeling is LTL over finite traces (LTL_f [21]). This logic is at the basis of concrete modeling

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

languages such as DECLARE. LTL$_f$ has the same syntax of LTL [22], but is interpreted on finite traces.

Definition 2 (LTL$_f$ formula). An LTL$_f$ formula is inductively defined as:

$$\varphi ::= \phi \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \bullet\varphi \mid \Diamond\varphi \mid \Box\varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2$$

where ϕ is a propositional formula over Σ, $\bigcirc$ is the *next* operator, $\bullet$ is *weak next*, $\Diamond$ is *eventually*, $\Box$ is *always*, $\mathcal{U}$ is *until*. □

Definition 3 (Semantics of LTL$_f$). The semantics of LTL$_f$ is defined by the satisfaction relation $\models$, inductively defined over the structure of the formula, given a trace τ and a position $1 \leqslant i \leqslant |\tau|$, as follows:

- $\tau, i \models \phi$ if $\tau(i)$ satisfies ϕ (in terms of propositional entailment);
- $\tau, i \models \neg\varphi$ if it is not the case that $\tau, i \models \varphi$;
- $\tau, i \models \varphi_1 \wedge \varphi_2$ if $\tau, i \models \varphi_1$, and $\tau, i \models \varphi_2$;
- $\tau, i \models \bigcirc\varphi$ if $i < |\tau|$, and $\tau, i+1 \models \varphi$;
- $\tau, i \models \bullet\varphi$ if either $i = |\tau|$, or $\tau, i \models \bigcirc\varphi$;
- $\tau, i \models \varphi_1 \,\mathcal{U}\, \varphi_2$ if there exists j such that $i \leq j \leq |\tau|$, $\tau, j \models \varphi_2$, and for every k such that $i \leq k < j$, we have $\tau, k \models \varphi_1$.

The other operators are obtained as combinations of the operators defined above (see [21]). □

Beyond LTL$_f$, more expressive logics have been exploited as well. Notable examples are regular expressions [23, 24, 25], which have been used to define an alternative semantics for DECLARE, and linear-dynamic logic over finite traces (LDL$_f$), exploited to monitor DECLARE patterns enriched with meta-constraints predicating over the truth values of other constraints [26]. Interestingly, LTL$_f$ is strictly less expressive than regular expressions, which, in turn, are expressively equivalent to LDL$_f$ and to monadic second-order logic over finite traces (MSO$_f$) [21, 27, 28].

The most widely adopted approach for consistency checking, enactment, monitoring, and mining of declarative process models is to leverage the automata-theoretic approach for temporal logics. This is done by exploiting the well-known result that a formula in each of the aforementioned logics can be captured by a corresponding deterministic finite-state automaton (DFA), which accepts all and only those traces that satisfy the formula.

To abstract away from the specific logic of interest, we employ the generic term *constraint* as a way to refer to a formula in any of the logics mentioned

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

above. We use LTL$_f$ in our examples just for presentation purposes. A *template* is a parametric LTL$_f$ formula. As pointed out above, all such logics can be characterized using DFAs. We use the term *constraint automaton* to refer to the DFA that captures a constraint of interest. When needed, we indicate a non-deterministic finite-state automaton with NFA.

Definition 4 (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 $\mathtt{t} \in \Sigma$, $\delta(s, \mathtt{t})$ is defined. □

Examples of algorithms that produce the constraint automaton of a given constraint expressed in LTL$_f$ or LDL$_f$ can be found in [21, 26, 29].

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 any $1 \leqslant i \leqslant |\tau|$, the i-th transition in the path matches with the i-th task in τ.

2.3. Declare

DECLARE is a declarative process modeling language originally introduced by Pesic and van der Aalst in [3, 6] to express rules over the control flow of a process. More recently, DECLARE has been extended to take into account other process perspectives that go beyond the pure control flow, such as time [30, 31] and data [32, 33, 34].

A DECLARE model consists of a set of constraints applied to tasks. Constraints, in turn, are based on templates. Templates have a graphical representation and their semantics can be formalized using different logics, the main one being LTL$_f$, making them verifiable and executable. Each constraint inherits the graphical representation and semantics from its template. The major benefit of using templates is that analysts do not have to be aware of the underlying logic-based formalization to understand the models. They

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

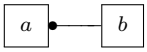
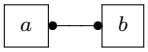
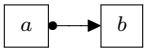
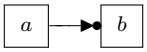
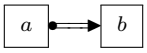
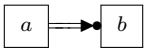
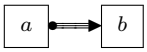
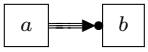
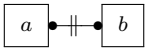
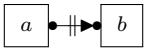
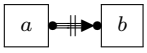
Template and explanation	Formalization	Graphical notation
Existence a occurs at least once	$\Diamond a$	
Absence2 a occurs at most once	$\neg\Diamond(a \wedge \bigcirc\Diamond a)$	
Responded existence If a occurs, then b occurs	$\Diamond a \rightarrow \Diamond b$	
Coexistence a occurs if and only if b occurs	$\Diamond a \leftrightarrow \Diamond b$	
Response If a occurs, then b occurs eventually afterwards	$\Box(a \rightarrow \Diamond b)$	
Precedence b occurs only if a occurred beforehand	$(\neg b \,\mathcal{U}\, a) \vee \Box(\neg b)$	
Alternate response If a occurs, then b occurs afterwards, before a recurs	$\Box(a \rightarrow \bigcirc(\neg a \,\mathcal{U}\, b))$	
Alternate precedence If b occurs, it is preceded by a and no other b can recur in between	$(\neg b \,\mathcal{U}\, a) \vee \Box(\neg b) \wedge \Box(b \rightarrow \bigcirc((\neg b \,\mathcal{U}\, a) \vee \Box(\neg b)))$	
Chain response If a occurs, then b occurs immediately after	$\Box(a \rightarrow \bigcirc b)$	
Chain precedence If b occurs, then a occurs immediately before	$\Box(\bigcirc b \rightarrow a)$	
Not coexistence a and b never occur together	$\Diamond a \rightarrow \neg\Diamond b$	
Not succession a never occurs before b	$\Box(a \rightarrow \neg\Diamond b)$	
Not chain succession a and b occur if and only if no b occurs immediately after a	$\Box(a \rightarrow \neg \bigcirc b)$	

Table 1: Graphical notation and LTL_f formalization of some standard DECLARE templates, referred to formal parameters a and b

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

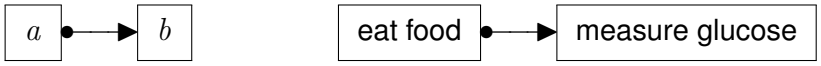
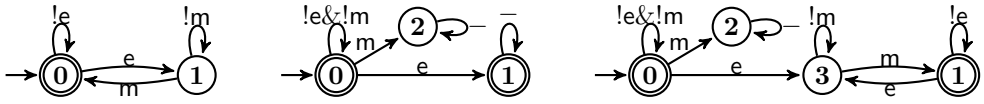


Figure 1: Response DECLARE template and a possible instantiation



(a) *response*(e, m) (b) *precedence*(e, m) (c) *succession*(e, m)

Figure 2: Automata of response, precedence, and succession constraints

work with the graphical representation of templates, while the underlying formulae remain hidden.

Table 1 summarizes some DECLARE templates, specifying their semantics by means of LTL_f formulae, and their graphical notation. The reader can refer to [3] for a full description of the language. All the templates of the list can be instantiated as constraints. For example, the *response* DECLARE 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 $\Box(\text{eat food} \rightarrow \Diamond\text{measure glucose})$.

Figure 2 shows the constraint automata representing the *response*, *precedence*, and *succession* DECLARE templates (which holds if and only if both response and precedence hold) grounded on two tasks eat food (e) and measure glucose (m). 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 !e is a shortcut for a set of transitions between the same two states, each one labeled with a task taken from $\Sigma \backslash \{\text{e}\}$. A transition labeled with "–" is a shortcut to denote all transitions corresponding to a task in Σ.

Notably, this compact notation allows us to use the same automaton regardless of Σ (just assuming that Σ contains the tasks mentioned by the constraint plus at least one additional task to express any "other" task). Following Definition 4, in Fig. 2, we do not *trim* the automata, i.e., we explicitly maintain all states, including the trap states from which it is not possible to reach any accepting state (such as state **2** in Fig. 2(b) and in Fig. 2(c)).

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

3. 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, the reason for satisfaction may be twofold. It could be the case that the trace actively interacts with the constraint or that the constraint is trivially satisfied because there is no interaction with the trace.

Consider the response constraint in Fig. 1. This constraint is satisfied in a trace where food is eaten and then the glucose level is eventually measured. In this case, the constraint is *relevant* to the trace, since, intuitively, it interacts with the trace creating an expectation towards measuring the glucose level when food is eaten. Such an expectation is finally fulfilled in fact. However, the same constraint is also satisfied by those traces where no food is ever eaten. This latter case is an example of the so-called *vacuous* satisfaction [8, 9]. Discriminating between these two situations is crucial in a variety of (declarative) process mining tasks, such as conformance checking and process discovery. In this section, we deepen the motivation behind our work, considering simple examples that show the need for considering relevance in process mining, and then reviewing the limitations of the two main existing approaches for vacuity detection.

3.1. Relevance and Activation Counting in Process Mining

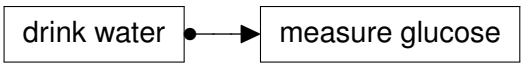
Consider an event log containing a single occurrence of trace:

$$\tau_w = \langle \mathsf{drink\ water}, \mathsf{measure\ glucose} \rangle,$$

and 10^3 occurrences of:

$$\tau_f = \langle \mathsf{eat\ food}, \mathsf{measure\ glucose} \rangle.$$

A common approach followed by different discovery algorithms for DECLARE is to extract all tasks present in the log, combine them to create constraints from the repertoire of patterns offered by DECLARE, and then computing their support in the log [25, 35, 36]. Support, in turn, depends on how many traces in the log satisfy or violate the constraint of interest. If one naively applies this approach without further considering constraint relevance, the resulting algorithm could return the response constraint φ_f shown in Fig. 1, and also:

$\varphi_w =$ 

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

with the same degree of importance, as they both have 100 % support in the log. However, it is a matter of fact that φ_f is more relevant than φ_w, since the input log is such that it satisfies φ_w only once non-vacuously, and 10^3 times in a trivial way. There is no possibility to tackle this fine-grained understanding of support unless relevance is explicitly taken into account. To do so, approaches exist for identifying *witnesses* for a given constraint, i.e., traces where the constraint is *non*-vacuously satisfied. These approaches either are based on a pre-analysis of the LTL$_f$ formula underlying the constraint, or are tailored to the standard DECLARE set of templates. However, the first approach is syntax-dependent, thus different formulations of the same constraint can lead to unequal results. The second approach, instead, cannot handle the case of custom constraints, which can be based on arbitrary, user-specified LTL$_f$ formulae.

Consider again constraint φ_f and its usage to monitor an evolving trace where new events are dynamically added at run-time, so as to track the execution of tasks. Consider now three different snapshots within the monitored execution:

- $\tau_0 = \varepsilon$;
- $\tau_1 = \langle \mathsf{eat\ food}, \mathsf{measure\ glucose} \rangle$;
- $\tau_2 = \tau_1 \cdot \tau_1 \cdot \tau_1 \cdot \tau_1$.

In all these snapshots, the monitored trace satisfies φ_f, but intuitively it does so in an increasingly relevant way. This can clearly be seen if one applies the intuition that φ_f is *activated* every time its source task eat food is executed, and computes a health indicator combining the number of times φ_f is activated with the number of times φ_f is brought back to a satisfied state. This requires to go beyond the mere notion of constraint satisfaction, which would simply judge τ_0, τ_1, and τ_2 as compliant with φ_f, without further distinctions.

