

# Monitoring Constraints and Metaconstraints with Temporal Logics on Finite Traces

GIUSEPPE DE GIACOMO, Sapienza University of Rome, Italy

RICCARDO DE MASELLIS, Uppsala University, Sweden

FABRIZIO MARIA MAGGI and MARCO MONTALI, Free University of Bozen-Bolzano, Italy

Runtime monitoring is a central operational decision support task in business process management. It helps process executors to check on-the-fly whether a running process instance satisfies business constraints of interest, providing an immediate feedback when deviations occur. We study runtime monitoring of properties expressed in  $LTL_f$ , a variant of the classical  $LTL$  (Linear-time Temporal Logic) that is interpreted over finite traces, and in its extension  $LDL_f$ , a powerful logic obtained by combining  $LTL_f$  with regular expressions. We show that  $LDL_f$  is able to declaratively express, in the logic itself, not only the constraints to be monitored, but also the de facto standard  $RV$ - $LTL$  monitors. On the one hand, this enables us to directly employ the standard characterization of  $LDL_f$  based on finite-state automata to monitor constraints in a fine-grained way. On the other hand, it provides the basis for declaratively expressing sophisticated metaconstraints that predicate on the monitoring state of other constraints, and to check them by relying on standard logical services instead of ad hoc algorithms. We then report on how this approach has been effectively implemented using Java to manipulate  $LDL_f$  formulae and their corresponding monitors, and the  $RUM$  rule mining suite as underlying infrastructure.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; • **Applied computing** → **Business process monitoring**; • **Theory of computation** → *Logic and verification*;

Additional Key Words and Phrases: Temporal logics, runtime verification, business process monitoring, operational decision support, process constraints, metaconstraints

## ACM Reference format:

Giuseppe De Giacomo, Riccardo De Masellis, Fabrizio Maria Maggi, and Marco Montali. 2022. Monitoring Constraints and Metaconstraints with Temporal Logics on Finite Traces. *ACM Trans. Softw. Eng. Methodol.* 31, 4, Article 68 (July 2022), 44 pages.  
<https://doi.org/10.1145/3506799>

The Work partially supported by the ERC Advanced Grant WhiteMech (No. 834228), the EU ICT-48 2020 project TAILOR (No. 952215), the PRIN projects RIPER (No. 20203FFYLK) and PINPOINT (No. 2020FNEB27), the UNIBZ projects WineID, VERBA, and SMART-APP.

Authors' addresses: G. De Giacomo, Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma, Via Ariosto 25, 00185 Roma, Italy; email: [degiacomo@diag.uniroma1.it](mailto:degiacomo@diag.uniroma1.it); R. De Masellis, Uppsala University, Box 337, 75105 Uppsala, Sweden; email: [riccardo.demasellis@it.uu.se](mailto:riccardo.demasellis@it.uu.se); F. M. Maggi and M. Montali, Faculty of Computer Science, Free University of Bozen-Bolzano, Piazza Domenicani 3, 39100 Bolzano, Italy; emails: {[maggi](mailto:maggi@inf.unibz.it), [montali](mailto:montali@inf.unibz.it)}@inf.unibz.it.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2022 Association for Computing Machinery.

1049-331X/2022/07-ART68 \$15.00

<https://doi.org/10.1145/3506799>

## 1 INTRODUCTION

In recent years, process mining has transformed business process management by accompanying traditional analysis techniques for operational processes with a full range of techniques based on the factual event data recorded during the actual execution of such processes.

While traditional process mining techniques start from event data of already completed process instances, *operational decision support* [53] lifts process mining to running, live process executions, whose constitutive events are dynamically recorded. Operational support techniques must hence deal with evolving, incomplete executions for which the past is completely known, but the future is yet to happen.

In this work, we concentrate on one of the most important tasks within operational support: (*prescriptive*) *monitoring*. The goal of monitoring is to check on-the-fly whether a running process instance conforms to business constraints of interest, tracking the evolving states of such constraints and promptly detecting violations [32]. The so-obtained information can be used to calculate and return to domain experts interpretable conformance results and aggregated conformance metrics. This, in turn, forms the basis for taking informed countermeasures aimed at mitigating the effect of undesired behaviors and at positively intervening on the future continuation of the monitored execution. In this respect, monitoring is an essential component to enable *action-oriented process mining*, where the runtime detection of deviations and the enforcement of suitable countermeasures are paired in a virtuous circle [41].

In addition to the setting of operational support, runtime monitoring is in general instrumental in all those scenarios where the system to be checked is either a black box (being unknown or not accessible for verification and analysis in its internal specification) or highly unstructured [4]. An example of black-box process is the behavior of third-party, non-trusted customers when accessing online services offered by the company of interest via APIs. Here, service invocations with their payload and sender information are logged by the IT infrastructure of the company. While it is impossible to certify *a priori* whether users will behave as expected, logging makes it possible to monitor customer flows against fraud-detection rules and to promptly intervene in case of violations. An example of highly unstructured process is that of an emergency ward in a sanitary facility. Here, healthcare professionals handle patients on a per-case basis, continuously adapting the emergency treatment depending on newly collected data and the specific patient record. Monitoring becomes in this setting instrumental to detect whether clinical pathway rules as well as background medical rules on treatments are indeed respected [12].

To build provably correct runtime monitors equipped with a well-defined semantics and a solid formal background, monitoring is typically rooted into the field of *formal verification*, the branch of formal methods aimed at checking whether a system meets some property of interest. Being the system dynamic, properties/constraints to be monitored are usually expressed in some temporal logic, that is, a modal logic whose modal operators predicate about the evolution of the system along time.

Among all temporal logics used in verification, **Linear-time Temporal Logic (LTL)** is particularly suited for monitoring, as it matches the fact that a system execution can be captured as of an evolving, linear sequence of events. Since the traditional interpretation of LTL is over infinite traces, an LTL monitor considers the trace of a running execution as the prefix of an infinite trace that will continue forever [4, 5]. This hypothesis falls short when monitoring executions of operational (business) processes, such as the aforementioned customer flows and clinical pathways. Here, the standard assumption is that each trace produced by the system is in fact finite, as each process instance is expected to eventually reach one of the foreseen ending states of the process [44]. In this setting, a monitored trace is the known prefix of a still-to-happen, finite-length full

trace. Importantly, the overall length of such a full trace is unknown and not even bounded by a maximum value. This makes monitoring technically challenging, as the future continuation of a monitored trace will be one of infinitely many different suffixes of finite length.

To handle this type of setting, finite-trace variants of LTL have to be considered. We stress, again, that in a finite-trace setting each trace contains finitely many elements, but no bound exists on the length of the continuation of the trace, nor on the number of continuations that have to be potentially taken into account. In this work, we start from the well-studied logic  $LTL_f$  (LTL on finite traces), introduced in Reference [19], which constitutes the formal basis for one of the main declarative process modeling approaches: DECLARE [33, 39, 44].

Following Reference [33], monitoring in  $LTL_f$  amounts to checking whether the current execution belongs to the set of admissible *prefixes* for the traces of a given  $LTL_f$  formula  $\varphi$ . To achieve such a task,  $\varphi$  is usually first translated into a corresponding finite-state automaton that recognizes all and only those *finite* traces that satisfy  $\varphi$ . This supports combined reasoning on the monitored trace and its possible future continuations, providing fine-grained feedback on the satisfaction of constraints, and in particular detection of violations as early as possible [33, 34]. Despite the presence of previous operational decision support techniques to monitoring  $LTL_f$  constraints over finite traces [33, 34], two main challenges have not yet been tackled in a systematic way.

First, several alternative semantics have been proposed to make LTL suitable for monitoring, considering the key fact that during monitoring, the truth value of a formula may change over time [5]. Among the different semantics for runtime verification of temporal logics, we consider the de facto standard RV monitor conditions [4], which interpret LTL formulae using four distinct truth values that account at once for the current trace and its possible future continuations. Specifically, in the RV-LTL framework, a formula is associated to a corresponding RV state, which may witness: (i) *permanent violation* (the formula is currently violated, and the violation cannot be repaired anymore); (ii) *temporary violation* (the formula is currently violated but it is possible to continue the execution in a way that makes the formula satisfied); (iii) *permanent satisfaction* (the formula is currently satisfied and it will stay satisfied no matter how the execution continues); (iv) *temporary satisfaction* (the formula is currently satisfied but may become violated in the future). The main issue here is that no comprehensive, formal framework based on finite traces is available to handle such RV states. On the one hand, this is because runtime verification for temporal logics has been systematically studied only in the infinite-trace setting [4]. On the other hand, the incorporation of such an RV semantics in a finite-trace setting has only been tackled so far with ad hoc techniques. This is in particular the case of Reference [33], which operationally proposes to “color” automata with the RV states, but it does not come with an underlying formal counterpart justifying the correctness of the approach.

A second, fundamental challenge is the incorporation of advanced forms of monitoring, going beyond what can be expressed with  $LTL_f$ . In particular, contemporary monitoring approaches do not systematically account for *metaconstraints* that predicate on the RV state of other constraints. This is especially important in a monitoring setting, where it is often of interest to consider certain constraints only when specific circumstances arise, such as when other constraints become violated.

In this article, we attack these two challenges by proposing an end-to-end formal and operational framework for monitoring constraints expressed in  $LTL_f$  and in its extension  $LDL_f$  [19].  $LDL_f$  is a powerful logic that completely captures Monadic Second-order Logic over finite traces and that is, in turn, expressively equivalent to the language of regular expressions.  $LDL_f$  does so by combining regular expressions with  $LTL_f$ , adopting the syntax of **Propositional Dynamic Logic (PDL)**. This provides a balanced integration between the expressiveness of regular expressions, and the declarativeness of  $LTL_f$ . Interestingly, in spite of the greater expressivity of  $LDL_f$  with respect to  $LTL_f$ ,

reasoning in  $LDL_f$  is performed with the same algorithms and hence has the same computational complexity of  $LTL_f$ .

Our first, technical contribution is the formal development, accompanied by a proof-of-concept implementation, of a framework for monitoring  $LTL_f$  and  $LDL_f$  constraints using the four truth values of the RV approach. The framework is based on standard finite-state automata, without a detour to Büchi automata that a treatment based on infinite trace setting would require. We do this in two steps. In the first step, we devise a direct translation of  $LDL_f$  (and hence of  $LTL_f$ ) formulae into nondeterministic automata. The technique is grounded on **alternating automata** (AFW), but it actually avoids their introduction all together: in fact, the technique directly produces a standard **non-deterministic finite-state automaton** (NFA), which can then be manipulated using conventional automata techniques (such as determinization and minimization). In the second step, we show that  $LDL_f$  is able to capture, in the logic itself, special formulae that capture all RV monitoring conditions. More specifically, given an arbitrary  $LDL_f$  formula  $\varphi$ , we show how to construct, for each RV monitor condition, another  $LDL_f$  formula that characterizes all and only those traces culminating in a time point where  $\varphi$  is associated to that RV state. By studying the so-obtained four  $LDL_f$  special formulae, we then describe how to construct a single automaton that, given a trace, outputs the RV state associated to  $\varphi$  by that trace. This, in turn, provides for the first time a proof of correctness of the “colored automata” approach proposed in Reference [33].

We exploit this meta-level ability of  $LDL_f$  in our second contribution, which shows how to use the logic to capture *metaconstraints*, and how to monitor them by relying on the standard logical reasoning procedures of satisfiability and model checking instead of ad hoc algorithms. Metaconstraints provide a well-founded, declarative basis to specify and monitor constraints depending on the monitoring state of other constraints. To concretely show the flexibility and effectiveness of our approach, we introduce and study three interesting classes of metaconstraints.

- (1) The first class is about *contextualizing* a constraint, by expressing that it has to be enforced only in those time points where another constraint is in a given RV state. This extends basic forms of contextualization based on the presence of special events, as those present in Reference [21].
- (2) The second class deals with different forms of *compensation constraints*, which capture that a compensating constraint has to be monitored when another constraint becomes permanently violated. This mechanism can be used to express temporal versions of so-called *contrary-to-duty obligations* [48] in normative reasoning, that is, obligations that are put in place only when other obligations are violated. While compensation is considered to be a fundamental compliance monitoring functionality [32], it is still largely unexplored, and to the best of our knowledge no existing framework supports it at the level of the constraint specification language.
- (3) The third and last metaconstraint class we consider targets the interesting case of *conflicting constraints*, that is, constraints that, depending on the circumstances, may contradict each other. In particular, we show how to express a *preference* on which constraint should be satisfied when a contradiction arises. This is of particular importance in all those settings where constraints are elicited from different sources, which may sometimes contradict each other; a prominent example here is that of clinical guidelines when dealing with comorbidities [45] and background medical knowledge [7].

In the final part of the article, we report on how our monitoring framework has been concretely implemented as part of the *RuM toolkit*,<sup>1</sup> the most actively maintained framework for Declare [1, 2].

<sup>1</sup><https://rulemining.org>.

We also evaluate the time and space required to construct monitors, using a real-life model in the medical domain as a basis.

This article is a largely extended version of the conference paper in Reference [16]. In relation with Reference [16], we expand all technical parts, including here full proofs of the obtained results and a completely novel part on the construction of “colored automata” for monitoring. In addition, we provide here a novel account on metaconstraints, defining them in general and then including three relevant metaconstraint classes that have not yet been investigated in prior work. We also report here on the complete implementation of our monitoring framework and its evaluation on a real example.

The rest of the article is structured as follows: In Section 2, we introduce syntax and semantics of  $LDL_f$  and  $LTL_f$  and discuss how  $LTL_f$  is employed by the declarative business process modeling language DECLARE. In Section 3, we show how an  $LDL_f/LTL_f$  formula can be translated into a corresponding NFA that accepts all and only the traces that satisfy the formula. In Section 4, we prove how  $LDL_f$  is able to capture the RV states in the logic itself and employ the automata-theoretic approach developed in Section 3 to construct RV monitors for  $LDL_f/LTL_f$  formulae. In Section 5, we discuss how the resulting framework can be applied in the context of the DECLARE constraint-based process modeling approach. In Section 6, we turn to metaconstraints, introducing the three interesting metaconstraint classes of contextualization, compensation, and preference in case of conflict. The implementation of our monitoring framework in Java and *RuM* is reported in Section 7. We end the article with conclusions and references to future work.

## 2 LINEAR TEMPORAL LOGICS ON FINITE TRACES AND APPLICATION TO BPM

In this section, we describe the two temporal logics on finite traces we adopt to monitor constraints and metaconstraints:  $LTL_f$  (**linear temporal logic interpreted on finite traces**) and its extension  $LDL_f$ . In addition, we recall how  $LTL_f$  is employed within BPM to capture declarative, constraint-based processes.

### 2.1 $LTL_f$ : LTL on Finite Traces

LTL on finite traces, called  $LTL_f$  [19], has exactly the same syntax as LTL on infinite traces [46]. Namely, given a set  $\mathcal{P}$  of propositional symbols,  $LTL_f$  formulae are obtained through the following grammar:

$$\phi ::= \phi \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \circ\phi \mid \bullet\phi \mid \diamond\phi \mid \square\phi \mid \phi_1 \mathcal{U} \phi_2 \mid \phi_1 \mathcal{R} \phi_2,$$

where  $\phi$  is a propositional formula over  $\mathcal{P}$  (employing the usual Boolean connectives).

Intuitively,  $\circ\phi$  says that  $\phi$  holds at the *next* instant;  $\bullet\phi$  is the *weak next* operator, that is,  $\phi$  holds in the next instant if such an instant exists<sup>2</sup>;  $\diamond\phi$  says that *eventually* in some future instant  $\phi$  holds;  $\square\phi$  says that  $\phi$  *always* hold, i.e., holds for all future instants in the trace;  $\phi_1 \mathcal{U} \phi_2$  says that at some eventually  $\phi_2$  holds and *until* that point  $\phi_1$  holds; and  $\phi_1 \mathcal{R} \phi_2$  says that  $\phi_1$  *releases*  $\phi_2$ , i.e.,  $\phi_2$  holds until and including once  $\phi_1$  is true.

All these operators can be reduced to only  $\circ\phi$  and  $\phi_1 \mathcal{U} \phi_2$ , exploiting negation. In fact,  $\bullet\phi$  is equivalent to  $\neg\circ\neg\phi$ ;  $\diamond\phi$  is equivalent to  $true \mathcal{U} \phi$ ;  $\square\phi$  is equivalent to  $\neg\diamond\neg\phi$ ;  $\phi_1 \mathcal{R} \phi_2$  is equivalent to  $\neg(\neg\phi_1 \mathcal{U} \neg\phi_2)$ .

However, sometimes it is convenient to drop negation altogether, keeping it only on propositional formulae.<sup>3</sup> This is possible without loss of generality by exploiting the equivalence above.

<sup>2</sup>Note that in LTL on infinite trace, we have that  $\neg\circ\phi$  is equivalent to  $\circ\neg\phi$ . This is not the case in  $LTL_f$ , since  $\neg\circ\phi$  is true if either the next instant does not exist in the trace (i.e.,  $\neg\circ\phi$  is evaluated at the last of the trace) or the next instant exists and  $\neg\phi$  holds in it. In other words:  $\neg\circ\phi$  is equivalent to  $\bullet\neg\phi$ .

<sup>3</sup>Also recall that, whenever  $\phi$  is a propositional formula,  $\phi' = \neg\phi$  is a propositional formula as well.

The resulting formula is said to be in *Negation Normal Form (NNF)*. An arbitrary  $\text{LTL}_f$  formula can be put in NNF in linear time.

*Example 2.1.* Consider the  $\text{LTL}_f$  formula  $\Box(a \rightarrow (\bullet b))$ .<sup>4</sup> The formula expresses that for the entire duration of the trace, whenever  $a$  is true then  $b$  is true in the next instant, unless  $a$  is at the last instant of the trace. The formula  $\Box(a \rightarrow (\circ b))$  is similar, however, it implies that  $a$  is not true at the last instant, since otherwise the formula would require the existence of a further instant where  $b$  is true.

The semantics of  $\text{LTL}_f$  is given in terms of *finite traces* denoting finite-length, possibly empty, sequences  $\pi = \pi_0, \dots, \pi_{n-1}$  of elements from the alphabet  $2^{\mathcal{P}}$ . Each  $\pi_i$  is a propositional interpretation of the propositional symbols in  $\mathcal{P}$ : A propositional symbol  $p \in \mathcal{P}$  is true in the time instance associated to  $\pi_i$  if  $p \in \pi_i$ , false otherwise. The length  $n$  of the trace  $\pi$  is denoted by  $\text{length}(\pi) = n$ . Note that if  $n = 0$ , then we get the *empty trace*, denoted by  $\epsilon$ . Notice that, here, differently from Reference [19], we allow the empty trace  $\epsilon$  as in Reference [8]. This is convenient for composing monitors, as it will become clear later on in the article. We denote by  $\pi(i, j)$ , the segment of the trace  $\pi$  starting at the  $i$ th instant and ending at the  $j$ th instant (excluded). In particular,  $\pi(0, \text{length}(\pi)) = \pi$ . If  $j > \text{length}(\pi)$ , then  $\pi(i, j) = \pi(i, \text{length}(\pi))$ . If  $i \geq \text{length}(\pi)$  or  $j \leq i$ , then we have  $\pi(i, j) = \epsilon$ , i.e., the empty trace. For convenience, we may denote by  $\pi(i) = \pi(i, i+1)$  the  $i$ th instant in the trace.

Given a finite trace  $\pi$ , we inductively define when an  $\text{LTL}_f$  formula  $\varphi$  is *true* at instant  $i$  written  $\pi, i \models \varphi$ , as follows (we assume the formula in NNF for convenience):

- $\pi, i \models \phi$ , with  $\phi$  a propositional formula, iff  $0 \leq i < \text{length}(\pi)$  and  $\phi$  is true in the propositional interpretation  $\pi(i)$ ;
- $\pi, i \models \varphi_1 \wedge \varphi_2$  iff  $\pi, i \models \varphi_1$  and  $\pi, i \models \varphi_2$ ;
- $\pi, i \models \varphi_1 \vee \varphi_2$  iff  $\pi, i \models \varphi_1$  or  $\pi, i \models \varphi_2$ ;
- $\pi, i \models \circ\varphi$  iff  $0 \leq i < \text{length}(\pi) - 1$  and  $\pi, i+1 \models \varphi$ ;
- $\pi, i \models \bullet\varphi$  iff  $0 \leq i < \text{length}(\pi) - 1$  implies  $\pi, i+1 \models \varphi$ ;
- $\pi, i \models \diamond\varphi$  iff for some  $j$  s.t.  $0 \leq i \leq j < \text{length}(\pi)$ , we have  $\pi, j \models \varphi$ ;
- $\pi, i \models \square\varphi$  iff for all  $j$  s.t.  $0 \leq i \leq j < \text{length}(\pi)$ , we have  $\pi, j \models \varphi$ ;
- $\pi, i \models \varphi_1 \mathcal{U} \varphi_2$  iff for some  $j$  s.t.  $1 \leq i \leq j < \text{length}(\pi)$ , we have  $\pi, j \models \varphi_2$ , and for all  $k$ ,  $i \leq k < j$ , we have  $\pi, k \models \varphi_1$ ;
- $\pi, i \models \varphi_1 \mathcal{R} \varphi_2$  iff for all  $j$  s.t.  $0 \leq i \leq j < \text{length}(\pi)$ , either we have  $\pi, j \models \varphi_2$  or for some  $k$ ,  $i \leq k < j$ , we have  $\pi, k \models \varphi_1$ .

As usual, we write  $\pi \models \varphi$  as a shortcut for  $\pi, 0 \models \varphi$ . Whenever  $\pi \models \varphi$ , we say that  $\varphi$  is true in  $\pi$  or, equivalently, that  $\pi$  satisfies  $\varphi$ .

*Example 2.2.* Consider the set  $\mathcal{P} = \{a, b, c\}$  of propositional symbols, and the formulae  $\varphi_1 = \Box(a \rightarrow (\bullet b))$  and  $\varphi_2 = \Box(a \rightarrow (\circ b))$  from Example 2.1. We evaluate  $\varphi_1$  and  $\varphi_2$  over the following three traces:

	0	1	2	3	4
$\pi_1$	{a, c},	{b}			
$\pi_2$	{a, c},	{b},	$\emptyset$ ,	{a, b},	
$\pi_3$	{a, c},	{b},	$\emptyset$ ,	{a, b},	{c}

<sup>4</sup>As usual  $\alpha \rightarrow \beta$  stands for  $\neg\alpha \vee \beta$ .



