LTLf and LDLf monitoring: a technical report

De Giacomo, G.; De Masellis, R.; Grasso, M.; Maggi, F.M.; Montali, M.

Published: 01/01/2014

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Citation for published version (APA):
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 (LTLf) and in its extension LDLf. LDLf is a powerful logic that captures all monadic second order logic on finite traces, which is obtained by combining regular expressions and LTLf, adopting the syntax of propositional dynamic logic (PDL). Interestingly, in spite of its greater expressivity, LDLf has exactly the same computational complexity of LTLf. We show that LDLf 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, and check them by relying on usual logical services 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 LDLf 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 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. In several context, and in particular often in BPM, we can assume that
the trace of the system is if face finite \cite{17}. 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 \cite{4}, and at the base of one of the main declarative process modeling approaches: DECLARE \cite{17,15,9}. Following \cite{9}, 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 finite-state automaton for $\varphi$, which recognizes all those finite executions that satisfy $\varphi$.

Despite the presence of previous operational decision support techniques to monitoring LTL$_f$ constraints over finite traces \cite{9,10}, 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 \cite{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 \cite{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 \cite{9} 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 so-called contrary-to-duty obligations \cite{19}, 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 \cite{8}, 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$_f$ \cite{4}. LDL$_f$ is a powerful logic that captures all monadic second order logic on finite traces, which is obtained by combining regular expressions and 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 monitor conditions. Indeed given an LDL$_f$ formula $\phi$, we show how to construct the LDL$_f$ formulas that captures whether prefixes of $\phi$ satisfy the various RV monitor 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 nondeterministic finite-state automaton (NFA). Notably, such technique has been concretely
implemented, and embedded into a monitoring plug-in for the PROM, which supports checking LDL₁ formulas and metaconstraints.

2 LTL₁ and LDL₁

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

LTL on finite traces, called LTL₁ [4], has exactly the same syntax as LTL on infinite traces [18]. Namely, given a set of \( \mathcal{P} \) of propositional symbols, LTL₁ formulas are obtained through the following:

\[
\varphi ::= \phi \mid \neg \varphi \mid \varphi_1 \land \varphi_2 \mid \varphi_1 \lor \varphi_2 \mid \bigcirc \varphi \mid \bullet \varphi \mid \diamond \varphi \mid \square \varphi \mid \varphi_1 \mathcal{U} \varphi_2
\]

where \( \phi \) is a propositional formula over \( \mathcal{P} \), \( \bigcirc \) is the next operator, \( \bullet \) is weak next, \( \diamond \) is eventually, \( \square \) is always, \( \mathcal{U} \) is until.

It is known that LTL₁ is as expressive as First Order Logic over finite traces, so strictly less expressive than regular expressions 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 these difficulties, in [4] Linear Dynamic Logic of Finite Traces, or LDL₁, has been proposed. This logic is as natural as LTL₁ but with the full expressive power of Monadic Second Order logic over finite traces. LDL₁ is obtained by merging LTL₁ with regular expression through the syntax of the well-know logic of programs PDL, Propositional Dynamic Logic [6,7] but adopting a semantics based on finite traces. This logic is an adaptation of LDL, introduced in [21], which, like LTL₁, is interpreted over infinite traces.

Formally, LDL₁ formulas are built as follows:

\[
\varphi ::= \phi \mid \tt \mid \ff \mid \neg \varphi \mid \varphi_1 \land \varphi_2 \mid \varphi_1 \lor \varphi_2 \mid \langle \rho \rangle \varphi \mid \lfloor \rho \rfloor \varphi
\]

\[
\rho ::= \phi \mid \varphi \mid \rho_1 + \rho_2 \mid \rho_1 \cdot \rho_2 \mid \rho^*
\]

where \( \phi \) is a propositional formula over \( \mathcal{P} \); \( \tt \) and \( \ff \) denote respectively the true and the false LDL₁ formulas (not to be confused with the propositional formula true and false), \( \rho \) denotes path expressions, which are RE₁ expressions over propositional formulas \( \phi \), with the addition of the test construct \( \varphi ? \) typical of PDL; and \( \varphi \) stand for LDL₁ formulas built by applying boolean connectives and the modal connectives \( \langle \rho \rangle \varphi \) and \( \lfloor \rho \rfloor \varphi \). In fact \( \lfloor \rho \rfloor \varphi \equiv \neg \langle \rho \rangle \neg \varphi \).

Intuitively, \( \langle \rho \rangle \varphi \) states that, from the current step in the trace, there exists an execution satisfying the regular expression \( \rho \) such that its last step satisfies \( \varphi \). While \( \lfloor \rho \rfloor \varphi \) states that, from the current step, all executions satisfying the regular expression \( \rho \) are such that their last step satisfies \( \varphi \). Tests are used to insert into the execution path checks for satisfaction of additional LDL₁ formulas.

As for LTL₁, the semantics of LDL₁ is given in terms of finite traces denoting a finite, possibly empty, sequence of consecutive steps in the trace, i.e., finite words \( \pi \) over the alphabet of \( 2^\mathcal{P} \), containing all possible propositional interpretations of the propositional symbols in \( \mathcal{P} \). We denote by \( \pi(i) \) the \( i \)th step in the trace. If the trace is shorter and does
not include an $i$th step $\pi(i)$ is undefined. We denote by $\pi(i, j)$ the segment of the trace $\pi$ starting at $i$th step end ending at the $j$th step (included). If $i$ or $j$ are out of range wrt the trace that $\pi(i, j)$ is undefined, except $\pi(i, i) = \epsilon$ (i.e., the empty trace).