3.2. Syntax-Dependent Vacuity Detection

The notion of constraint relevance discussed so far is intimately connected to the well-known notion of *vacuity detection* in model checking. Specifically, [9] introduces an approach for vacuity detection in temporal model checking for LTL (over infinite traces) to determine whether a given trace is a witness for an LTL formula. The method extends an LTL formula φ to a new formula $witness(\varphi)$ that, when satisfied, ensures that the original formula φ is non-vacuously satisfied. In this way, whenever a trace complies with $witness(\varphi)$, it complies with φ, and φ is also non-vacuously satisfied in the trace. Formula

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

$witness(\varphi)$ is generated by considering that a trace τ satisfies φ non-vacuously if τ satisfies φ, and τ satisfies a set of additional conditions that guarantee that every subformula of φ does really affect the truth value of φ in τ. 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, for example, the response constraint $\Box(\mathsf{eat\ food} \rightarrow \Diamond\mathsf{measure\ glucose})$. The vacuity detection condition is $\Diamond\mathsf{eat\ food}$, so that the 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 can seamlessly be lifted to LTL$_f$, and it was indeed applied to DECLARE in [35] for vacuity detection in the context of process discovery, so as to tackle the issues discussed in Section 3.1. However, the algorithm introduced in [9] may generate different results for LTL$_f$ formulae that are semantically equivalent but syntactically different. Consider, for instance, the following logically equivalent formulae (expressing the alternate response DECLARE template):

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

$$\varphi' \;=\; \Box(a \rightarrow \bigcirc(\neg a\,\mathcal{U}\,b)).$$

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

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

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

$$\neg(\Box(\neg a \vee \Diamond b) \vee \Diamond(a \wedge \neg \bigcirc (\neg a\,\mathcal{U}\,b))).$$

Considering that the conjunction of $\neg(\Box(\neg a \vee \Diamond b) \vee \Diamond(a \wedge \neg \bigcirc (\neg a\,\mathcal{U}\,b))$ with φ is always false, this is sufficient to conclude that $witness(\varphi) = \mathit{false}$.

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

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

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

$$\psi'_6 = a(2), \text{ and } \psi'_7 = b.$$

The corresponding vacuity detection conditions are: *(i) true* for ψ'_1 and ψ'_2; *(ii)* $\neg(\Box(\bigcirc(\neg a \,\mathcal{U}\, b))) \equiv \Diamond(\neg \bigcirc (\neg a \,\mathcal{U}\, b))$ for ψ'_3; *(iii)* $\neg(\Box(\neg a \vee \mathit{false})) \equiv \Diamond a$ for ψ'_4 and ψ'_5; *(iv)* $\neg(\Box(\neg a \vee \bigcirc(\mathit{false}\,\mathcal{U}\, b))) \equiv \Diamond(a \wedge \neg \bigcirc (b))$ for ψ'_6, whose conjunction with φ' is not always false.

Declarative languages such as 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 with respect to these constraints.

3.3. Ad-Hoc Approaches

An alternative to the syntax-dependent vacuity detection is to restrict the constraint language considering a pre-defined family of constraint patterns (e.g., DECLARE) rather than a full-fledged temporal logic, and provide ad-hoc approaches to vacuity detection, explicitly handling each constraint pattern. For example, [10, 12, 13, 37] introduce ad hoc approaches to vacuity detection for DECLARE. However, these approaches fail when DECLARE is extended with new templates, a feature that has been deemed essential, since the very first seminal papers on this language [6]. The following example introduces a template that cannot be expressed by using standard DECLARE.

Example 1. We call *progression* of a tuple of tasks $\langle \mathsf{t}_1, \ldots, \mathsf{t}_n \rangle$ a sequence that starts with t_1, 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 tasks, 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 formally be 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

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

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:

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

To overcome the issue of vacuity detection, the algorithms described in [10, 37] use an adapted version of the Apriori algorithm first proposed in [38] as a pre-processing step to detect sets of tasks that are often co-occurring in the traces. Thereupon, they compute the support of only those constraints that are exerted on tasks in such sets.

However, even if this approach is reportedly effective for standard DECLARE templates, it would not suffice for a custom DECLARE constraint such as the progression response described earlier. In fact, frequent task sets do not account for the mutual order of occurrence of the items. Therefore, even if fin, pay, del, rec always occur in this specific order in the traces of the event log, the progression response constraints expressed over any permutation of those tasks would all be considered as satisfied. However, only the one mentioned in Example 1 would be relevant, whilst the other $\left(\!\binom{4}{2}\!\right) \times \left(\left(\!\binom{4}{2}\!\right) - 1\right) - 1 = 89$ constraints would only be vacuously satisfied. The calculation is made on the basis of the following: all possible 2-combinations with repetitions of tasks from the set of size 4 ($\{\mathsf{fin}, \mathsf{pay}, \mathsf{del}, \mathsf{rec}\}$) is eligible as a source tuple, e.g., $\langle \mathsf{pay}, \mathsf{del} \rangle$; for every source, all possible 2-combinations with repetitions of tasks from that set is a possible target tuple, excluding the combination that coincides with the source (in the example, all tuples but $\langle \mathsf{pay}, \mathsf{del} \rangle$ are acceptable). The reason for this last exclusion can intuitively be justified on the basis of the observation reported in [39]: a (progression) response constraint that has coinciding source and target is unsatisfiable over finite traces, because every occurrence of the source would require the occurrence of the same tuple later on, thus activating the constraint again, and requiring another occurrence of that tuple, etc.

Another common approach to vacuity detection consists in pruning those constraints that are assigned with a low confidence, i.e., the support scaled by the number of traces in which the activation of the constraint occurs [10, 12, 13]. However, activations are well defined only for standard DECLARE templates (e.g., an activation of response$(\mathsf{e}, \mathsf{m}) = \Box(\mathsf{e} \rightarrow \Diamond\mathsf{m})$ corresponds to an occurrence of e in a trace). Instead, no such concept is formally defined for

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

custom templates such as the aforementioned progression response. Therefore, the confidence-based approach would not be applicable, too.

The definition of relevance and activation counting for the progression constraint shown in Example 1 can neither be hijacked from the ad-hoc definitions provided for the core DECLARE patterns, nor easily obtained using human ingenuity. This is why we aim at achieving a semantical, general treatment of vacuity, making it possible to seamlessly apply declarative process mining techniques using constraint patterns that go beyond standard DECLARE such as the progression response of Example 1.

4. Relevance and 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. We focus here on relevance, without attempting to "measure" the degree of relevance by counting activations. The problem of counting will be discussed in Section 8.

Our approach has three distinctive features:

- It is *fully semantical*, in the sense that it detects when a trace is a witness for a constraint, in a way that is completely independent of the specific syntactic form of the constraint.
- It is *general*, in the sense that it does not focus on a specific constraint language, but seamlessly work for all the aforementioned temporal logics, including LTL_f, LDL_f, and MSO_f.
- *It can seamlessly be applied at run-time or a posteriori*, i.e., it can directly be used to assess relevance of running, evolving traces.

Our approach consists of three steps. Section 4.1 describes the first step, in which we gain more details about the different states in which a constraint can be, going beyond the coarse-grained characterization of satisfied vs. violated. Section 4.2 illustrates the second step, in which we leverage those additional details to semantically characterize the notion of "witness", which in turn constitutes the basis for understanding whether a trace activates a constraint or not. Finally, Section 5 details the last step, in which we mirror this approach into the automata-based characterization of the aforementioned logics to make it operational.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

4.1. Activation States and Relevant Task Executions

To understand in detail how a trace relates to a constraint, we build on the four truth values provided by RV-LTL [40], which considers LTL in the light of run-time verification. This approach has recently been adopted for conformance checking and monitoring of LTL_f and LDL_f constraints [26, 41, 42]. RV-LTL brings two main advantages in the context of this paper. On the one hand, it makes our approach directly applicable to monitoring and online operational support [43, 44]. On the other hand, it provides the basis to check whether an execution trace actively interacts with a constraint or not.

Definition 5 (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 is satisfied by the current trace (i.e., $\tau \models \varphi$ as per Definition 3), 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., $\tau \not\models \varphi$), 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 to the constraint, since such truth values are indeed unmodifiable. Temporary states instead are those for which relevant 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.

Definition 6 (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. □

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

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 in a finer-grained way by checking whether the task execution causes at least one change in the set of permitted tasks.

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

Definition 7 (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., $V \in \{\mathtt{ps}, \mathtt{pv}, \mathtt{ts}, \mathtt{tv}\}$, and $\Lambda \subseteq \Sigma$ is a set of *permitted* tasks. □

As per Definition 5 and Definition 6, not all activation states are meaningful. For example, we know that if the current RV-LTL value is $\mathtt{pv}$, then no task can be permitted. We systematize this notion by identifying those activation states that are "legal".

Definition 8 (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 $\emptyset \subseteq \Lambda \subseteq \Sigma$;
- $\langle \mathtt{tv}, \Lambda \rangle$, 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 legal activation states over Σ.

Definition 9 (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 $[\tau \models \varphi]_{RV} = v$ for $v \in \{\mathtt{ps}, \mathtt{pv}, \mathtt{ts}, \mathtt{tv}\}$ (as per Definition 5); (2) for every $\mathtt{t} \in \Sigma$, we have $\mathtt{t} \in \Lambda$ iff $\mathtt{t}$ is permitted by φ after τ (as per Definition 6). The *initial activation state* is the activation state computed for $\tau = \varepsilon$.

Trace activation states enjoy the following property.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

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

Example 2. Consider the response constraint $\varphi_f = \Box(\mathsf{eat\ food} \rightarrow \Diamond\mathsf{measure\ glucose})$ over Σ. The initial activation state of φ_f is $\langle \mathtt{ts}, \Sigma\rangle$: all tasks are permitted, and φ_f is temporarily satisfied, since there are traces culminating in the violation of the constraint. Consider now the trace $\langle \mathsf{eat\ food}\rangle$: we get $actState_{\varphi_f}(\mathsf{eat\ food}) = \langle \mathtt{tv}, \Sigma\rangle$. Indeed, all tasks are still permitted, but φ_f is temporarily violated because it expects the future occurrence of measure glucose. ■

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 to the constraint or not.

Definition 10 (Relevant task execution). Let φ be a constraint over Σ, $\mathsf{t} \in \Sigma$ be a task, and τ an execution trace over Σ^*. Let $\langle V, \Lambda\rangle = actState_\varphi(\tau)$ and $\langle V', \Lambda'\rangle = actState_\varphi(\tau \cdot \mathsf{t})$ respectively be the trace activation states of φ in τ and the one obtained as the result of executing t after τ. We say that t *is a relevant execution for φ after τ* (or equivalently that t *is a relevant execution for φ in $actState_\varphi(\tau)$*) if $V \neq V'$ or $\Lambda \neq \Lambda'$. □

4.2. Activations, Vacuity, and Witnesses

Definition 10 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.

Definition 11 (Activation). A constraint φ over Σ is *activated* by a trace τ over Σ^* if there exists $\mathsf{t} \in \Sigma$ s.t.: (1) $\tau = \tau_{pre} \cdot \mathsf{t} \cdot \tau_{suf}$; (2) t is a relevant execution for φ after τ_{pre} (as per Definition 10). □

Example 3. Consider the response constraint of Example 2, and the execution trace $\tau = \langle \mathsf{d}, \mathsf{m}, \mathsf{e}, \mathsf{m}, \mathsf{m}, \mathsf{e}, \mathsf{e}, \mathsf{m}\rangle$ (where d stands for drink water). By making trace activation states along τ explicit, we get:

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

$$\langle \mathtt{ts}, \Sigma \rangle \ \mathtt{d} \ \langle \mathtt{ts}, \Sigma \rangle \ \mathtt{m} \ \langle \mathtt{ts}, \Sigma \rangle \ \underset{\uparrow}{\mathtt{e}} \ \langle \mathtt{tv}, \Sigma \rangle \ \underset{\uparrow}{\mathtt{m}} \ \langle \mathtt{ts}, \Sigma \rangle \ \mathtt{m} \ \langle \mathtt{ts}, \Sigma \rangle \ \underset{\uparrow}{\mathtt{e}} \ \langle \mathtt{tv}, \Sigma \rangle \ \mathtt{e} \ \langle \mathtt{tv}, \Sigma \rangle \ \underset{\uparrow}{\mathtt{m}} \ \langle \mathtt{ts}, \Sigma \rangle$$

Arrows indicate the relevant task executions. In fact, the first relevant task execution is e, 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 m is also relevant, because it triggers the opposite change. The second following m, instead, is irrelevant, because it keeps the activation state unchanged. A similar pattern can be recognized for the following two es: the first one is relevant, the second one is not. Notice that τ complies with φ_f. Now, consider the *not coexistence* DECLARE constraint $\varphi_{nc} = \neg(\Diamond \mathsf{e} \wedge \Diamond \mathsf{m})$, and the same execution trace τ as before. We obtain:

$$\langle \mathtt{ts}, \Sigma \rangle \ \mathtt{d} \ \langle \mathtt{ts}, \Sigma \rangle \ \underset{\uparrow}{\mathtt{m}} \ \langle \mathtt{ts}, \Sigma \setminus \{\mathtt{e}\} \rangle \ \underset{\uparrow}{\mathtt{e}} \ \langle \mathtt{pv}, \emptyset \rangle \ \mathtt{m} \ \langle \mathtt{pv}, \emptyset \rangle \ \mathtt{m} \ \langle \mathtt{pv}, \emptyset \rangle \ \ldots \ \langle \mathtt{pv}, \emptyset \rangle$$

The constraint is, in fact, initially temporarily satisfied, and every task is permitted. In the second position of τ, the relevant execution of m 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 e is also relevant, because it causes a permanent violation of the constraint. A permanent violation corresponds to an irreversible activation state, therefore all consequent task executions are irrelevant independently on how the trace continues. ■

Considering the two constraints in Example 3, in one case, the trace contains relevant task executions, and satisfies the constraint (so that the trace is a witness for the constraint), whereas in the second case the trace violates the constraint. In the case of satisfaction, two cases may arise: either the trace satisfies the constraint and the constraint is relevant to the trace, 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.

