Monitoring Business Metaconstraints Based on LTL & LDL for Finite Traces

Giuseppe De Giacomo[1], Riccardo De Masellis[1], Marco Grasso[1], Fabrizio Maria Maggi[2], and Marco Montali[3]

[1] Sapienza Università di Roma, Via Ariosto, 25, 00185 Rome, Italy
(degiacomo|demasellis)@dis.uniroma1.it
[2] University of Tartu, J. Liivi 2, 50409 Tartu, Estonia
f.m.maggi@ut.ee
[3] Free University of Bozen-Bolzano, Piazza Domenicani 3, 39100 Bolzano, Italy
montali@inf.unibz.it

Abstract. Runtime monitoring is one of the central tasks to provide operational decision support to running business processes, and check on-the-fly whether they comply with constraints and rules. We study runtime monitoring of properties expressed in LTL on finite traces (LTL_f) and its extension LDL_f. LDL_f is a powerful logic that captures all monadic second order logic on finite traces, which is obtained by combining regular expressions with LTL_f, adopting the syntax of propositional dynamic logic (PDL). Interestingly, in spite of its greater expressivity, LDL_f has exactly the same computational complexity of LTL_f. We show that LDL_f is able to capture, in the logic itself, not only the constraints to be monitored, but also the de-facto standard RV-LTL monitors. This makes it possible to declaratively capture monitoring metaconstraints, i.e., constraints about the evolution of other constraints, and check them by relying on usual logical services for temporal logics instead of ad-hoc algorithms. This, in turn, enables to flexibly monitor constraints depending on the monitoring state of other constraints, e.g., "compensation" constraints that are only checked when others are detected to be violated. In addition, we devise a direct translation of LDL_f formulas into nondeterministic automata, avoiding to detour to Büchi automata or alternating automata, and we use it to implement a monitoring plug-in for the ProM suite.

1 Introduction

Runtime monitoring is one of the central tasks to provide *operational decision support* [21] to running business processes, and check on-the-fly whether they comply with constraints and rules. In order to provide well-founded and provably correct runtime monitoring techniques, this area is usually rooted into that of *verification*, the branch of formal analysis aiming at checking whether a system meets some property of interest. Being the system dynamic, properties are usually expressed by making use of modal operators accounting for the time.

Among all the temporal logics used in verification, Linear-time Temporal Logic (LTL) is particularly suited for monitoring, as an actual system execution is indeed linear. However, the LTL semantics is given in terms of infinite traces, hence monitoring must check whether the current trace is a prefix of an infinite trace, that will never be completed [7, 2]. In several context, and in particular often in BPM, we can assume that

the trace of the system is in fact finite [18]. For this reason, finite-trace variant of the LTL have been introduced. Here we use the logic LTL_f (LTL on finite traces), investigated in detail in [4], and at the base of one of the main declarative process modeling approaches: DECLARE [18, 16, 11]. Following [11], monitoring in LTL_f amounts to check whether the current execution belongs to the set of admissible *prefixes* for the traces of a given LTL_f formula φ. To achieve such a task, φ is usually first translated into a finite-state automaton for φ, which recognizes all those *finite* executions that satisfy φ.

Despite the presence of previous operational decision support techniques to monitoring LTL_f constraints over finite traces [11, 12], two main challenges have not yet been tackled in a systematic way. First of all, several alternative semantics have been proposed to make LTL suitable for runtime verification (such as the de-facto standard RV monitor conditions [2]), but no comprehensive technique based on finite-state automata is available to accommodate them. On the one hand, runtime verification for such logics typically considers finite partial traces whose continuation is however infinite [2], with the consequence that the corresponding techniques detour to Büchi automata for building the monitors. On the other hand, the incorporation of such semantics in the BPM setting (where also continuations are finite) has only been tackled so far with effective but ad-hoc techniques (cf. the "coloring" of automata in [11] to support the RV conditions), without a corresponding formal underpinning.

A second, key challenge is the incorporation of advanced forms of monitoring, where some constraints become of interest only in specific, critical circumstances (such as the violation of other constraints). This is the basis for supporting monitoring of compensation constraints and the so-called contrary-to-duty obligations [20], i.e., obligations that are put in place only when other obligations have not been fulfilled. While this feature is considered to be a fundamental compliance monitoring functionality [10], it is still an open challenge, without any systematic approach able to support it at the level of the constraint specification language.

In this paper, we attack these two challenges by studying runtime monitoring of properties expressed in LTL_f and in its extension LDL_f [4]. LDL_f is a powerful logic that captures all monadic second order logic on finite traces, which is obtained by combining regular expressions with LTL_f, adopting the syntax of propositional dynamic logic (PDL). Interestingly, in spite of its greater expressivity, LDL_f has exactly the same computational complexity of LTL_f. We show that LDL_f is able to capture, in the logic itself, not only the usual LDL_f constraints to be monitored, but also the de-facto standard RV conditions. Indeed given an LDL_f formula φ, we show how to construct the LDL_f formulas that captures whether prefixes of φ satisfy the various RV conditions. This, in turn, makes it possible to declaratively capture *monitoring metaconstraints*, and check them by relying on usual logical services 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, such as "compensation" constraints that are only checked when others are violated.

Interestingly, in doing so we devise a direct translation of LDL_f (and hence of LTL_f) formulas into nondeterministic automata, which avoid the usual detour to Büchi automata. The technique is grounded on alternating automata (AFW), but it actually avoids also their introduction all together, and directly produces a standard non-deterministic finite-state automaton (NFA). Notably, such technique has been implemented and em-

bedded into a monitoring plug-in for the PROM, which supports the check of LDL_f constraints and metaconstraints.

2 LTL_f and LDL_f

In this paper we will adopt the standard LTL and its variant LDL interpreted on finite runs.

LTL on finite traces, called LTL_f [4], has exactly the same syntax as LTL on infinite traces [19]. Namely, given a set of $\mathcal{P}$ of propositional symbols, LTL_f formulas are obtained through the following:

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

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

It is known that LTL_f is as expressive as First Order Logic over finite traces, so strictly less expressive than regular expressons which in turn are as expressive as Monadic Second Order logic over finite traces. On the other hand, regular expressions are a too low level formalism for expressing temporal specifications, since, for example, they miss a direct construct for negation and for conjunction [4].

To overcome this difficulties, in [4] *Linear Dynamic Logic of Finite Traces*, or LDL_f, has been proposed. This logic is as natural as LTL_f but with the full expressive power of Monadic Second Order logic over finite traces. LDL_f is obtained by merging LTL_f with regular expression through the syntax of the well-know logic of programs PDL, *Propositional Dynamic Logic*, [8, 9] but adopting a semantics based on finite traces. This logic is an adaptation of LDL, introduced in [22], which, like LTL, is interpreted over infinite traces.