Trace  $\pi_1 = \{a, c\}, \{b\}$  has a length of 2, as it contains two time instants, the first where  $a$  and  $c$  are true, and the second where  $b$  is true. It satisfies both  $\varphi_1$  and  $\varphi_2$ . Intuitively,  $\pi_1$  satisfies  $\varphi_2$  because every time  $a$  is in the propositional interpretation of an instant of  $\pi_1$  (which only happens at instant 1), the next instant is so  $b$  is true there; this also means that  $\varphi_1$  is satisfied by  $\pi_1$  as well, being  $\varphi_1$  a weaker form of  $\varphi_2$ . Trace  $\pi_2$ , of which  $\pi_1$  is a prefix, satisfies  $\varphi_1$  but violates  $\varphi_2$ . This is because the second occurrence of  $a$  in  $\pi_1$ , which happens at time instant 3, is in the final instant of the trace, which is not compatible with  $\varphi_2$ : To be satisfied,  $\varphi_2$  would require the existence of time instant 4 in  $\pi_2$ , with  $b$  belonging to the propositional interpretation of such an instant. Finally, both  $\varphi_1$  and  $\varphi_2$  are false in  $\pi_3$ , which extends  $\pi_2$  with a further time instant 4 where  $b$  is false.

Let us show how the intuitive discussion carried out so far is formalized by the semantics of the logic. By definition of  $\Box$ , formula  $\varphi_1$  is satisfied by a trace if, in every time instant of the trace, the inner formula  $a \rightarrow (\bullet b) = \neg a \vee \bullet b$  is true. This holds if every instant falls under one of the following three cases:

- (1) it does not contain  $a$ , or
- (2) it satisfies  $\bullet b$ , which in turn holds if:
  - (a) it does not have a next instant in the trace, or
  - (b) it has a next instant, and  $b$  is true therein.

With these cases at hand, it is immediate to see how  $\varphi_1$  is evaluated over the three traces:

- $\pi_1$  satisfies  $\varphi_1$ : instant 0 obeys to case (2b), and instant 1 to case (1).
- $\pi_2$  also satisfies  $\varphi_1$ : instant 0 obeys to case (2b), instants 1 and 2 to case (1), and instant 3 to case (2a).
- $\pi_3$  does not satisfy  $\varphi_1$ : instant 3 does not obey to any of the three cases (1), (2a), and (2b).

The same line of reasoning can be applied to  $\varphi_2$ , noticing that every instant of the trace must now obey to one of the following two cases:

- (1) it does not contain  $a$ , or
- (2) it satisfies  $\circ b$ , which in turn requires that it has a next instant in the trace, and  $b$  is true therein.

We now show how the semantics defined above applies to the case where the  $LTL_f$  formula  $\varphi$  of interest is evaluated over trace  $\pi$  in a position that exceeds the length of  $\pi$ , which also indicates what happens when  $\varphi$  is evaluated over the empty trace. We start by noticing that that  $\pi, i \models \varphi$  iff  $\pi(i, length(\pi)), i \models \varphi$ . Hence, if  $i \geq length(\pi)$ , that is,  $\pi(i, length(\pi)) = \epsilon$ , then we get:

- $\pi, i \not\models \phi$ , with  $\phi$  a propositional formula;
- $\pi, i \models \varphi_1 \wedge \varphi_2$  iff  $\pi, i \models \varphi_1$  and  $\pi, i \models \varphi_2$ ;
- $\pi, i \models \varphi_1 \vee \varphi_2$  iff  $\pi, i \models \varphi_1$  or  $\pi, i \models \varphi_2$ ;
- $\pi, i \not\models \circ \varphi$ ;
- $\pi, i \models \bullet \varphi$ ;
- $\pi, i \not\models \diamond \varphi$ ;
- $\pi, i \models \square \varphi$ ;
- $\pi, i \not\models \varphi_1 \mathcal{U} \varphi_2$ ;
- $\pi, i \models \varphi_1 \mathcal{R} \varphi_2$ .

Observe that all formulas that have a sub-formula required to hold in the current instant or in the future are trivially false for  $i \geq length(\pi)$ . Indeed, by looking at the semantics of  $\phi$  propositional,  $\circ \varphi$ ,  $\diamond \varphi$ , and  $\varphi_1 \mathcal{U} \varphi_2$ , we notice that they all require the existence of an instant  $j < length(\pi)$  ( $i < length(\pi)$  for  $\phi$  propositional) where a sub-formula must hold. Such an instant does not exist

in the empty trace or when  $i \geq \text{length}(\pi)$ . This includes the case of propositional formulae, which indeed require the existence of an instance over which the formula is evaluated.

*Example 2.3.* We have  $\epsilon \models \Box(a \rightarrow (\circ b))$ , since the  $\Box$  operator requires that for every instant in the trace, the inner formula  $a \rightarrow (\circ b)$  must be true; this trivially holds for  $\epsilon$ , since it does not contain any instant at all. Instead, we have  $\epsilon \not\models a \rightarrow (\circ b)$ : This would require either that at instant 0 formula  $\neg a$  is true, or that at instant 1 formula  $b$  is true; none of these two alternatives hold for  $\epsilon$ , as it does not contain any instant.

## 2.2 The DECLARE Constraint-based Process Modeling Language

DECLARE<sup>5</sup> is a language and framework for the declarative, constraint-based modeling of processes and services based on  $\text{LTL}_f$ . A thorough treatment of constraint-based processes can be found in References [37, 42]. As a modeling language, DECLARE takes a complementary approach to that of classical, imperative process modeling. In imperative process modeling, all allowed control flows among tasks must be explicitly represented, and execution traces not falling within this set are implicitly considered as forbidden. Instead of this procedural and “closed” approach, DECLARE has a declarative, “open” flavor: The agents responsible for the process execution can freely choose in which order to perform the process tasks, provided that the resulting execution trace satisfies the business constraints of interest. This is the reason why, alongside traditional control-flow constraints such as sequence (called in DECLARE *chain succession*), DECLARE supports a variety of more refined constraints that impose loose temporal orderings, and/or that explicitly account for negative information, i.e., the explicit prohibition of task execution.

Given a set  $\mathcal{P}$  of tasks, a DECLARE model  $\mathcal{M}$  is a set of  $\text{LTL}_f$  (and hence  $\text{LDL}_f$ ) constraints over  $\mathcal{P}$ . A finite trace  $\pi$  over  $\mathcal{P}$  *complies with*  $\mathcal{M}$  if it satisfies every constraint in  $\mathcal{M}$ , that is, for every constraint  $\varphi \in \mathcal{M}$ , we have that  $\pi \models \varphi$  in  $\text{LTL}_f$  terms. Since propositional symbols in  $\mathcal{P}$  conceptually represent tasks, the presence of task  $a \in \mathcal{P}$  in an instant of a trace indicates that  $a$  is executed in that instant. Often, process traces are totally ordered, that is, they indicate one and only one task execution per instant. This is simply captured by traces where each instant comes with an interpretation that only contains one and only one symbol from  $\mathcal{P}$ .

Among all possible  $\text{LTL}_f$  constraints, some specific *patterns* have been singled out as particularly meaningful for expressing DECLARE processes, taking inspiration from Reference [21]. Such patterns are grouped into four families:

- *existence* (unary) constraints, stating that the target task must/cannot be executed (for an indicated number of repetitions);
- *choice* (binary) constraints, accounting for alternative tasks;
- *relation* (binary) constraints, connecting a source task to a target task and expressing that, whenever the source task is executed, then the target task must also be executed (possibly with additional temporal conditions);
- *negation* (binary) constraints, capturing that whenever the source task is executed, then the target task is prohibited (possibly with additional temporal conditions).






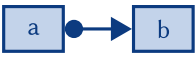


Table 1 summarizes some of these patterns. See Reference [39] for the full list of patterns.

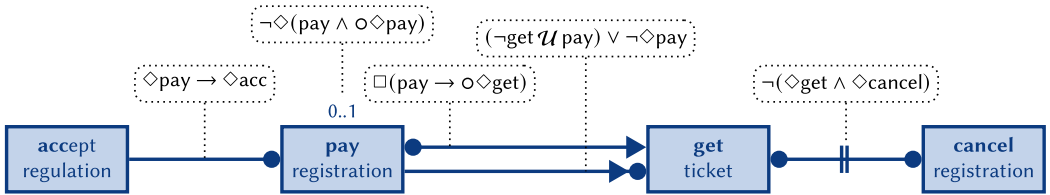
*Example 2.4.* Consider the fragment of a ticket booking process illustrated in Figure 1 using DECLARE. The process fragment consists of four tasks and four constraints, but in spite of its simplicity clearly illustrates the main features of declarative, constraint-based process modeling.

<sup>5</sup><http://www.win.tue.nl/declare/>.



Table 1. Some DECLARE Constraint Templates: Name, Graphical Representation, Intuitive Explanation, and  $LTL_f$  Semantics

NAME	NOTATION	$LTL_f$	DESCRIPTION
existence		$\diamond a$	a is executed at least once
absence		$\neg \diamond a$	a cannot be executed at all
absence 2		$\neg \diamond (a \wedge \circ \diamond a)$	a can be executed at most once
choice;		$\diamond (a \vee b)$	At least one between a and b must be executed
responded existence		$\diamond a \rightarrow \diamond b$ ;	If a is executed, then b has also to be executed (either before or later);
response		$\square (a \rightarrow \circ \diamond b)$	Whenever a is executed, then b must be executed later
precedence		$\neg b \mathcal{U} a \vee \neg \diamond b$	b can only be executed if a has been executed before
not coexistence		$\neg (\diamond a \wedge \diamond b)$	a and b cannot be both executed

Fig. 1. Fragment of a booking process in DECLARE, showing also the  $LTL_f$  formalization of the constraints used therein.

Specifically, each process instance is focused on the management of a specific customer registration to a booking event. For simplicity, we assume that the type of registration is selected upon instantiating the process and is therefore not explicitly captured as a set of tasks within the process itself. The process fragment then consists of four tasks:

- **accept regulation** is the task used to accept the regulation of the booking company for the specific type of registration the customer is interested in;
- **pay registration** is the task used to pay for the registration;
- **get ticket** is the task used to physically withdraw the ticket containing the registration details;
- **cancel registration** is the task used to abort the instance of the registration process.

The execution of the aforementioned tasks is subject to the following behavioral constraints: First, within an instance of the booking process, a customer may pay for the registration at most once. This is captured in `DECLARE` by constraining the `pay` registration task with an `absence 2` constraint.

After executing the payment, the customer must eventually get the corresponding ticket. However, the ticket can be obtained only after having performed the payment. This is captured in `DECLARE` by constraining `pay` registration and `get` ticket with a response constraint going from the first task to the second, and with a precedence constraint going from the second task to the first. This specific combination is called *succession*.

When a payment is executed, the customer must accept the regulation of the registration. There is no particular temporal order required for accepting the registration: Upon the payment, if the regulation has been already accepted, then no further steps are required; otherwise, the customer is expected to accept the regulation afterwards. This is captured in `DECLARE` by connecting `pay` registration to `accept` regulation by means of a `responded existence` constraint.

Finally, a customer may always decide to cancel the registration, with the only constraint that the cancellation is incompatible with the possibility of getting the registration ticket. This means that, when the ticket is withdrawn, cancellation is not available anymore; however, once the registration is canceled, the ticket cannot be issued. This is captured in `DECLARE` by relating `get` ticket to `cancel` registration through a `not coexistence` constraint.

We close the example by considering some compliant and non-compliant traces (using abbreviations for tasks):

- The empty trace  $\epsilon$  is compliant. The booking process does not mandatorily prescribe any task execution, but reactively constrains which executions are possible or forbidden, depending on which tasks are actually executed.
- Trace  $\{\text{pay}\}$  is not compliant, since the response constraint relating `pay` to `get` requires that, in an instant strictly following instant 0, `get` is executed.
- Traces  $\{\text{pay}\}$ ,  $\{\text{acc}\}$ ,  $\{\text{get}\}$  and  $\{\text{acc}\}$ ,  $\{\text{pay}\}$ ,  $\{\text{get}\}$  are both compliant. The presence of an instant where `pay` is executed interacts with three constraints. The first is the `responded existence` that requires the presence of `acc` in some instant (which follows the payment in the first trace and precedes the payment in the second). The second is the `absence 2` constraint over `pay`, which forbids a second execution of the same task. The third is the *response* constraint requiring an execution of `get` in a later instant (which occurs immediately afterwards in the second trace and after two steps in the first trace).
- Trace  $\{\text{pay}\}$ ,  $\{\text{acc}\}$ ,  $\{\text{get}\}$ ,  $\{\text{cancel}\}$  is not compliant, as it violates the `not coexistence` constraint indicating that `get` and `cancel` cannot be both present in the same trace.

### 2.3 $LDL_f$ : LDL on Finite Traces

One may wonder whether  $LTL_f$  is expressive enough to capture relevant properties of finite traces. A natural choice to go beyond  $LTL_f$  would be to adopt *regular expressions*, which are indeed more expressive than  $LTL_f$ . This is witnessed by the fact that  $LTL_f$  is as expressive as **First-Order Logic (FO)** over finite traces [19], which in turn corresponds to *star-free regular expressions*, a strict subset of regular expressions that in fact correspond to **Monadic Second-Order Logic (MSO)** over finite traces.

Hence, one could use regular expressions instead of  $LTL_f$  to express properties of finite traces with an expressiveness advantage. However, regular expressions miss explicit constructs for negation (complementation) and conjunction, and if we add them to the language, then we get a formalism for which reasoning becomes non-elementary [50].

To be as expressive as regular expressions and, at the same time, convenient as a temporal logic, **Linear Dynamic Logic on finite traces** ( $\text{LDL}_f$ ) has been proposed in Reference [19].  $\text{LDL}_f$  is obtained by merging  $\text{LTL}_f$  with regular expressions through the syntax of the well-known logic of programs **Propositional Dynamic Logic** ( $\text{PDL}$ ) [23, 27], but adopting a semantics based on finite traces.

Formally,  $\text{LDL}_f$  formulae are built as follows:

$$\begin{aligned} \varphi & ::= tt \mid ff \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \langle\rho\rangle\varphi \mid [\rho]\varphi \\ \rho & ::= \phi \mid \varphi? \mid \rho_1 + \rho_2 \mid \rho_1; \rho_2 \mid \rho^*, \end{aligned}$$

where:

- $tt$  and  $ff$  denote, respectively, the true and the false  $\text{LDL}_f$  formula (not to be confused with the propositional formula *true* and *false*);
- $\phi$  denotes propositional formulae over  $\mathcal{P}$ ;
- $\rho$  denotes path expressions, which are regular expressions over propositional formulae  $\phi$  over  $\mathcal{P}$  with the addition of the test construct  $\varphi?$  typical of  $\text{PDL}$ , and are used to insert into the execution path checks for satisfaction of additional  $\text{LDL}_f$  formulae;
- $+$ , likewise the Boolean “or” separates different alternatives;
- $;$  is the concatenation operator;
- $*$  is the Kleene star denoting zero or more occurrences of the preceding element;
- $\varphi$  stands for an  $\text{LDL}_f$  formula built by applying Boolean connectives and the modal operators  $\langle\rho\rangle\varphi$  and  $[\rho]\varphi$ , which are dual, since  $[\rho]\varphi$  is equivalent to  $\neg\langle\rho\rangle\neg\varphi$ .

Intuitively,  $\langle\rho\rangle\varphi$  states that, from the current instant in the trace, there exists an execution satisfying the regular expression  $\rho$  such that its last instant satisfies  $\varphi$ . Instead,  $[\rho]\varphi$  states that, from the current instant, all executions satisfying the regular expression  $\rho$  are such that their last instant satisfies  $\varphi$ .

As defined above,  $\text{LDL}_f$  only includes propositional formulae  $\phi$  as path expressions, and not directly as  $\text{LDL}_f$  formulae. However, the latter can be immediately introduced as abbreviations:  $\phi \doteq \langle\phi\rangle tt$ . For example, to say that eventually proposition  $a$  holds, instead of writing  $\langle true^* \rangle a$ , we can write  $\langle true^* \rangle a tt$ . This is analogous to what happens in (extensions with regular expressions of) XPath, a well-known formalism developed for navigating XML documents and graph databases [10, 15, 36]. We may keep  $\phi$  as  $\text{LDL}_f$  formulae for convenience, however, we have to be careful of the difference we get if we apply negation to  $\phi$  as a propositional formula or as an  $\text{LDL}_f$  formula (i.e., to  $\langle\phi\rangle tt$ ). In the first case, we get  $\neg\phi$ , which is equivalent to  $\langle\neg\phi\rangle tt$ . In the latter case, we get  $[\phi]ff$ . This, in turn, is equivalent to  $[true^*]ff \vee \langle\neg\phi\rangle tt$ , which states that either the trace is empty<sup>6</sup> or  $\neg\phi$  holds in the current instant. This is the reason why if we decide to adopt the abbreviation  $\phi$  as a direct  $\text{LDL}_f$  formula, then it is convenient to restrict  $\text{LDL}_f$  to NNF; this has the effect of dropping the negation except in propositional formulas and avoids the aforementioned ambiguity. Like for the case of  $\text{LTL}_f$ , every  $\text{LDL}_f$  formula can be easily transformed in NNF in linear time.

It is also convenient to introduce the following abbreviations, specific for dealing with the finiteness of the traces:

- $end = [true]ff$ , which denotes that the trace has been completed (the current instant is out of the range of the trace or the remaining fragment of the trace is empty);
- $last = \langle true \rangle end$ , which denotes the last instant of the trace.

<sup>6</sup> $[true^*]ff$  is indeed satisfied only by the empty trace, since it expresses that every continuation of the trace must culminate in a state where  $ff$  holds, which means that no such state should exist.

*Example 2.5.* Let us consider a couple of  $\text{LDL}_f$  examples. To express that *before entering restricted area a, a person must have permission for a*, we can write

$$\langle (\neg \text{inArea}_a)^* \rangle (\text{getPerm}_a \wedge \neg \text{inArea}_a) \vee [\text{true}^*] \neg \text{inArea}_a.$$

To express the more demanding requirement that *each time a person enters the restricted area a it must have a new permission for a*, we can write:

$$\varphi_{\text{perm}} = \langle (\neg \text{inArea}_a^*; \text{getPerm}_a \wedge \neg \text{inArea}_a; \neg \text{inArea}_a^*; \text{inArea}_a)^* \rangle \text{end}.$$

Note that while the first property can also be easily expressed in  $\text{LTL}_f$  as  $\neg \text{inArea}_a \mathcal{U} (\text{getPerm}_a \wedge \neg \text{inArea}_a) \vee \square \neg \text{inArea}_a$ , this is not the case for the second one.

As for  $\text{LTL}_f$ , the semantics of  $\text{LDL}_f$  is given in terms of *finite traces*, including *empty traces*. An  $\text{LDL}_f$  formula  $\varphi$  is true at instant  $i$ , written  $\pi, i \models \varphi$ , if:

- $\pi, i \models tt$ ;
- $\pi, i \not\models ff$ ;
- $\pi, i \models \neg\varphi$  iff  $\pi, i \not\models \varphi$ ;
- $\pi, i \models \varphi_1 \wedge \varphi_2$  iff  $\pi, i \models \varphi_1$  and  $\pi, i \models \varphi_2$ ;
- $\pi, i \models \varphi_1 \vee \varphi_2$  iff  $\pi, i \models \varphi_1$  or  $\pi, i \models \varphi_2$ ;
- $\pi, i \models \langle \rho \rangle \varphi$  iff for some  $j$  s.t.  $i \leq j$ , we have  $\pi(i, j) \in \mathcal{L}(\rho)$  and  $\pi, j \models \varphi$ ;
- $\pi, i \models [\rho] \varphi$  iff for all  $j$  s.t.  $i \leq j$  and  $\pi(i, j) \in \mathcal{L}(\rho)$ , we have  $\pi, j \models \varphi$ .

The relation  $\pi(i, j) \in \mathcal{L}(\rho)$  is defined inductively as follows:

- $\pi(i, j) \in \mathcal{L}(\phi)$  iff  $j = i + 1$  and  $0 \leq i < \text{length}(\pi)$  and  $\pi, i \models \phi$  ( $\phi$  propositional);
- $\pi(i, j) \in \mathcal{L}(\varphi?)$  iff  $j = i$  and  $\pi, i \models \varphi$ ;
- $\pi(i, j) \in \mathcal{L}(\rho_1 + \rho_2)$  iff  $\pi(i, j) \in \mathcal{L}(\rho_1)$  or  $\pi(i, j) \in \mathcal{L}(\rho_2)$ ;
- $\pi(i, j) \in \mathcal{L}(\rho_1; \rho_2)$  iff exists  $k$  s.t.  $\pi(i, k) \in \mathcal{L}(\rho_1)$  and  $\pi(k, j) \in \mathcal{L}(\rho_2)$ ;
- $\pi(i, j) \in \mathcal{L}(\rho^*)$  iff  $j = i$  or exists  $k$  s.t.  $\pi(i, k) \in \mathcal{L}(\rho)$  and  $\pi(k, j) \in \mathcal{L}(\rho^*)$ .

Note that if  $i \geq \text{length}(\pi)$ , hence, e.g., for  $\pi = \epsilon$ , the above definitions still apply; though,  $\langle \phi \rangle \varphi$  (with  $\phi$  propositional) and  $\langle \psi \rangle \varphi$  become trivially false. As before, we write  $\pi \models \varphi$  as a shortcut for  $\pi, 0 \models \varphi$ .

Finally, in agreement with the semantics given above, we can introduce the path expression  $\epsilon$ , matching with the empty trace, as abbreviation of  $(\text{false})^*$ .

*Example 2.6.* Consider formula  $\varphi_{\text{perm}}$  from Example 2.5 and a set  $\mathcal{P} = \{\text{inArea}_a, \text{getPerm}_a\}$  of propositional symbols only containing the two symbols mentioned in the formula. We discuss some examples of satisfying and non-satisfying traces for  $\varphi_{\text{perm}}$ . In general, satisfying traces are those and only those characterized by the regular expression contained within the  $\langle \cdot \rangle$  operator. In addition, recall that interpretation  $\{\text{getPerm}_a\}$  satisfies both the propositional formulae  $\text{getPerm}_a$  and  $\neg \text{inArea}_a$ , thus providing different ways of matching the sub-expressions used in  $\varphi_{\text{perm}}$ ; recall also that interpretation  $\emptyset$  satisfies  $\neg \text{inArea}_a$ . With these observations at hand, we get the following:

- $\emptyset, \{\text{getPerm}_a\}, \emptyset, \{\text{getPerm}_a\}$  satisfies  $\varphi_{\text{perm}}$ , since, according to the formula, permissions can be asked multiple times and may not be necessarily used to enter the area.
- $\{\text{getPerm}_a\}, \{\text{inArea}_a\}, \emptyset, \emptyset$  satisfies  $\varphi_{\text{perm}}$ , providing a sample trace where a single permission is asked and used to access the area later.
- $\{\text{getPerm}_a\}, \{\text{inArea}_a\}, \emptyset, \{\text{inArea}_a\}$  does not satisfy  $\varphi_{\text{perm}}$ , since, after instant 1, it is only possible to find a later instant containing  $\text{inArea}_a$  if, in between, there is an instant where  $\text{getPerm}_a$  holds.