Definition 12 ((Non-)vacuous satisfaction/witness). Let φ be a constraint over Σ, and τ a trace over Σ^* that complies with φ (as per Definition 5). If φ is activated by τ (as per Definition 11), then τ *non-vacuously satisfies* φ and we call τ a *witness* for φ. Otherwise τ *vacuously satisfies* φ. □

Example 4. In Example 3, trace τ activates both the response (φ_f) and not coexistence (φ_{nc}) constraints. Now, consider the execution trace $\tau_2 =$

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

$\langle \mathsf{d}, \mathsf{d}, \mathsf{m}, \mathsf{d}, \mathsf{m} \rangle$. Since τ_2 contains m, τ_2 activates φ_{nc}: when the first occurrence of m happens, the set of permitted tasks moves from the whole Σ to $\Sigma \setminus \{\mathsf{e}\}$. Furthermore, τ_2 does not contain both e and m, therefore it complies with φ_{nc}. Consequently, we have that τ_2 non-vacuously satisfies φ_{nc}, i.e., τ_2 is a witness for φ_{nc}. Since τ_2 does not contain occurrences of e, it does not activate the response constraint. More specifically, τ_2 never changes the initial activation state of φ_f, which corresponds to $\langle \mathtt{ts}, \Sigma \rangle$. This also shows that τ_2 complies with φ_f, and, therefore, that τ_2 vacuously satisfies φ_f. ∎

5. Checking Relevance Using Automata

We now make vacuity detection operational by leveraging the automata-theoretic approach.

5.1. Activation-Aware Automaton

To check the relevance of a constraint to a trace, we exploit a combination of the automata construction technique in [26] with the notion of *colored automata* [41]. Colored automata augment DFAs 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 state; on the other hand, relevant executions are marked in the automaton by "coloring" their corresponding transitions. We consequently obtain the following type of automaton.

Definition 13 (Activation-Aware Automaton). The *activation-aware automaton* $\mathcal{A}^{act}_{\varphi}$ of an LTL$_f$ 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 of φ (as per Definition 4 and [26]);
- $\alpha : S \longrightarrow \mathbb{S}_\Sigma$ is the function that maps each state $s \in S$ to the corresponding activation state $\alpha(s) = \langle V, \Lambda \rangle$, where:
  1. $V = \mathtt{ts}$ iff $s \in F$, and there exists a state $s' \in S$ s.t. $\delta^*(s, s')$ and $s' \notin F$;
  2. $V = \mathtt{ps}$ iff $s \in F$, and for every state $s' \in S$ s.t. $\delta^*(s, s')$, we have $s' \in F$;
  3. $V = \mathtt{tv}$ iff $s \notin F$, and there exists a state $s' \in S$ s.t. $\delta^*(s, s')$ and $s' \in F$;

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

4. $V = \mathtt{pv}$ iff $s \notin F$, and for every state $s' \in S$ s.t. $\delta^*(s, s')$, we have $s' \notin F$;
5. Λ contains task $\mathtt{t} \in \Sigma$ iff there exists a state $s' \in S$ s.t. $s' = \delta(s, \mathtt{t})$ and $\alpha(s')$ has an RV-LTL truth value different from $\mathtt{pv}$.

- $\rho \subseteq \mathit{Domain}(\delta)$ is the set of transitions in δ that are relevant to φ, i.e.: $\rho = \{\langle s, \mathtt{t}\rangle \mid \langle s, \mathtt{t}\rangle \in \mathit{Domain}(\delta)$ and $\mathtt{t}$ is a relevant execution for φ in $\alpha(s)\}$

□

Consider a trace τ, a constraint φ and the activation-aware automaton $\mathcal{A}_\varphi^{act}$ of φ. By definition, there exists one and only one state s of $\mathcal{A}_\varphi^{act}$ obtained by replaying τ over $\mathcal{A}_\varphi^{act}$. Intuitively, the specification of α in Definition 13 makes Definition 5 operational, by drawing the following parallel between the conditions in Definition 5, and those in Definition 13:

- "φ satisfied by τ" translates into checking that s is an accepting state;
- "φ violated by τ" translates into checking that s is non-accepting;
- "φ remains satisfied for every possible continuation of τ" translates into checking that all reachable states from s are accepting;
- "φ remains violated for every possible continuation of τ" translates into checking that all reachable states from s are non-accepting;
- "there exists a continuation of τ leading to a violation of φ" translates into checking that it is possible to reach a non-accepting state from s;
- "there exists a continuation of τ leading to a satisfaction of φ" translates into checking that it is possible to reach an accepting state from s.

In addition, condition 5 of Definition 13 correctly reconstructs the notion of permitted task as a task that does not lead to a permanent violation of φ, i.e., to a successor state of s that is labeled with a permanent violation. As a consequence of this, the set of permitted tasks in s simply corresponds to all labels associated to outgoing transitions from s in the trimmed version of $\mathcal{A}_\varphi^{act}$ (i.e., the version of $\mathcal{A}_\varphi^{act}$ where states associated to a permanent violation are actually removed).

We now prove that the activation-aware automaton correctly reconstructs activation and relevance as defined in Section 4.2. We recall that our approach applies to all those logics that have automata-theoretic characterization in terms of finite state automata. Therefore, we restrict the following discussion to LTL_f although the results could seamlessly be extended to more expressive logics, such as regular expressions or LDL_f.

Theorem 1. *Let φ be an LTL_f formula over Σ, and $\mathcal{A}_\varphi^{act} = \langle \Sigma, S, s_0, \delta, F, \alpha, \rho\rangle$ the activation-aware automaton of φ. Let $\tau = \langle \mathtt{t}_1 \cdots \mathtt{t}_n\rangle$ be a non-empty, finite*

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

trace over Σ*, and* $s_0 \cdots s_n$ *the sequence of states such that* $\delta(s_{i-1}, \mathsf{t}_i) = s_i$ *for* $1 \leqslant i \leqslant n$.[1] *Then, the following two properties hold:*

- $\alpha(s_n) = actState_\varphi(\tau)$*, i.e.,* α *correctly produces the trace activation state of* φ *in* τ *(in the sense of Definition 9);*
- τ *non-vacuously satisfies* φ *(in the sense of Definition 12) if and only if* $\langle s_{i-1}, \mathsf{t_i} \rangle \in \rho$ *for some* $1 \leqslant i \leqslant n$.

Proof 2. From the correctness of the constraint automaton construction (as per Definition 4 and [26]), we know that τ satisfies φ iff τ is accepted by $\mathcal{A}_\varphi^{act}$ (i.e., iff $s_n \in F$). This corresponds to the notion of compliance in Definition 5. The proof of the first claim is then obtained by observing that all tests in Definition 13, which characterize the RV-LTL values and permitted tasks of the automaton states, perfectly mirror Definition 5 and 6. In particular, notice that the labeling of states with RV-LTL values agrees with the construction of "local colored automata" in [41], proven to be correct in [26].

The second claim immediately follows from the first one, by observing that: *(i)* in Definition 13, ρ is constructed by relying on the notion of relevance in a given activation state (as per Definition 10) as dictated by Definition 11; *(ii)* Definition 12 directly builds upon Definition 11. □

It has to be noted that the notion of activation-aware automaton is affected if, instead of considering as constraint automaton the unique, minimal deterministic finite-state automaton for the constraint of interest, one considers instead a non-deterministic finite-state automaton or a non-minimal deterministic finite-state automaton.

We first observe that it is essential for the construction of the activation-aware automaton that the constraint automaton of the formula of interest is actually deterministic. In fact, for a non-deterministic automaton it is not possible, in general, to unambiguously associate a state to a corresponding RV-LTL value. On the other hand, we know that each non-deterministic finite-state automaton can be determinized [45], consequently resolving such potential ambiguity.

In addition, our approach is *robust* with respect to non-minimal equivalent constraint automata. In fact, one could ask whether two equivalent finite-state automata would lead to non-equivalent corresponding activation-aware

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

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

automata, i.e., automata that would assign different activation states to the same trace. This is not possible, due to the simple argument that:

- Equivalent deterministic finite-state automata are also equivalent to the same minimal deterministic finite-state automaton (which is unique up to isomorphism).
- The necessary and sufficient condition to collapse two states s_1 and s_2 in a non-minimal deterministic finite-state automaton is that for each (suffix) trace τ, the state obtained by replaying τ from s_1 is accepting if and only if the state obtained by replaying τ from s_2 is accepting. As a consequence, the RV-LTL truth value, as well as the set of permitted tasks, coincide for s_1 and s_2, i.e., s_1 and s_2 correspond to the same activation state.

5.2. Construction of the Activation-Aware Automaton

We now comment on how to effectively construct the activation-aware automaton of a given constraint φ. This can be done as a rather straightforward application of Definition 13, retaining the same computational complexity of the standard automaton construction. More specifically, the activation-aware automaton can be obtained by applying the following procedure:

1. The constraint automaton $\mathcal{A}_\varphi$ of φ is built by applying the LDL$_f$2NFA procedure of [26], and then the standard determinization procedure to the obtained automaton (thus getting a DFA). $\mathcal{A}_\varphi$ is then enriched with α and ρ from Definition 13 so as to obtain the corresponding activation-aware automaton.
2. Function α is constructed in two iterations.
   (a) In the first iteration, the RV-LTL truth value of each state in $\mathcal{A}_\varphi$ is computed. This is done by picking each state s of $\mathcal{A}_\varphi$ and checking whether: *(i)* s is accepting or not, *(ii)* s may reach an accepting state, and *(iii)* s may reach a non-accepting state. Depending on the answers obtained from these three checks, the RV-LTL truth value of s is unambiguously determined. Notice that this iteration can be performed in time that is polynomial in the size of the automaton. This level of complexity will henceforth be denoted as PTIME for the sake of brevity.
   (b) The second iteration goes again over each state s of $\mathcal{A}_\varphi$ using the RV-LTL truth value of s and of its successor states so as to compute the set of permitted tasks. This can simply be done by picking each outgoing transition of s, and adding the corresponding label to the set of permitted tasks in s provided that the transition

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

points to a state whose RV-LTL value is different from permanent violation. Also this iteration can be performed 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 2 and Fig. 3 respectively list the activation-aware automata of some standard DECLARE patterns and the activation-aware automaton of 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 Definition 10. 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.

Example 5. Consider the precedence DECLARE constraint φ_p expressing that measure glucose cannot occur until eat food occurs. Its constraint automaton $\mathcal{A}_{\varphi_p}$ is shown in Fig. 2(b). The corresponding activation-aware automaton $\mathcal{A}^{act}_{\varphi_p}$ is depicted in Table 2. We explain how $\mathcal{A}^{act}_{\varphi_p}$ is obtained from $\mathcal{A}_{\varphi_p}$.

We start with the RV-LTL truth values computed for the three states of $\mathcal{A}_{\varphi_p}$:

- state **0** is temporarily satisfied, since state **0** is an accepting state, but it is connected to state **2**, which is non-accepting;
- state **1** is permanently satisfied, since state **1** is an accepting "trap" state only connected to itself;
- state **2** is permanently violated, since state **2** is a non-accepting "trap" state only connected to itself (this state would in fact be filtered out if one trims $\mathcal{A}_{\varphi_p}$).