Formally, LDL_f formulas are built as follows:

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

where ϕ is a propositional formula over $\mathcal{P}$; tt and ff denote respectively the true and the false LDL_f formula (not to be confused with the propositional formula true and false); ρ denotes path expressions, which are regular expressions over propositional formulas ϕ with the addition of the test construct $\varphi?$ typical of PDL; and φ stand for LDL_f formulas built by applying boolean connectives and the modal connectives $\langle\rho\rangle\varphi$ and $[\rho]\varphi$. In fact $[\rho]\varphi \equiv \neg\langle\rho\rangle\neg\varphi$.

Intuitively, $\langle\rho\rangle\varphi$ states that, from the current step in the trace, there exists an execution satisfying the regular expression ρ such that its last step satisfies φ. While $[\rho]\varphi$ states that, from the current step, all executions satisfying the regular expression ρ are such that their last step satisfies φ. Tests are used to insert into the execution path checks for satisfaction of additional LDL_f formulas.

As for LTL_f, the semantics of LDL_f is given in terms of *finite traces* denoting a finite, possibly empty, sequence of consecutive steps in the trace, i.e., finite words π over the alphabet of $2^{\mathcal{P}}$, containing all possible propositional interpretations of the propositional symbols in $\mathcal{P}$. We denote by n the length of the trace, and by $\pi(i)$ the i-th step in the trace. If $i > n$, then $\pi(i)$ is undefined. We denote by $\pi(i, j)$ the segment of the trace π

starting at i-th step end ending at the j-th step (included). If i or j are out of range wrt the trace then $\pi(i,j)$ is undefined, except $\pi(i,i) = \epsilon$ (i.e., the empty trace).

The semantics of LDL_f is as follows: an LDL_f formula φ *is true* at a step i, in symbols $\pi, i \models \varphi$, as follows:

- $\pi, i \models tt$
- $\pi, i \not\models ff$
- $\pi, i \models \phi$ iff $1 \leq i \leq n$ and $\pi(i) \models \phi$ (ϕ propositional).
- $\pi, i \models \neg\varphi$ 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 we have $\pi(i,j) \in \mathcal{L}(\rho)$ and $\pi, j \models \varphi$.
- $\pi, i \models [\rho]\varphi$ iff for all j such that $\pi(i,j) \in \mathcal{L}(\rho)$ we have $\pi, j \models \varphi$.

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

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

Observe that for $i > n$, hence e.g., for $\pi = \epsilon$ we get:

- $\pi, i \models tt$
- $\pi, i \not\models ff$
- $\pi, i \not\models \phi$ (ϕ propositional).
- $\pi, i \models \neg\varphi$ 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 $\pi(i,i) \in \mathcal{L}(\rho)$ and $\pi, i \models \varphi$.
- $\pi, i \models [\rho]\varphi$ iff $\pi(i,i) \in \mathcal{L}(\rho)$ implies $\pi, i \models \varphi$.

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

- $\pi(i,i) \notin \mathcal{L}(\phi)$ (ϕ propositional)
- $\pi(i,i) \in \mathcal{L}(\varphi?)$ if $\pi, i \models \varphi$
- $\pi(i,i) \in \mathcal{L}(\rho_1 + \rho_2)$ if $\pi(i,i) \in \mathcal{L}(\rho_1)$ or $\pi(i,i) \in \mathcal{L}(\rho_2)$
- $\pi(i,i) \in \mathcal{L}(\rho_1; \rho_2)$ if $\pi(i,i) \in \mathcal{L}(\rho_1)$ and $\pi(i,i) \in \mathcal{L}(\rho_2)$
- $\pi(i,i) \in \mathcal{L}(\rho^*)$

Notice we have the usual boolean equivalences such as $\varphi_1 \vee \varphi_2 \equiv \neg\varphi_1 \wedge \neg\varphi_2$, furthermore we have that: $\phi \equiv \langle\phi\rangle tt$, and $[\rho]\varphi \equiv \neg\langle\rho\rangle\neg\varphi$. It is also convenient to introduce the following abbreviations:

- $end = [true?]ff$ that denotes that the traces is been completed (the remaining trace is ϵ the empty one)
- $last = \langle true\rangle end$, which denotes the last step of the trace.

It easy to encode LTL_f into LDL_f: it suffice to observe that we can express the various LTL_f operators by recursively applying the following translations:

- $\bigcirc\varphi$ translates to $\langle true\rangle\varphi$;
- $\bullet\varphi$ translates to $\neg\langle true\rangle\neg\varphi = [true]\varphi$ (notice that $\bullet a$ is translated into $[true][\neg a]ff$, since a is equivalent to $\langle a\rangle tt$);
- $\Diamond\varphi$ translates to $\langle true^*\rangle\varphi$;
- $\Box\varphi$ translates to $[true^*]\varphi$ (notice that $\Box a$ is translated into $[true^*][\neg a]ff$);
- $\varphi_1 \,\mathcal{U}\, \varphi_2$ translates to $\langle(\varphi_1?; true)^*\rangle\varphi_2$.

It is also easy to encode regular expressions, used as a specification formalism for traces into LDL_f: ρ translates to $\langle\rho\rangle end$.

We say that a trace satisfies an LTL_f or LDL_f formula φ, written $\pi \models \varphi$ if $\pi, 1 \models \varphi$. (Note that if π is the empty trace, and hence 1 is out of range, still the notion of $\pi, 1 \models \varphi$ is well defined). Also sometimes we denote by $\mathcal{L}(\varphi)$ the set of traces that satisfy φ: $\mathcal{L}(\varphi) = \{\pi \mid \pi \models \varphi\}$.

3 LDL_f Automaton

We can associate with each LDL_f formula φ an NFA A_φ (exponential in the size of the formula) that accepts exactly those traces that make φ true. Here, we provide a simple direct algorithm for computing the NFA corresponding to an LDL_f formula. The correctness of the algorithm is based on the fact that (*i*) we can associate each LDL_f formula φ with a polynomial *alternating automaton on words* (AFW) $\mathcal{A}_\varphi$ which accepts exactly the traces that make φ true [4], and (*ii*) every AFW can be transformed into an NFA, see, e.g., [4]. However, to formulate the algorithm we do not need these notions, but we can work directly on the LDL_f formula. In order to proceed with the construction of the AFW $\mathcal{A}_\varphi$, we put LDL_f formulas φ in negation normal form $nnf(\varphi)$ by exploiting equivalences and pushing negation inside as much as possible, until is eliminated except in propositional formulas. Note that computing $nnf(\varphi)$ can be done in linear time. In other words, wlog, we consider as syntax for LDL_f the one in the previous section but without negation. Then we define an auxiliary function δ that takes an LDL_f formula ψ (in negation normal form) and a propositional interpretation Π for $\mathcal{P}$ (including $last$), or a special symbol ϵ, returning a positive boolean formula whose atoms are (quoted) ψ subformulas.