- $\{getPerm_a\}, \{inArea_a, getPerm_a\}, \{inArea_a\}$  does not satisfy  $\varphi_{perm}$ , since instant 0 correctly matches the regular expression  $(\neg inArea_a)^*$ ;  $getPerm_a \wedge \neg inArea_a$ , instant 1 correctly matches the following regular expression  $(\neg inArea_a)^*$ ;  $inArea_a$ , but instant 2 matches none of the two consequent possible regular expressions, which are either  $(\neg inArea_a)^*$  or  $(\neg inArea_a)^*$ ;  $getPerm_a \wedge \neg inArea_a$ ; this correctly indicates that a permission can be used to access the area only if it occurs when being outside the area.
- $\{getPerm_a\}, \emptyset, \emptyset, \{inArea_a\}, \{getPerm_a\}, \{inArea_a\}$  satisfies  $\varphi_{perm}$ , providing a sample trace where there is a proper alternation between permissions and accesses to the area, possibly done in a later instant (not necessarily occurring immediately after the obtained permission).

It is easy to encode  $LTL_f$  into  $LDL_f$ : Following Reference [19], we do so by introducing a translation function  $tr$  defined by induction on the structure of the input  $LTL_f$  formula as follows (for convenience, we assume the  $LTL_f$  formula to be in NNF):

$$\begin{aligned}
tr(\phi) &= \langle \phi \rangle tt \quad (\phi \text{ propositional}) \\
tr(\varphi_1 \wedge \varphi_2) &= tr(\varphi_1) \wedge tr(\varphi_2) \\
tr(\varphi_1 \vee \varphi_2) &= tr(\varphi_1) \vee tr(\varphi_2) \\
tr(\bigcirc \varphi) &= \langle true \rangle (tr(\varphi) \wedge \neg end) \\
tr(\bullet \varphi) &= [true](tr(\varphi) \vee end) \\
tr(\heartsuit \varphi) &= \langle true^* \rangle (tr(\varphi) \wedge \neg end) \\
tr(\square \varphi) &= [true^*](tr(\varphi) \vee end) \\
tr(\varphi_1 \mathcal{U} \varphi_2) &= \langle (tr(\varphi_1)?; true)^* \rangle (tr(\varphi_2) \wedge \neg end) \\
tr(\varphi_1 \mathcal{R} \varphi_2) &= [((\neg tr(\varphi_1))?; true)^*](tr(\varphi_2) \vee end).
\end{aligned}$$

By induction on the structure of the input formula, and by exploiting the semantics of  $LTL_f$  and  $LDL_f$ , we can prove the following correctness result for the translation function  $tr$ :

**PROPOSITION 2.7.** *Let  $\psi$  an  $LTL_f$  formula (in NNF),  $\pi$  a trace (possibly the empty trace  $\epsilon$ ) and  $i > 0$ . Then  $\pi, i \models \psi$  if and only if  $\pi, i \models tr(\psi)$ .*

*Example 2.8.* Consider the  $LTL_f$  formula  $\square(a \rightarrow (\bullet b))$  of Example 2.1. The  $LDL_f$  counterpart obtained by the translation is  $[true^*](\langle \neg a \rangle tt \vee ([true](\langle b \rangle tt \vee end)) \vee end)$ .

Thanks to Proposition 2.7, we can use  $LDL_f$  to express all patterns defined for DECLARE (cf. Section 2.2).

Finally, it is also easy to encode regular expressions, used as a specification formalism for traces, into  $LDL_f$ : Regular expression  $\rho$  simply translates to  $\langle \rho \rangle end$ .

We say that a trace satisfies an  $LTL_f/LDL_f$  formula  $\varphi$ , written  $\pi \models \varphi$ , if  $\pi, 0 \models \varphi$ . Note that if  $\pi$  is the empty trace, and hence 0 is out of range, then still the notion of  $\pi, 0 \models \varphi$  is well defined. Also, sometimes we denote by  $\mathcal{L}(\varphi)$  the set of traces that satisfy  $\varphi$ , i.e.,  $\mathcal{L}(\varphi) = \{\pi \mid \pi \models \varphi\}$ .

### 3 $LDL_f$ AUTOMATON

We can associate to each  $LDL_f$  formula  $\varphi$  a corresponding (exponential) nondeterministic finite-state automaton. This automaton provides the basis for carrying out reasoning in  $LDL_f$ , as well as advanced monitoring tasks, as illustrated in Section 4.1.

A **nondeterministic finite-state automaton** (NFA) is a tuple  $A = (\Sigma, \mathcal{S}, s_0, \varrho, \mathcal{S}_f)$ , where: (i)  $\Sigma$  is a finite alphabet of symbols; (ii)  $\mathcal{S}$  is a finite set of states; (iii)  $s_0 \in \mathcal{S}$  is the initial state; (iv)  $\varrho : \mathcal{S} \times \Sigma \times \mathcal{S}$  is a ( $\Sigma$ -labeled) transition relation; (v)  $\mathcal{S}_f \subseteq \mathcal{S}$  is the set of final states. We say that the NFA  $A$  is a **deterministic finite-state automaton** (DFA) if, for each state  $s \in \mathcal{S}$  and symbol  $\ell \in \Sigma$ , there exists at most one state  $s' \in \mathcal{S}$  such that  $(s, \ell, s') \in \varrho$ . A finite sequence  $\ell_1, \dots, \ell_n \in \Sigma^*$  is accepted by  $A$  if there exists a sequence of states  $s_0 \dots s_n$  starting from the initial state  $s_0$  and ending in a

$$\begin{aligned}
\delta("tt", \Pi) &= true \\
\delta("ff", \Pi) &= false \\
\delta("ϕ", \Pi) &= \delta("<\phi>tt", \Pi) \quad (\phi \text{ prop.}) \\
\delta("<\phi_1 \wedge \phi_2>", \Pi) &= \delta("<\phi_1>", \Pi) \wedge \delta("<\phi_2>", \Pi) \\
\delta("<\phi_1 \vee \phi_2>", \Pi) &= \delta("<\phi_1>", \Pi) \vee \delta("<\phi_2>", \Pi) \\
\delta("<\phi>\varphi", \Pi) &= \begin{cases} \mathbf{E}("<\varphi>") & \text{if } \Pi \models \phi \quad (\phi \text{ prop.}) \\ false & \text{if } \Pi \not\models \phi \end{cases} \\
\delta("<\psi?>\varphi", \Pi) &= \delta("<\psi>", \Pi) \wedge \delta("<\varphi>", \Pi) \\
\delta("<\rho_1 + \rho_2>\varphi", \Pi) &= \delta("<\rho_1>\varphi", \Pi) \vee \delta("<\rho_2>\varphi", \Pi) \\
\delta("<\rho_1; \rho_2>\varphi", \Pi) &= \delta("<\rho_1><\rho_2>\varphi", \Pi) \\
\delta("<\rho^*>\varphi", \Pi) &= \delta("<\varphi>", \Pi) \vee \delta("<\rho>\mathbf{F}_{<\rho^*>\varphi}", \Pi) \\
\delta("[\phi]\varphi", \Pi) &= \begin{cases} \mathbf{E}("<\varphi>") & \text{if } \Pi \models \phi \quad (\phi \text{ prop.}) \\ true & \text{if } \Pi \not\models \phi \end{cases} \\
\delta("[\psi?]\varphi", \Pi) &= \delta(nnf("<\neg\psi>"), \Pi) \vee \delta("<\varphi>", \Pi) \\
\delta("[\rho_1 + \rho_2]\varphi", \Pi) &= \delta("[\rho_1]\varphi'", \Pi) \wedge \delta("[\rho_2]\varphi", \Pi) \\
\delta("[\rho_1; \rho_2]\varphi", \Pi) &= \delta("[\rho_1][\rho_2]\varphi", \Pi) \\
\delta("[\rho^*]\varphi", \Pi) &= \delta("<\varphi>", \Pi) \wedge \delta("[\rho]\mathbf{T}_{[\rho^*]\varphi}", \Pi) \\
\delta("<\mathbf{F}_\psi>", \Pi) &= false \\
\delta("<\mathbf{T}_\psi>", \Pi) &= true
\end{aligned}$$

Fig. 2. Definition of  $\delta$ , where  $\mathbf{E}("<\varphi>")$  recursively replaces in " $\varphi$ " all occurrences of atoms of the form  $\mathbf{T}_\psi$  and  $\mathbf{F}_\psi$  by  $\psi$ .

final state  $s_n \in S_f$  such that for every  $i \in \{1, n\}$ , we have  $(s_{i-1}, \ell_i, s_i) \in \varrho$ . The empty sequence  $\epsilon$  is accepted by  $A$  if  $s_0 \in S_f$ . The language  $\mathcal{L}(A)$  of  $A$  is the set of all and only sequences from  $\Sigma^*$  that are accepted by  $A$ . It is well known that every NFA can be determinized into a language-preserving DFA.<sup>7</sup>

We now provide an algorithm that, given an  $\text{LDL}_f$  formula  $\varphi$ , computes a corresponding NFA  $A(\varphi)$  that accepts exactly those traces that make  $\varphi$  true. To do so, we first define an auxiliary function  $\delta$  that takes an  $\text{LDL}_f$  quoted formula " $\psi$ " in negation normal form, to be considered as an atom (i.e., a propositional atomic formula), and a propositional interpretation  $\Pi$  for  $\mathcal{P}$  and returns a positive Boolean formula whose atoms are quoted sub-formulae of " $\psi$ ". The function  $\delta$  is detailed in Figure 2.<sup>8</sup>

The intuition behind  $\delta("<\psi>", \Pi)$  is the following: The function *unfolds* the temporal operators in  $\psi$  by one (temporal) step only, therefore telling us what is yet to be checked to satisfy  $\psi$  (in the future) when we see the propositional symbols in  $\Pi$  (in the current instant). In other words, let us

<sup>7</sup>Note that, while the determinization of an NFA can generate an exponentially larger DFA, in practice this is often not the case, especially after minimization. Indeed, the size of a minimized DFA is often similar, if not smaller, to that of the original NFA. This phenomenon has been observed over the years (e.g., in Reference [52]) and recently it is at the base of the best-performing tools for synthesis from  $\text{LTL}_f$  and  $\text{LDL}_f$  specifications (e.g., References [17, 51, 55]).

<sup>8</sup>The function  $\delta$  is in fact the transition function of a *finite alternating automaton on words* (AFW), and indeed, we will exploit this fact when proving correctness below. However, to understand and implement the construction, we do not need to detour to AFW, as it is sufficient to consider  $\delta$  as an auxiliary function to be used by the algorithm in Figure 3.



```

1: algorithm LDLf2NFA
2: input LDLf formula  $\varphi$ 
3: output NFA  $A(\varphi) = (2^{\mathcal{P}}, \mathcal{S}, s_0, \varrho, \mathcal{S}_f)$ 
4:  $s_0 \leftarrow \{\text{"}\varphi\text{"}\}$  ▷ set the initial state
5:  $\mathcal{S}_f \leftarrow \{\emptyset\}$  ▷ set final states
6: if  $(\delta(\text{"}\varphi\text{"}, \epsilon) = \text{true})$  then ▷ check if initial state is also final
7:    $\mathcal{S}_f \leftarrow \mathcal{S}_f \cup \{s_0\}$ 
8:  $\mathcal{S} \leftarrow \{s_0, \emptyset\}, \varrho \leftarrow \emptyset$ 
9: while ( $\mathcal{S}$  or  $\varrho$  change) do
10:  for ( $s \in \mathcal{S}$ ) do
11:    if ( $s' \models \bigwedge_{\langle \psi \rangle \in \mathcal{S}} \delta(\text{"}\psi\text{"}, \Pi)$ ) then ▷ add new state and transition
12:       $\mathcal{S} \leftarrow \mathcal{S} \cup \{s'\}$ 
13:       $\varrho \leftarrow \varrho \cup \{(s, \Pi, s')\}$ 
14:      if ( $\bigwedge_{\langle \psi \rangle \in \mathcal{S}'} \delta(\text{"}\psi\text{"}, \epsilon) = \text{true}$ ) then ▷ check if new state is also final
15:         $\mathcal{S}_f \leftarrow \mathcal{S}_f \cup \{s'\}$ 

```

Fig. 3. NFA construction.

assume to have a formula  $\psi$  and a trace  $\pi = \pi_0, \pi_1, \dots, \pi_{n-1}$  and our objective is to check whether  $\pi, 0 \models \psi$  (recall that each instant  $i$  of  $\pi$  is a propositional interpretation  $\pi_i$  over  $\mathcal{P}$ ). We start from the first time instant  $\pi_0$  and we call  $\delta(\text{"}\psi\text{"}, \pi_0)$ . Then, we move one step forward and we call  $\delta$  on the result of the previous call and on  $\pi_1$ , and so on.

In defining  $\delta$ , apart from quoted sub-formulae of  $\psi$ , we also make use of extra atoms  $\mathbf{F}_{\langle \rho^* \rangle \varphi}$  and  $\mathbf{T}_{[\rho^*] \varphi}$ . These are used for handling the recursion induced by sub-formulae of the form  $\langle \rho^* \rangle \varphi$  and  $[\rho^*] \varphi$ . Such extra symbols act in  $\delta$  as if they were additional quoted formulas, except that during the recursive computation of  $\delta$  they disappear, either because evaluated to *true* or *false* or because they are syntactically replaced by  $\langle \rho \rangle \varphi$  and  $[\rho] \varphi$ , respectively, when the current instant of the trace (i.e., the current symbol in the word) is consumed and a new Boolean combination of quoted formulae is returned. For performing the syntactic substitution, we use an auxiliary function  $\mathbf{E}$ , which takes as input a quoted formula “ $\varphi$ ” and returns the quoted formula obtained by recursively substituting all occurrences of these extra atoms  $\mathbf{F}_{\psi}$  and  $\mathbf{T}_{\psi}$  with the formula  $\psi$  itself.

The function  $\delta$  is immediately extended to the case where the trace is empty: We add a special symbol  $\epsilon$  to the propositional interpretations  $\Pi$  and extend  $\delta$  in Figure 2 to  $\epsilon$  by keeping exactly the same rules except for the following cases:

$$\begin{aligned} \delta(\text{"}\langle \phi \rangle \varphi\text{"}, \epsilon) &= \text{false} \quad (\phi \text{ propositional}), \\ \delta(\text{"}[\phi] \varphi\text{"}, \epsilon) &= \text{true} \quad (\phi \text{ propositional}). \end{aligned}$$

As a result,  $\delta(\text{"}\varphi\text{"}, \epsilon)$  always returns a Boolean combination of quoted sub-formulae, which trivially simplifies to either *true* or *false*.

Using  $\delta$  as an auxiliary function, the algorithm LDL<sub>f</sub>2NFA in Figure 3 takes as input an LDL<sub>f</sub> formula  $\varphi$  and produces, in a forward fashion, its corresponding NFA  $A(\varphi) = (2^{\mathcal{P}}, \mathcal{S}, s_0, \varrho, \mathcal{S}_f)$ , where: (i)  $2^{\mathcal{P}}$  (i.e., the set of all interpretations over  $\mathcal{P}$ ) is the alphabet of the automaton; (ii)  $\mathcal{S}$  is a set of states; (iii)  $s_0 \in \mathcal{S}$  is the initial state; (iv)  $\varrho : \mathcal{S} \times 2^{\mathcal{P}} \times \mathcal{S}$  is a transition relation; (v)  $\mathcal{S}_f \subseteq \mathcal{S}$  is the set of final states.

Each NFA state  $s \in \mathcal{S}$  is a set of quoted sub-formulae of  $\varphi$  to be interpreted as a conjunction of atoms. Examples of states are  $s_1 = \{[\rho_1] \varphi, \text{"}c\text{"}\}$  and  $s_2 = \{\text{"}a \vee b\text{"}\}$ . Note that the empty set  $\emptyset$  stands for empty conjunction, i.e., for *true*. Hence, we trivially have  $(\emptyset, \Pi, \emptyset) \in \varrho$  for every  $\Pi$ .

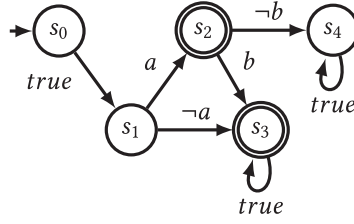


Fig. 4. Graphical representation of the DFA for the LTL<sub>f</sub> formula  $O(a \rightarrow (\bullet b))$ .

In line 11,  $s'$  is any set of quoted sub-formulae such that  $s' \models \bigwedge_{(\psi \in s)} \delta(\psi, \Pi)$ . In fact, it is enough to consider the minimal ones, that is: If  $s', s'' \models \bigwedge_{(\psi \in s)} \delta(\psi, \Pi)$  and  $s' \subseteq s''$ , then we consider only  $s'$ . The reason is that if  $s' \subseteq s''$ , then  $s' \models s''$  and hence, if  $s'' \models \bigwedge_{(\psi \in s)} \delta(\psi, \Pi)$ , also  $s' \models \bigwedge_{(\psi \in s)} \delta(\psi, \Pi)$ . In addition, note that, since  $\bigwedge_{(\psi \in s)} \delta(\psi, \Pi)$  is a positive Boolean formula, to check whether  $s' \models \bigwedge_{(\psi \in s)} \delta(\psi, \Pi)$  it is sufficient to consider  $s'$  as an interpretation (implicitly setting all atoms not occurring in  $s'$  as false and evaluate  $\bigwedge_{(\psi \in s)} \delta(\psi, \Pi)$ ). Indeed, in the evaluation those atoms that are false will not be considered.

We can show the correctness of the algorithm following Reference [19]. The correctness is based on the fact that (i) we can associate each LDL<sub>f</sub> formula  $\varphi$  with an *alternating automaton on words* (AFW) that accepts exactly those traces that make  $\varphi$  true and (ii) every AFW can be transformed into an equivalent NFA. Indeed, the auxiliary function  $\delta$  implicitly constructs the AFW for  $\varphi$ , and then LDL<sub>f</sub>2NFA transforms such AFW into a corresponding NFA. As a result, we can state the following result:

**PROPOSITION 3.1.** *Let  $\varphi$  be an LDL<sub>f</sub> formula and  $A(\varphi)$  the NFA obtained by algorithm LDL<sub>f</sub>2NFA to  $\varphi$ . Then for every trace  $\pi$  over  $\mathcal{P}$ , we have that  $\pi \models \varphi$  iff  $\pi \in \mathcal{L}(A(\varphi))$ .*

Moreover, it is easy to see that the algorithm LDL<sub>f</sub>2NFA computes in at most an exponential number of steps in the size of the input formula  $\varphi$  the (exponential) NFA corresponding to LDL<sub>f</sub> formula  $\varphi$ . Indeed, the computation of the auxiliary function  $\delta$  is polynomial in the size of  $\varphi$ , and the algorithm generates at most as many states as all subsets of (quoted) sub-formulae of  $\varphi$ .

**PROPOSITION 3.2.** *Let  $\varphi$  be an LDL<sub>f</sub> formula. Then LDL<sub>f</sub>2NFA terminates in at most an exponential number of steps, and the resulting NFA  $A(\varphi)$  has a number of states that is at most exponential in the size of  $\varphi$ .*

Obtaining the NFA from an LDL<sub>f</sub> formula provides an operational basis for reasoning. In particular, we can check the satisfiability of an LDL<sub>f</sub> formula  $\varphi$  by checking whether its corresponding NFA  $A(\varphi)$  is nonempty. The same applies for validity and logical implication, which are linearly reducible to satisfiability. Notably,  $A(\varphi)$  can be built on-the-fly, and hence, we can check non-emptiness in PSPACE in the size of  $\varphi$ . Considering that it is known that satisfiability in LDL<sub>f</sub> (in fact already for its fragment LTL<sub>f</sub>) is PSPACE-hard, we can conclude that the proposed construction is optimal with respect to the computational complexity for satisfiability (see Reference [19] for details). Note that these results hold for LTL<sub>f</sub> as well, since we can translate it into LDL<sub>f</sub> as demonstrated before.

Figure 4 shows the NFA obtained by the application of LDL<sub>f</sub>2NFA to the LTL<sub>f</sub> formula  $O(a \rightarrow (\bullet b))$ . Notice that the automaton is actually a DFA. In the figure (and in the remainder of the article),  $s_0$  always indicates the initial state, and final states are double-circled. In addition, for the sake of readability, we use logical formulae over  $\mathcal{P}$  as compact labels of transitions in an NFA, where each formula implicitly denotes all its satisfying interpretations from  $2^{\mathcal{P}}$ . For the same reason, edge-labels in the automaton are logical formulae; each formula is a shortcut for every interpretation

satisfying that formula, e.g., by assuming  $\mathcal{P} = \{a, b\}$ , the edge labeled with  $\neg a$  from state  $s_1$  to  $s_2$  is a shortcut for the two transitions  $(s_1, \{b\}, s_2), (s_1, \emptyset, s_2) \in \varrho$ . Examples of automata constructed following, step-by-step, the algorithmic construction of  $\text{LDL}_f\text{2NFA}$  can be found in Reference [22, Section. 2.5].

Finally, we observe that performing efficiently the transformation from  $\text{LTL}_f/\text{LDL}_f$  into  $\text{NFA}/\text{DFA}$  is a currently active area of research, and quite advanced tools are available [3, 17, 31, 55].

#### 4 RUNTIME MONITORING

From a high-level perspective, the monitoring problem amounts to observing an evolving system execution and reporting the violation or satisfaction of properties of interest at the earliest possible time. As the system execution progresses, the trace of events it generates gets extended. At each step, the monitor checks whether the current trace conforms to the properties, not only checking the events collected so far, but also the possible future continuations. This evolving aspect has a significant impact on the monitoring output: At each step, indeed, the outcome may have a degree of uncertainty due to the fact that future executions are yet unknown. We informally discuss this aspect in the next example.