Consequently:

- the permitted tasks in state **0** are all tasks in Σ except measure glucose, since its execution would lead to state **2**, which is permanently violated;
- the permitted tasks in states **1** and **2** are respectively the entire set Σ and the empty set, since they are respectively associated to a permanent satisfaction/violation. ■

Example 6. Consider the response DECLARE constraint φ_f expressing that whenever eat food occurs, measure glucose is expected to eventually occur. Its constraint automaton $\mathcal{A}_{\varphi_f}$ is shown in Fig. 2(a). The corresponding activation-aware automaton $\mathcal{A}^{act}_{\varphi_f}$ is depicted in Table 2. We explain how $\mathcal{A}^{act}_{\varphi_f}$ is obtained from $\mathcal{A}_{\varphi_f}$.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

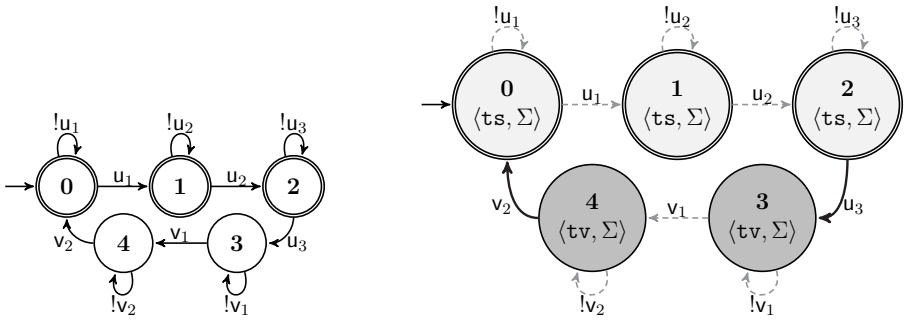


Figure 3: Constraint automaton and activation-aware automaton of a *progression response* constraint (with three sources and two targets)

We start with the RV-LTL truth values computed for the two states of $\mathcal{A}_{\varphi_f}$:

- state **0** is temporarily satisfied, since state **0** is an accepting state, but it is connected to state **1**, which is non-accepting;
- state **1** is temporarily violated, since state **0** is non-accepting, but it is connected to state **0**, which is accepting.

Since there is no state in $\mathcal{A}_{\varphi_f}$ associated to a permanent violation, for both states **0** and **1** the permitted tasks are all tasks in Σ. ■

6. Implementation

The implementation of our approach is based on activation-aware automata. The rationale is to replay the traces on the labeled automata that represent the constraints under analysis, to check the traversed transitions and the state reached at the end. The reached state informs about the compliance of the trace with respect to the constraint. The traversed transitions specify whether the compliance is vacuous or not.

The implementation of our approach consists of two phases. Section 6.1 describes the first phase, in which the activation-aware automata are computed. Section 6.2 illustrates the second phase, in which the traces of an event log are replayed on those automata.

6.1. Generation of Activation-Aware Template Automata

We resort on the technique described in [26] to generate constraint automata. Thereupon, we enrich the obtained automata to associate every

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

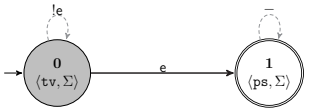
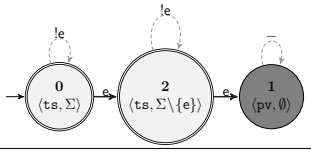
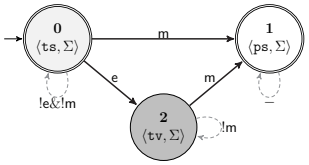
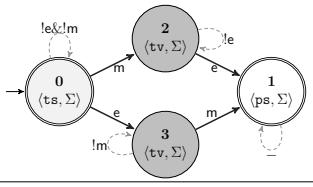
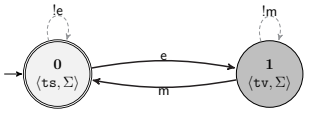
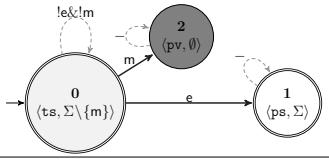
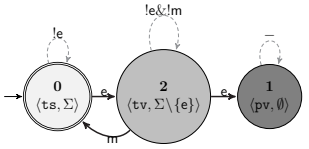
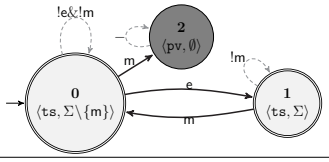
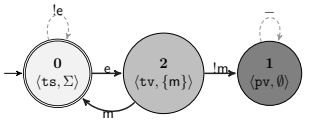
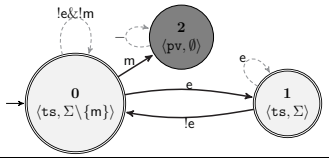
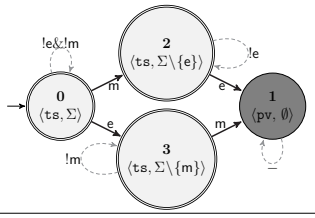
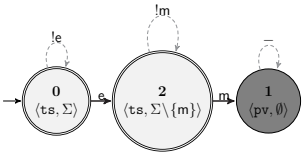
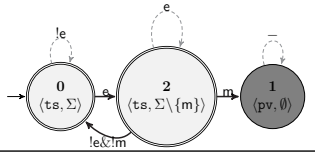
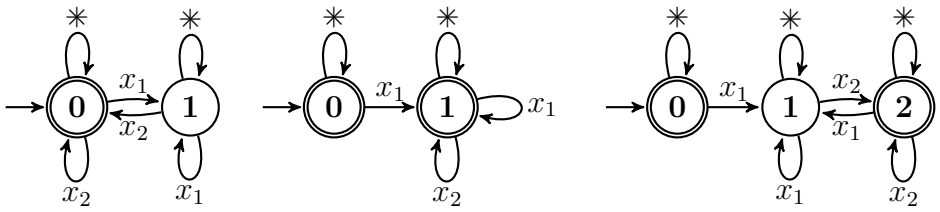
TEMPLATE	ACTIVATION-AWARE AUTOMATON	TEMPLATE	ACTIVATION-AWARE AUTOMATON
Existence		**Absence2**	
Responded existence		**Coexistence**	
Response		**Precedence**	
Alternate response		**Alternate precedence**	
Chain response		**Chain precedence**	
Not coexistence		**Not succession**	
Not chain succession			

Table 2: Extended constraint automata for some standard DECLARE templates applied to tasks e and m

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011



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

Figure 4: Minimized deterministic automata for the *precedence*, *response* and *succession* DECLARE templates

state to its activation state. Automata operations are, however, reportedly memory-intensive. In order to save space, we adapt the approach by leveraging two strategies.

First, minimized (hence trimmed) automata are initially used, to decrease the amount of states; states for formerly trimmed transitions are artificially added at the end of the automata generation. In particular, we reduce the number of states of the original translation through the algorithm of Hopcroft [46], which formally guarantees the returned automaton to be unique for equivalent input automata up to isomorphism, i.e., except for a renaming of states [47].

Second, the symbols of the alphabet of activation-aware automata are made parametric, so as to have one automaton per template, rather than one per constraint. We call this type of automaton *template automaton*. The use of template automata instead of constraint automata becomes vital when there is the necessity to check the compliance with respect to large sets of constraints. This is the case of many discovery algorithms, which instantiate each template with all possible combinations of tasks that can be found in an event log, and check their compliance with respect to the log [25, 35, 36]. In the following, we describe in detail the implemented approach.

The labeling domain of template automata is in our implementation such that one distinct symbol x_i is included for every parameter of the template, and another one is added to act as a wild-card ($\ast$). "$\ast$" is a symbol denoting an equivalence class for all those symbols that are different from any x_i. The labeling domain of a template automaton of the DECLARE template response(x_1, x_2) has thus three symbols: $\{x_1, x_2, \ast\}$. We henceforth denote such a domain as $\Sigma^x_\ast$. The template automata of the precedence, response, and succession DECLARE templates are shown in Fig. 4. As an example, we can assume to have a set of tasks (input alphabet) consisting of $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}$. If

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

we consider the response constraint response(a, b), then a is assigned to the first parameter of the response template, x_1, and b to its second parameter, x_2. $*$ thus signifies any other task but a and b, i.e., c and d.

Denoting the mapping function for the assignment of parameters to tasks as $\mu : \Sigma_*^x \to \Sigma$, we remark that μ is surjective, and (only for standard DECLARE templates) injective for $\Sigma_*^x \setminus \{*\}$. In other words, for standard DECLARE, we assume that different parameters are not assigned to the same task, in line with the limitation posed by the definition of DECLARE constraints. A detailed discussion about the unsatisfiability over finite traces of constraints violating this restriction, such as response(a, a), can be found in [39]. For other templates, such as the progression response, the injectivity does not hold.

Once the (trimmed) template automata are obtained, their states are enriched so as to bear information about the activation states. As per Definition 5, the activation state is a pair, consisting of *(i)* a truth value in RV-LTL, V, and *(ii)* the set of permitted tasks, Λ. The truth value of every state is assigned as follows:

$V =$ pv if the state is non-accepting and has no outgoing transition;

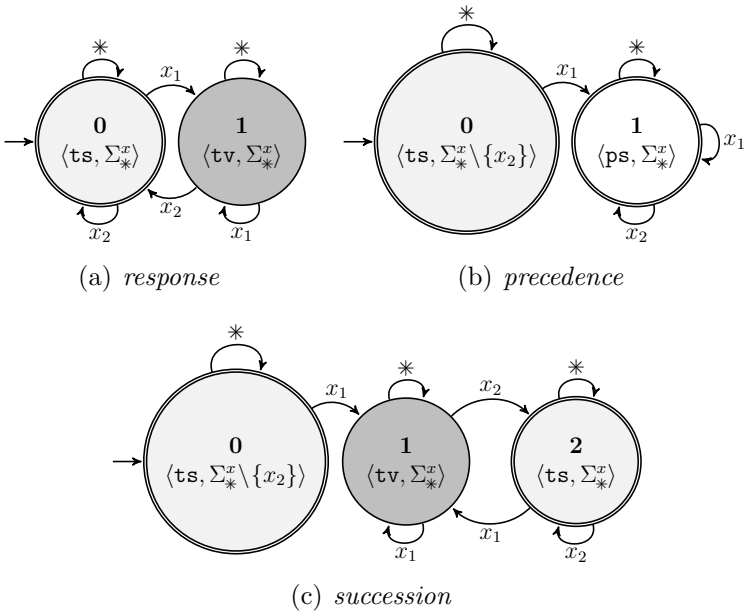
$V =$ tv if the state is non-accepting yet it has at least one outgoing transition;

$V =$ ts if the state is accepting and has either no outgoing transitions, or at least one non-looping outgoing transition;

$V =$ ps if the state is accepting and only has looping transitions.

Note that no state in the template automaton can be assigned a truth value of pv, unless the automaton has only an initial non-accepting state and no outgoing transitions. Indeed, transitions leading to non-accepting states that are sink nodes do not appear in the trimmed automaton. In other words, the trimming leads to automata that only keep the permitted tasks as transitions going out of states, as it can be seen by comparing Figs. 4(b) and 4(c) to Figs. 2(b) and 2(c), respectively. Owing to this, it is sufficient to have at least one outgoing transition for a non-accepting state to be assigned a truth value of tv. Examples are states labeled as **1** in the automata in Fig. 4(a) and Fig. 4(c): both for response(x_1, x_2) and succession(x_1, x_2), the occurrence of the task assigned to x_1 makes the constraints temporarily violated, because a following occurrence of the x_2-task can make the trace still compliant. An accepting state whose outgoing transitions are all looping is assigned with ps because it marks a stage at which the trace is compliant, no matter what follows. This is the case, e.g., for state **1** in the automaton in Fig. 4(b): once

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011



(a) *response* (b) *precedence*

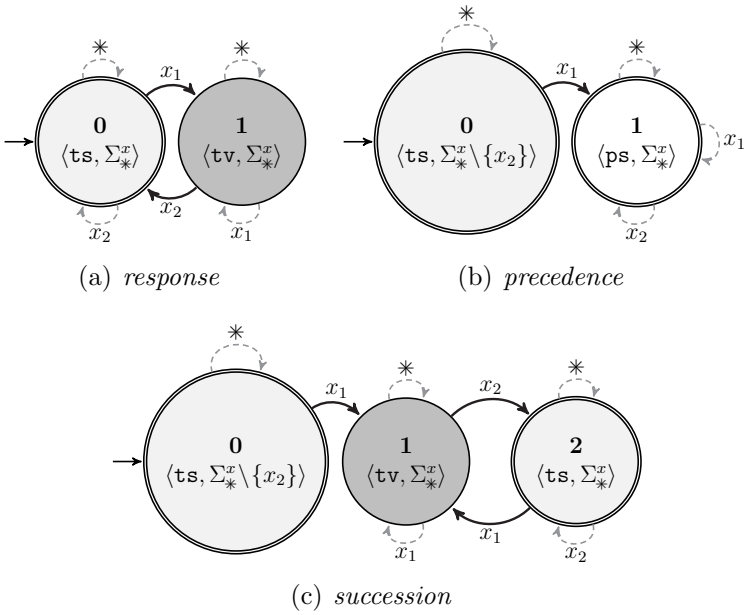
(c) *succession*

Figure 5: Minimized deterministic automata for the *precedence*, *response* and *succession* DECLARE templates with activation states

the task assigned to x_1 for precedence(x_1, x_2) occurs, the constraint becomes permanently satisfied. An accepting state with at least one non-looping transition is assigned a truth value of `ts`, because the automaton is minimal [46]. This is the case, e.g., for state **0** in the automata in Fig. 4(a) and Fig. 4(b), and states **0** and **2** in the automaton in Fig. 4(c).