$$
\begin{aligned}
\delta(\texttt{"}tt\texttt{"}, \Pi) &= true \\
\delta(\texttt{"}ff\texttt{"}, \Pi) &= false \\
\delta(\texttt{"}\phi\texttt{"}, \Pi) &= \begin{cases} true \text{ if } \Pi \models \phi \\ false \text{ if } \Pi \not\models \phi \end{cases} \quad (\phi \text{ propositional}) \\
\delta(\texttt{"}\varphi_1 \wedge \varphi_2\texttt{"}, \Pi) &= \delta(\texttt{"}\varphi_1\texttt{"}, \Pi) \wedge \delta(\texttt{"}\varphi_2\texttt{"}, \Pi) \\
\delta(\texttt{"}\varphi_1 \vee \varphi_2\texttt{"}, \Pi) &= \delta(\texttt{"}\varphi_1\texttt{"}, \Pi) \vee \delta(\texttt{"}\varphi_2\texttt{"}, \Pi) \\
\delta(\texttt{"}\langle\phi\rangle\varphi\texttt{"}, \Pi) &= \begin{cases} \texttt{"}\varphi\texttt{"} \text{ if } last \notin \Pi \text{ and } \Pi \models \phi \quad (\phi \text{ propositional}) \\ \delta(\texttt{"}\varphi\texttt{"}, \epsilon) \text{ if } last \in \Pi \text{ and } \Pi \models \phi \\ false \text{ if } \Pi \not\models \phi \end{cases} \\
\delta(\texttt{"}\langle\psi?\rangle\varphi\texttt{"}, \Pi) &= \delta(\texttt{"}\psi\texttt{"}, \Pi) \wedge \delta(\texttt{"}\varphi\texttt{"}, \Pi) \\
\delta(\texttt{"}\langle\rho_1 + \rho_2\rangle\varphi\texttt{"}, \Pi) &= \delta(\texttt{"}\langle\rho_1\rangle\varphi\texttt{"}, \Pi) \vee \delta(\texttt{"}\langle\rho_2\rangle\varphi\texttt{"}, \Pi) \\
\delta(\texttt{"}\langle\rho_1; \rho_2\rangle\varphi\texttt{"}, \Pi) &= \delta(\texttt{"}\langle\rho_1\rangle\langle\rho_2\rangle\varphi\texttt{"}, \Pi) \\
\delta(\texttt{"}\langle\rho^*\rangle\varphi\texttt{"}, \Pi) &= \begin{cases} \delta(\texttt{"}\varphi\texttt{"}, \Pi) & \text{if } \rho \text{ is test-only} \\ \delta(\texttt{"}\varphi\texttt{"}, \Pi) \vee \delta(\texttt{"}\langle\rho\rangle\langle\rho^*\rangle\varphi\texttt{"}, \Pi) & \text{o/w} \end{cases} \\
\delta(\texttt{"}[\phi]\varphi\texttt{"}, \Pi) &= \begin{cases} \texttt{"}\varphi\texttt{"} \text{ if } last \notin \Pi \text{ and } \Pi \models \phi \quad (\phi \text{ propositional}) \\ \delta(\texttt{"}\varphi\texttt{"}, \epsilon) \text{ if } last \in \Pi \text{ and } \Pi \models \phi \quad (\phi \text{ propositional}) \\ true \text{ if } \Pi \not\models \phi \end{cases} \\
\delta(\texttt{"}[\psi?]\varphi\texttt{"}, \Pi) &= \delta(\texttt{"}nnf(\neg\psi)\texttt{"}, \Pi) \vee \delta(\texttt{"}\varphi\texttt{"}, \Pi)
\end{aligned}
$$

1: **algorithm** LDL$_f$2NFA()
2: **input** LTL$_f$ formula φ
3: **output** NFA $A_\varphi = (2^{\mathcal{P}}, \mathcal{S}, \{s_0\}, \varrho, \{s_f\})$
4: $s_0 \leftarrow \{\text{"}\varphi\text{"}\}$ ▷ single initial state
5: $s_f \leftarrow \emptyset$ ▷ single final state
6: $\mathcal{S} \leftarrow \{s_0, s_f\}$, $\varrho \leftarrow \emptyset$
7: **while** ($\mathcal{S}$ or ϱ change) **do**
8: **if** ($q \in \mathcal{S}$ and $q' \models \bigwedge_{(\text{"}\psi\text{"} \in q)} \delta(\text{"}\psi\text{"}, \Theta)$) **then**
9: $\mathcal{S} \leftarrow \mathcal{S} \cup \{q'\}$ ▷ update set of states
10: $\varrho \leftarrow \varrho \cup \{(q, \Theta, q')\}$ ▷ update transition relation

Fig. 1. NFA construction

$$\delta(\text{"}[\rho_1 + \rho_2]\varphi\text{"}, \Pi) = \delta(\text{"}[\rho_1]\varphi\text{"}, \Pi) \wedge \delta(\text{"}[\rho_2]\varphi\text{"}, \Pi)$$
$$\delta(\text{"}[\rho_1; \rho_2]\varphi\text{"}, \Pi) = \delta(\text{"}[\rho_1][\rho_2]\varphi\text{"}, \Pi)$$
$$\delta(\text{"}[\rho^*]\varphi\text{"}, \Pi) = \begin{cases} \delta(\text{"}\varphi\text{"}, \Pi) & \text{if } \rho \text{ is test-only} \\ \delta(\text{"}\varphi\text{"}, \Pi) \wedge \delta(\text{"}[\rho][\rho^*]\varphi\text{"}, \Pi) & \text{o/w} \end{cases}$$

where $\delta(\text{"}\varphi\text{"}, \epsilon)$, i.e., the interpretation of LDL$_f$ formula in the case the (remaining fragment of the) trace is empty, is defined as follows:

$$\begin{aligned}
\delta(\text{"}tt\text{"}, \epsilon) &= \mathit{true} \\
\delta(\text{"}ff\text{"}, \epsilon) &= \mathit{false} \\
\delta(\text{"}\phi\text{"}, \epsilon) &= \mathit{false} \quad (\phi \text{ propositional}) \\
\delta(\text{"}\varphi_1 \wedge \varphi_2\text{"}, \epsilon) &= \delta(\text{"}\varphi_1\text{"}, \epsilon) \wedge \delta(\text{"}\varphi_2\text{"}, \epsilon) \\
\delta(\text{"}\varphi_1 \vee \varphi_2\text{"}, \epsilon) &= \delta(\text{"}\varphi_1\text{"}, \epsilon) \vee \delta(\text{"}\varphi_2\text{"}, \epsilon) \\
\delta(\text{"}\langle\phi\rangle\varphi\text{"}, \epsilon) &= \mathit{false} \quad (\phi \text{ propositional}) \\
\delta(\text{"}\langle\psi?\rangle\varphi\text{"}, \epsilon) &= \delta(\text{"}\psi\text{"}, \epsilon) \wedge \delta(\text{"}\varphi\text{"}, \epsilon) \\
\delta(\text{"}\langle\rho_1 + \rho_2\rangle\varphi\text{"}, \epsilon) &= \delta(\text{"}\langle\rho_1\rangle\varphi\text{"}, \epsilon) \vee \delta(\text{"}\langle\rho_2\rangle\varphi\text{"}, \epsilon) \\
\delta(\text{"}\langle\rho_1; \rho_2\rangle\varphi\text{"}, \epsilon) &= \delta(\text{"}\langle\rho_1\rangle\langle\rho_2\rangle\varphi\text{"}, \epsilon) \\
\delta(\text{"}\langle\rho^*\rangle\varphi\text{"}, \epsilon) &= \delta(\text{"}\varphi\text{"}, \epsilon) \\
\delta(\text{"}[\phi]\varphi\text{"}, \epsilon) &= \mathit{true} \quad (\phi \text{ propositional}) \\
\delta(\text{"}[\psi?]\varphi\text{"}, \epsilon) &= \delta(\text{"}\mathit{nnf}(\neg\psi)\text{"}, \epsilon) \vee \delta(\text{"}\varphi\text{"}, \epsilon) \\
\delta(\text{"}[\rho_1 + \rho_2]\varphi\text{"}, \epsilon) &= \delta(\text{"}[\rho_1]\varphi\text{"}, \epsilon) \wedge \delta(\text{"}[\rho_2]\varphi\text{"}, \epsilon) \\
\delta(\text{"}[\rho_1; \rho_2]\varphi\text{"}, \epsilon) &= \delta(\text{"}[\rho_1][\rho_2]\varphi\text{"}, \epsilon) \\
\delta(\text{"}[\rho^*]\varphi\text{"}, \epsilon) &= \delta(\text{"}\varphi\text{"}, \epsilon)
\end{aligned}$$