*Example 4.1.* Consider again formula  $\Phi := \circ(a \rightarrow (\bullet b))$  from Figure 4. The formula requires that in the next step, if  $a$  is performed, then either the trace ends or, if it continues, it does so by executing  $b$ . We use  $\Phi$  to monitor one execution  $\pi$  of a system of interest. At the beginning, the execution trace is empty, namely,  $\pi_0 := \epsilon$ . Upon evaluating if  $\pi_0$  satisfies  $\Phi$ , the best we can say is that *currently* it does not. Indeed, due to the semantics of  $\circ$ ,  $\Phi$  requires the trace to contain at least two steps. At the same time, the execution can still satisfy  $\Phi$  in the future if the trace  $\pi_0$  is suitably extended. All in all, we can then indicate that  $\pi_0$  *temporarily violates*  $\Phi$ . Assume now that  $\{a, c\}$  is observed, thus evolving the current trace into  $\pi_1 := \{a, c\}$ . The monitoring verdict does not change, as the  $\circ$  operator requires at least one further step, and such a further step can indeed possibly occur if the execution suitably extends  $\pi_1$ . Upon collecting a further step  $\{b\}$ , trace  $\pi_1$  evolves into  $\pi_2 := \{a, c\}, \{b\}$ . The semantics of  $\circ$  is now fulfilled, and considering that  $\{b\} \not\models a$ , the inner implication  $a \rightarrow (\bullet b)$  is (vacuously) true, in turn making the whole formula  $\Phi$  true. We can therefore conclude that the execution *currently* satisfies  $\Phi$ , and we can even make this claim stronger. In fact, it is now certain that, no matter how the execution will continue by extending  $\pi_2$ ,  $\Phi$  will continue to stay satisfied. Consequently, we can assert that  $\pi_2$  *permanently satisfies*  $\Phi$ .

Several variants of monitoring semantics have been proposed to systematically refine the notion of satisfaction as intuitively shown in Example 4.1 (see Reference [4] for a survey). In this work, we adopt the 4-valued semantics in Reference [33], which is essentially the finite-trace variant of the infinite-trace  $\text{RV}$  semantics in Reference [4]. As we will show next, in our finite-trace setting the  $\text{RV}$  semantics can be elegantly defined, since both trace prefixes and their continuations are finite.

Given an  $\text{LTL}_f/\text{LDL}_f$  formula  $\varphi$  and a trace  $\pi$ , the monitor returns one among the following four  $\text{RV}$  states:

- *temp\_true*, meaning that  $\pi$  *temporarily satisfies*  $\varphi$ , i.e., it satisfies  $\varphi$ , but there is at least one possible continuation of  $\pi$  that violates  $\varphi$ ;
- *temp\_false*, meaning that  $\pi$  *temporarily violates*  $\varphi$ , i.e.,  $\varphi$  is not satisfied by  $\pi$ , but there is at least one possible continuation of  $\pi$  that does so;
- *perm\_true*, meaning that  $\pi$  *permanently satisfies*  $\varphi$ , i.e.,  $\varphi$  is satisfied by  $\pi$  and it will always be, no matter how  $\pi$  is extended;
- *perm\_false*, meaning that  $\pi$  *permanently violates*  $\varphi$ , i.e.,  $\varphi$  is not satisfied by  $\pi$  and it will never be, no matter how  $\pi$  is extended.

Formally, let  $\varphi$  be an  $\text{LDL}_f/\text{LTL}_f$  formula, and let  $\pi$  be a trace. Then, we define that  $\varphi$  is in rv state  $s \in \{\text{temp\_true}, \text{temp\_false}, \text{true}, \text{false}\}$  (written  $\llbracket \varphi = \text{temp\_true} \rrbracket$ ) on trace  $\pi$  as follows:

- $\pi \models \llbracket \varphi = \text{temp\_true} \rrbracket$  if  $\pi \models \varphi$  and there exists a trace  $\pi'$  such that  $\pi\pi' \not\models \varphi$ , where  $\pi\pi'$  denotes the trace obtained by concatenating  $\pi$  with  $\pi'$ ;
- $\pi \models \llbracket \varphi = \text{temp\_false} \rrbracket$  if  $\pi \not\models \varphi$  and there exists a trace  $\pi'$  such that  $\pi\pi' \models \varphi$ ;
- $\pi \models \llbracket \varphi = \text{perm\_true} \rrbracket$  if  $\pi \models \varphi$  and for every trace  $\pi'$ , we have  $\pi\pi' \models \varphi$ ;
- $\pi \models \llbracket \varphi = \text{perm\_false} \rrbracket$  if  $\pi \not\models \varphi$  and for every trace  $\pi'$ , we have  $\pi\pi' \not\models \varphi$ .

By inspecting the definition of rv states, it is straightforward to see that a formula  $\varphi$  is in one and only one rv state on a trace  $\pi$ .

The rv states  $\text{temp\_true}$  and  $\text{temp\_false}$  are not definitive: They may change into any other rv state as the system progresses. This reflects the general unpredictability of how a system execution unfolds. Conversely, the rv states  $\text{perm\_true}$  and  $\text{perm\_false}$  are stable since, once emitted by the monitor, they will by definition not change anymore. Observe that a stable rv state can be reached in two different situations: (i) when the system execution terminates; (ii) when the formula that is being monitored can be fully evaluated by observing a partial trace only. The first case is indeed trivial, as when the execution ends, there are no possible future evolutions and hence it is enough to evaluate the finite (and now complete) trace seen so far according to the  $\text{LDL}_f$  semantics. In the second case, instead, it is irrelevant whether the system continues its execution or not, since some  $\text{LDL}_f$  properties, such as eventualities or safety properties, can be fully evaluated as soon as something happens, e.g., when the eventuality is verified or the safety requirement is violated. Notice also that, when a stable state is returned by the monitor, the monitoring analysis can be stopped.

From a more theoretical viewpoint, given an  $\text{LDL}_f$  property  $\varphi$ , the monitor looks at the trace seen so far, assesses if it is a *prefix* of a full trace not yet completed, and categorizes it according to its potential for satisfying or violating  $\varphi$  in the future. We call a prefix *possibly good* for an  $\text{LDL}_f$  formula  $\varphi$  if there exists an extension of it that satisfies  $\varphi$ . More precisely, given an  $\text{LDL}_f$  formula  $\varphi$ , we define the set of *possibly good prefixes* for  $\mathcal{L}(\varphi)$  as the set:

$$\mathcal{L}_{\text{poss\_good}}(\varphi) = \{\pi \mid \text{there exists } \pi' \text{ such that } \pi\pi' \in \mathcal{L}(\varphi)\}. \quad (1)$$

Prefixes for which every possible extension satisfies  $\varphi$  are instead called *necessarily good*. More precisely, given an  $\text{LDL}_f$  formula  $\varphi$ , we define the set of *necessarily good prefixes* for  $\mathcal{L}(\varphi)$  as the set:

$$\mathcal{L}_{\text{nec\_good}}(\varphi) = \{\pi \mid \text{for every } \pi' \text{ it holds that } \pi\pi' \in \mathcal{L}(\varphi)\}. \quad (2)$$

The set of *necessarily bad prefixes*  $\mathcal{L}_{\text{nec\_bad}}(\varphi)$  can be defined analogously as:

$$\mathcal{L}_{\text{nec\_bad}}(\varphi) = \{\pi \mid \text{for every } \pi' \text{ it holds that } \pi\pi' \notin \mathcal{L}(\varphi)\}. \quad (3)$$

Observe that the necessarily bad prefixes for  $\varphi$  are the necessarily good prefixes for  $\neg\varphi$ , i.e.,  $\mathcal{L}_{\text{nec\_bad}}(\varphi) = \mathcal{L}_{\text{nec\_good}}(\neg\varphi)$ .

Such language-theoretic notions allow us to capture all the rv states defined before. More precisely, it is immediate to show the following:

**PROPOSITION 4.2.** *Let  $\varphi$  be an  $\text{LDL}_f$  formula and  $\pi$  a trace. Then:*

- $\pi \models \llbracket \varphi = \text{temp\_true} \rrbracket$  iff  $\pi \in \mathcal{L}(\varphi) \setminus \mathcal{L}_{\text{nec\_good}}(\varphi)$ ;
- $\pi \models \llbracket \varphi = \text{temp\_false} \rrbracket$  iff  $\pi \in \mathcal{L}(\neg\varphi) \setminus \mathcal{L}_{\text{nec\_bad}}(\varphi)$ ;
- $\pi \models \llbracket \varphi = \text{perm\_true} \rrbracket$  iff  $\pi \in \mathcal{L}_{\text{nec\_good}}(\varphi)$ ;
- $\pi \models \llbracket \varphi = \text{perm\_false} \rrbracket$  iff  $\pi \in \mathcal{L}_{\text{nec\_bad}}(\varphi)$ .

*Example 4.3.* Consider formula  $\Phi := \circ(a \rightarrow (\bullet b))$  from Example 4.1, together with the three traces witnessing the progression of the execution described there: (i)  $\pi_0 = \epsilon$ , (ii)  $\pi_1 = \{a, c\}$ , and (iii)  $\pi_2 = \{a, c\}, \{b\}$ . By applying Proposition 4.2, we can now back up formally the intuitive discussion on monitoring  $\Phi$  done in Example 4.1. In fact, all traces  $\pi_0, \pi_1$ , and  $\pi_2$  belong to  $\mathcal{L}_{\text{poss\_good}}(\Phi)$ , since either they satisfy  $\Phi$ , or can be completed into traces that do so. However, only  $\pi_2$  is in  $\mathcal{L}_{\text{nec\_good}}(\Phi)$ . Consequently, we have: (i)  $\pi_0 \models \llbracket \Phi = \text{temp\_false} \rrbracket$ ; (ii)  $\pi_1 \models \llbracket \Phi = \text{temp\_false} \rrbracket$ ; (iii)  $\pi_2 \models \llbracket \Phi = \text{perm\_true} \rrbracket$ .

We now establish interesting relationships over the different types of prefixes introduced before. We start by observing that the set of all finite words over the alphabet  $2^P$  is the union of the language of  $\varphi$  and its complement  $\mathcal{L}(\varphi) \cup \mathcal{L}(\neg\varphi) = (2^P)^*$ . Also, any language and its complement are disjoint  $\mathcal{L}(\varphi) \cap \mathcal{L}(\neg\varphi) = \emptyset$ . Since from the definition of possibly good prefixes we have  $\mathcal{L}(\varphi) \subseteq \mathcal{L}_{\text{poss\_good}}(\varphi)$  and  $\mathcal{L}(\neg\varphi) \subseteq \mathcal{L}_{\text{poss\_good}}(\neg\varphi)$ , we also have that  $\mathcal{L}_{\text{poss\_good}}(\varphi) \cup \mathcal{L}_{\text{poss\_good}}(\neg\varphi) = (2^P)^*$ . Also,  $\mathcal{L}_{\text{poss\_good}}(\varphi) \cap \mathcal{L}_{\text{poss\_good}}(\neg\varphi)$  corresponds to:

$$\{\pi \mid \text{there exists } \pi' \text{ such that } \pi\pi' \in \mathcal{L}(\varphi) \text{ and there exists } \pi'' \text{ such that } \pi\pi'' \in \mathcal{L}(\neg\varphi)\},$$

meaning that the set of possibly good prefixes for  $\varphi$  and the set of possibly good prefixes for  $\neg\varphi$  do intersect, and in such intersection there are paths that can be extended to satisfy  $\varphi$ , but can also be extended to satisfy  $\neg\varphi$ . In addition:  $\mathcal{L}(\varphi) = \mathcal{L}_{\text{poss\_good}}(\varphi) \setminus \mathcal{L}(\neg\varphi)$ .

Turning to necessarily good prefixes and necessarily bad prefixes, we have that  $\mathcal{L}_{\text{nec\_good}}(\varphi) = \mathcal{L}_{\text{poss\_good}}(\varphi) \setminus \mathcal{L}_{\text{poss\_good}}(\neg\varphi)$ , that  $\mathcal{L}_{\text{nec\_bad}}(\varphi) = \mathcal{L}_{\text{poss\_good}}(\neg\varphi) \setminus \mathcal{L}_{\text{poss\_good}}(\varphi)$ , and also that  $\mathcal{L}_{\text{nec\_good}}(\varphi) \subseteq \mathcal{L}(\varphi)$  and  $\mathcal{L}_{\text{nec\_good}}(\varphi) \not\subseteq \mathcal{L}(\neg\varphi)$ .

Notably, necessarily good, necessarily bad, and possibly good prefixes partition all finite traces. In fact, by directly applying the definitions of necessarily good, necessarily bad, possibly good prefixes of  $\mathcal{L}(\varphi)$  and  $\mathcal{L}(\neg\varphi)$ , we obtain the following:

PROPOSITION 4.4. *The set of all traces  $(2^P)^*$  can be partitioned into*

$$\mathcal{L}_{\text{nec\_good}}(\varphi) \quad \mathcal{L}_{\text{poss\_good}}(\varphi) \cap \mathcal{L}_{\text{poss\_good}}(\neg\varphi) \quad \mathcal{L}_{\text{nec\_bad}}(\varphi)$$

so

$$\begin{aligned} \mathcal{L}_{\text{nec\_good}}(\varphi) \cup (\mathcal{L}_{\text{poss\_good}}(\varphi) \cap \mathcal{L}_{\text{poss\_good}}(\neg\varphi)) \cup \mathcal{L}_{\text{nec\_bad}}(\varphi) &= (2^P)^* \\ \mathcal{L}_{\text{nec\_good}}(\varphi) \cap (\mathcal{L}_{\text{poss\_good}}(\varphi) \cap \mathcal{L}_{\text{poss\_good}}(\neg\varphi)) \cap \mathcal{L}_{\text{nec\_bad}}(\varphi) &= \emptyset. \end{aligned}$$

#### 4.1 Monitoring $\text{LDL}_f$ Formulae

As pointed out in the previous section, the core issue in monitoring is prefix recognition.  $\text{LTL}_f$  is not expressive enough to talk about prefixes of its own formulae. Roughly speaking, given an  $\text{LTL}_f$  formula, the language of its possibly good prefixes, in general, cannot be described as an  $\text{LTL}_f$  formula. For such reasons, building a monitor usually requires direct manipulation of the automaton for the formula.

$\text{LDL}_f$ , instead, can capture any nondeterministic automaton as a formula, and it has the capability of expressing properties on prefixes. We can exploit such an extra expressivity to capture the monitoring condition in a direct and elegant way. We start by showing how to construct formulae representing (the language of) prefixes of other formulae, and then we show how to use them in the context of monitoring.

Technically, given an  $\text{LDL}_f$  formula  $\varphi$ , it is possible to express the language  $\mathcal{L}_{\text{poss\_good}}(\varphi)$  with an  $\text{LDL}_f$  formula  $\varphi'$ . Such a formula is obtained in two steps.

LEMMA 4.5. *Given an  $\text{LDL}_f$  formula  $\varphi$ , there exists a regular expression  $\text{pref}_\varphi$  such that  $\mathcal{L}(\text{pref}_\varphi) = \mathcal{L}_{\text{poss\_good}}(\varphi)$ .*

PROOF. The proof is constructive. We build the NFA  $A(\varphi)$  for  $\varphi$ . We then build a new NFA  $A_{\text{poss\_good}}(\varphi)$  by taking  $A(\varphi)$  and setting as final states all states from which we can reach a final state of  $A(\varphi)$ . The so-obtained NFA  $A_{\text{poss\_good}}(\varphi)$  is such that  $\mathcal{L}(A_{\text{poss\_good}}(\varphi)) = \mathcal{L}_{\text{poss\_good}}(\varphi)$ . Since NFAs are exactly as expressive as regular expressions, we can set  $\text{pref}_\varphi$  to be the regular expression that encodes  $A_{\text{poss\_good}}(\varphi)$ .  $\square$

Since  $\text{LDL}_f$  is as expressive as regular expressions (cf. Reference [19]), we can translate  $\text{pref}_\varphi$  into an equivalent  $\text{LDL}_f$  formula, obtaining the following key result:

THEOREM 4.6. *Given an  $\text{LDL}_f$  formula  $\varphi$ ,*

$$\begin{aligned} \pi \in \mathcal{L}_{\text{poss\_good}}(\varphi) &\text{ iff } \pi \models \langle \text{pref}_\varphi \rangle \text{end} \\ \pi \in \mathcal{L}_{\text{nec\_good}}(\varphi) &\text{ iff } \pi \models \langle \text{pref}_\varphi \rangle \text{end} \wedge \neg \langle \text{pref}_{\neg\varphi} \rangle \text{end}. \end{aligned}$$

PROOF. Any regular expression  $\rho$ , and hence any regular language, can be captured in  $\text{LDL}_f$  as  $\langle \rho \rangle \text{end}$ . Specifically, the language  $\mathcal{L}_{\text{poss\_good}}(\varphi)$  is captured by  $\langle \text{pref}_\varphi \rangle \text{end}$ , and the language  $\mathcal{L}_{\text{nec\_good}}(\varphi)$ , which is equivalent to  $\mathcal{L}_{\text{poss\_good}}(\varphi) \setminus \mathcal{L}_{\text{poss\_good}}(\neg\varphi)$ , is captured by  $\langle \text{pref}_\varphi \rangle \text{end} \wedge \neg \langle \text{pref}_{\neg\varphi} \rangle \text{end}$ .  $\square$

In other words, given an  $\text{LDL}_f$  formula  $\varphi$ , formula  $\varphi' = \langle \text{pref}_\varphi \rangle \text{end}$  is an  $\text{LDL}_f$  formula satisfying  $\mathcal{L}(\varphi') = \mathcal{L}_{\text{poss\_good}}(\varphi)$ . Similarly for  $\mathcal{L}_{\text{nec\_good}}(\varphi)$ .

Exploiting this result, and the results in Proposition 4.2, we reduce the evaluation of rv states to the standard evaluation of  $\text{LDL}_f$  formulae over a (partial) trace. Formally:

THEOREM 4.7. *Let  $\pi$  be a trace. The following equivalences hold:*

- $\pi \models \llbracket \varphi = \text{temp\_true} \rrbracket$  iff  $\pi \models \varphi \wedge \langle \text{pref}_{\neg\varphi} \rangle \text{end}$ ;
- $\pi \models \llbracket \varphi = \text{temp\_false} \rrbracket$  iff  $\pi \models \neg\varphi \wedge \langle \text{pref}_\varphi \rangle \text{end}$ ;
- $\pi \models \llbracket \varphi = \text{perm\_true} \rrbracket$  iff  $\pi \models \langle \text{pref}_\varphi \rangle \text{end} \wedge \neg \langle \text{pref}_{\neg\varphi} \rangle \text{end}$ ;
- $\pi \models \llbracket \varphi = \text{perm\_false} \rrbracket$  iff  $\pi \models \langle \text{pref}_{\neg\varphi} \rangle \text{end} \wedge \neg \langle \text{pref}_\varphi \rangle \text{end}$ .

PROOF. The theorem follows directly from Proposition 4.2 and Theorem 4.6.  $\square$

This result provides an actual procedure to return the rv state of an  $\text{LDL}_f$  formula  $\varphi$ : We build four automata, one for each of the four formulae above, and then follow the evolution of the trace  $\pi$  simultaneously on each one of them. Since Proposition 4.4 proves that the languages of the four automata are a partition for the set of all languages over  $(2^P)^*$ , we are guaranteed that, at each step, one and only one automaton is in a final state, and hence one and only one truth value is unambiguously returned as output of the monitoring procedure.

Example 4.8. Consider the DFA of formula  $\Phi := \circ(a \rightarrow (\bullet b))$ , shown in Figure 4.

Figures 5(a)–5(d) represent the four automata for monitoring the different rv truth values of  $\Phi$ . More specifically:

- The DFA in Figure 5(a) checks whether  $\pi \models \llbracket \varphi = \text{temp\_true} \rrbracket$ . Indeed, its final state is  $s_2$ , which corresponds to the subset of the final states in the original automaton from which some non-final state (in this case  $s_4$ ) can be reached.
- The DFA in Figure 5(b) checks whether  $\pi \models \llbracket \varphi = \text{temp\_false} \rrbracket$ . Indeed, its final states are  $s_0$  and  $s_1$ , which correspond to the subset of the non-final states in the original automaton from which some final state (in this case  $s_3$ ) can be reached.
- The DFA in Figure 5(c) checks whether  $\pi \models \llbracket \varphi = \text{perm\_true} \rrbracket$ . Indeed, its final state is  $s_3$ , which corresponds to the subset of the final states in the original automaton from which none of the non-final states can be reached.



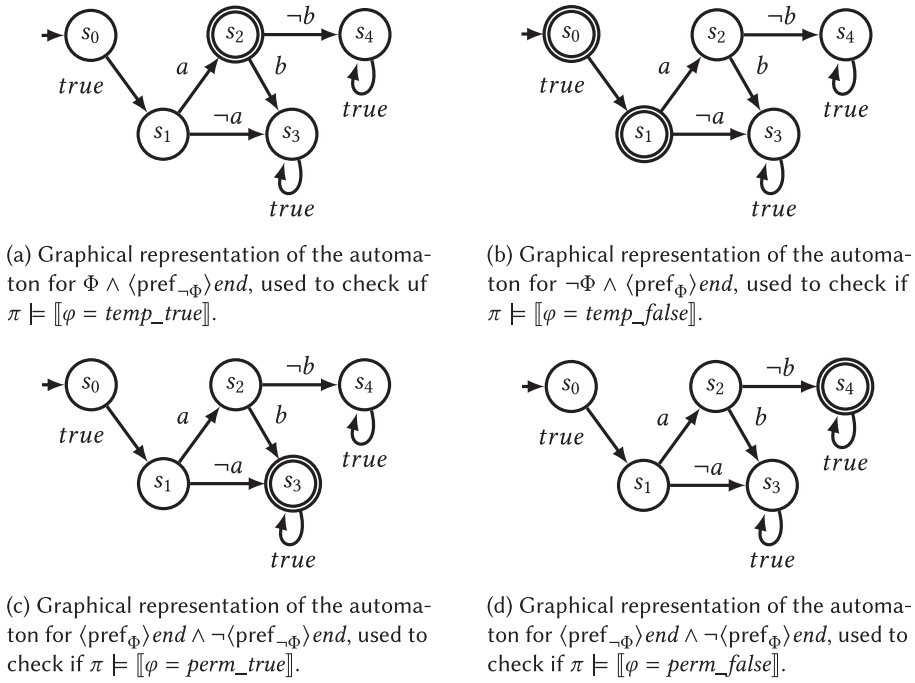


Fig. 5. Automata used to monitor the four rv truth values of formula  $O(a \rightarrow (\bullet b))$ .

- The DFA in Figure 5(d) checks whether  $\pi \models \llbracket \varphi = \text{perm\_false} \rrbracket$ . Indeed, its final state is  $s_4$ , which corresponds to the subset of the final states in the original automaton from which no final state can be reached.

By monitoring the execution discussed in Examples 4.1 and 4.3 using these four automata, we get the following:

- (1) At the beginning (trace  $\pi_0 = \epsilon$ ), all automata are in their respective  $s_0$  state; this is final for the DFA in Figure 5(b), hence the monitor returns *temp\_false*.
- (2) Upon receiving  $\{a, c\}$  (overall trace  $\pi_1 = \{a, c\}$ ), all automata move from  $s_0$  to  $s_1$ , since  $\{a, c\} \models \text{true}$ ; this new state continues to be final for the DFA in Figure 5(b), so *temp\_false* is kept as monitoring outcome.
- (3) Upon receiving  $\{b\}$  (overall trace  $\pi_2 = \{a, c\}, \{b\}$ ), given that  $\{b\} \models \neg a$  all automata move from  $s_1$  to  $s_3$ , which is final for the DFA in Figure 5(c), leading to *perm\_true* as the monitor output. This produced result is irrevocable, as  $s_3$  has a single transition only, pointing to  $s_3$  itself.