At this stage, the calculation of the set of permitted tasks is rather straightforward: for every state, the permitted tasks correspond to all the labels of the outgoing transitions. This is due to the fact that *(i)* the outgoing transitions leading to non-accepting sink-node states do not appear in trimmed minimal automata, and *(ii)* no outgoing transitions share the same label in a deterministic automaton. The outcome of the aforementioned steps over the template automata in Fig. 4 are depicted in Fig. 5. Since the evaluation of the activation state is made by locally inspecting the state under analysis and its outgoing transitions, no state needs to be processed more than once. A traversal of the automaton starting from the initial state is thus sufficient, and, in the worst case, all transitions have to be gone through once. Therefore, the time complexity of the computation of the activation states of a template

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011



(a) *response* (b) *precedence*

(c) *succession*

Figure 6: Minimized deterministic activation-aware automata for the *precedence*, *response*, and *succession* DECLARE templates

automaton is $O(|S| \cdot |\Sigma^x_*|)$, where S is the set of states and Σ^x_* is the labeling domain.

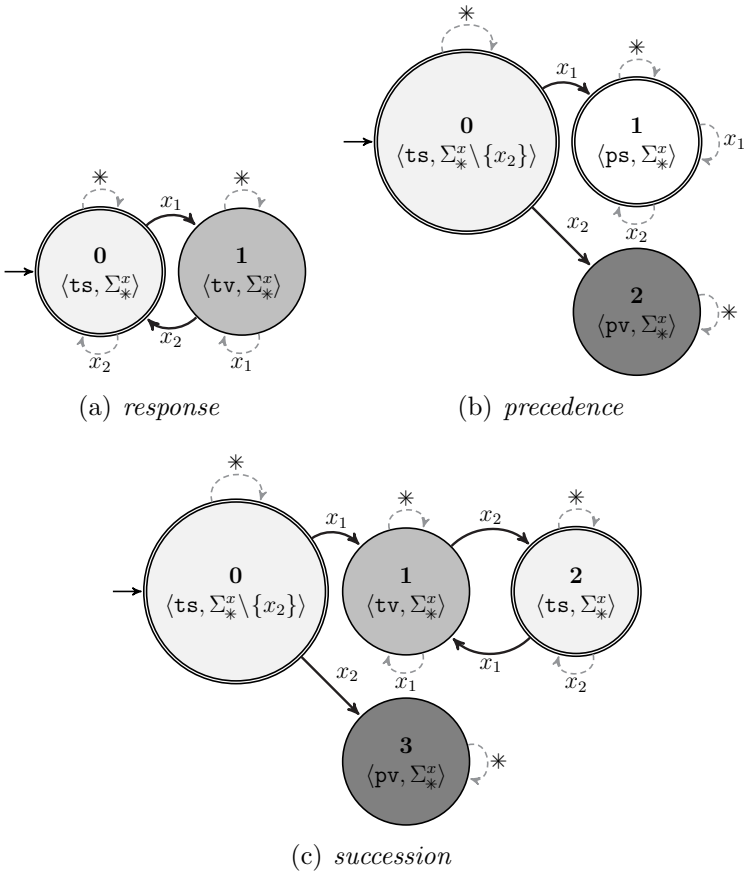
Once the activation states are assigned to states, the template automata are made activation-aware by computing the set of relevant transitions (as per Definition 13). To do so, every state is visited once, and transitions are included in the set of relevant ones if and only if:

1. the RV-LTL truth value of the next state differs from the one of the current state (e.g., $\langle \mathbf{0}, x_1, \mathbf{1} \rangle$ and $\langle \mathbf{1}, x_2, \mathbf{0} \rangle$ in Fig. 5(a)), or
2. the set of permitted tasks of the next state differs from the one of the current state (e.g., $\langle \mathbf{0}, x_1, \mathbf{1} \rangle$ in Fig. 5(b)).

The evaluation of the stated conditions over the template automata of Fig. 5 are depicted in Fig. 6. Dashed, gray transitions are irrelevant, as opposed to the black, solid ones. The time complexity of this operation is again $O(|S| \cdot |\Sigma^x_*|)$.

Finally, template automata are augmented so as to make the transition function left-total as per Definition 4, i.e., to make them capable of replaying non-compliant traces. To that extent, nodes having activation state $\langle \mathtt{pv}, \Sigma^x_* \rangle$

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011



(a) *response* (b) *precedence*

(c) *succession*

Figure 7: Deterministic activation-aware automata for the *precedence*, *response* and *succession* DECLARE templates

are added and linked to every state whose set of permitted tasks is strictly contained in Σ^x_* (e.g., states labeled as **0** in Fig. 6(b) and Fig. 6(c)). The transitions linking the source state to the new state are labeled by the tasks of Σ^x_* that are not included in the set of permitted tasks (e.g., x_2 for the **0**-states both in Fig. 6(b) and Fig. 6(c)). The activation-aware template automata of the response, precedence, and succession DECLARE templates are depicted in Fig. 7. The activation-aware template automaton of the progression response is depicted in Fig. 8. The time complexity of this last operation remains $O(|S| \cdot |\Sigma^x_*|)$.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

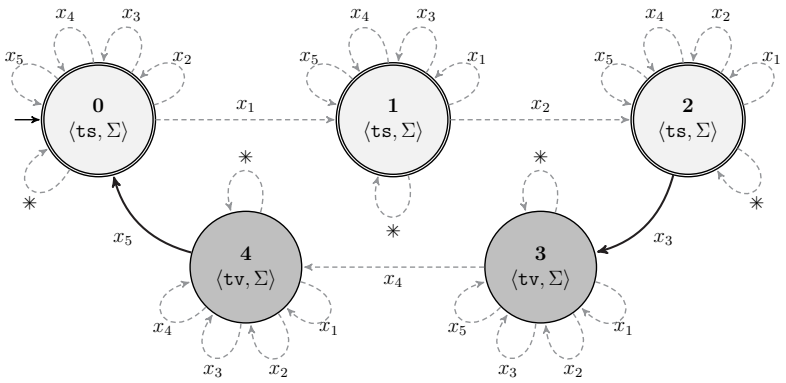


Figure 8: Activation-aware automaton for the *progression response* template (with three sources and two targets)

6.2. Replay of an Event Log on a Template Automaton

Having built the activation-aware template automata, we can verify for a given event log whether each trace violates or not a constraint. If it complies with it, we can also decide whether the satisfaction is vacuous or not. To this end, we rely on the notion of *constraint cursor*. The constraint cursor is meant to act as a pointer which walks along the automaton as the trace unfolds. In a template automaton, we create one cursor for every constraint we want to check, i.e., for every assignment of tasks in Σ to the n parameters of the template. In this way, we avoid to keep in memory multiple copies of the automata, i.e., one for each constraint instantiating the templates to be checked against the log. Nevertheless, we are able to keep information about the status of every constraint during the replay of the log. For example, given a set of tasks $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathsf{f}, \mathsf{g}, \mathsf{h}\}$ and the succession DECLARE template, we have one cursor for every assignment of the parameters to an ordered pair of tasks, e.g., succession(a, b), succession(b, a), succession(f, h), and succession(e, f). Every cursor maps the assigned tasks to the respective parameters that label the transitions of the template automaton. Any other task in Σ is mapped to $\ast$. For instance, the cursor of succession(a, b) maps a to x_1 and b to x_2. Tasks c, d, ..., h are all mapped to $\ast$.

Every step of the cursor is subsequently dictated by the occurring tasks in the trace. At the beginning, the cursor points at the initial state of the template automaton. Considering the automaton depicted in Fig. 7(c), the succession(a, b)-cursor points at state **0**. For each task in the trace, the cursor moves from the current state along the outgoing transition mapped to the task.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

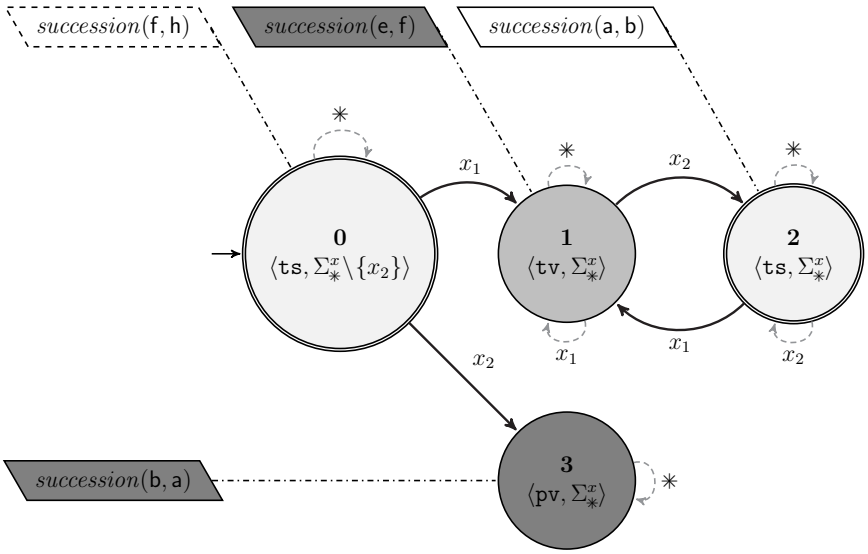


Figure 9: Deterministic activation-aware automaton of the *succession* DECLARE template with some constraint cursors at the end of the replay of trace $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{d}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathsf{g}, \mathsf{a}, \mathsf{b} \rangle$

A *walk* on the automaton is the replay of a trace through a constraint cursor. Given the example trace $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{d}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathsf{g}, \mathsf{a}, \mathsf{b} \rangle$, the succession$(\mathsf{a}, \mathsf{b})$-cursor first moves to state **1** of the template automaton depicted in Fig. 7(c) along transition $\langle \mathbf{0}, x_1, \mathbf{1} \rangle$, since the first event in the trace is an occurrence of a, which is mapped to x_1. Thereupon, the next move is along $\langle \mathbf{1}, x_1, \mathbf{1} \rangle$. The third task, b, moves the cursor along $\langle \mathbf{1}, x_2, \mathbf{2} \rangle$. The fourth task, d, moves the cursor along $\langle \mathbf{2}, *, \mathbf{2} \rangle$. At the end of the trace, the cursor points at state **2**.

The replay of a full trace leads to three possible assessments of the trace with respect to the constraint under analysis.

1. If the walk ends in a state denoting a temporary (`tv`) or permanent violation (`pv`), then the trace is considered as *non-compliant*.
2. If the walk terminates in a state denoting a temporary (`ts`) or permanent satisfaction (`ps`), then the trace is considered as *compliant* with the constraint.
3. The fulfillment is categorized as *non-vacuous* only if a relevant transition has been traversed at least once during the walk. Otherwise, the trace is *vacuously* compliant.

Figures 9 and 10 show constraint cursors, depicted as trapezia, at the end of the replay of the example trace $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{d}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathsf{g}, \mathsf{a}, \mathsf{b} \rangle$. The state they point at is

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

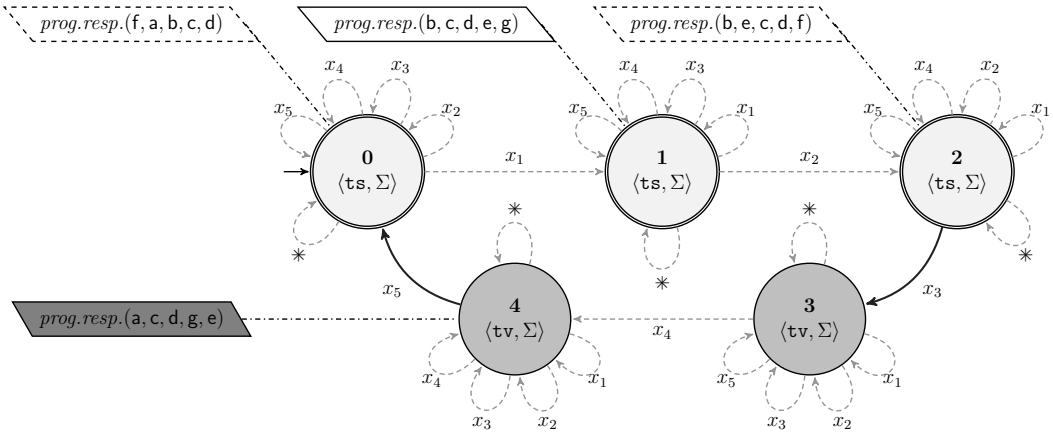


Figure 10: Deterministic activation-aware automaton of the *progression response* template (with three sources and two targets) with some constraint cursors at the end of the replay of trace $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{d}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathsf{g}, \mathsf{a}, \mathsf{b} \rangle$

linked by a dash-dotted line. The shape's bounding line and color indicate the compliance of the trace with the constraint. Dashed white trapezia denote vacuously satisfied constraints. Examples of such constraint cursors are the ones referring to succession(f, h), in Fig. 9, and progression response$(\mathsf{b}, \mathsf{e}, \mathsf{c}, \mathsf{d}, \mathsf{f})$, in Fig. 10. Solid white trapezia indicate non-vacuously satisfied ones (see succession(a, b) in Fig. 9 and progression response$(\mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathsf{g})$ in Fig. 10). Cursors of constraints that are violated by the trace are filled with a grey color, such as succession(e, f) in Fig. 9 and progression response$(\mathsf{a}, \mathsf{c}, \mathsf{d}, \mathsf{g}, \mathsf{e})$ in Fig. 10.