Notice also that for ϕ propositional, $\delta(\text{"}\phi\text{"}, \Pi) = \delta(\text{"}\langle\phi\rangle tt\text{"}, \Pi)$ and $\delta(\text{"}\phi\text{"}, \epsilon) = \delta(\text{"}\langle\phi\rangle tt\text{"}, \epsilon)$, as a consequence of the equivalence $\phi \equiv \langle\phi\rangle tt$.
Using the auxiliary function δ we can build the NFA A_φ of an LDL$_f$ formula φ in a forward fashion as described in Figure 1), where: states of A_φ are sets of atoms (recall that each atom is quoted φ subformulas) to be interpreted as a conjunction; the empty conjunction $\emptyset$ stands for true; Θ is either a propositional interpretation Π over $\mathcal{P}$ or the empty trace ϵ (this gives rise to epsilon transition either to true or

false) and q' is a set of quoted subformulas of φ that denotes a minimal interpretation such that $q' \models \bigwedge_{(\texttt{"}\psi\texttt{"} \in q)} \delta(\texttt{"}\psi\texttt{"}, \Theta)$. (Note: we do not need to get all q such that $q' \models \bigwedge_{(\texttt{"}\psi\texttt{"} \in q)} \delta(\texttt{"}\psi\texttt{"}, \Theta)$, but only the minimal ones.) Notice that trivially we have $(\emptyset, a, \emptyset) \in \varrho$ for every $a \in \Sigma$.

The algorithm LDL$_f$2NFA terminates in at most exponential number of steps, and generates a set of states $\mathcal{S}$ whose size is at most exponential in the size of φ.

Theorem 1. *Let φ be an* LDL$_f$ *formula and A_φ the* NFA *constructed as above. Then $\pi \models \varphi$ iff $\pi \in \mathcal{L}(A_\varphi)$ for every finite trace π.*

Proof (sketch). Given a LDL$_f$ formula φ, δ grounded on the subformulas of φ becomes the transition function of the AFW, with initial state $\texttt{"}\varphi\texttt{"}$ and no final states, corresponding to φ [4]. Then LDL$_f$2NFA essentially transforms the AFW into a NFA. □

Notice that above we have assumed to have a special proposition $last \in \mathcal{P}$. If we want to remove such an assumption, we can easily transform the obtained automaton $A_\varphi = (2^{\mathcal{P}}, \mathcal{S}, \{\texttt{"}\varphi\texttt{"}\}, \varrho, \{\emptyset\})$ into the new automaton

$$A'_\varphi = (2^{\mathcal{P}-\{last\}}, \mathcal{S} \cup \{\mathit{ended}\}, \{\texttt{"}\varphi\texttt{"}\}, \varrho', \{\emptyset, \mathit{ended}\})$$

where: $(q, \Pi', q') \in \varrho'$ iff $(q, \Pi', q') \in \varrho$, or $(q, \Pi' \cup \{last\}, true) \in \varrho$ and $q' = \mathit{ended}$.

It is easy to see that the NFA obtained can be built on-the-fly while checking for nonemptiness, hence we have:

Theorem 2. *Satisfiability of an* LDL$_f$ *formula can be checked in PSPACE by nonemptiness of A_φ (or A'_φ).*

Considering that it is known that satisfiability in LDL$_f$ is a PSPACE-complete problem, we can conclude that the proposed construction is optimal wrt computational complexity for satisfiability, as well as for validity and logical implication which are linearly reducible to satisfiability in LDL$_f$ (see [4] for details).

4 Run-time Monitoring

From an high-level perspective, the monitoring problem amounts to observe an evolving system execution and to report the violation or satisfaction of properties of interest at the earliest possible time. As the system progresses, its execution trace increases, and at each step the monitor checks whether the trace seen so far conforms to the properties, by considering that the execution can still continue. 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.

Several variant of monitoring semantics have been proposed (see [2] for a survey). In this paper we adopt the semantics in [11], which is basically the finite-trace variant of the RV semantics in [2]: given a LTL$_f$ or LDL$_f$ formula φ, when the system evolves, the monitor returns one among the following truth values:

- $[\varphi]_{RV} = \mathit{temp\_true}$, meaning that the current execution trace *temporarily satisfies* φ, i.e., it is currently compliant with φ, but a possible system future prosecution may lead to falsify φ;

- $[\varphi]_{RV} = temp\_false$, meaning that the current trace *temporarily falsify* φ, i.e., φ is not current compliant with φ, but a possible system future prosecution may lead to satisfy φ;
- $[\varphi]_{RV} = true$, meaning that the current trace *satisfies* φ and it will always do, no matter how it proceeds;
- $[\varphi]_{RV} = false$, meaning that the current trace *falsifies* φ and it will always do, no matter how it proceeds.

The first two conditions are unstable because they may change into any other value as the system progresses. This reflects the general unpredictability of system possible executions. Conversely, the other two truth values are stable since, once outputted, they will not change anymore. Observe that a stable truth value 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 LDL_f semantics. In the second case, instead, it is irrelevant whether the systems continues its execution or not, since some 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 value is outputted, the monitoring analysis can be stopped.

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

$$\mathcal{L}_{poss\_good}(\varphi) = \{\pi \mid \exists \pi'.\pi\pi' \in \mathcal{L}(\varphi)\} \tag{1}$$

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

$$\mathcal{L}_{nec\_good}(\varphi) = \{\pi \mid \forall \pi'.\pi\pi' \in \mathcal{L}(\varphi)\}. \tag{2}$$

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

$$\mathcal{L}_{nec\_bad}(\varphi) = \{\pi \mid \forall \pi'.\pi\pi' \notin \mathcal{L}(\varphi)\}. \tag{3}$$

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

Using this language theoretic notions, we can provide a precise characterization of the semantics four standard monitoring evaluation functions [11].