## 4.2 Monitoring Using Colored Automata

Figure 5 clearly shows that the four monitoring automata for  $O(a \rightarrow (\bullet b))$  all have the “the same shape”, in the sense that they all have the same structure of the automaton for  $O(a \rightarrow (\bullet b))$  depicted in Figure 4, only differing in which states are declared as final.

In this section, we prove that this property holds for every  $\text{LDL}_f$  formula. Consequently, we show that using four automata is indeed redundant, and that monitoring can be performed by making use of just a single automaton with “colored” states retaining, at once, all the necessary monitoring information for the different rv truth values. Specifically, we build one automaton

only and then mark its states with four different colors, each corresponding to the final states of a specific formula in Theorem 4.7, hence each representing one among the four RV truth values. The intention of using a single automaton for runtime verification is not novel [33], but here, for the first time, we provide a formal justification of its correctness.

As a first step, we formally define the notion of *shape equivalence* to capture the intuition that two automata have the same “shape”, i.e., they have corresponding states and transitions, but possibly differ in their final states.

Formally, let  $A_1 = (2^{\mathcal{P}}, \mathcal{S}^1, s_0^1, \varrho^1, S_f^1)$  and  $A_2 = (2^{\mathcal{P}}, \mathcal{S}^2, s_0^2, \varrho^2, S_f^2)$  be two DFA defined over a set  $\mathcal{P}$  of propositional symbols. We say that  $A_1$  and  $A_2$  are *shape equivalent*, written  $A_1 \sim A_2$ , if there exists a bijection  $h : \mathcal{S}^1 \rightarrow \mathcal{S}^2$  such that:

- (1)  $h(s_0^1) = s_0^2$ ;
- (2) for each  $(s_1^1, \Pi, s_2^1) \in \varrho^1$ , we have that  $(h(s_1^1), \Pi, h(s_2^1)) \in \varrho^2$ ; and
- (3) for each  $(s_1^2, \Pi, s_2^2) \in \varrho^2$ , we have that  $(h^{-1}(s_1^2), \Pi, h^{-1}(s_2^2)) \in \varrho^1$ .

We write  $A_1 \stackrel{h}{\sim} A_2$  to explicitly indicate the bijection  $h$  from  $A_1$  to  $A_2$  that induces their shape equivalence.

It is easy to see that bijection  $h$  preserves initial states (condition (1)) and transitions (conditions (2) and (3)), but does not require a correspondence between final states.

LEMMA 4.9. *Shape equivalence  $\sim$  is an equivalence relation.*

PROOF. Reflexivity: The identity function trivially satisfies (1)–(3) above. Symmetry: Let  $A \stackrel{h}{\sim} A$ . Given that  $h$  is a bijection, then  $A \stackrel{h^{-1}}{\sim} A$ . Transitivity: Let  $A_1 \stackrel{h}{\sim} A_2$  and  $A_2 \stackrel{g}{\sim} A_3$ . Then  $A_1 \stackrel{h \circ g}{\sim} A_3$ , where  $h \circ g$  is the composition of  $h$  and  $g$ .  $\square$

Hence,  $\sim$  induces (equivalence) classes of automata with the same shape. Automata for the basic formulae in Theorem 4.7 belong to the same class.

LEMMA 4.10. *For each  $\text{LDL}_f$  formula  $\varphi$ ,  $A(\varphi)$ ,  $A(\neg\varphi)$ ,  $A(\langle \text{pref}_\varphi \rangle \text{end})$ , and  $A(\langle \text{pref}_{\neg\varphi} \rangle \text{end})$  are in the same equivalence class by  $\sim$ .*

PROOF. From automata theory,  $A(\neg\varphi)$  can be obtained from  $A(\varphi)$  by switching the final states with the non-final ones. Hence, the identity  $i : \mathcal{S}^\varphi \rightarrow \mathcal{S}^\varphi$  is such that  $A(\varphi) \stackrel{i}{\sim} A(\neg\varphi)$ . Moreover,  $A(\varphi) \sim A(\langle \text{pref}_\varphi \rangle \text{end}) \sim A(\langle \text{pref}_{\neg\varphi} \rangle \text{end})$  as  $A(\langle \text{pref}_\varphi \rangle \text{end})$ , respectively,  $A(\langle \text{pref}_{\neg\varphi} \rangle \text{end})$ , can be obtained from  $A(\varphi)$ , respectively,  $A(\neg\varphi)$ , by setting as final states all states from which there exists a non-zero length path to a final state of  $A(\varphi)$ , respectively,  $A(\neg\varphi)$ , as explained in the proof of Lemma 4.5. Hence, again, the identity relation  $i$  is such that  $A(\varphi) \stackrel{i}{\sim} A(\langle \text{pref}_\varphi \rangle \text{end}) \stackrel{i}{\sim} A(\langle \text{pref}_{\neg\varphi} \rangle \text{end})$ .  $\square$

As the last step for proving that automata for the four formulae in Theorem 4.7 are in the same class, we show that conjunction of formulae preserves shape equivalence, in the following precise sense:

THEOREM 4.11. *Let  $\varphi_1, \varphi_2, \psi_1$  and  $\psi_2$  be  $\text{LDL}_f$  formulae so  $A(\varphi_1) \sim A(\psi_1)$  and  $A(\varphi_2) \sim A(\psi_2)$ . Then  $A(\varphi_1 \wedge \varphi_2) \sim A(\psi_1 \wedge \psi_2)$ .*

PROOF. From the semantics of  $\text{LDL}_f$  and Theorem 3.1, it follows that  $A(\varphi_1 \wedge \varphi_2) \equiv A(\varphi_1) \cap A(\varphi_2)$ . Recall that states of  $A(\varphi_1) \cap A(\varphi_2)$  are ordered pairs  $(s^{\varphi_1}, s^{\varphi_2}) \in \mathcal{S}^{\varphi_1} \times \mathcal{S}^{\varphi_2}$ . Let  $h_1$  and  $h_2$  be bijections such that  $A(\varphi_1) \stackrel{h_1}{\sim} A(\psi_1)$  and  $A(\varphi_2) \stackrel{h_2}{\sim} A(\psi_2)$ . We use  $h_1$  and  $h_2$  to construct a new bijection  $h : \mathcal{S}^{\varphi_1} \times \mathcal{S}^{\varphi_2} \rightarrow \mathcal{S}^{\psi_1} \times \mathcal{S}^{\psi_2}$  such that  $h(s^{\varphi_1}, s^{\varphi_2}) = (h_1(s^{\varphi_1}), h_2(s^{\varphi_2}))$ . We show

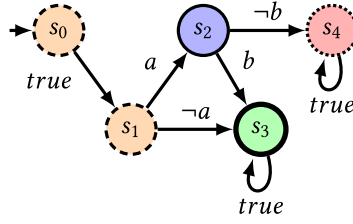


Fig. 6. Graphical representation of the colored automaton for  $\bigcirc(a \rightarrow (\bullet b))$ , with the following color coding: Automaton states corresponding to the rv state *temp\_false* are represented in orange, dashed line; those corresponding to *perm\_true* in green, thick solid line; those corresponding to *temp\_true* in blue, thin solid line; those corresponding to *perm\_false* in red, dotted line.

that  $h$  satisfies criteria (1)–(3) of shape equivalence, hence inducing  $A(\varphi_1 \wedge \varphi_2) \stackrel{h}{\sim} A(\psi_1 \wedge \psi_2)$ . The starting state  $s_0^{\varphi_1 \wedge \varphi_2}$  of  $A(\varphi_1 \wedge \varphi_2)$  corresponds to  $(s_0^{\varphi_1}, s_0^{\varphi_2})$  by definition of  $A(\varphi_1) \cap A(\varphi_2)$ . At the same time,  $s_0^{\psi_1 \wedge \psi_2} = (h_1(s_0^{\varphi_1}), h_2(s_0^{\varphi_2})) = (s_0^{\varphi_1}, s_0^{\varphi_2})$  by definition of  $h$ , which proves (1). Now, consider a transition  $((s_1^{\varphi_1}, s_1^{\varphi_2}), \Pi, (s_2^{\varphi_1}, s_2^{\varphi_2}))$  in  $\varrho^{\varphi_1 \wedge \varphi_2}$ . By construction, this means that there exist transitions  $(s_1^{\varphi_1}, \Pi, s_2^{\varphi_1}) \in \varrho^{\varphi_1}$  and  $(s_1^{\varphi_2}, \Pi, s_2^{\varphi_2}) \in \varrho^{\varphi_2}$ . Since  $A(\varphi_1) \stackrel{h_1}{\sim} A(\psi_1)$  and  $A(\varphi_2) \stackrel{h_2}{\sim} A(\psi_2)$ , we have that  $(h_1(s_1^{\varphi_1}), \Pi, h_1(s_2^{\varphi_1})) \in \varrho^{\psi_1}$  and  $(h_2(s_1^{\varphi_2}), \Pi, h_2(s_2^{\varphi_2})) \in \varrho^{\psi_2}$ . It follows that  $((h_1(s_1^{\varphi_1}), h_2(s_1^{\varphi_2})), \Pi, (h_1(s_2^{\varphi_1}), h_2(s_2^{\varphi_2}))) \in \varrho^{\psi_1 \wedge \psi_2}$ , which proves (2). Condition (3) is proved analogously with  $h^{-1}$ .  $\square$

**COROLLARY 4.12.** *Given an LDL<sub>f</sub> formula  $\varphi$ , automata  $A(\varphi \wedge \langle \text{pref}_{-\varphi} \rangle \text{end})$ ,  $A(\neg\varphi \wedge \langle \text{pref}_{\varphi} \rangle \text{end})$ ,  $A(\langle \text{pref}_{\varphi} \rangle \text{end} \wedge \neg\langle \text{pref}_{-\varphi} \rangle \text{end})$ , and  $A(\langle \text{pref}_{-\varphi} \rangle \text{end} \wedge \neg\langle \text{pref}_{\varphi} \rangle \text{end})$  are in the same equivalence class by  $\sim$ .*

This result tells that the automata of the formulae used to capture the RV states of an LDL<sub>f</sub> formula of interest, as captured by Theorem 4.7, are identical modulo final states. In addition, by definition of the four LDL<sub>f</sub> formulae, we directly get that each state is marked as final by one and only one of such automata. This, in turn, allows us to merge all the four automata together into a single automaton, provided that we recall, for each state in the automaton, which of the four formulae marks it as final (which corresponds to declare to which of the four RV states it corresponds). In practice, we can simply build the automaton  $A(\varphi)$  for  $\varphi$ , and “color” each state in the automaton according to its corresponding RV state. This can be realized with the following, direct procedure: We first build  $A(\varphi) = (2^{\mathcal{P}}, \mathcal{S}, s_0, \varrho, S_f)$ , with  $S_f$  the set of its final states, and, for each  $s \in \mathcal{S}$ , we compute the set  $\text{Reach}(s)$  of states reachable from  $s$ . Then:

- if (i)  $s \in S_f$ , (ii)  $\text{Reach}(s) \not\subseteq S_f$ , and (iii)  $\text{Reach}(s) \cap \{s\} \neq \emptyset$ , then we mark  $s$  as *temp\_true*;
- if (i)  $s \notin S_f$ , (ii)  $\text{Reach}(s) \not\subseteq (\mathcal{S} \setminus S_f)$ , and (iii)  $\text{Reach}(s) \cap S_f \neq \emptyset$ , then we mark  $s$  as *temp\_false*;
- if (i)  $s \in S_f$  and (ii)  $\text{Reach}(s) \subseteq S_f$ , then we mark  $s$  as *perm\_true*;
- if (i)  $s \notin S_f$  and (ii)  $\text{Reach}(s) \subseteq (\mathcal{S} \setminus S_f)$ , then we mark  $s$  as *perm\_false*.

It is easy to see that the four bullets above match the four ones of Theorem 4.7. The soundness of the marking immediately follows from the definitions and results in the previous section.

This solution is very flexible, as the reachability analysis can be performed on-the-fly: Indeed, this is the procedure we actually implemented in our runtime verification tool, as explained in Section 7.

**Example 4.13.** Figure 6 depicts the colored automaton for the formula in Example 4.1. The colored automaton is the merging of the four automata in Figures 5(a)–5(d), suitably coloring each

state, depending on which one of the four automata marks that state as final. In particular, states  $s_0$  and  $s_1$ , which are final in the automaton for  $\pi \models \llbracket \varphi = \text{temp\_false} \rrbracket$  (Figure 5(b)), are now marked as *temp\_false* (orange, dashed line); state  $s_2$ , which is final in the automaton for  $\pi \models \llbracket \varphi = \text{temp\_true} \rrbracket$  (Figure 5(a)), is marked as *temp\_true* (blue, solid thin line); state  $s_3$ , which is final in the automaton for  $\pi \models \llbracket \varphi = \text{perm\_true} \rrbracket$  (Figure 5(c)), is marked with *perm\_true* (green, solid thick line); and, last, state  $s_4$ , which was final in the automaton for  $\pi \models \llbracket \varphi = \text{perm\_false} \rrbracket$  (Figure 5(d)), is marked with *perm\_false* (red, dotted line).

## 5 MONITORING DECLARE CONSTRAINTS

We now ground our monitoring approach to the case of `DECLARE` (cf. Section 2.2).


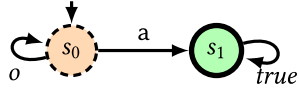

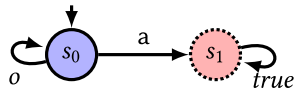

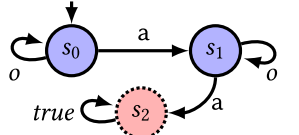

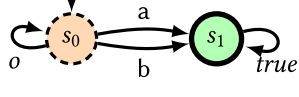

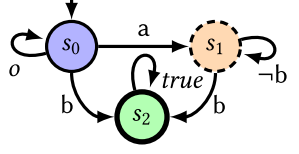

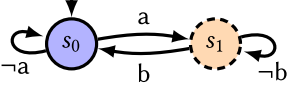

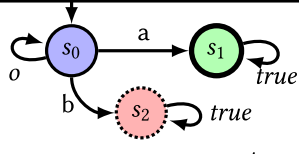

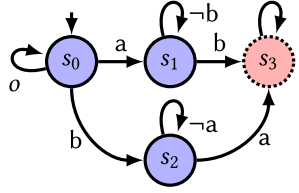
Several logic-based techniques have been proposed to support end-users in defining, checking, and enacting `DECLARE` models [37, 39, 42, 43]. More recently, the  $LTL_f$  characterization of `DECLARE`, together with its operational automata-theoretic counterpart, have been exploited to provide advanced monitoring and runtime verification facilities [33, 34]. In particular, monitoring `DECLARE` models amounts to:

- Tracking the evolution of a single `DECLARE` constraint against an evolving trace, providing a fine-grained feedback on how the truth value of the constraint evolves when tasks are performed. This is done by adopting the RV semantics for  $LTL_f$ . Specifically, in Reference [33], the evolution of `DECLARE` constraints through the different RV states is tackled using the ad hoc “colored automaton” construction technique that we have formally justified in Section 4.2.
- Tracking the compliance of an evolving trace to the entire `DECLARE` model, by considering all its constraints together. This is done by constructing the colored automaton for the conjunction of all constraints in the model. Monitoring the evolving trace against such a “global” automaton is crucial for inferring complex violations that cannot be ascribed to the interaction of the current trace with a single constraint in the model, but arise due to the interplay between the current trace and multiple constraints at once. Such violations emerge when the current trace induces a *conflict* over two or more constraints. Technically, this means that, in the current circumstances, such constraints contradict each other and consequently cannot be all satisfied anymore [34]. By considering all constraints together, the presence of this kind of conflict can be detected immediately, without waiting for the later moment when an explicit violation of one of the single constraints involved in the conflict eventually arises. This important feature has been classified as *early detection of violations* in a reference monitoring survey [32].

**Monitoring Declare Constraints with  $LDL_f$ .** Since  $LDL_f$  includes  $LTL_f$ , `DECLARE` patterns can be directly encoded in  $LDL_f$  using their standard formalization [39, 44]. Good prefixes of such formulae can be also defined, providing the basis for the definition of the monitoring formulae as in Theorem 4.7. Table 2 reports the good prefix characterization of some of the `DECLARE` patterns. At the same time, we can operationally obtain a monitor for each `DECLARE` pattern by simply: (i) encoding its formula into a corresponding NFA following the approach of Section 3; (ii) (if needed), determinizing it using standard techniques to obtain a DFA; (iii) coloring the DFA via reachability queries as described in Section 4.2.

This approach only works for single constraints, but not for a whole `DECLARE` model, which combines multiple constraints at once. To account for single constraints and their interplay, given a `DECLARE` model  $\mathcal{M}$ , we proceed as follows:

Table 2. DECLARE Constraint Templates from Table 1, Showing, Together with Their  $LTL_f$  Semantics, Their Good Prefix Characterization and Colored DFA

CONSTRAINT	$LTL_f$	pref	COLORED DFA
$1..*$  existence	$\diamond a$	$true^*$	
$0$  absence	$\neg \diamond a$	$o^*$	
$0..1$  absence 2	$\neg \diamond (a \wedge \diamond a)$	$o^* + (o^*; a; o^*)$	
 choice	$\diamond (a \vee b)$	$true^*$	
 responded existence	$\diamond a \rightarrow \diamond b$	$true^*$	
 response	$\square (a \rightarrow \diamond b)$	$true^*$	
 precedence	$\neg b \mathcal{U} a \vee \neg \diamond b$	$(\neg b)^* + (o^*; a; true^*)$	
 not coexistence	$\neg (\diamond a \wedge \diamond b)$	$(a + o)^* + (b + o)^*$	

- (1) For every constraint  $c \in \mathcal{M}$ , we derive its  $LTL_f$  formula  $\varphi_c$  and construct its corresponding deterministic colored automaton  $A(c)$ . This colored automaton acts as *local monitor* for its constraint  $c$ . This can be used to track the rv state of  $c$  as tasks are executed.
- (2) We build the  $LTL_f$  formula  $\Phi_{\mathcal{M}}$  standing for the conjunction of the  $LTL_f$  formulae encoding all constraints in  $\mathcal{M}$  and construct its corresponding deterministic colored automaton  $A(\mathcal{M})$ .

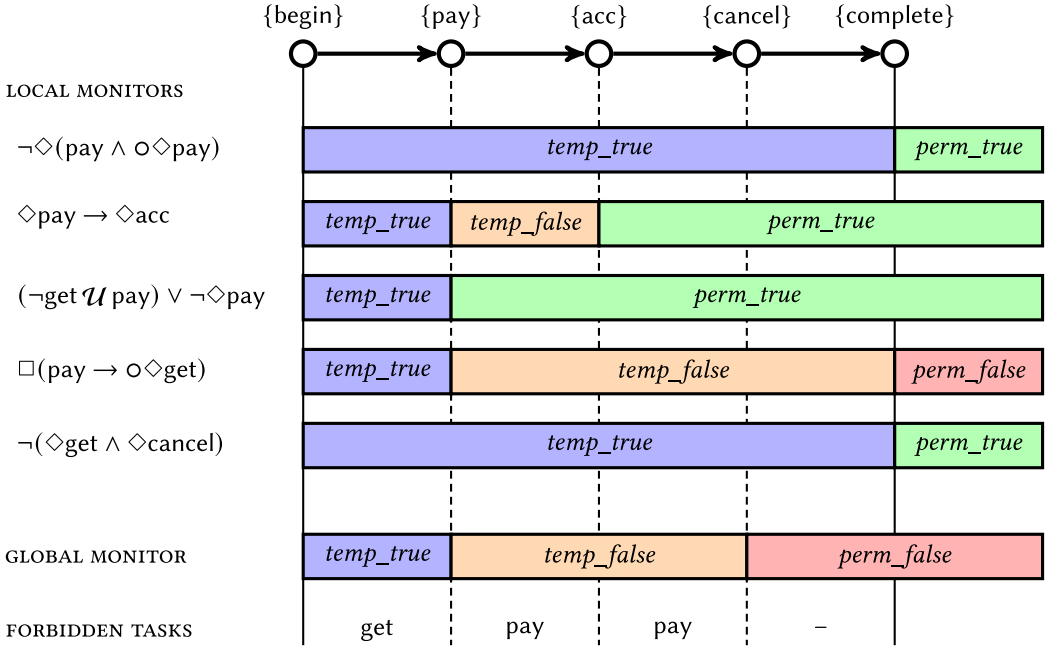


Fig. 7. Result computed by monitoring the DECLARE model of Figure 1 against the non-compliant trace  $\{\text{begin}\}, \{\text{pay}\}, \{\text{acc}\}, \{\text{cancel}\}, \{\text{complete}\}$  using local monitors for each constraint and the global monitor accounting for all of them at once. Graphically, the trace evolves from left to right, and in a given step, the evolution of the RV states for the local and global monitors is the one shown from  $\{\text{begin}\}$  up to that point.

This colored automaton acts as *global monitor* for the entire DECLARE model  $\mathcal{M}$ . This can be used to track the overall RV state of  $\mathcal{M}$  as tasks are executed and early detect violations arising from conflicting constraints.

- (3) When the monitoring of a process execution starts, which can be signaled to the monitor through a special, initial event (*begin* in Figure 7), the RV truth value attached to the initial state of each local monitor, as well as that of the global monitor, are returned.
- (4) Whenever an event witnessing the execution of a task is tracked, it is delivered to each local monitor and to the global monitor. The new, current state of each monitor is then computed, based on the current state and on the received task name, in turn returning the truth value associated to the new state.
- (5) When the process execution is completed (i.e., no further events are expected to occur), which can be signaled to the monitor through a special, final event (*end* in Figure 7), each monitor produces a final verdict, according to the following procedure: If, upon completion, the current state of the monitor is colored by *perm\_true* or *temp\_true*, then the trace is judged as compliant (i.e., *perm\_true*, considering that no change will occur anymore); if, instead, the color is *perm\_false* or *temp\_false*, then the trace is judged as non-compliant (i.e., *perm\_false*).
- (6) The global monitor can be inquired to obtain additional information about how the monitored trace interacts with all the model constraints considered together. For example, when the current state of the global monitor is *temp\_false* or *temp\_true*, retrieving the names of tasks whose execution leads to a *perm\_false* state is useful to return which tasks are currently forbidden by the model. This information is irrelevant when the monitor is in a *perm\_true* or *perm\_false* state, as no task can lead to change the current verdict.



*Example 5.1.* Figure 7 depicts the result computed by monitoring the DECLARE model introduced in Example 2.4 and shown in Figure 1, against a trace where a registration is paid, the corresponding regulation is accepted, and then the registration is canceled.