7. Evaluation

In order to validate our approach, we have embedded it into a Java software prototype for the discovery of constraints from an event log (based on the algorithm presented in [35]).[2] The approach has been run on two real-life event logs taken from the collection of the IEEE Task Force on Process Mining: the log used for the Business Process Intelligence (BPI) challenge 2013 [48], and a log pertaining to a road traffic fines management process [49]. The tests have been conducted on a machine equipped with an

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

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

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*, *coexistence*, and *not 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.

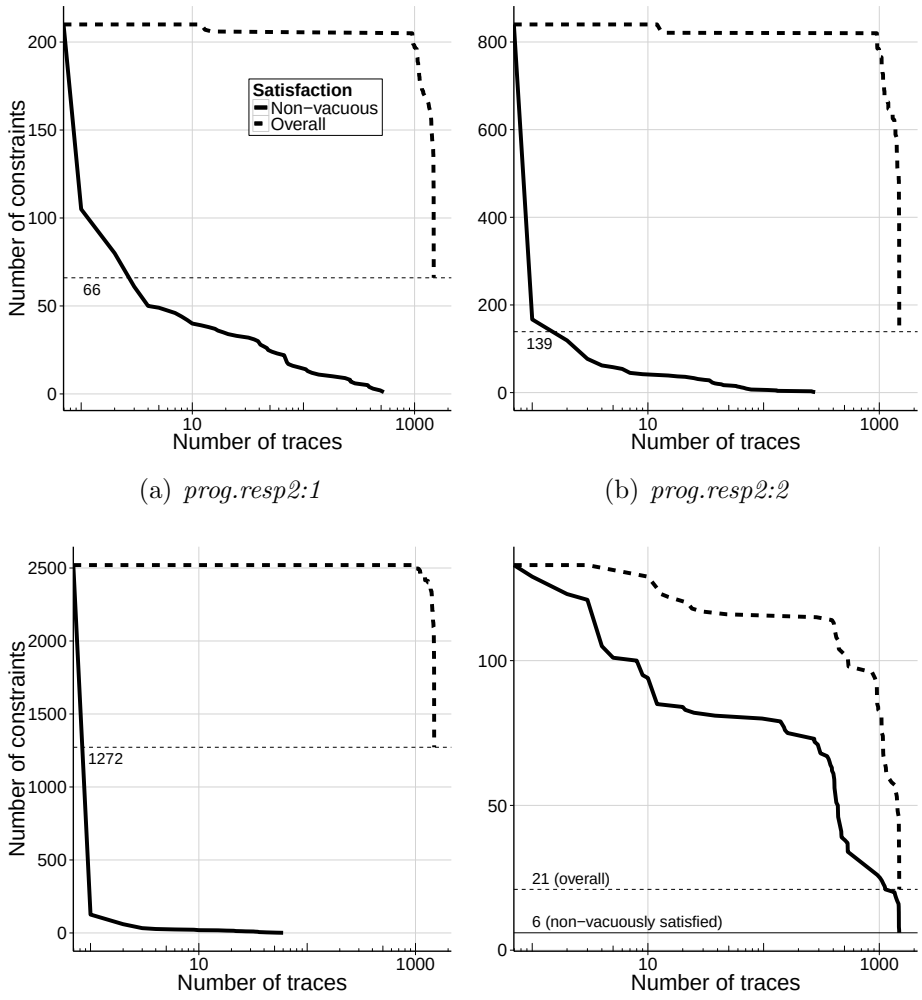
Figure 11 shows the trends of the number of progression response constraints discovered from the BPI challenge 2013 log with respect to the number of traces (vacuously and non-vacuously) satisfying them. Figures 11(a) to 11(c) relate to progression response templates with an increasing number of parameters. 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 significant number of irrelevant constraints. The discovery algorithm detected, indeed, that 66 *prog.resp2:1*, 139 *prog.resp2:2*, and 1 272 *prog.resp3:2* constraints 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 *prog.resp3:2* can only be activated when three different tasks occur sequentially, whereas *prog.resp2:1* and *prog.resp2:2* 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 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. Figure 11(d) shows the same trend when using the standard DECLARE templates mentioned above for the discovery. Overall, the computation took 9.442 s, out of which 0.426 s were spent to build the automata, and the remaining 9.016 s to check the log.

We illustrate 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 MINERful [25] and the Declare Miner [10]. The constraints discovered by our tool, using a minimum threshold of 100 % of witnesses in the log, are:

- Existence(Create fine)
- Alt. precedence(Create fine, Add penalty)

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011



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

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

Figure 11: Trends of the number of the discovered constraints with respect to the number of traces satisfying them

- Not 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)
- Not chain succession(Create fine, Insert fine notification)
- Alt. precedence(Create fine, Notify result appeal to offender)

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

- Not chain succession(Create fine, Notify result appeal to offender)
- Alt. precedence(Create fine, Receive result appeal from prefecture)
- Not chain succession(Create fine, Receive result appeal from prefecture)
- Alt. precedence(Create fine, Send appeal to prefecture)
- Not chain succession(Create fine, Send appeal to prefecture)
- Alt. precedence(Create fine, Send fine)
- Alt. precedence(Create fine, Send for credit collection)
- Not chain succession(Create fine, Send for credit collection)

Such constraints are a subset of the ones returned by MINERful using the same templates, because 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 tasks. In addition, some tasks cannot directly follow Create fine. Also, we discovered that the following progression response constraints are non-vacuously 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 traces. Similar results cannot be obtained with MINERful and the Declare Miner. The former is indeed not designed to discover non-standard DECLARE constraints. The latter offers that facility, but only provides an ad-hoc mechanism for vacuity detection. As mentioned in Section 3, this means that, in order to discriminate between vacuously and non-vacuously satisfied progression response constraints using the Declare Miner, a new, ad hoc procedure should be implemented and embedded in the Declare Miner code. This should be done for every type of constraint that is not covered by standard DECLARE.

8. From Relevance to Counting and Related Issues

The natural further step in this line of research is to extend our approach towards the possibility of *counting activations*, and use this fine-grained information to define the degree of adherence of a trace to a constraint. This is not only relevant to process monitoring but also crucial in the context of declarative process discovery, when the discovery algorithm extracts those

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

constraints that are not fully supported by the input log [50]. 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, defined in an ad-hoc way for each of the core DECLARE patterns.

In this section, we argue that the framework presented in this paper may be used as a formal basis for counting activations, limiting ourselves to the case of satisfied constraints. We introduce a possible way of counting and relate this to the previous, ad-hoc approaches for counting activations in DECLARE. Our conclusion is that the problem of counting cannot satisfactorily be approached without enriching the constraints of interest with further information.

8.1. Counting Activations

As shown in Section 4, and, in particular, in Definitions 10 and 11, our semantic notion of activation is applied to single tasks present in a trace. Consequently, Definition 11 lends itself to the following notion of *strength*, based on relevant task counting.

Definition 14 (Satisfaction strength). A trace τ over Σ^* has *satisfaction strength* $1 \leq k \leq |\tau|$, for a constraint φ, if τ complies with φ, and there are exactly k positions $j_1, \ldots, j_k$ such that, for each $1 \leq i \leq k$, task $\tau(j_i)$ is a relevant task execution for φ after the prefix of τ ending at position $j_i - 1$.□

Intuitively, Definition 14 states that a trace compliant with a constraint has satisfaction strength k if it contains exactly k tasks which, in their position within the trace, represent activations of the constraint. As a consequence, we also get that a trace has satisfaction strength 0 for a constraint if it vacuously satisfies that constraint.

We show how this notion of counting behaves in some simple cases.

Example 7. Consider the response(e, m) constraint, together with trace $\tau = \langle \mathsf{d}, \mathsf{m}, \mathsf{e}, \mathsf{m}, \mathsf{m}, \mathsf{e}, \mathsf{e}, \mathsf{m} \rangle$ of Example 3. As clearly witnessed by the discussion in Example 3, this constraint has satisfaction strength 4 over τ, since there are four positions leading to a relevant evolution of the activation state. ■

Example 8. Consider the precedence(e, m) constraint. Its satisfaction strength is either 0 or 1. In fact, this constraint has satisfaction strength 0 if the considered trace does not contain any execution of e or m, and it has satisfaction strength 1 if the considered trace contains at least one execution of e that is not preceded by any execution of m. When e occurs, the constraint becomes

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

permanently satisfied, and, therefore, all suffixes will not be relevant to it. Considering traces $\tau_0 = \varepsilon$, $\tau_1 = \langle \mathsf{e}, \mathsf{m} \rangle$, and $\tau_2 = \tau_1 \cdot \tau_1 \cdot \tau_1 \cdot \tau_1 = \langle \mathsf{e}, \mathsf{m}, \mathsf{e}, \mathsf{m}, \mathsf{e}, \mathsf{m}, \mathsf{e}, \mathsf{m} \rangle$ from Section 3.1, precedence(e, m) has satisfaction strength equal to 0 over τ_0, and equal to 1 over both τ_1 and τ_2. ■

We use Examples 7 and 8 to discuss how this approach relates to the ones present in the literature [14, 51], and argue that a suitable notion of counting requires to have additional information attached to the constraint of interest.

8.2. On the Suitability of Counting Strategies

By considering the response DECLARE constraint, [14, 51] count activations similarly to the approach presented here: the more times the constraint is moved from temporary satisfaction to temporary violation by the trace of interest, the more weight the traces get in terms of compliance. In general, this correspondence holds over those DECLARE relation templates that have a forward orientation in time, such as response, alternate response, and chain response in Table 1. There is still quite an important difference: the approaches in [14, 51] do not consider, as we do here, the *activation of the constraint as a whole*, but they instead focus on the *activation of the constraint source*. For unary constraints applied to single tasks, the source is that task. For binary constraints (such as response), instead, the source is intuitively the task that "triggers" the constraint, i.e., the task associated to the dot graphical element when it comes to the graphical representation of DECLARE patterns. In the case of Examples 3 and 7, this means that those approaches increase the strength of compliance every time e is executed, whereas in our case such execution only matters if it has an impact on the constraint.

This distinction creates a discrepancy between our approach and those in [14, 51] when it comes to negation templates (such as not coexistence, not succession, and not chain succession), as well as for those relation templates that either do not have any temporal orientation (such as responded existence and coexistence), or have a backward orientation (such as precedence, alternate precedence, and chain precedence). Take, for example, the case of precedence. With our approach (see Example 8), the satisfaction strength ranges from 0 to 1. With [14, 51], instead, the constraint strength increases every time the source task is executed, even when the precedence is in a permanent state of satisfaction.

The difference between counting on tasks as opposed to constraints as a whole requires further investigation. There is, in addition, an orthogonal

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

dimension that deeply impacts counting and cannot be captured when tasks are just represented as propositions. This dimension is about *data* carried by tasks, as well as *effects* induced by task executions within the organizational setting.

Data may be used to correlate tasks. Such correlations may be used, in turn, to create multiple instantiations of the same constraint and track their simultaneous evolution depending not only on the execution of tasks, but also on the data they carry. This is, e.g., studied in [51, 52]. To appreciate this fine-grained distinction, consider again trace τ in Example 3, where task e stands for "eat food", and task m stands for "measure glucose". Measuring the satisfaction strength would in this case be radically different if all those task executions refer to the same person (thus having that two consequent executions of e are idempotent), or whether they refer to different persons. This distinction can only be considered and reflected into counting, if tasks are enriched with data attributes (in this example, the person identifier).