Proposition 1. *Let φ be an LDL_f formula and π a trace. Then:*
- $\pi \models [\varphi]_{RV} = temp\_true$ *iff* $\pi \in \mathcal{L}(\varphi) \setminus \mathcal{L}_{nec\_good}(\varphi)$*;*
- $\pi \models [\varphi]_{RV} = temp\_false$ *iff* $\pi \in \mathcal{L}(\neg\varphi) \setminus \mathcal{L}_{nec\_bad}(\varphi)$*;*
- $\pi \models [\varphi]_{RV} = true$ *iff* $\pi \in \mathcal{L}_{nec\_good}(\varphi)$*;*
- $\pi \models [\varphi]_{RV} = false$ *iff* $\pi \in \mathcal{L}_{nec\_bad}(\varphi)$.

Proof (sketch). Immediate from the definitions in [11] and the language theoretic definitions above. □

We close this section by exploiting the language theoretic notions to better understand the relationships between the various kinds of prefixes. We start by observing that, the set of all finite words over the alphabet $2^{\mathcal{P}}$ is the union of the language of φ and its complement $\mathcal{L}(\varphi) \cup \mathcal{L}(\neg\varphi) = (2^{\mathcal{P}})^*$. 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 $\mathcal{L}(\varphi) \subseteq \mathcal{L}_{poss\_good}(\varphi)$ and $\mathcal{L}(\neg\varphi) \subseteq \mathcal{L}_{poss\_good}(\neg\varphi)$, we also have that $\mathcal{L}_{poss\_good}(\varphi) \cup \mathcal{L}_{poss\_good}(\neg\varphi) = (2^{\mathcal{P}})^*$. Also from the definition it is easy to see that $\mathcal{L}_{poss\_good}(\varphi) \cap \mathcal{L}_{poss\_good}(\neg\varphi) = \{\pi \mid \exists\pi'.\pi\pi' \in \mathcal{L}(\varphi) \wedge \exists\pi''.\pi\pi'' \in \mathcal{L}(\neg\varphi)\}$ meaning that the set of possibly good prefixes for φ and the set of possibly good prefixes for $\neg\varphi$ do intersect, and in such an intersection are paths that can be extended to satisfy φ but can also be extended to satisfy $\neg\varphi$. It is also easy to see that $\mathcal{L}(\varphi) = \mathcal{L}_{poss\_good}(\varphi) \setminus \mathcal{L}(\neg\varphi)$.

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

Interestingly, necessarily good, necessarily bad, possibly good prefixes partition all finite traces. Namely

Proposition 2. *The set of all traces $(2^{\mathcal{P}})^*$ can be partitioned into*

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

$$\textit{such that } \mathcal{L}_{nec\_good}(\varphi) \cup (\mathcal{L}_{poss\_good}(\varphi) \cap \mathcal{L}_{poss\_good}(\neg\varphi)) \cup \mathcal{L}_{nec\_bad}(\varphi) = (2^{\mathcal{P}})^*$$
$$\mathcal{L}_{nec\_good}(\varphi) \cap (\mathcal{L}_{poss\_good}(\varphi) \cap \mathcal{L}_{poss\_good}(\neg\varphi)) \cap \mathcal{L}_{nec\_bad}(\varphi) = \emptyset.$$

Proof (sketch). Follows from the definitions of the necessarily good, necessarily bad, possibly good prefixes of $\mathcal{L}(\varphi)$ and $\mathcal{L}(\neg\varphi)$. □

5 Runtime Monitors in LDL_f

As discussed in the previous section the core issue in monitoring is prefix recognition. LTL_f is not expressive enough to talk about prefixes of its own formulas. Roughly speaking, given a LTL_f formula, the language of its possibly good prefixes cannot be in general described as an LTL_f formula. For such a reason, building a monitor usually requires direct manipulation of the automaton for the formula.

LDL_f instead can capture any nondeterministic automata 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 formulas representing (the language of) prefixes of other formulas, and then we exploit them in the context of monitoring.

More precisely, given an LDL_f formula φ, it is possible to express the language $\mathcal{L}_{possgood}(\varphi)$ with an LDL_f formula φ'. Such a formula is obtained in two steps.

Lemma 1. *Given a LDL_f formula φ, there exists a regular expression pref_φ such that $\mathcal{L}(\mathsf{pref}_\varphi) = \mathcal{L}_{poss\_good}(\varphi)$.*

Proof (sketch). The proof is constructive. We can build the NFA A for φ following the procedure described in Section 3. We then set as final all states of A from which there exists a path to a final state. This new finite state machine $A_{poss\_good}(\varphi)$ is such that $\mathcal{L}(A_{poss\_good}(\varphi)) = \mathcal{L}_{poss\_good}(\varphi)$. Since NFA are exactly as expressive as regular expressions, we can translate $A_{poss\_good}(\varphi)$ to a regular expression pref_{φ}. □

Given that LDL$_f$ is as expressive as regular expression (cf. [4]), we can translate pref_{φ} into an equivalent LDL$_f$ formula, as the following states.

Theorem 3. *Given a* LDL$_f$ *formula* φ*,*

$$\pi \in \mathcal{L}_{poss\_good}(\varphi) \textit{ iff } \pi \models \langle \mathsf{pref}_{\varphi} \rangle end$$
$$\pi \in \mathcal{L}_{nec\_good}(\varphi) \textit{ iff } \pi \models \langle \mathsf{pref}_{\varphi} \rangle end \wedge \neg \langle \mathsf{pref}_{\neg\varphi} \rangle end$$

Proof (sketch). Any regular expression ρ, and hence any regular language, can be captured in LDL$_f$ as $\langle \rho \rangle end$. Hence the language $\mathcal{L}_{nec\_good}(\varphi)$ can be captured by $\langle \mathsf{pref}_{\varphi} \rangle end$ and the language $\mathcal{L}_{nec\_good}(\varphi)$ which is equivalent $\mathcal{L}_{poss\_good}(\varphi) \setminus \mathcal{L}_{poss\_good}(\neg\varphi)$ can be captured by captured by $\langle \mathsf{pref}_{\varphi} \rangle end \wedge \neg \langle \mathsf{pref}_{\neg\varphi} \rangle end$. □

In other words, given a LDL$_f$ formula φ, formula $\varphi' = \langle \mathsf{pref}_{\varphi} \rangle end$ is a LDL$_f$ formula such that $\mathcal{L}(\varphi') = \mathcal{L}_{possgood}(\varphi)$. Similarly for $\mathcal{L}_{nec\_good}(\varphi)$.