When monitoring starts, all local monitors are in state *temp\_true*, and so is the global monitor. Task *get ticket* is forbidden, since according to the precedence constraint connecting *get ticket* to *pay registration* (i.e., formula  $(\neg \text{get } \mathcal{U} \text{ pay}) \vee \neg \diamond \text{pay}$ ), a previous execution of *pay registration* is needed. When the payment is executed:

- the local monitor for the responded existence constraint linking *pay registration* to *accept regulation* (i.e., formula  $\diamond \text{pay} \rightarrow \diamond \text{acc}$ ) moves to *temp\_false*, because it requires acceptance of the regulation (which has not been done yet);
- the local monitor for the precedence constraint linking *get ticket* to *pay registration* (i.e., formula  $(\neg \text{get } \mathcal{U} \text{ pay}) \vee \neg \diamond \text{pay}$ ) moves to *perm\_true*, enabling once and for all the possibility of executing *get ticket*;
- the local monitor for the response constraint linking *pay registration* to *get ticket* (i.e., formula  $\Box(\text{pay} \rightarrow \circ \diamond \text{get})$ ) moves to *temp\_false*, because its satisfaction now demands a consequent execution of the *get ticket* task.

The global monitor also moves to *temp\_false*, since there are two tasks that must be executed to satisfy the responded existence and response constraints, and it is indeed possible to execute them without violating other constraints. At the same time, further payments are now forbidden, due to the absence 2 constraint attached to the *pay registration* task (i.e., formula  $\neg \diamond(\text{pay} \wedge \circ \diamond \text{pay})$ ).

The consequent execution of *accept regulation* turns the state of the responded existence constraint linking *pay registration* to *accept regulation* (i.e., formula  $\diamond \text{pay} \rightarrow \diamond \text{acc}$ ) to *perm\_true*: Since the regulation has now been accepted, the constraint is satisfied and will stay so no matter how the execution is continued.

The most interesting transition is the one triggered by the consequent execution of the *cancel registration* task. While this event does not trigger any state change in the local monitors, it actually induces a transition of the global monitor to the permanent, *perm\_false* rv state. In fact, no continuation of the trace will be able to satisfy all constraints of the considered model. More specifically, the sequence of events received so far induces a so-called *conflict* [34] for the response constraint linking *pay registration* to *get ticket* (i.e., formula  $\Box(\text{pay} \rightarrow \circ \diamond \text{get})$ ), and the not coexistence constraint relating *get ticket* and *cancel registration* (i.e., formula  $\neg(\diamond \text{get} \wedge \diamond \text{cancel})$ ). In fact, the response constraint requires a future execution of the *get ticket* task, which is, however, forbidden by the not coexistence constraint. Consequently, no continuation of the current trace will ever be able to satisfy both constraints at once.

Since no further task execution actually happens, the trace is finally declared to be complete, with no execution of the *get ticket* task. This has the effect of, respectively, moving the response and not coexistence constraints to *perm\_false* and *perm\_true*. Also, the absence 2 constraint on payment becomes *perm\_true*, witnessing that no double payment occurred in the trace.

## 6 MODELING AND MONITORING METACONSTRAINTS

In Section 4, we have demonstrated that  $\text{LDL}_f$  has the ability of expressing formulae that capture the rv state of other formulae. This can be interpreted as the ability of  $\text{LDL}_f$  to express meta-level properties of  $\text{LDL}_f$  constraints within the logic itself. Such properties, which we call *metaconstraints*, can, in turn, be themselves monitored using the automata-theoretic approach described in Section 4.

In this section, we elaborate on this observation: We discuss how metaconstraints can be built, illustrate interesting metaconstraint patterns, and substantiate our proposal in the context of DECLARE, extending the approach introduced in Section 5. To do so, we extensively rely on the ability of  $LDL_f$  to capture bad/good prefixes of other formulae and to describe properties over those prefixes. In addition, we also directly exploit the power of  $LDL_f$  (in particular, that of regular expressions) within metaconstraints (see, e.g., formula (4) below). This goes beyond the expressiveness of  $LTL_f$  and is essential to capture real-life properties, as witnessed by industrial standard languages for temporal properties such as the *Property Specification Language* [29].

### 6.1 Modeling Metaconstraints

Theorem 4.7 shows that, for an arbitrary  $LDL_f$  formula  $\varphi$ , four  $LDL_f$  formulae can be automatically constructed to express whether  $\varphi$  is in one of the four RV states. Consequently, given  $s \in \{temp\_true, temp\_false, true, false\}$  and an  $LTL_f/LDL_f$  formula  $\varphi$ , we can consider formulae of the form  $\llbracket \varphi = s \rrbracket$  as special atoms of the logic itself.

Such special atoms are used to check whether a trace brings  $\varphi$  in state  $s$ . However, they cannot be used to explicitly characterize, which are the paths that lead  $\varphi$  to RV state  $s$ , i.e., that make formula  $\llbracket \varphi = s \rrbracket$  true. Such paths can be readily obtained by constructing the regular expression for language  $\mathcal{L}(\llbracket \varphi = s \rrbracket)$ , which we denote as  $re_{\llbracket \varphi = s \rrbracket}$ . For example,  $re_{\llbracket \varphi = perm\_false \rrbracket} = \mathcal{L}(\langle \text{pref}_{\neg\varphi} \rangle end \wedge \neg \langle \text{pref}_{\varphi} \rangle end)$  describes all paths culminating in a permanent violation of  $\varphi$ .

With these notions at hand, we can build  $LTL_f/LDL_f$  metaconstraints as standard  $LTL_f/LDL_f$  formulae that include:

- formulae of the form  $\llbracket \varphi = s \rrbracket$  as atoms;
- formulae of the form  $re_{\llbracket \varphi = s \rrbracket}$  as path expressions.

A metaconstraint is then translated back into a standard  $LDL_f$  formula by replacing each subformula of the form  $\llbracket \varphi = s \rrbracket$  with its corresponding  $LDL_f$  formula, and  $re_{\llbracket \varphi = s \rrbracket}$  with its corresponding path expression. To do so, given  $s \in \{temp\_true, temp\_false, true, false\}$  and  $\varphi$ , one can proceed as follows: The  $LDL_f$  formula for  $\llbracket \varphi = s \rrbracket$  is directly obtained from the correspondence established in Theorem 4.7. A direct (non-optimized) way to calculate the regular expression for  $re_{\llbracket \varphi = s \rrbracket}$  is through the following two steps: First, the  $LDL_f$  formula  $\llbracket \varphi = s \rrbracket$  (obtained as before) is transformed into its corresponding automaton. Then, the automaton can be folded back into a path expression using standard techniques for translating automata into regular expressions [28] (see Reference [26] for a fine-grained account of the computational complexity of this step).

*Example 6.1.* Consider the  $LTL_f$  formula  $\varphi = \circ(a \rightarrow (\bullet b))$  from Figure 6. By inspecting the DFA from that figure, we can easily obtain the path expressions for  $re_{\llbracket \varphi = s \rrbracket}$ , with  $s \in \{temp\_true, temp\_false, true, false\}$ :

- $re_{\llbracket \varphi = temp\_false \rrbracket} = \epsilon + true$ , since the  $temp\_false$  states of the automation are  $s_0$  and  $s_1$ , respectively, reachable with the empty trace and the trace containing just one step (regardless of the propositional interpretation of that step).
- $re_{\llbracket \varphi = temp\_true \rrbracket} = true; a$  as the only  $temp\_true$  state of the automaton is  $s_2$ , reachable with traces of length 2 whose second step contains  $a$ .
- $re_{\llbracket \varphi = false \rrbracket} = true; a; \neg b; true^*$  as the only  $false$  state of the automation is  $s_4$ , reachable from  $s_2$  by executing a further step whose propositional interpretation does not contain  $b$ , followed by an arbitrary prefix (cf. the looping edge of  $s_4$ ).
- Considering the two possible ways of reaching  $s_3$ , and the loop on  $s_3$  itself, we have  $re_{\llbracket \varphi = true \rrbracket} = true; ((a; b; true^*) + (\neg a; true^*))$ .

Now consider the responded existence DECLARE constraint and its corresponding DFA, both shown in Table 2. Given two tasks  $a$  and  $b$ , its corresponding LTL<sub>f</sub> formula is  $\psi = \diamond a \rightarrow \diamond b$ . By inspecting the DFA of the formula, we get the following:

- Considering that no state in the automaton is *false*, we have  $re_{\llbracket \psi = \text{false} \rrbracket} = \text{false}$  (i.e., no trace will satisfy this expression).
- Considering  $s_0$  and recalling that  $o$  is a shortcut for  $\neg a \wedge \neg b$ , we have  $re_{\llbracket \psi = \text{temp\_true} \rrbracket} = (\neg a \wedge \neg b)^*$ .
- Considering  $s_1$  and the ways to reach it, we have  $re_{\llbracket \psi = \text{temp\_false} \rrbracket} = (\neg a \wedge \neg b)^*; a; (\neg b)^*$ .
- Considering  $s_2$  and the ways to reach it, we have  $re_{\llbracket \psi = \text{true} \rrbracket} = ((\neg a \wedge \neg b)^*; b; \text{true}^*) + ((\neg a \wedge \neg b)^*; a; (\neg b)^*; b; \text{true}^*)$ .

## 6.2 Some Relevant Metaconstraint Patterns

We present three types of metaconstraints, demonstrating the sophistication and versatility of the resulting framework.

*Contextualizing constraints.* This type of metaconstraint is used to express that a constraint must hold *while* another constraint is in some rv state. The latter constraint, together with the specified state, consequently provides a monitoring *context* for the former, *contextualized* constraint. This form of scoping can be seen as a generalization of the mechanism used in Reference [21], where the scope of application of LTL patterns is defined using special atomic events.

Let us specifically consider the case of a *contextualized absence*, where, given a task  $a$ , the contextualized constraint has the form  $\Box \neg a$ , and the context is provided by an arbitrary regular expression  $\rho$ . This is formalized as:

$$[\rho](\neg a \vee \text{end}), \quad (4)$$

where *end* denotes the end of the trace, as defined in Section 2; this is needed, since, in LDL<sub>f</sub>,  $\neg a$  expresses that some task different than  $a$  is executed, while we also want to accept the case where no task is performed at all (and the trace completes).

The idea of formula (4) is to relativize the unrestricted  $\Box$  operator to all and only those paths that satisfy  $\rho$ . This can be further refined, for example by choosing as context the fact that another constraint  $\varphi$  is in a given rv state  $s$ . This is done by setting  $\rho$  to the regular expression  $re_{\llbracket \varphi = s \rrbracket}$ , describing those paths that lead to rv state  $s$  for  $\varphi$ . We then obtain:

$$[re_{\llbracket \varphi = s \rrbracket}](\neg a \vee \text{end}). \quad (5)$$

A monitor for formula (5) returns *temp\_true* either when  $\varphi$  is not in state  $s$ , but may evolve into such a state, or when  $\varphi$  is in state  $s$ . In the latter situation, by inspecting the monitor, one can see that task  $a$  is forbidden; this also means that upon the execution of  $a$ , the monitor evolves into *perm\_false*. Finally, the monitor returns *perm\_true* if  $\varphi$  is not in state  $s$  and will never be able to enter into state  $s$ , no matter how the trace evolves.

*Example 6.2.* Consider the constraint model in Figure 1. We want to express that it is not possible to get the ticket after the payment is done until the regulation is accepted (if it was accepted before, then no restriction applies). This can be seen as a *contextualized absence* constraint forbidding get ticket when the responded existence that links pay registration to accept regulation (i.e., formula  $\diamond \text{pay} \rightarrow \diamond \text{acc}$ ) is *temporarily violated*, which in turn describes the execution state in which pay registration has been done, but accept regulation not. Formally, this is encoded by instantiating formula (5) into:

$$[re_{\llbracket \{\diamond \text{pay} \rightarrow \diamond \text{acc}\} = \text{temp\_false} \rrbracket}](\neg \text{get} \vee \text{end}),$$

which, in turn, expands into:

$$[(\text{get} + o)^*; \text{pay}; (\neg \text{acc})^*](\neg \text{get} \vee \text{end}),$$

where  $o$  denotes any task different from  $\text{pay}$ ,  $\text{get}$ , and  $\text{acc}$  (so  $(\text{get} + o)^*$  is the regular expression capturing that tasks different from  $\text{pay}$  and  $\text{acc}$  are arbitrarily repeated).

*Compensation constraints.* In general terms, compensation refers to a behavior that has to be enforced when the current execution reaches an unexpected/undesired state. In our setting, the undesired state triggering a compensation is the *permanent violation* of a property that captures a desired behavior, which, in turn, triggers the fact that *another* formula, capturing the compensating behavior, has to be satisfied. We call the first formula the *default constraint* and the second formula its *compensating constraint*.

Previous works have tackled this issue through ad hoc techniques that work directly with monitors, not declaratively with the formulae to be monitored declarative counterpart [33, 34]. Dealing with this at the formula level has the advantage that the compensating behavior is semantically well-characterized. Interestingly, compensating based on the permanent violation of a constraint allows us to express, in our setting, a temporal variant of what in legal reasoning would be called *contrary-to-duty obligations* [48], that is, obligations that are enforced when other obligations are violated. In addition, incorporating compensation within monitoring frameworks is considered as a necessary feature, which surprisingly has not been extensively investigated in the past [32].

To explain how compensation works in our setting, let us consider the general case of a default  $\text{LDL}_f$  constraint  $\varphi$  and a compensating  $\text{LDL}_f$  constraint  $\psi$ . By noticing that once a trace permanently violates a constraint, then every possible continuation still permanently violates that constraint, we capture the *compensation* of  $\varphi$  by  $\psi$  as:

$$\llbracket \varphi = \text{perm\_false} \rrbracket \rightarrow \psi. \quad (6)$$

The intuitive interpretation of formula (6) is that either  $\varphi$  never enters into the *perm\_false* rv state or  $\psi$  holds. No requirement is placed regarding *when*  $\psi$  should be monitored in case  $\varphi$  gets permanently violated. In fact, the overall compensation formula (6) gets temporarily/permanently satisfied even when the compensating constraint  $\psi$  is temporarily/permanently satisfied *before* the moment when the default constraint  $\varphi$  gets permanently violated. This may sound counterintuitive, as it is usually intended that the compensating behavior has to be exhibited *as a reaction* to the violation. We can capture this intuition by turning formula (6) into the following *reactive compensation* formula:

$$\llbracket \varphi = \text{perm\_false} \rrbracket \rightarrow \langle \text{re}_{\llbracket \varphi = \text{perm\_false} \rrbracket} \rangle \psi. \quad (7)$$

This formula imposes that, in case of a permanent violation of  $\varphi$ , the compensating constraint  $\psi$  must hold *after*  $\varphi$  has become permanently violated.

Assuming that  $\varphi$  can be potentially violated (which is the reason why we want to express a compensation), a monitor for formula (7) starts by emitting *temp\_true*. As soon as the monitored execution is so  $\varphi$  cannot be permanently violated anymore, the monitor switches to *perm\_true*. If instead the monitored execution leads to permanently violate  $\varphi$ , then from the moment of the violation onwards, the evolution of the monitor follows that of  $\psi$ .

*Example 6.3.* Consider the not coexistence constraint in Figure 1. We want to model that, whenever this constraint is permanently violated, that is, whenever a ticket is retrieved and the registration is canceled, then a return ticket (return for short) task must be executed. This has to occur in reaction to the permanent violation. Hence, we rely on template (7) and instantiate it into:

$$\llbracket \{\neg(\diamond \text{get} \wedge \diamond \text{cancel})\} = \text{perm\_false} \rrbracket \rightarrow \langle \text{re}_{\llbracket \{\neg(\diamond \text{get} \wedge \diamond \text{cancel})\} = \text{perm\_false} \rrbracket} \rangle \diamond \text{return}.$$

This formula is equivalent to

$$(\diamond\text{get} \wedge \diamond\text{cancel}) \rightarrow \langle \text{re}_{\{\diamond\text{get} \wedge \diamond\text{cancel}\}} \rangle \diamond\text{return},$$

which, in turn, becomes

$$(\diamond\text{get} \wedge \diamond\text{cancel}) \rightarrow \left\langle \begin{array}{l} (o^*; \text{get}; (\neg\text{cancel})^*; \text{cancel}; \text{true}^*) \\ +(o^*; \text{cancel}; \neg\text{get}^*; \text{get}; \text{true}^*) \end{array} \right\rangle \diamond\text{return},$$

where  $o$  is a shortcut notation for any task different than get and cancel.

*Constraint priority for conflict resolution.* Thanks to the fact that RV states take into considerations all possible future evolutions of a monitored execution, our framework handles the subtle situation where the execution reaches a state of affairs in which the conjunction of two constraints is permanently violated, while none of the two is so if considered in isolation. This situation of *conflict* has been already recalled in Section 4.1 in the case of DECLARE. A situation of conflict involving two constraints  $\varphi$  and  $\psi$  in a given situation witnesses that even though none of  $\varphi$  and  $\psi$  is permanently violated in that situation, they contradict each other, and hence in every possible future course of execution from that situation, at least one of them will eventually become permanently violated. In such a state of affairs, it may become relevant to specify which one of the two constraints has *priority* over the other, that is, which one should be preferably satisfied.

Expressing preferences of this form is particularly important in those situations where conflicts emerge from the interplay of different sets of constraints, possibly elicited by different parties, and that may indeed contradict each other. This is, for example, the case in the medical domain, in particular when dealing with clinical guidelines. In fact, each clinical guideline is typically modeled in isolation, that is, by assuming ideal patients who only present those characteristics that are relevant for the guideline at hand. When executing such a guideline on an actual patient, the circumstances are obviously much more complex. On the one hand, the patient may present so-called comorbidities, which in turn would call for the concurrent execution of multiple guidelines at once [45]. On the other hand, during the execution, medical experts combine the specific process knowledge conveyed by the guideline with their own background medical knowledge [7]. Such different sources of knowledge induce constraints that may contradict each other, in turn calling for conflict resolution strategies that depend on the specific conflicting rules [7, 45]. In this respect, the metaconstraint pattern described next can be used to make such conflict-resolution strategies explicit and inform monitors correspondingly.

Formally, a trace culminates in a conflict for two  $\text{LDL}_f$  constraints  $\varphi$  and  $\psi$  if it satisfies the following metaconstraint:

$$\llbracket \{\varphi \wedge \psi\} = \text{perm\_false} \rrbracket \wedge \neg \llbracket \varphi = \text{perm\_false} \rrbracket \wedge \neg \llbracket \psi = \text{perm\_false} \rrbracket. \quad (8)$$

Specifically, assuming that  $\varphi$  and  $\psi$  can potentially enter into a conflict, a monitor for formula (8) proceeds as follows:

- Initially, the monitor outputs *temp\_false*, witnessing that no conflict has been seen so far, but it may actually occur in the future.
- From this initial situation, the monitor can evolve in one of the following two ways:
  - the monitor turns to *perm\_false*, witnessing that from this moment on neither of the two constraints will ever be violated anymore, irrespectively of how the trace continues;
  - the monitor turns to *temp\_true*, whenever the monitored execution indeed culminates in a conflict—this witnesses that a conflict is currently in place.
- From the latter situation witnessing the presence of a conflict, the monitor evolves then to *perm\_false* when one of the two constraints indeed becomes permanently violated; this

witnesses that the conflict is not anymore in place, due to the fact that now the permanent violation can actually be ascribed to one of the two constraints taken in isolation from the other.

Using this monitor, we can identify all points in the trace where a conflict is in place by simply checking when the monitor returns *temp\_true*.

Notice that the monitor never outputs *perm\_true*, since a conflicting situation will always eventually permanently violate  $\varphi$  or  $\psi$ , in turn permanently violating (8). In addition, the notion of conflict defined in formula (8) is inherently “non-monotonic,” as it ceases to exist as soon as one of the two involved constraints becomes permanently violated alone. This is the reason why we cannot directly employ formula (8) as a basis to define which constraint we *prefer* over the other when a conflict arises. To declare that  $\varphi$  is *preferred over*  $\psi$ , we then relax formula (8) by simply considering the violation of the composite constraint  $\varphi \wedge \psi$ , which may occur due to a conflict or due to the permanent violation of one of the two constraints  $\varphi$  and  $\psi$ . We then create a formula expressing that whenever the composite constraint is violated, then we want to satisfy the preferred constraint  $\varphi$ :

$$\langle re_{\llbracket \{\varphi \wedge \psi\} = perm\_false \rrbracket} \rangle tt \rightarrow \varphi. \quad (9)$$

In the typical situation where a permanent violation of  $\varphi \wedge \psi$  does not manifest itself at the beginning of the trace, but may indeed occur in the future, a monitor for formula (9) starts by emitting *temp\_true*. When the composite constraint  $\varphi \wedge \psi$  becomes permanently violated (either because of a conflict or because of a permanent violation of one of its components), formula  $\llbracket \{\varphi \wedge \psi\} = perm\_false \rrbracket$  turns to *perm\_true*, and the monitor consequently switches to observe the evolution of  $\varphi$  (that is, of the head of the implication in formula (9)).