Beyond data attributes, one may also consider the effect of tasks. Imagine, for example, to have a response DECLARE constraint relating a task "add item to order", and a task "pay order". In this case, the more items are added to an order, the more paying the order may be deemed as important, intuitively connecting the notion of compliance to the price of the order. Again, taking this aspect into consideration, when measuring compliance strength, is only possible if the representation of constraints and tasks is suitably enriched.

In conclusion, we believe that a suitable notion of counting cannot be defined in general, unless the impact of time, resources, and data is explicitly incorporated into the constraint language. This is matter of future work.

9. Related work

Our research relates to process mining applied, in particular, to the field of declarative process mining [18]. The idea of employing mining in the context of workflow management systems has been originally introduced in [53]. Processes were modeled as directed graphs in which vertices represented activities and edges stood for the dependencies between them. Cook and Wolf, at the same time, investigated similar issues in the context of software engineering processes [54]. From [53] onwards, several techniques have been proposed in the stream of process mining [55].

In the area of declarative process mining, [35] first proposed an automated discovery algorithm based on the instantiation of a set of candidate DECLARE

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

constraints later checked against event logs by replaying the traces on constraint automata to determine their support. In our approach, we resort on constraint automata as well, though we introduce the notion of activation-aware automaton to detect the cases in which constraints are only vacuously satisfied. The approach presented in [35] has been improved in [10] where the number of candidates to be checked is reduced through the Apriori algorithm, originally developed by Agrawal and Srikant for mining association rules [38]. That was the first attempt to avoid that vacuously satisfied constraints could enter the final result set using ad-hoc techniques (valid for standard DECLARE). The Apriori algorithm and the DECLARE-specific vacuity detection approach have also been utilized in [37], where a sequence analysis-based algorithm has been proposed for DECLARE discovery. In [56], the same approach has been applied for the refinement and repair of DECLARE models based on logs, and for guiding the discovery task based on prior domain knowledge. Our approach is generic in that it applies to any constraint whose semantics is expressible by means of a finite state automaton.

In [57, 58], the authors presented an approach for the mining of declarative process models expressed through probabilistic logics. The approach first extracts a set of integrity constraints from a log. Then, the learned constraints are translated into Markov Logic formulae that allow for a probabilistic classification of the traces. In [59, 60], the authors describe an approach based on Inductive Logic Programming techniques to discover DECLARE models. These approaches are not equipped with vacuity detection techniques such as the one presented in this paper.

In [23, 24, 25], the authors introduce MINERful, a two-step algorithm for the discovery of DECLARE constraints. The first step of the approach is the building of a knowledge base, with information about temporal statistics about the (co-)occurrence of tasks within the log. Then, the validity [23] and the support [24, 25] of constraints is computed by querying that knowledge base. The value assigned to support is calculated by counting the activations not leading to violations of constraints. However, there is no specific measure to tackle vacuity and the approach has not been extended beyond the default repertoire of DECLARE templates. In [12, 13], the authors propose an extension of MINERful to discover target-branched DECLARE constraints, i.e., constraints in which the target parameter is replaced by a disjunction of actual tasks. Branched DECLARE constraints and, in particular, target-branched DECLARE constraint can be expressed using LTL_f. Therefore, our solution can be utilized for branched DECLARE models as well.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

Another approach for the discovery of DECLARE models is described in [36]. The presented technique is based on the translation of DECLARE templates into SQL queries on a relational database instance, where the event log has previously been stored. The query answer assigns the free variables with those tasks that lead to the satisfaction of the constraint in the event log. The methodology has later been extended towards multi-perspective DECLARE discovery [61], to include data in the formulation of constraints, as described in [34]. Beforehand, an approach for discovering data-ware constraints was proposed in [62], and an approach for dealing with the time perspective was introduced in [63]. Recently, approaches for taking into account activity lifecycles in DECLARE discovery have been proposed in [64, 65]. The consideration of event data and activity lifecycles in the analysis of constraints relevance is still an interesting open challenge that we aim at tackling in future work.

An algorithm for discovering DCR graphs has been proposed in [66]. Since DCR graphs employ temporal logics to express their semantics, our approach can be applied to this set of constraints, too. Approaches for the online discovery of DECLARE models have been presented in [67, 68, 69]. Our approach can be used to complement these approaches since it is based on the RV-LTL semantics that is suitable to be used to check running, evolving traces.

Finally, we remark that studies have been conducted to remove inconsistencies and redundancies from discovered declarative models [50, 70]. The proposed solutions resort on the language-inclusion and cross-product of automata underlying constraints. Our approach is complementary to them: the removal of inconsistencies and redundancies is a correction and simplification step to be carried out ex-post, analyzing the mutual entailment or contradiction of constraints, whilst we operate on the set of constraints while being discovered to ensure their relevance in terms of non-vacuity to the event log.

10. Conclusion

This paper presents the first semantical characterization of relevance for constraints (expressed with temporal logics over finite traces) to an event log. As a side result, we also obtain a semantical notion of vacuous satisfaction for such logics. Our characterization comes with a concrete approach for monitoring and checking relevance to running or complete traces, achieved by

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

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 beyond the DECLARE patterns.

The presented solution generalizes the ad-hoc approaches previously proposed in the literature for tackling conformance checking and discovery of DECLARE constraints. The solution 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.

The yet unresolved problem of activation counting paves the path of our future work. Furthermore, it is in our plans to integrate the presented approach in a full-fledged framework for declarative process discovery including *(i)* the option for the user to define custom templates [71], *(ii)* the integration of user-specified domain knowledge [56], and *(iii)* the automated removal of inconsistencies and redundancies [50, 70]. Thereupon, we will assess the efficacy of the proposed approach in the context of thorough business process analysis initiatives to be conducted with the collaboration of real-world organizations.

From a more theoretical point of view, a challenge of interest for future research endeavor is the extension of the notion of vacuity towards multi-perspective declarative process models [32, 33, 34], where not only tasks but also artifacts, timestamps, resources, and other event data are considered in the formulation of constraints. A formal framework that accounts for relevance detection under multiple perspectives will be of clear benefit to the expressiveness of the discovered models.

Studies will also be conducted on the approach as a stand-alone module, to allow for its usage in those fields beyond the scope of process mining yet resorting on constraint-based specifications such as, e.g., knowledge acquisition and refinement of constraint-based recommender systems [72], configuration problems in declarative representation of knowledge bases [73], services discovery and behavioral matching [74], and behavioral requirements elicitation in software engineering [75, 76].

References

[1] W. M. P. van der Aalst, A. Adriansyah, A. K. A. de Medeiros, F. Arcieri, T. Baier, T. Blickle, R. P. J. C. Bose, P. van den Brand, R. Brandtjen, J. C. A. M. Buijs, A. Burattin, J. Carmona, M. Castellanos, J. Claes,

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

J. Cook, N. Costantini, F. Curbera, E. Damiani, M. de Leoni, P. Delias, B. F. van Dongen, M. Dumas, S. Dustdar, D. Fahland, D. R. Ferreira, W. Gaaloul, F. van Geffen, S. Goel, C. W. Günther, A. Guzzo, P. Harmon, A. H. M. ter Hofstede, J. Hoogland, J. E. Ingvaldsen, K. Kato, R. Kuhn, A. Kumar, M. L. Rosa, F. M. Maggi, D. Malerba, R. S. Mans, A. Manuel, M. McCreesh, P. Mello, J. Mendling, M. Montali, H. R. M. Nezhad, M. zur Muehlen, J. Munoz-Gama, L. Pontieri, J. Ribeiro, A. Rozinat, H. S. Pérez, R. S. Pérez, M. Sepúlveda, J. Sinur, P. Soffer, M. Song, A. Sperduti, G. Stilo, C. Stoel, K. D. Swenson, M. Talamo, W. Tan, C. Turner, J. Vanthienen, G. Varvaressos, E. Verbeek, M. Verdonk, R. Vigo, J. Wang, B. Weber, M. Weidlich, T. Weijters, L. Wen, M. Westergaard, M. T. Wynn, Process mining manifesto, in: BPM Workshops, 2011, pp. 169–194. doi:10.1007/978-3-642-28108-2_19.

[2] C. Di Ciccio, A. Marrella, A. Russo, Knowledge-intensive Processes: Characteristics, requirements and analysis of contemporary approaches, J. Data Semantics 4 (1) (2015) 29–57. doi:10.1007/s13740-014-0038-4.

[3] W. M. P. van der Aalst, M. Pesic, H. Schonenberg, Declarative workflows: Balancing between flexibility and support, Computer Science - R&D 23 (2) (2009) 99–113. doi:10.1007/s00450-009-0057-9.

[4] P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, H. A. Reijers, Imperative versus declarative process modeling languages: An empirical investigation, in: Business Process Management Workshops (1), 2011, pp. 383–394. doi:10.1007/978-3-642-28108-2_37.

[5] M. Reichert, B. Weber, Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies, Springer, 2012. doi:10.1007/978-3-642-30409-5.

[6] M. Pesic, H. Schonenberg, W. M. P. van der Aalst, DECLARE: Full support for loosely-structured processes, in: EDOC, 2007, pp. 287–300. doi:10.1109/EDOC.2007.14.

[7] T. T. Hildebrandt, R. R. Mukkamala, T. Slaats, Nested dynamic condition response graphs, in: FSEN, 2011, pp. 343–350. doi:10.1007/978-3-642-29320-7_23.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

[8] I. Beer, S. Ben-David, C. Eisner, Y. Rodeh, Efficient detection of vacuity in temporal model checking, Formal Methods in System Design 18 (2) (2001) 141–163. doi:10.1023/A:1008779610539.

[9] O. Kupferman, M. Y. Vardi, Vacuity detection in temporal model checking, International Journal on Software Tools for Technology Transfer 4 (2) (2003) 224–233. doi:10.1007/s100090100062.

[10] F. M. Maggi, R. P. J. C. Bose, W. M. P. van der Aalst, Efficient discovery of understandable declarative process models from event logs, in: CAiSE, 2012, pp. 270–285. doi:10.1007/978-3-642-31095-9_18.

[11] D. M. M. Schunselaar, F. M. Maggi, N. Sidorova, Patterns for a log-based strengthening of declarative compliance models, in: IFM, 2012, pp. 327–342. doi:10.1007/978-3-642-30729-4_23.

[12] C. Di Ciccio, F. M. Maggi, J. Mendling, Discovering Target-Branched Declare constraints, in: BPM, Springer, 2014, pp. 34–50. doi:10.1007/978-3-319-10172-9_3.

[13] C. Di Ciccio, F. M. Maggi, J. Mendling, Efficient discovery of Target-Branched Declare constraints, Information Systems 56 (2016) 258–283. doi:10.1016/j.is.2015.06.009.

[14] A. Burattin, F. M. Maggi, W. M. P. van der Aalst, A. Sperduti, Techniques for a posteriori analysis of declarative processes, in: EDOC, 2012, pp. 41–50. doi:10.1109/EDOC.2012.15.

[15] F. M. Maggi, M. Montali, W. M. P. van der Aalst, An operational decision support framework for monitoring business constraints, in: FASE, 2012, pp. 146–162. doi:10.1007/978-3-642-28872-2_11.

[16] M. de Leoni, F. M. Maggi, W. M. P. van der Aalst, Aligning event logs and declarative process models for conformance checking, in: BPM, 2012, pp. 82–97. doi:10.1007/978-3-642-32885-5_6.

[17] M. de Leoni, F. M. Maggi, W. M. P. van der Aalst, An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data, Information Systems 47 (2015) 258 – 277. doi:10.1016/j.is.2013.12.005.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

[18] F. M. Maggi, Declarative process mining with the declare component of ProM, in: BPM (Demos), 2013.
URL http://ceur-ws.org/Vol-1021/paper_8.pdf

[19] F. M. Maggi, M. Montali, C. Di Ciccio, J. Mendling, Semantical vacuity detection in declarative process mining, in: BPM, 2016, pp. 158–175. doi:10.1007/978-3-319-45348-4_10.

[20] M. Montali, Specification and Verification of Declarative Open Interaction Models: a Logic-Based Approach, Vol. 56 of Lecture Notes in Business Information Processing, Springer, 2010. doi:10.1007/978-3-642-14538-4.

[21] G. De Giacomo, M. Y. Vardi, Linear temporal logic and linear dynamic logic on finite traces, in: IJCAI, 2013, pp. 854–860.
URL http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6997

[22] A. Pnueli, The temporal logic of programs, in: FOCS, 1977, pp. 46–57. doi:10.1109/SFCS.1977.32.

[23] C. Di Ciccio, M. Mecella, Mining constraints for artful processes, in: BIS, Springer, 2012, pp. 11–23. doi:10.1007/978-3-642-30359-3_2.