Exploiting this result, and the results in Proposition 1, we reduce runtime monitoring to the standard evaluation of LDL$_f$ formulas over a (partial) trace. Formally:

Theorem 4. *Let* π *be a (typically partial) trace. The following equivalences hold:*

- $\pi \models [\varphi]_{RV} = \mathit{temp\_true}$ *iff* $\pi \models \varphi \wedge \langle \mathsf{pref}_{\neg\varphi} \rangle end$*;*
- $\pi \models [\varphi]_{RV} = \mathit{temp\_false}$ *iff* $\pi \models \neg\varphi \wedge \langle \mathsf{pref}_{\varphi} \rangle end$*;*
- $\pi \models [\varphi]_{RV} = \mathit{true}$ *iff* $\langle \mathsf{pref}_{\varphi} \rangle end \wedge \neg \langle \mathsf{pref}_{\neg\varphi} \rangle end$*;*
- $\pi \models [\varphi]_{RV} = \mathit{false}$ *iff* $\langle \mathsf{pref}_{\neg\varphi} \rangle end \wedge \neg \langle \mathsf{pref}_{\varphi} \rangle end$*.*

Proof (sketch). Follows from Proposition 1 and Theorem 3 using the language theoretic equivalences discussed in Secton 4. □

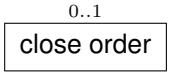
6 Monitoring Declare Constraints and Metaconstraints

We now ground our monitoring approach to the case of DECLARE monitoring. DECLARE[4] is a language and framework for the declarative, constraint-based modelling of processes and services. A thorough treatment of constraint-based processes can be found in [17, 14]. As a modelling language, DECLARE takes a complementary approach to that of classical, imperative process modeling, in which all allowed control-flows among tasks must be explicitly represented, and every other execution trace is 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 how to perform the involved tasks, provided that the resulting execution trace complies with the modeled business constraints. This is the reason why, alongside traditional control-flow constraints such as sequence (called in DECLARE *chain succession*), DECLARE supports a plethora of peculiar constraints that do not impose specific temporal orderings, or that explicitly account with negative information, i.e., prohibition of task execution.

[4] `http://www.win.tue.nl/declare/`

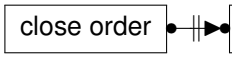
Given a set $\mathcal{P}$ of tasks, a DECLARE model is a set $\mathcal{C}$ of LTL_f (and hence LDL_f) constraints over $\mathcal{P}$, used to restrict the allowed execution traces. Among all possible LTL_f constraints, some specific *patterns* have been singled out as particularly meaningful for expressing DECLARE processes, taking inspiration from [6]. Such patterns are grouped into four families: *(i) existence* (unary) constraints, stating that the target task must/cannot be executed (a certain amount of times); *(ii) choice* (binary) constraints, modeling choice of execution; *(iii) relation* (binary) constraints, modeling that whenever the source task is executed, then the target task must also be executed (possibly with additional requirements); *(iv) negation* (binary) constraints, modeling that whenever the source task is executed, then the target task is prohibited (possibly with additional restrictions). Table 1 reports some of these patterns.

Example 1. Consider a fragment of a purchase order process, where we consider three key business constraints. First, an order can be closed at most once. In DECLARE, this can be tackled with a `absence 2` constraint, visually and formally represented as:



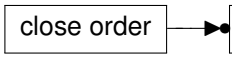
$$\varphi_{close} = \neg\Diamond(\text{close order} \wedge \bigcirc\Diamond\text{close order})$$

Second, an order can be canceled only until it is closed. This can be captured by a `negation succession` constraint, which states that after the order is closed, it cannot be canceled anymore:



$$\varphi_{canc} = \Box(\text{close order} \rightarrow \neg\Diamond\text{cancel order})$$

Finally, after the order is closed, it becomes possible to do supplementary payments, for various reasons (e.g., to speed up the delivery of the order).



$$\varphi_{pay} = (\neg\text{pay suppl}\,\mathcal{U}\,\text{close order}) \vee \neg\Diamond\text{close order}$$

Beside modeling and enactment of constraint-based processes, previous works have also focused on runtime verification of DECLARE models. A family of DECLARE monitoring approaches rely on the original LTL_f formalization of DECLARE, and employ corresponding automata-based techniques to track running process instances and check whether they satisfy the modeled constraints or not [11, 12]. Such techniques have been in particular used for:

- monitoring single DECLARE constraints so as to provide a fine-grained feedback; this is done by adopting the RV semantics for LTL_f, and tracking the evolution each constraint through the four RV truth values.
- Monitoring the global DECLARE model by considering all its constraints together (i.e., constructing a DFA for the conjunction of all constraints); this is important for computing the *early detection* of violations, i.e., violations that cannot be explicitly found in the execution trace collected so far, but that cannot be avoided in the future.

We now discuss how LDL_f can be adopted for monitoring DECLARE constraints, with a twofold advantage. First, as shown in Section 5, LDL_f is able to encode the RV semantics directly into the logic, without the need of introducing ad-hoc modifications in the corresponding standard logical services. Second, beside being able to reconstruct all the aforementioned monitoring techniques, our approach also provides a declarative, well-founded basis for monitoring metaconstraints, i.e., constraints that involve both the execution of tasks and the monitoring outcome obtained by checking other constraints.

Monitoring Declare Constraints with LDL_f**.** Since LDL_f includes LTL_f, DECLARE constraints can be directly encoded in LDL_f using their standard formalization [18, 16]. Thanks to the translation into NFAs discussed in Section 3 (and, if needed, their determinization into corresponding DFAs), the obtained automaton can then be used to check whether a (partial) finite trace satisfies this constraint or not. This is not very effective, as the approach does not support the detection of fine-grained truth values as those of RV. By relying on Theorem 4, however, we can reuse the same technique, this time supporting all RV. In fact, by formalizing the good prefixes of each DECLARE pattern, we can immediately construct the four LDL_f formulas that embed the different RV truth values, and check the current trace over each of the corresponding automata. Table 1 reports the good prefix characterization of some of the DECLARE patterns; it can be seamlessly extended to all other patterns as well.

Example 2. Let us consider the `absence 2` constraint φ_{close} in Example 1. Following Table 1, its good prefix characterization is $\mathsf{pref}_{\varphi_{close}} = o^* + (o^*; \mathsf{close\ order}; o^*)$, where o is a shortcut for all the tasks involved in the purchase order process but close order. This can be used to construct the four formulas mentioned in Theorem 4, which in turn provide the basis to produce, e.g., the following result:

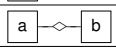
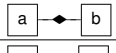
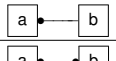
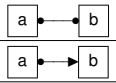
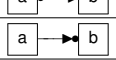
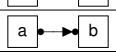
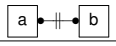
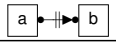
	start	do "close order"	do "pay suppl."	do "close order"
0..1 close order	$temp\_true$			$false$

Observe that this baseline approach can be extended along a number of directions. For example, as shown in Table 1, the majority of DECLARE patterns does not cover all the four RV truth values. This is the case, e.g., of `absence 2`, which can never be evaluated to $true$ (since it is always possible to continue the execution so as to perform a twice), nor to $temp\_false$ (the only way of violating the constraint is to perform a twice, and in this case it is not possible to "repair" to the violation anymore). This information can be used to restrict the generation of the automata only to those cases that are relevant to the constraint. Furthermore, it is possible to reconstruct exactly the approach in [11], where every state in the DFAs corresponding to the constraints to be monitored, is enriched with a "color" accounting for one of the four RV truth values. To do so, we have simply to combine the four DFAs generated for each constraint. This is possible because such DFAs are generated from formulas built on top of the good prefix characterization of the original formula, and hence they all produce the same automaton, but with different final states. In fact, this observation provides a formal justification to the correctness of the approach in [11].