Finally, notice that both patterns (8) and (9) can be generalized to conflicts and preferences expressed over a set of  $n$  formulae. In particular, formula (8) would in this case need to conjoin the fact that the conjunction of all  $n$  constraints is permanently violated with the fact that this is not the case for any of the conjunctions over the maximal subsets of  $n - 1$  such constraints. For example, for a set  $\{\varphi_1, \varphi_2, \varphi_3\}$  of three constraints, we would obtain:

$$\llbracket \varphi_1 \wedge \varphi_2 \wedge \varphi_3 = perm\_false \rrbracket \\ \wedge \neg \llbracket \varphi_1 \wedge \varphi_2 = perm\_false \rrbracket \wedge \neg \llbracket \varphi_1 \wedge \varphi_3 = perm\_false \rrbracket \wedge \neg \llbracket \varphi_2 \wedge \varphi_3 = perm\_false \rrbracket.$$

Formula (9), instead, would require to identify, among the maximal subsets containing  $n - 1$  constraints over the  $n$  ones, which is the preferred one. Assuming that the set of constraints is  $\{\varphi_1, \dots, \varphi_n\}$  and that the preferred maximal subset is the one containing all such constraints but the one indexed by  $j$ , we then get:

$$\langle re_{\llbracket \bigwedge_{i \in \{1, \dots, n\}} \varphi_i = perm\_false \rrbracket} \rangle tt \rightarrow \bigwedge_{i \in \{1, \dots, n\}, i \neq j} \varphi_i.$$

*Example 6.4.* Consider again Figure 1, and in particular the response and not coexistence constraints, respectively, linking pay registration to get ticket and get ticket to cancel registration, which we compactly refer to as  $\psi_r$  and  $\varphi_{nc}$ . These two constraints enter in a conflict when a registration is paid and canceled, but the ticket is not retrieved (this would indeed lead to a permanent violation of  $\varphi_{nc}$  alone). Let  $o$  denote any task that is different from pay, get, and cancel. The traces that culminate in a conflict for  $\psi_r$  and  $\varphi_{nc}$  are those that satisfy the regular expression:

$$(o^*; \text{pay}; (o + \text{pay})^*; \text{cancel}; (\neg \text{get})^*) + (o^*; \text{cancel}; (o + \text{cancel})^*; \text{pay}; (\neg \text{get})^*) \quad (10)$$



Recall that, as specified in Section 2, testing whether a trace satisfies this regular expression can be done by encoding it in  $LDL_f$  as:

$$\langle (o^*; \text{pay}; (o + \text{pay})^*; \text{cancel}; (\neg \text{get})^*) + (o^*; \text{cancel}; (o + \text{cancel})^*; \text{pay}; (\neg \text{get})^*) \rangle \text{end}. \quad (11)$$

We want to express that we prefer the `not coexistence` constraint over the `response` one, i.e., that, upon cancellation, the ticket should not be retrieved even if the payment has been done. To this end, we first notice that, for an evolving trace, the composite constraint  $\psi_r \wedge \varphi_{nc}$  is permanently violated either when  $\varphi_{nc}$  is so or when a conflict arises. The first situation arises when the trace contains both `cancel` and `get` (in whatever order), whereas the second arises when the trace contains both `cancel` and `pay` (in whatever order). Consequently, we have that  $re_{\llbracket \{\varphi_{nc} \wedge \psi_r \} = \text{perm\_false} \rrbracket}$  corresponds to the regular expression:

$$\begin{aligned} & (o^*; \text{pay}; (\neg \text{cancel})^*; \text{cancel}; (\text{true})^*) \\ & + (o^*; \text{get}; (\neg \text{cancel})^*; \text{cancel}; (\text{true})^*) \\ & + (o^*; \text{cancel}; (\text{cancel} + o)^*; (\text{get} + \text{pay}); (\text{true})^*). \end{aligned}$$

We then use this regular expression together with  $\varphi_{nc}$  to instantiate formula (9) as follows:

$$\left\langle \begin{array}{l} (o^*; \text{pay}; (\neg \text{cancel})^*; \text{cancel}; (\text{true})^*) \\ + (o^*; \text{get}; (\neg \text{cancel})^*; \text{cancel}; (\text{true})^*) \\ + (o^*; \text{cancel}; (\text{cancel} + o)^*; (\text{get} + \text{pay}); (\text{true})^*) \end{array} \right\rangle tt \rightarrow \neg(\diamond \text{get} \wedge \diamond \text{cancel})$$

We now consider a final example showing the evolution of the monitors for the metaconstraints discussed in the various examples of this section.

*Example 6.5.* Figure 8 reports the result computed by the monitors for the metaconstraints discussed in Examples 6.2, 6.3, and 6.4 on a sample trace. When the payment occurs, the contextual absence constraint forbids to get tickets. The prohibition is then permanently removed upon the consequent acceptance of the regulation, which ensures that the selected context will never appear again.

The execution of the third step, consisting in the cancellation of the order, induces a conflict for  $\neg(\diamond \text{get} \wedge \diamond \text{cancel})$  and  $\square(\text{pay} \rightarrow \circ \diamond \text{get})$ , since they, respectively, forbid and require to eventually get the ticket. The monitor for the `conflict` metaconstraint witnesses this by switching to *temp\_true*. The preference stays instead *temp\_true*, but while up to this point it was emitting *temp\_true* because no conflict had occurred yet, it now emits *temp\_true*, because this is the current *rv* state of the preferred, `not coexistence` constraint.

The execution of the `get ticket` task induces a permanent violation for constraint  $\neg(\diamond \text{get} \wedge \diamond \text{cancel})$ , which, in turn, triggers a number of effects:

- Since the preference metaconstraint is now following the evolution of the preferred constraint  $\neg(\diamond \text{get} \wedge \diamond \text{cancel})$ , it also moves to *perm\_false*.
- The conflict is not present anymore and will never be encountered again, given that one of its two constraints is permanently violated on its own. Thus, the monitor for the `conflict` metaconstraint turns to *perm\_false*.
- The reactive `compensation` is triggered by the permanent violation of  $\neg(\diamond \text{get} \wedge \diamond \text{cancel})$  and asserts that, from now on, the compensating constraint  $\diamond \text{return}$  must be satisfied; since the ticket is yet to be returned, the metaconstraint turns to *temp\_false*.

The execution of the last step, consisting in returning the ticket, has the effect of permanently satisfying the `compensation` metaconstraint, which was indeed waiting for this task to occur.

All in all, consider the `DECLARE` model in Figure 1 with two modifications: The responded existence constraint shown therein is substituted with the contextualized version of Example 6.2;

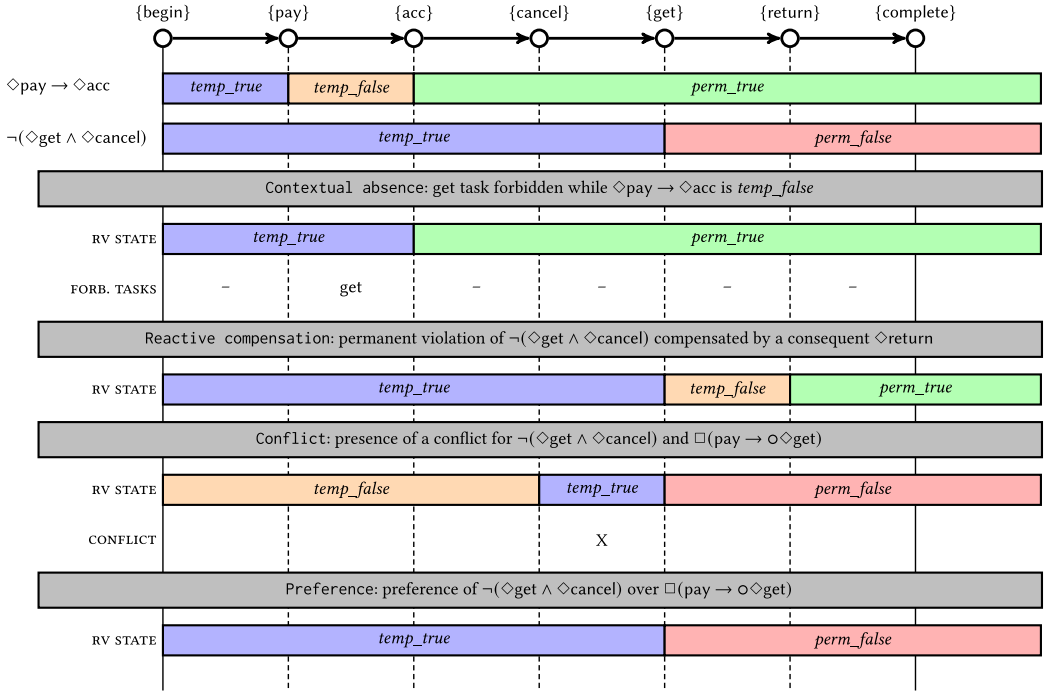


Fig. 8. Result computed by monitoring the metaconstraints in Examples 6.2, 6.3, and 6.4 against the trace  $\{\text{begin}\}, \{\text{pay}\}, \{\text{acc}\}, \{\text{cancel}\}, \{\text{get}\}, \{\text{return}\}$ ; for readability, we also report the evolution of the monitors for the constraints mentioned by the metaconstraints.

the compensation and preference constraints of Examples 6.3 and 6.4 are added. The overall trace does not comply with the original DECLARE model in Figure 1, since it violates not coexistence constraint between get and cancel. However, it complies with the revised model, since it properly compensates to the violation of not coexistence by suitably returning the ticket. Notice that, in doing so, the trace does not respect the preference formulated in Example 6.4, as the preferred not coexistence constraint is indeed violated.

## 7 IMPLEMENTATION

The entire monitoring approach illustrated in this article has been implemented in the MOBUCONLDL component of the RUM rule mining framework [1, 2]. RUM is the most complete and updated framework for process mining with DECLARE.<sup>9</sup> In particular, MOBUCONLDL appears in the monitoring section of the tool.

The reasoning component of MoBuConLdL is called FLLOAT (“From LTL<sub>f</sub>/LDL<sub>f</sub> To Automata”). The source code of FLLOAT is available at <https://github.com/RiccardoDeMasellis/FLLOAT>, while an introductory webpage containing information on the architecture of the code, its main APIs, and the examples/experiments contained in this article, can be found at <https://mobucon.inf.unibz.it>.

MOBUCONLDL requires as inputs an event log and a reference model defined using DECLARE. Each constraint and activity in the reference model can be used in RUM to build metaconstraints based on parameterized templates expressing the behaviors introduced in Section 6, i.e., *contextual*

<sup>9</sup>See <https://rulemining.org>.

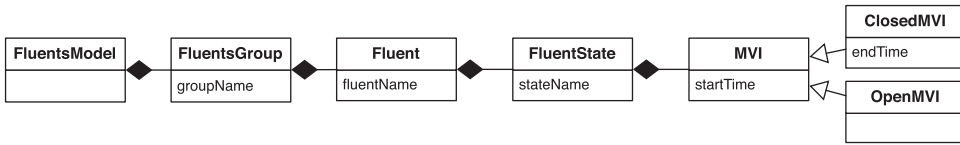


Fig. 9. Fluent model used to store the evolution of constraints.

*absence, reactive compensation, conflict, and preference.* The monitor is implemented in a way that it reads events incrementally from the input log so the log is actually replayed and monitored in a simulation environment. During the replay, the current processed event is sent to a reasoning component that computes the current status of the constraints in the reference model and returns the actual result's computation to the graphical interface. The reasoning component is specifically dedicated to the construction and manipulation of NFAs from  $LDL_f/LTL_f$  formulae (detailed in Section 7.1), concretely implementing the technique presented in Section 3.

The events processed by the monitor are expressed using the XES format ([www.xes-standard.org/](http://www.xes-standard.org/)) for event data. XES is an extensible XML-based standard recently adopted by the IEEE task force on process mining. The response produced by MoBUConLDL contains the temporal information related to the evolution of each monitored constraint from the beginning of the trace up to now. At each time point, a constraint can be in one state, which models whether it is currently: *(permanently) satisfied*, i.e., the current execution trace complies with the constraint; *possibly satisfied*, i.e., the current execution trace is compliant with the constraint, but it is possible to violate it in the future; *(permanently) violated*, i.e., the process instance is not compliant with the constraint; *possibly violated*, i.e., the current execution trace is not compliant with the constraint, but it is possible to satisfy it by generating some sequence of events. This state-based evolution is encapsulated in a *fluent model* that obeys to the schema sketched in Figure 9; this model, initially introduced in References [9, 14], is exploited in the monitoring interface, following Reference [33]. A fluent model aggregates groups of fluents, containing sets of correlated fluents. Each fluent models a multi-state property that changes over time. In our setting, fluent names refer to the constraints of the reference model. The fact that the constraint was in a certain state along a (maximal) time interval is modeled by associating a closed MVI (Maximal Validity Interval) to that state. MVIs are characterized by their starting and ending timestamps. Current states are associated to open MVIs, which have an initial fixed timestamp but an end that will be bounded to a currently unknown future value. Figure 10 shows the interface running with the example in Figure 8.

## 7.1 Reasoning Component

FLLOAT implements the logics of the runtime verification by building the automaton of the reference model  $LDL_f$  constraints with the algorithms presented in Section 3. The FLLOAT code has been implemented in Java by using an object-oriented paradigm, and it has been tested with JUnit test cases. It is made up by several conceptual modules and makes use of external libraries as shown by the UML diagram in Figure 11, where the main classes are depicted by the usual rectangles, their surrounding boxes represent the conceptual modules they belong to, and dashed arrows show relevant dependencies. We briefly comment on each module.

**Formulae.** It contains classes and methods to represent and manipulate logical formulae, including the  $\delta$  function in Figure 2, a method for translating in negation normal form, and the translation  $LTL_f$  to  $LDL_f$  presented in Section 2.

**Automaton construction.** The main functionality provided by FLLOAT is the construction of the NFA and DFA for an  $LTL_f/LDL_f$  formula  $\varphi$  given as input. The whole procedure works as follows:

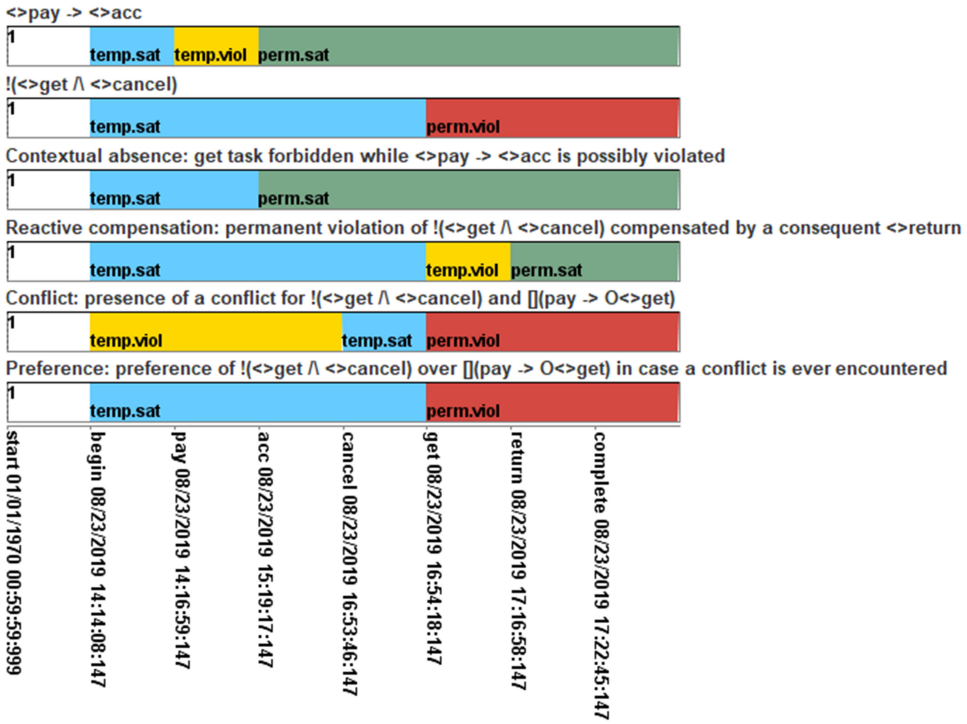


Fig. 10. Screenshot of the MobuconLDL graphical interface in RuM.

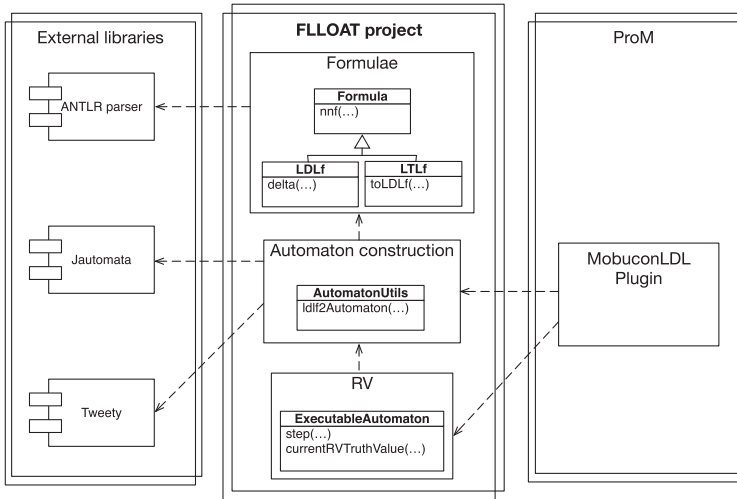


Fig. 11. UML-like diagram of the backend main components.

First, the formula is parsed and an LTLf or LDLf object is created. This is achieved by classes that are previously and automatically generated by ANTLR<sup>10</sup> starting from grammar files. If the input formula is LDL<sub>f</sub>, then it is translated in negation normal form and the algorithm for the

<sup>10</sup><http://www.antlr.org>.

automata generation is called. Conversely, if it is  $LTL_f$ , then it is first converted to  $LDL_f$ . Once an  $LDL_f$  formula  $\varphi$  in negation normal form has been obtained, the automaton is generated with Algorithm  $LDL_f2NFA$  in Figure 3. The method *ldlf2Automaton* consists of two nested cycles: the outer on states in  $\mathcal{S}$  yet to be analyzed and the inner on interpretation for  $\mathcal{P}$  or the empty trace (line 8 of Algorithm  $LDL_f2NFA$ ). At each iteration  $\delta(s, \Theta)$  is computed, where  $s \in \mathcal{S}$  and  $\Theta \in 2^{\mathcal{P}} \cup \epsilon$ , possibly generating new states  $q'$  to be added to the set of states to be analyzed along with the respective transitions (lines 8–9 of Algorithm  $LDL_f2NFA$ ). The implementation employs *explicit* automata, where each transition is decorated with a specific symbol from  $\mathcal{P}$  (thus, in general, two states may be linked via multiple transitions, each with a distinguished symbol). The Tweety library<sup>11</sup> is used to compute the models, i.e.,  $q'$  states, of the formula  $\bigwedge_{(\psi \in q)} \delta(\psi, \Theta)$  in line 8 of Algorithm  $LDL_f2NFA$ . The procedure starts by analyzing  $\varphi$ , the only state in  $\mathcal{S}$ , and ends when all states have been analyzed and no further states have been generated in the meanwhile. The data structures for automata are defined in the *jautomata* library,<sup>12</sup> which provides methods for automata manipulation.

**Runtime Verification.** The runtime verification functionalities are provided by the `ExecutableAutomaton` class. An executable automaton is essentially a DFA with a reference to the current state. When an executable automaton is created, the current state is set to the initial state (by construction, there is always a unique initial state). The idea is to navigate the automaton and return RV truth values while events are executed. Recalling the results presented in Section 4.2, each automaton state represents an RV truth value. Hence, an operative way to implement an RV monitor is to analyze one-by-one the occurring events and to perform the corresponding transitions on the automaton of the constraints. Each time a state change is triggered by a transition leading to state  $s$ , we calculate  $Reach(s)$  and return the corresponding truth value.

## 7.2 Building Monitors for DECLARE

We now focus on the construction of monitors for DECLARE to realize the approach described in Section 5. Consider a declare model  $\mathcal{M}$ ,  $n$  constraints  $\varphi_1, \dots, \varphi_n$ . To monitor  $\mathcal{M}$ , one needs  $n + 1$  colored DFAS:

- $n$  local monitors, where monitor  $i$  tracks the evolution of the RV state of the single constraint  $\varphi_i$ ;
- one global monitor, accounting for the interplay of all  $n$  constraints at once.

The global monitor has, in the worst case, a size that is doubly exponential in the size of the formula expressing the conjunction of all  $n$  constraints in the model. The local monitors can instead be precomputed for the different DECLARE templates (as shown in Table 2), then simply replacing the task placeholders used in the template with the corresponding, actual tasks. It is also interesting to observe that, by employing the technique described in Reference [33], one could solely rely on the global monitor, which can be constructed in such a way that each state therein retains the local color that each single constraint has in that state.

We thus focus, in the remainder of the section, on the construction of the global monitor. A naive way of constructing the monitor would be to simply build the overall formula of the DECLARE model by conjoining all its constraint  $LTL_f$  formulae, then invoking the  $LDL_f2NFA$  algorithm of Section 3, followed by a determinization step. However, as just recalled, the size of the global monitor is, in the worst case, double exponential, which would make this way of operating highly impractical.

<sup>11</sup><http://tweetyproject.org>.

<sup>12</sup><https://github.com/abailly/jautomata>.

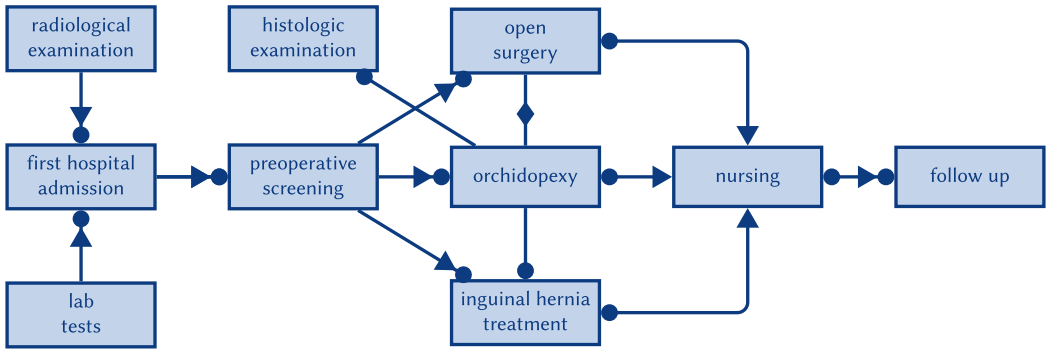


Fig. 12. DECLARE model for a cryptorchidism treatment clinical guideline (from [49]).

Fortunately, there are two important facts hinting at a better complexity in the average case, and at a more clever way to construct the monitor. First, the determinization step often produces a DFA that is of similar size, if not smaller, than the NFA from which it is generated [52]. Second, in the specific case of DECLARE, each DECLARE template comes with a pre-computed automaton of polynomial size, which is actually the minimal DFA of its corresponding  $LTL_f$  formula (as shown in Table 2 for some templates). Combining such DFAs with Boolean operators (conjunction, disjunction, negation) preserves determinism, but not minimality.

We exploit this latter observation to construct the global monitor  $\mathcal{A}$  through the following *lazy construction* procedure: Set  $\mathcal{A} = A(\varphi_1)$ , that is,  $\mathcal{A}$  is the local DFA for the first constraint of the DECLARE model  $\mathcal{M}$ . Then loop over the remaining constraints, where the  $i$ th iteration step is as follows:

- (1) get the DFA  $A_i = A(\varphi_{i+1})$  for constraint  $\varphi_{i+1}$  from  $\mathcal{M}$ ;
- (2) intersect the current global DFA  $\mathcal{A}$  with  $A_i$ , recalling that the intersection is still a DFA;
- (3) minimize the so-obtained intersection, assigning the resulting DFA to  $\mathcal{A}$ .

At the end of the loop,  $\mathcal{A}$  corresponds to the minimal DFA for  $\mathcal{M}$  (that is, for the formula  $\bigwedge_{i \in \{1, \dots, n\}} \varphi_i$ ). Minimizing at each step guarantees that the size of  $\mathcal{A}$  is kept as controlled as possible during the whole computation. Finally,  $\mathcal{A}$  can be colored with RV values either during the construction procedure or through a simple post-processing step.