[24] C. Di Ciccio, M. Mecella, A two-step fast algorithm for the automated discovery of declarative workflows, in: CIDM, IEEE, 2013, pp. 135–142. doi:10.1109/CIDM.2013.6597228.

[25] C. Di Ciccio, M. Mecella, On the discovery of declarative control flows for artful processes, ACM Trans. Manage. Inf. Syst. 5 (4) (2015) 24:1–24:37. doi:10.1145/2629447.

[26] G. De Giacomo, R. De Masellis, M. Grasso, F. M. Maggi, M. Montali, Monitoring business metaconstraints based on LTL & LDL for finite traces, in: BPM, 2014, pp. 1–17. doi:10.1007/978-3-319-10172-9_1.

[27] J. R. Büchi, Weak second-order arithmetic and finite automata, Mathematical Logic Quarterly 6 (1-6) (1960) 66–92, proof of MSO-FSA equivalence. doi:10.1002/malq.19600060105.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

[28] V. Diekert, P. Gastin, First-order definable languages, in: Logic and Automata, 2008, pp. 261–306.

[29] D. Giannakopoulou, K. Havelund, Automata-based verification of temporal properties on running programs, in: ASE, 2001, pp. 412–416. doi:10.1109/ASE.2001.989841.

[30] M. Westergaard, F. M. Maggi, Looking into the future. using timed automata to provide a priori advice about timed declarative process models, in: OTM, 2012, pp. 250–267. doi:10.1007/978-3-642-33606-5_16.

[31] F. M. Maggi, M. Westergaard, Using timed automata for *a Priori* warnings and planning for timed declarative process models, Int. J. Cooperative Inf. Syst. 23 (1). doi:10.1142/S0218843014400036.

[32] M. Montali, F. Chesani, P. Mello, F. M. Maggi, Towards data-aware constraints in Declare, in: SAC, 2013, pp. 1391–1396. doi:10.1145/2480362.2480624.

[33] R. De Masellis, F. M. Maggi, M. Montali, Monitoring data-aware business constraints with finite state automata, in: H. Zhang, L. Huang, I. Richardson (Eds.), ICSSP, ACM, 2014, pp. 134–143. doi:10.1145/2600821.2600835.

[34] A. Burattin, F. M. Maggi, A. Sperduti, Conformance checking based on multi-perspective declarative process models, Expert Syst. Appl. 65 (2016) 194–211. doi:10.1016/j.eswa.2016.08.040.

[35] F. M. Maggi, A. J. Mooij, W. M. P. van der Aalst, User-guided discovery of declarative process models, in: CIDM, IEEE, 2011, pp. 192–199. doi:10.1109/CIDM.2011.5949297.

[36] S. Schönig, A. Rogge-Solti, C. Cabanillas, S. Jablonski, J. Mendling, Efficient and customisable declarative process mining with SQL, in: CAiSE, 2016, pp. 290–305. doi:10.1007/978-3-319-39696-5_18.

[37] T. Kala, F. M. Maggi, C. Di Ciccio, C. Di Francescomarino, Apriori and sequence analysis for discovering declarative process models, in: EDOC, 2016, pp. 1–9. doi:10.1109/EDOC.2016.7579378.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

[38] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: VLDB, 1994, pp. 487–499.

[39] G. De Giacomo, R. De Masellis, M. Montali, Reasoning on LTL on finite traces: Insensitivity to infiniteness, in: AAAI, 2014, pp. 1027–1033. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8575

[40] A. Bauer, M. Leucker, C. Schallhart, Runtime verification for LTL and TLTL, ACM Trans. Softw. Eng. Methodol. 20 (4) (2011) 14:1–14:64. doi:10.1145/2000799.2000800.

[41] F. M. Maggi, M. Montali, M. Westergaard, W. M. P. van der Aalst, Monitoring business constraints with Linear Temporal Logic: An approach based on colored automata, in: BPM, 2011, pp. 132–147. doi:10.1007/978-3-642-23059-2_13.

[42] F. M. Maggi, M. Westergaard, M. Montali, W. M. van der Aalst, Runtime verification of LTL-based declarative process models, in: RV, 2011, pp. 131–146. doi:10.1007/978-3-642-29860-8_11.

[43] M. Westergaard, F. M. Maggi, Modeling and verification of a protocol for operational support using coloured petri nets, in: PETRI NETS, 2011, pp. 169–188. doi:10.1007/978-3-642-21834-7_10.

[44] F. M. Maggi, M. Westergaard, Designing software for operational decision support through coloured petri nets, Enterprise IS 11 (5) (2017) 576–596. doi:10.1080/17517575.2015.1067723.

[45] M. O. Rabin, D. S. Scott, Finite automata and their decision problems, IBM Journal of Research and Development 3 (2) (1959) 114–125. doi:10.1147/rd.32.0114.

[46] J. E. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, in: Z. Kohavi (Ed.), Theory of Machines and Computations, Stanford University, Stanford, CA, USA, 1971, pp. 189–196.

[47] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to Automata Theory, Languages, and Computation (3rd Edition), Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

[48] W. Steeman, Real-life event logs – an incident management process: closed problems, BPIC (2013). doi:10.4121/uuid:500573e6-accc-4b0c-9576-aa5468b10cee.

[49] M. de Leoni, F. Mannhardt, Road traffic fine management process (2015). doi:10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5.

[50] C. Di Ciccio, F. M. Maggi, M. Montali, J. Mendling, Ensuring model consistency in declarative process discovery, in: BPM, Springer, 2015, pp. 144–159. doi:10.1007/978-3-319-23063-4_9.

[51] M. Montali, F. M. Maggi, F. Chesani, P. Mello, W. M. van der Aalst, Monitoring business constraints with the event calculus, ACM TIST 5 (1) (2013) 17:1–17:30. doi:10.1145/2542182.2542199.

[52] R. P. J. C. Bose, F. M. Maggi, W. M. P. van der Aalst, Enhancing Declare maps based on event correlations, in: BPM, 2013, pp. 97–112. doi:10.1007/978-3-642-40176-3_9.

[53] R. Agrawal, D. Gunopulos, F. Leymann, Mining process models from workflow logs, in: EDBT, 1998, pp. 469–483. doi:10.1007/BFb0101003.

[54] J. E. Cook, A. L. Wolf, Discovering models of software processes from event-based data, ACM Trans. Softw. Eng. Methodol. 7 (3) (1998) 215–249. doi:10.1145/287000.287001.

[55] W. M. P. van der Aalst, Process Mining - Data Science in Action, Second Edition, Springer, 2016. doi:10.1007/978-3-662-49851-4.

[56] F. M. Maggi, R. J. C. Bose, W. M. van der Aalst, A knowledge-based integrated approach for discovering and repairing declare maps, in: CAiSE, 2013, pp. 433–448. doi:10.1007/978-3-642-38709-8_28.

[57] E. Bellodi, F. Riguzzi, E. Lamma, Probabilistic logic-based process mining, in: CILC, 2010.
URL http://ceur-ws.org/Vol-598/paper17.pdf

[58] E. Bellodi, F. Riguzzi, E. Lamma, Probabilistic declarative process mining, in: KSEM, 2010, pp. 292–303. doi:10.1007/978-3-642-15280-1_28.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

[59] E. Lamma, P. Mello, M. Montali, F. Riguzzi, S. Storari, Inducing declarative logic-based models from labeled traces, in: BPM, 2007, pp. 344–359. doi:10.1007/978-3-540-75183-0_25.

[60] F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi, S. Storari, Exploiting inductive logic programming techniques for declarative process mining, T. Petri Nets and Other Models of Concurrency 2 (2009) 278–295.
URL http://dx.doi.org/10.1007/978-3-642-00899-3_16

[61] S. Schönig, C. Di Ciccio, F. M. Maggi, J. Mendling, Discovery of multi-perspective declarative process models, in: ICSOC, 2016, pp. 87–103. doi:10.1007/978-3-319-46295-0_6.

[62] F. M. Maggi, M. Dumas, L. García-Bañuelos, M. Montali, Discovering data-aware declarative process models from event logs, in: BPM, 2013, pp. 81–96. doi:10.1007/978-3-642-40176-3_8.

[63] F. M. Maggi, Discovering metric temporal business constraints from event logs, in: Perspectives in Business Informatics Research - 13th International Conference, BIR 2014, Lund, Sweden, September 22-24, 2014. Proceedings, 2014, pp. 261–275. doi:10.1007/978-3-319-11370-8_19.

[64] M. L. Bernardi, M. Cimitile, C. D. Francescomarino, F. M. Maggi, Using discriminative rule mining to discover declarative process models with non-atomic activities, in: ECAI, 2014, pp. 281–295. doi:10.1007/978-3-319-09870-8_21.

[65] M. L. Bernardi, M. Cimitile, C. D. Francescomarino, F. M. Maggi, Do activity lifecycles affect the validity of a business rule in a business process?, Inf. Syst. 62 (2016) 42–59. doi:10.1016/j.is.2016.06.002.

[66] S. Debois, T. T. Hildebrandt, P. H. Laursen, K. R. Ulrik, Declarative process mining for DCR graphs, in: SAC, 2017, pp. 759–764. doi:10.1145/3019612.3019622.

[67] F. M. Maggi, A. Burattin, M. Cimitile, A. Sperduti, Online process discovery to detect concept drifts in ltl-based declarative process models, in: OTM, 2013, pp. 94–111. doi:10.1007/978-3-642-41030-7_7.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

[68] A. Burattin, M. Cimitile, F. M. Maggi, Lights, camera, action! business process movies for online process discovery, in: BPM Workshops, 2014, pp. 408–419. doi:10.1007/978-3-319-15895-2_34.

[69] A. Burattin, M. Cimitile, F. M. Maggi, A. Sperduti, Online discovery of declarative process models from event streams, IEEE Trans. Services Computing 8 (6) (2015) 833–846. doi:10.1109/TSC.2015.2459703.

[70] C. Di Ciccio, F. M. Maggi, M. Montali, J. Mendling, Resolving inconsistencies and redundancies in declarative process models, Information Systems 64 (2017) 425–446. doi:10.1016/j.is.2016.09.005.

[71] M. Räim, C. Di Ciccio, F. M. Maggi, M. Mecella, J. Mendling, Log-based understanding of business processes through temporal logic query checking, in: CoopIS, Springer, 2014, pp. 75–92. doi:10.1007/978-3-662-45563-0_5.

[72] A. Felfernig, R. D. Burke, Constraint-based recommender systems: technologies and research issues, in: ICEC, 2008, pp. 3:1–3:10. doi:10.1145/1409540.1409544.

[73] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, Consistency-based diagnosis of configuration knowledge bases, Artif. Intell. 152 (2) (2004) 213–234. doi:10.1016/S0004-3702(03)00117-6.

[74] R. Thiagarajan, W. Mayer, M. Stumptner, Semantic service discovery by consistency-based matchmaking, in: APWeb/WAIM, Berlin, Heidelberg, 2009, pp. 492–505. doi:10.1007/978-3-642-00672-2_43.

[75] W. Damm, D. Harel, Lscs: Breathing life into message sequence charts, Formal Methods in System Design 19 (1) (2001) 45–80. doi:10.1023/A:1011227529550.

[76] H. Kaindl, A scenario-based approach for requirements engineering: Experience in a telecommunication software development project, Systems Engineering 8 (3) (2005) 197–210. doi:10.1002/sys.20030.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2018.01.011

This document is a pre-print copy of the manuscript
(Di Ciccio et al. 2018)
published by Elsevier.

The final version of the paper is identified by DOI: 10.1016/j.is.2018.01.011

References

Di Ciccio, Claudio, Fabrizio Maria Maggi, Marco Montali, and Jan Mendling (2018). "On the Relevance of a Business Constraint to an Event Log". In: *Information Systems* 78, pp. 144–161. ISSN: 0306-4379. DOI: 10.1016/j.is.2018.01.011.

BibTeX

```
@Article{         DiCiccio.etal/IS2018:RelevanceofBusinessConstrainttoEventLog,
  author        = {Di Ciccio, Claudio and Maggi, Fabrizio Maria and Montali,
                   Marco and Mendling, Jan},
  title         = {On the Relevance of a Business Constraint to an Event
                   Log},
  journal       = {Information Systems},
  year          = {2018},
  volume        = {78},
  pages         = {144--161},
  issn          = {0306-4379},
  doi           = {10.1016/j.is.2018.01.011},
  keywords      = {Vacuity Detection, Declarative Process Mining, Constraint
                   Activation, Linear Temporal Logic, Finite State Automata},
  publisher     = {Elsevier}
}
```