Metaconstraints. Thanks to the ability of LDL_f to directly encode into the logic DECLARE constraints but also their RV monitoring states, we can formalize metaconstraints that relate the RV truth values of different constraints. Intuitively, such metaconstraints allow one to capture that *we become interested in monitoring some constraint only when other constraints are evaluated to be in a certain RV truth value*. This, in turn, provides the basis to declaratively capture two classes of properties that are of central importance in the context of runtime verification:

- *Compensation constraints*, that is, constraints that should be enforced by the agents executing the process in the case other constraints are violated, i.e., are evaluated to be $false$. Previous works have been tackled this issue through ad-hoc techniques, with no declarative counterpart [11, 12].

Table 1. Some DECLARE constraints, together with their prefix characterization, minimal bad prefix charaterization, and possible RV states; for each constraint, *o* is a shortcut for "other tasks", i.e., tasks not involved in the constraint itself.

	NAME	NOTATION	pref	POSSIBLE RV STATES
EXISTENCE	**Existence**		$(a+o)^*$	$temp\_false$, $true$
	Absence 2		$o^* + (o^*; a; o^*)$	$temp\_true$, $false$
CHOICE	**Choice**		$(a+b+o)^*$	$temp\_false$, $true$
	Exclusive Choice		$(a+o)^* + (b+o)^*$	$temp\_false$, $temp\_true$, $false$
RELATION	**Resp. existence**		$(a+b+o)^*$	$temp\_true$, $temp\_false$, $true$
	Coexistence		$(a+b+o)^*$	$temp\_true$, $temp\_false$, $true$
	Response		$(a+b+o)^*$	$temp\_true$, $temp\_false$
	Precedence		$o^*; (a; (a+b+o)^*)^*$	$temp\_true$, $true$, $false$
	Succession		$o^*; (a; (a+b+o)^*)^*$	$temp\_true$, $temp\_false$, $false$
NEGATION	**Not Coexistence**		$(a+o)^* + (b+o)^*$	$temp\_true$, $false$
	Neg. Succession		$(b+o)^*; (a+o)^*$	$temp\_true$, $false$

– Recovery mechanisms resembling *contrary-to-duty obligations* in legal reasoning [20], i.e., obligations that are put in place only when other obligations are not met.

Technically, a generic form for metaconstraints is the pattern $\Phi_{pre} \rightarrow \Psi_{exp}$, where:

– Φ_{pre} is a boolean formula, whose atoms are membership assertions of the involved constraints to the RV truth values;
– Ψ_{exp} is a boolean formula whose atoms are the constraints to be enforced when Φ_{pre} evaluates to true.

This pattern can be used, for example, to state that whenever constraints c_1 and c_2 are permanently violated, then either constraint c_3 or c_4 have to be enforced. Observe that the metaconstraint so constructed is a standard LDL_f formula. Hence, we can reapply Theorem 4 to it, getting four LDL_f formulas that can be used to track the evolution of the metaconstraint among the four RV values.

Example 3. Consider the DECLARE constraints of Example 1. We want to enhance it with a compensation constraint stating that whenever φ_{canc} is violated (i.e., the order is canceled after it has been closed), then a supplement payment must be issued. This can be easily captured in LDL_f as follows. First of all, we model the compensation constraint, which corresponds, in this case, to a standard `existence` constraint over the pay supplement task. Let φ_{dopay} denote the LDL_f formalization of such a compensation constraint. Second, we capture the intended compensation behavior by using the following LDL_f metaconstraint:

$$\{[\varphi_{canc}]_{RV} = \mathit{false}\} \rightarrow \varphi_{dopay}$$

which, leveraging Theorem 4, corresponds to the standard LDL_f formula:

$$(\langle \mathsf{pref}_{\neg\varphi_{canc}} \rangle \mathit{end} \wedge \neg \langle \mathsf{pref}_{\varphi_{canc}} \rangle \mathit{end}) \rightarrow \varphi_{dopay}$$

A limitation of this form of metaconstraint is that the right-hand part Ψ_{exp} is monitored *from the beginning of the trace*. This is acceptable in many cases. E.g., in Example 1, it is ok if the user already paid a supplement before the order cancelation caused constraint φ_{canc} to be violated. In other situations, however, this is not satisfactory, because we would like to enforce the compensating behavior only *after* Φ_{pre} evaluates to true, e.g., after the violation of a given constraint has been detected. In general, we can extend the aforementioned metaconstraint pattern as follows: $\Phi_{pre} \rightarrow [\rho]\Psi_{exp}$, where ρ is a regular expression denoting the paths after which Ψ_{exp} is expected to be enforced.

By constructing ρ as the regular expression accounting for the paths that make Φ_{pre} true, we can then exploit this improved metaconstraint to express that Ψ_{exp} is expected to become true after all prefixes of the current trace that made Φ_{pre} true.

Example 4. We modify the compensation constraint of Example 3, so as to reflect that when a closed order is canceled (i.e., φ_{canc} is violated), then a supplement must be paid *afterwards*. This is captured by the following metaconstraint:

$$\{[\varphi_{canc}]_{RV} = \mathit{false}\} \rightarrow [re_{\{[\varphi_{canc}]_{RV}=\mathit{false}\}}]\varphi_{dopay}$$

where $re_{\{[\varphi_{canc}]=\mathit{false}\}}$ denotes the regular expression for the language $\mathcal{L}(\{[\varphi_{canc}] = \mathit{false}\}) = \mathcal{L}(\langle \mathsf{pref}_{\neg\varphi_{canc}} \rangle \mathit{end} \wedge \neg\langle \mathsf{pref}_{\varphi_{canc}} \rangle \mathit{end})$. This regular expression describes all paths containing a violation for constraint φ_{canc}.