### 7.3 Performance Indications

Even though this article is not primarily concerned with performance and optimization issues, we close this section by commenting on the feasibility of our monitoring technique, considering the case of DECLARE. We focus on the monitoring construction step, which is computationally the most demanding: Once the DFA monitor for the input DECLARE model is built, monitoring a trace at runtime becomes a trivial operation, which amounts to following transitions matched by activity occurrences received by the monitor. For our experiments, we use the simple procedure described in Section 7.2. The experiments can be replicated by downloading the FLOAT package and by following the procedure described at <https://mobucon.inf.unibz.it>.

We start by considering the construction of a monitor for a real DECLARE process in the medical domain [49].

*Example 7.1.* Consider the DECLARE model in Figure 12, capturing a real clinical guideline for cryptorchidism. The global monitor for this model is the colored DFA corresponding to the conjunction of the 13  $LTL_f$  formulae that encode the 13 constraints shown in the figure. By relying



on the equivalence of logical conjunction and automata intersection exploited in Section 7.2, we fix an order over the 13 constraints and incrementally build the global monitor  $\mathcal{A}$  using the lazy construction procedure described there. On a Macbook Pro M1 with 16 MB of RAM, this procedure takes 0, 58 s to produce the global, minimal DFA of the global monitor, consisting of 49 states and 1,029 transitions.

We now scale up the model from Example 7.1 to obtain some indications on the performance of the approach. This is done by replicating the model sequentially, as previously done in Reference [40] to guarantee that the resulting model is still meaningful in terms of how constraints are interrelated. The approach differs from the one of random model generation adopted in Reference [54], which does not provide any guarantee on the meaningfulness of the obtained models, nor on their satisfiability.

*Example 7.2.* To give an indication about how the lazy construction procedure of Example 7.1 scales, we perform the following experiment: We measure the time and automaton size<sup>13</sup> by starting from the model that contains only the 10 activities of the DECLARE model of Figure 12 and no constraints, and then incorporate the constraints one-by-one. Once all 13 constraints are incorporated, we produce a replica of the 10 activities and add a precedence constraint connecting the first follow up activity to the second hospital admission one. We then proceed as before, adding one-by-one the 13 constraints shown in Figure 12, this time on the replicated activities. We then repeat this process.

Figure 13 reports how the size of the automaton and the lazy construction time change as the number of constraints increases. Entries with values 0, 14, and 28 for the x-axis are those where the model is expanded with new activities. One of these activities comes with the precedence constraint relating it to the previous follow up, and consequently participates to the complexity through that constraint. The other activities instead come completely unconstrained, and therefore they only participate to increase the number of transitions in the monitor; this happens because, as pointed out before, FLLOAT employs an explicit representation of transitions.

The case of Example 7.1 corresponds to the size and time for value 13 on the x-axis. The last experiment is for a DECLARE model consisting of 30 activities and 30 constraints, in turn corresponding to a conjunctive  $LTL_f$  formula of size 244 (measured as the number of activities, logical and temporal operators employed); for such a model, the monitor has a size of  $\sim 2.5 \cdot 10^5$ , and building it takes approximately 18 minutes. One can notice that while moving from 0 to 13 constraints, the automaton size goes approximately from 0 to 10K, while that of the timing from 0 to 1 s; while moving from 13 to 26 constraints, the two trends, respectively, move from 10K to 30K and from 1 s to 30 s; they finally reach 250K and 1K s for 30 constraints. In addition, it is immediate to see that the two curves are uneven. This is because the addition of a constraint, in turn calling for an automata intersection step, impacts differently, depending on semantics of the added constraint and how it interplays with the other constraints.

We close with three observations. First, the lazy procedure adopted here does not incorporate further algorithmic improvements and optimizations, nor semi-symbolic and symbolic techniques for representing transitions compactly. These aspects have been studied for DECLARE [54] and for the whole  $LTL_f$  in References [3, 17, 31, 55]. Importantly, they can be seamlessly integrated in our approach. Second, recall that the global monitor is “only” required for detecting permanent violations at the earliest moment possible. If this is not needed, then local monitors suffice for the purpose, as whenever the global monitor detects a permanent violation, then eventually at least

<sup>13</sup>Computed by simply summing up the number of states and the number of transitions of the automaton.

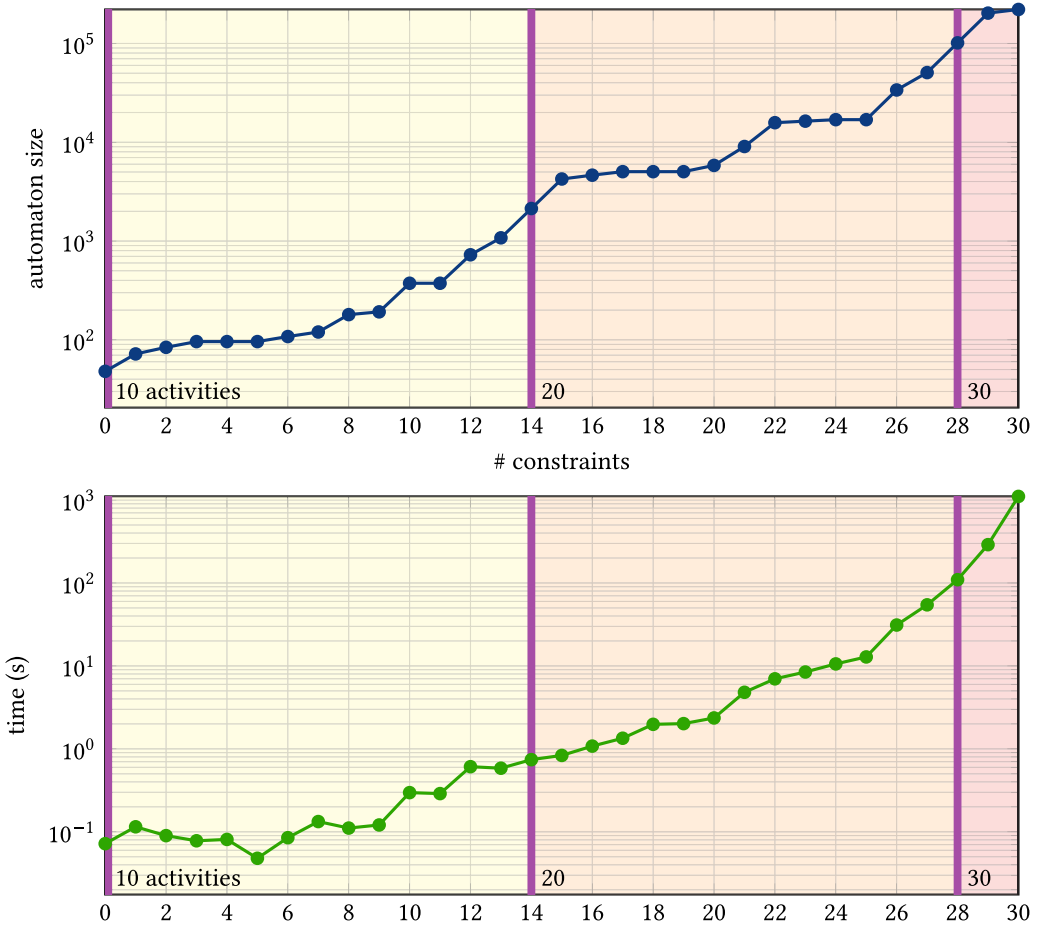


Fig. 13. Trends of time and size of the global monitor as the number of constraints increases, in the experiment of Example 7.2.

one of the local monitors will detect a permanent violation. Third, experimentally, the setting does not change when metaconstraints are considered, since they are always represented as standard  $LDL_f$  formulae.

## 8 CONCLUSION

In this article, we have brought forward a foundational and practical approach to formalize and monitor linear temporal constraints and metaconstraints, under the assumption that the traces generated by the system under study are *finite*. This is the typical case in the context of business process management and service-oriented architectures, where each execution of a business process or service invocation leads from a starting state to a completion state in a possibly unbounded, yet finite, number of steps.

The main novelty of our approach is to adopt a more powerful specification logic, that is,  $LDL_f$  (which corresponds to Monadic Second-order Logic over finite traces), instead of the typical choice of  $LTL_f$  (which corresponds to First-order Logic over finite traces). Like in the case of  $LTL_f$ , also  $LDL_f$  comes with an automata-theoretic characterization that employs standard finite-state

automata. Differently from  $LTL_f$ , though,  $LDL_f$  can declaratively express, within the logic, not only constraints that predicate on the dynamics of task executions, but also constraints that predicate on the monitoring state of other constraints.

The approach has been fully implemented as an independent library to specify  $LDL_f/LTL_f$  formulae as well as obtain and manipulate their corresponding automata, which is then invoked by a process-monitoring infrastructure that has been developed within the state-of-the-art RuM rule mining framework. A natural next step is to incorporate, and study, the different automata-construction optimizations studied in the literature and study their (combined) impact on time and space complexity incurred when constructing monitors for formulae of increasing length.

As main continuation for this work, we intend to incorporate other monitoring perspectives, such as in particular data objects carried by the monitored events. This setting is reminiscent of stream query languages and event calculi. For example, a reactive form [14] of the logic-based Event Calculus [30] has been applied to process monitoring against data-aware extensions of the DECLARE language in Reference [38], also considering some specific forms of compensation [13]. However, these approaches are only meant to query and reason over the events collected so far in the monitored trace, without conducting any form of speculative reasoning on its possible future continuations, as we do in our approach. Genuine investigation is then required to understand under which conditions it is possible to lift the automata-based techniques presented here to the case where events are equipped with a data payload and constraints are expressed in (fragments of) first-order temporal logics over finite traces. So far, this has been attempted only by interpreting data over a fixed, finite quantification domain [20], which makes the approach directly reducible to the propositional setting studied here.

Another interesting line we want to pursue is to consider monitoring in an adversarial environment where multiple *external actors* interact, possibly adversarially, to progress the process. This requires to link the *rv* conditions to key notions like “forcing” (in spite of the environment). This relates to synthesis in formal methods [47] and planning in nondeterministic domains in Artificial intelligence [24]. Among the various works, in the infinite-trace setting, Reference [35] studies GR(1) specifications (a computational web behaved fragment of LTL, [6]) obeying to the patterns introduced in Reference [21], which are indeed at the basis of the DECLARE process modeling language [37, 43]. However, recently in planning there has been a lot of interest on specifications expressed in  $LTL_f$  on finite traces [11, 18]. Interestingly,  $LTL_f$  can be seen as a fragment of LTL that is orthogonal to GR(1), and for which synthesis can be effectively scaled up as in the case of GR(1) (see Reference [25] for a discussion).

## ACKNOWLEDGMENTS

The authors would like to thank Anti Alman for the advice and support in the implementation effort.

## REFERENCES

- [1] Anti Alman, Claudio Di Ciccio, Dominik Haas, Fabrizio Maria Maggi, and Alexander Nolte. 2020. Rule mining with RuM. In *Proceedings of the 2nd International Conference on Process Mining*. 121–128.
- [2] Anti Alman, Claudio Di Ciccio, Fabrizio Maria Maggi, Marco Montali, and Han van der Aa. 2021. RuM: Declarative process mining, distilled. *Lecture Notes in Computer Science*, Vol. 12875, Artem Polyvyanyy, Moe Thandar Wynn, Amy Van Looy, and Manfred Reichert (Eds.). Springer, 23–29.
- [3] Suguman Bansal, Yong Li, Lucas M. Tabajara, and Moshe Y. Vardi. 2020. Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications. In *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press, 9766–9774.
- [4] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2010. Comparing LTL semantics for runtime verification. *Log. Computat.* 20, 3 (2010).

- [5] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* 20, 4 (2011), 14:1–14:64.
- [6] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa’ar. 2012. Synthesis of Reactive(1) designs. *J. Comput. Syst. Sci.* 78, 3 (2012), 911–938.
- [7] Alessio Bottrighi, Federico Chesani, Paola Mello, Marco Montali, Stefania Montani, and Paolo Terenziani. 2011. Conformance checking of executed clinical guidelines in presence of basic medical knowledge. In *Proceedings of the 19th International Conference on Business Process Management (BPM’21) (Lecture Notes in Business Information Processing, Vol. 100)*, Florian Daniel, Kamel Barkaoui, and Schahram Dustdar (Eds.). Springer, 200–211.
- [8] Ronen I. Brafman, Giuseppe De Giacomo, and Fabio Patrizi. 2018. LTLf/LDLf non-Markovian rewards. In *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press, 1771–1778.
- [9] Stefano Bragaglia, Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni. 2012. Reactive event calculus for monitoring global computing applications. In *Logic Programs, Norms and Action - Essays in Honor of Marek J. Sergot on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 7360)*, Alexander Artikis, Robert Craven, Nihan Kesim Cicekli, Babak Sadighi, and Kostas Stathis (Eds.). Springer, 123–146.
- [10] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2009. An automata-theoretic approach to regular XPath. *Lecture Notes in Computer Science*, Vol. 5708. Springer, 18–35.
- [11] Alberto Camacho and Sheila A. McIlraith. 2019. Strong fully observable non-deterministic planning with LTL and LTLf goals. In *Proceedings of the International Joint Conference on Artificial Intelligence*. ijcai.org, 5523–5531.
- [12] Federico Chesani, Catherine G. Enright, Marco Montali, and Michael G. Madden. 2015. Monitoring in the healthcare setting. In *Foundations of Biomedical Knowledge Representation—Methods and Applications (Lecture Notes in Computer Science, Vol. 9521)*, Arjen Hommersom and Peter J. F. Lucas (Eds.). Springer, 71–80.
- [13] Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni. 2008. Verification of choreographies during execution using the reactive event calculus. In *Proceedings of the 5th International Workshop on Web Services and Formal Methods (WS-FM’08) (Lecture Notes in Computer Science, Vol. 5387)*. Springer.
- [14] Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni. 2010. A logic-based, reactive calculus of events. *Fundam. Informaticae* 105, 1–2 (2010), 135–161.
- [15] James Clark and Steve DeRose. 1999. *XML Path Language (XPath) Version 1.0*. W3C Recommendation. World Wide Web Consortium. <https://www.w3.org/TR/1999/REC-xpath-19991116/>.
- [16] Giuseppe De Giacomo, Riccardo De Masellis, Marco Grasso, Fabrizio Maria Maggi, and Marco Montali. 2014. Monitoring business metaconstraints based on LTL and LDL for finite traces. *Lecture Notes in Computer Science*, Vol. 8659. Springer.
- [17] Giuseppe De Giacomo and Marco Favorito. 2021. Compositional approach to translate LTLf/LDLf into deterministic finite automata. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS’21)*, Susanne Biundo, Minh Do, Robert Goldman, Michael Katz, Qiang Yang, and Hankui Hankui Zhuo (Eds.). AAAI Press, 122–130.
- [18] Giuseppe De Giacomo and Sasha Rubin. 2018. Automata-theoretic foundations of FOND planning for LTLf/LDLf goals. In *Proceedings of the International Joint Conference on Artificial Intelligence*. 4729–4735.
- [19] Giuseppe De Giacomo and Moshe Y. Vardi. 2013. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI’13)*. AAAI.
- [20] Riccardo De Masellis, Fabrizio Maria Maggi, and Marco Montali. 2014. Monitoring data-aware business constraints with finite state automata. In *Proceedings of the International Conference on Software and System Processes (ICSSP’14)*. ACM, 134–143. DOI: [10.1145/2600821.2600835](https://doi.org/10.1145/2600821.2600835)
- [21] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. Patterns in property specifications for finite-state verification. In *Proceedings of the International Conference on Software Engineering (ICSE’99)*, Barry W. Boehm, David Garlan, and Jeff Kramer (Eds.). ACM Press.
- [22] Marco Favorito. 2018. *Reinforcement Learning for LTLf/LDLf Goals: Theory and Implementation*. MEng Thesis. Sapienza University of Rome. Retrieved from <https://github.com/marcofavorito/master-thesis/blob/master/thesis.pdf>.
- [23] Michael J. Fischer and Richard E. Ladner. 1979. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.* 18 (1979).
- [24] Hector Geffner and Blai Bonet. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publishers.
- [25] Lucas M. Tabajara Moshe Vardi Shufang Zhu Giuseppe De Giacomo, Antonio Di Stasio. 2021. Finite-trace and generalized-reactivity specifications in temporal synthesis. In *Proceedings of the International Joint Conference on Artificial Intelligence*. ijcai.org.
- [26] Hermann Gruber and Markus Holzer. 2015. From finite automata to regular expressions and back—A summary on descriptive complexity. *Int. J. Found. Comput. Sci.* 26, 8 (2015), 1009–1040.

- [27] David Harel, Dexter Kozen, and Jerzy Tiuryn. 2000. *Dynamic Logic*. The MIT Press.
- [28] J. E. Hopcroft, R. Motwani, and J. D. Ullman. 2007. *Introduction to Automata Theory, Languages, and Computation* (3rd ed.). Addison Wesley Publishing Co.
- [29] IEEE. 2005. IEEE standard for property specification language (PSL). *IEEE Std 1850-2005*, 1–143. DOI: <https://doi.org/10.1109/IEEESTD.2005.97780>.
- [30] Robert A. Kowalski and Marek J. Sergot. 1986. A logic-based calculus of events. *New Gen. Comput.* 4, 1 (1986), 67–95. DOI: <https://doi.org/10.1007/BF03037383>.
- [31] Jianwen Li, Lijun Zhang, Shufang Zhu, Geguang Pu, Moshe Y. Vardi, and Jifeng He. 2018. An explicit transition system construction approach to LTL satisfiability checking. *Form. Asp. Comput.* 30, 2 (2018), 193–217.
- [32] Linh Thao Ly, Fabrizio Maria Maggi, Marco Montali, Stefanie Rinderle-Ma, and Wil M. P. van der Aalst. 2013. A framework for the systematic comparison and evaluation of compliance monitoring approaches. In *Proceedings of the 17th IEEE International Enterprise Distributed Object Computing Conference (EDOC'13)*. IEEE.
- [33] Fabrizio Maria Maggi, Marco Montali, Michael Westergaard, and Wil M. P. van der Aalst. 2011. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In *Proceedings of the 9th International Conference on Business Process Management (BPM'11) (Lecture Notes in Computer Science, Vol. 6896)*. Springer.
- [34] Fabrizio Maria Maggi, Michael Westergaard, Marco Montali, and Wil M. P. van der Aalst. 2012. Runtime verification of LTL-based declarative process models. *Lecture Notes in Computer Science, Vol. 7186*. Springer.
- [35] Shahar Maoz and Jan Oliver Ringert. 2015. GR(1) synthesis for LTL specification patterns. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 96–106.
- [36] Maarten Marx. 2004. XPath with conditional axis relations. *Lecture Notes in Computer Science, Vol. 2992*. Springer, 477–494.
- [37] Marco Montali. 2010. *Specification and Verification of Declarative Open Interaction Models: A Logic-based Approach*. *Lecture Notes in Business Information Processing, Vol. 56*. Springer.
- [38] Marco Montali, Fabrizio Maria Maggi, Federico Chesani, Paola Mello, and Wil M. P. van der Aalst. 2013. Monitoring business constraints with the event calculus. *ACM Trans. Intell. Syst. Technol.* 5, 1 (2013).
- [39] Marco Montali, Maja Pesic, Wil M. P. van der Aalst, Federico Chesani, Paola Mello, and Sergio Storari. 2010. Declarative specification and verification of service choreographies. *ACM Trans. Web* 4, 1 (2010).
- [40] Marco Montali, Paolo Torroni, Federico Chesani, Paola Mello, Marco Alberti, and Evelina Lamma. 2010. Abductive logic programming as an effective technology for the static verification of declarative business processes. *Fundam. Informaticae* 102, 3–4 (2010), 325–361.
- [41] Gyunam Park and Wil M. P. van der Aalst. 2020. A general framework for action-oriented process mining. *Lecture Notes in Business Information Processing, Vol. 397*, Adela del-Río-Ortega, Henrik Leopold, and Flávia Maria Santoro (Eds.). Springer, 206–218.
- [42] Maja Pesic. 2008. *Constraint-based Workflow Management Systems: Shifting Controls to Users*. Ph.D. Dissertation. Beta Research School for Operations Management and Logistics, Eindhoven.
- [43] Maja Pesic, Helen Schonenberg, and Wil M. P. van der Aalst. 2007. DECLARE: Full support for loosely-structured processes. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC'07)*. IEEE Computer Society, 287–300.
- [44] Maja Pesic and Wil M. P. van der Aalst. 2006. A declarative approach for flexible business processes management. In *Proceedings of the BPM Workshops (Lecture Notes in Computer Science, Vol. 4103)*. Springer.
- [45] Luca Piovesan, Paolo Terenziani, and Daniele Theseider Dupré. 2020. Conformance analysis for comorbid patients in Answer Set Programming. *J. Biomed. Inform.* 103 (2020), 103377.
- [46] Amir Pnueli. 1977. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*. IEEE.
- [47] Amir Pnueli and Roni Rosner. 1989. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'89)*. 179–190.
- [48] Henry Prakken and Marek J. Sergot. 1996. Contrary-to-duty obligations. *Studia Logica* 57, 1 (1996).
- [49] Marcella Rovani, Fabrizio Maria Maggi, Massimiliano de Leoni, and Wil M. P. van der Aalst. 2015. Declarative process mining in healthcare. *Expert Syst. Appl.* 42, 23 (2015), 9236–9251. DOI: <https://doi.org/10.1016/j.eswa.2015.07.040>.
- [50] L. J. Stockmeyer and A. R. Meyer. 1973. Word problems requiring exponential time. In *Proceedings of the Symposium on Theory of Computing (STOC'73)*. 1–9.
- [51] Lucas M. Tabajara and Moshe Y. Vardi. 2020. LTLf synthesis under partial observability: From theory to practice. In *Proceedings of the 11th International Symposium on Games, Automata, Logics, and Formal Verification (GandALF'20) (Electronic Proceedings in Theoretical Computer Science, Vol. 326)*, Jean-François Raskin and Davide Bresolin (Eds.). 1–17.

- [52] Deian Tabakov and Moshe Y. Vardi. 2005. Experimental evaluation of classical automata constructions. In *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05) (Lecture Notes in Computer Science, Vol. 3835)*, Geoff Sutcliffe and Andrei Voronkov (Eds.). Springer, 396–411.
- [53] Wil M. P. van der Aalst. 2016. *Process Mining—Data Science in Action, Second Edition*. Springer. DOI : <https://doi.org/10.1007/978-3-662-49851-4>.
- [54] Michael Westergaard. 2011. Better algorithms for analyzing and enacting declarative workflow languages using LTL. In *Proceedings of the International Conference on Business Process Management*.
- [55] Shufang Zhu, Lucas M. Tabajara, Jianwen Li, Geguang Pu, and Moshe Y. Vardi. 2017. Symbolic LTLf synthesis. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI'17)*, Carles Sierra (Ed.). ijcai.org, 1362–1369.

Received December 2019; revised December 2021; accepted December 2021