7 Implementation

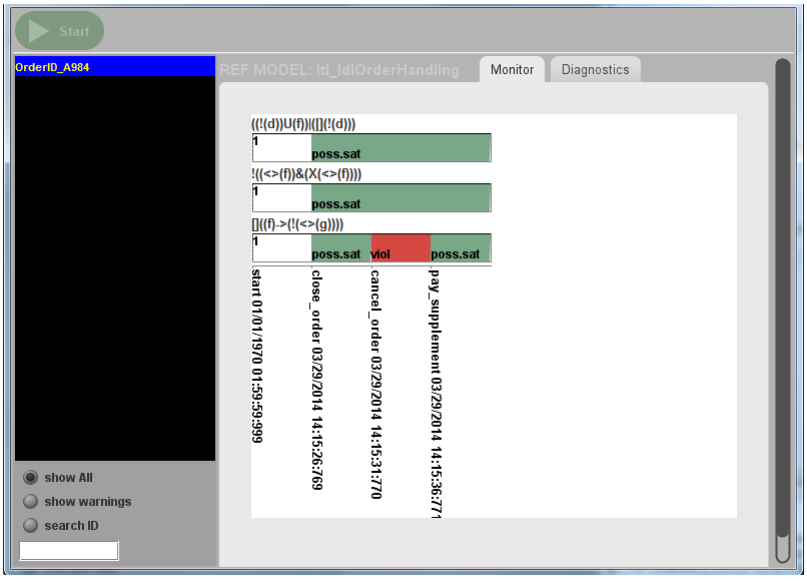


Fig. 2. Screenshot of our operational support provider's output.

The entire approach has been implemented as an *operational decision support (OS) provider* for the ProM 6 process mining framework[5]. ProM 6 provides a generic OS environment [23] that supports the interaction between an external workflow management system at runtime (producing events) and ProM. In particular, it provides an OS service that receives a stream of events from the external world, updates and orchestrates the registered OS providers implementing different types of online analysis to be applied on the stream, and reports the produced results back to the external world.

At the back-end of the plug-in, there is a software module specifically dedicated to the construction and manipulation of NFAs from LDL_f formulas, concretely implementing the technique presented in Section 3. To manipulate regular expressions and automata, we used the fast, well-known library `dk.brics.automaton` [13].

Figure 2 shows a graphical representation of the evolution of constraints described in Example 1 and 4 of Section 6 when monitored using our operational support provider. Events are displayed on the horizontal axis, while the vertical axis shows the three constraints expressed as LDL_f formulas, where the literals f, g and d respectively stand for tasks close order, cancel order, and pay supplement. Note that after the violation of the negation succession constraint a compensation meta-constraint is triggered to enforce that pay supplement is required to occur after the violation.

8 Conclusion

We have proposed an effective approach for monitoring dynamic (business) constraints on finite traces that represent the executions of running process instances. Our contribution can be seen as an extension of the declarative process specification approach at the basis of DECLARE, in which we tackle the monitoring problem with a more powerful temporal logic: LDL_f, i.e., Monadic Second Order logic over finite traces, instead of LTL_f, i.e., First Order logic over finite traces. Notably, this declarative approach to monitoring seamlessly supports the specification and monitoring of metaconstraints, i.e., constraints that do not only predicate about the dynamics of task executions, but also about the truth values of other constraints. We have grounded this approach on DECLARE itself, showing how to declaratively specify compensation constraints.

The next step will be to incorporate recovery mechanisms into the approach, in particular providing a formal underpinning to the ad-hoc recovery mechanisms studied in [11]. Furthermore, we intend to extend our approach to *data-aware* business constraints [1, 5], mixing temporal operators with first-order queries over the data attached to the monitored events. This setting has been studied using the Event Calculus [15], also considering some specific forms of compensation in DECLARE [3]. However, the resulting approach can only query the partial trace accumulated so far, and not reason upon its possible future continuations, as automata-based techniques are able to do. To extend the approach presented here to the case of data-aware business constraints, we will build on recent, interesting decidability results for the static verification of data-aware business processes against sophisticated variants of first-order temporal logics [1].

Acknowledgments. This research has been partially supported by the EU IP project *Optique: Scalable End-user Access to Big Data*, grant agreement n. FP7-318338, and by the Sapienza Award 2013 "SPIRITLETS*: Spiritlet-based smart spaces*".

[5] `http://www.promtools.org/prom6/`

References

1. B. Bagheri Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, and M. Montali. Verification of relational data-centric dynamic systems with external services. In *32nd ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*, 2013.
2. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Logic and Computation*, 2010.
3. F. Chesani, P. Mello, M. Montali, and P. Torroni. Verification of choreographies during execution using the reactive event calculus. In *5th Int. Workshop on Web Services and Formal Methods (WS-FM)*, LNCS. Springer, 2008.
4. G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *23rd Int. Joint Conf. on Artificial Intelligence (IJCAI)*. AAAI, 2013.
5. R. De Masellis, F. M. Maggi, and M. Montali. Monitoring data-aware business constraints with finite state automata. In *Int. Conf. on Software and System Proc. (ICSSP)*. ACM, 2014.
6. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In B. W. Boehm, D. Garlan, and J. Kramer, editors, *Proc. of the 1999 International Conf. on Software Engineering (ICSE)*. ACM Press, 1999.
7. C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout. Reasoning with temporal logic on truncated paths. In *15th Int. Conf. on Computer Aided Verification (CAV)*, LNCS. Springer, 2003.
8. M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Science*, 1979.
9. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
10. L. T. Ly, F. M. Maggi, M. Montali, S. Rinderle-Ma, and W. M. P. van der Aalst. A framework for the systematic comparison and evaluation of compliance monitoring approaches. In *Proc. of the 17th IEEE Int. Enterprise Distributed Object Computing Conf. (EDOC)*. IEEE, 2013.
11. F. M. Maggi, M. Montali, M. Westergaard, and W. M. P. van der Aalst. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In *9th Int. Conf. on Business Process Management (BPM)*, LNCS. Springer, 2011.
12. F. M. Maggi, M. Westergaard, M. Montali, and W. M. P. van der Aalst. Runtime verification of LTL-based declarative process models. In *2nd Int. Conf. on Runtime Verification (RV)*, LNCS. Springer, 2012.
13. A. Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010.
14. M. Montali. *Specification and Verification of Declarative Open Interaction Models: a Logic-Based Approach*. LNBIP. Springer, 2010.
15. M. Montali, F. M. Maggi, F. Chesani, P. Mello, and W. M. P. van der Aalst. Monitoring business constraints with the event calculus. *ACM TIST*, 2013.
16. M. Montali, M. Pesic, W. M. P. van der Aalst, F. Chesani, P. Mello, and S. Storari. Declarative specification and verification of service choreographies. *ACM Trans. on the Web*, 2010.
17. M. Pesic. *Constraint-Based Workflow Management Systems: Shifting Controls to Users*. PhD thesis, Beta Research School for Operations Management and Logistics, Eindhoven, 2008.
18. M. Pesic and W. M. P. van der Aalst. A declarative approach for flexible business processes management. In *Proc. of the BPM 2006 Workshops*, LNCS. Springer, 2006.
19. A. Pnueli. The temporal logic of programs. In *18th Ann. Symp. on Foundations of Computer Science (FOCS)*. IEEE, 1977.
20. H. Prakken and M. J. Sergot. Contrary-to-duty obligations. *Studia Logica*, 1996.
21. W. M. P. van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.
22. M. Vardi. The rise and fall of linear time logic, 2011. 2nd Int. Symp. on Games, Automata, Logics and Formal Verification.
23. M. Westergaard and F. Maggi. Modelling and verification of a protocol for operational support using coloured petri nets. In *32nd Int. Conf. on Appl. and Theory of Petri Nets*, 2011.