The semantics of $\text{LDL}_f$ is as follows: an $\text{LDL}_f$ formula $\varphi$ is true at a step $i$, in symbols $\pi, i \models \varphi$, as follows:

- $\pi, i \models \top$
- $\pi, i \not\models \bot$
- $\pi, i \models \varphi \iff 1 \leq i \leq n$ and $\pi(i) \models \varphi$ (\(\varphi\) propositional).
- $\pi, i \models \neg \varphi \iff \pi, i \not\models \varphi$.
- $\pi, i \models \varphi_1 \land \varphi_2 \iff \pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$.
- $\pi, i \models \varphi_1 \lor \varphi_2 \iff \pi, i \models \varphi_1$ or $\pi, i \models \varphi_2$.
- $\pi, i \models \langle \rho \rangle \varphi \iff$ for some $i$ we have $\pi(i, j) \in \mathcal{L}(\rho)$ and $\pi, j \models \varphi$.
- $\pi, i \models [\rho] \varphi \iff$ for all $i$ such that $\pi(i, j) \in \mathcal{L}(\rho)$ we have $\pi, j \models \varphi$.

The relation $\pi(i, j) \in \mathcal{L}(\varphi)$ if $j = i + 1 \leq n$ and $\pi(i) \models \varphi$ (\(\varphi\) 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 $\pi(k, j) \in \mathcal{L}(\rho^*)$.

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

- $\text{end} = [\top] \bot$ that denotes that the traces is been completed (the remaining trace is $\epsilon$ the empty one)
- $\text{last} = \langle \text{true} \rangle \text{end}$, which denotes the last step of the trace.

It easy to encode $\text{LTL}_f$ into $\text{LDL}_f$: it suffice to observe that we can express the various $\text{LTL}_f$ operators by recursively applying the following translations:

- $\Diamond \varphi$ translates to $\langle \text{true} \rangle \varphi$;
- $\bullet \varphi$ translates to $\neg \langle \text{true} \rangle \neg \varphi = [\text{true}] \varphi$ (notice that $\bullet a$ is translated into $[\text{true}] [\neg a] \bot$, since $a$ is equivalent to $\langle a \rangle \top$);
- $\Diamond \varphi$ translates to $\langle \text{true}^* \rangle \varphi$;
- $\Box \varphi$ translates to $[\text{true}^*] \varphi$ (notice that $\Box a$ is translated into $[\text{true}^*][\neg a] \bot$);
\[ \varphi_1 \cup \varphi_2 \] translates to \((\varphi_1 \land \text{true})^* \varphi_2\).

It is also easy to encode regular expressions, used as a specification formalism for traces into \(\text{LDL}_f\): \(\rho\) translates to \(\langle \rho \rangle \text{end}\).

We say that a trace satisfies an \(\text{LTL}_f\) or \(\text{LDL}_f\) formula \(\varphi\), written \(\pi \models \varphi\), if \(\pi \models \varphi\) if \(\varphi\) is well defined). Also sometimes we denote by \(L(\varphi)\) the set of traces that satisfy \(\varphi\): 
\[ L(\varphi) = \{ \pi \mid \pi \models \varphi \}. \]

### 3 LDL\(_f\) Automaton

We can associate with each \(\text{LDL}_f\) formula \(\varphi\) an (exponential) NFA \(A_\varphi\) that accepts exactly the traces that make \(\varphi\) true. Here, we provide a simple direct algorithm for computing the NFA corresponding to an \(\text{LDL}_f\) formula. The correctness of the algorithm is based on the fact that (i) we can associate with each \(\text{LDL}_f\) formula \(\varphi\) a polynomial alternating automaton on words (AFW) \(A_\varphi\) which accepts exactly the traces that make \(\varphi\) 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 \(\text{LDL}_f\) formula. In order to proceed with the construction of the AFW \(A_\varphi\), we put \(\text{LDL}_f\) formulas \(\varphi\) in negation normal form \(\text{nnf}(\varphi)\) by exploiting equivalences and pushing negation inside as much as possible, until is eliminated except in propositional formulas. Note that computing \(\text{nnf}(\varphi)\) can be done in linear time. In other words, wlog, we consider as syntax for \(\text{LDL}_f\) the one in the previous section but without negation. Then we define an auxiliary function \(\delta\) that takes an \(\text{LDL}_f\) formula \(\psi\) (in negation normal form) and a propositional interpretation \(\Pi\) for \(P\) (including \text{last},) or a special symbol \(\epsilon\), returning a positive boolean formula whose atoms are (quoted) \(\psi\) subformulas.

\[
\begin{align*}
\delta("tt", \Pi) &= \text{true} \\
\delta("ff", \Pi) &= \text{false} \\
\delta("\phi", \Pi) &= \begin{cases} 
\text{true} & \text{if } \Pi \models \phi \\
\text{false} & \text{if } \Pi \not\models \phi 
\end{cases} \quad \text{(}\phi\text{ propositional)}
\end{align*}
\]

\[
\begin{align*}
\delta("\varphi_1 \land \varphi_2", \Pi) &= \delta("\varphi_1", \Pi) \land \delta("\varphi_2", \Pi) \\
\delta("\varphi_1 \lor \varphi_2", \Pi) &= \delta("\varphi_1", \Pi) \lor \delta("\varphi_2", \Pi)
\end{align*}
\]

\[
\begin{align*}
\delta("\langle \rho_1, \rho_2 \rangle\varphi", \Pi) &= \delta("\langle \rho_1 \rangle\varphi", \Pi) \lor \delta("\langle \rho_2 \rangle\varphi", \Pi) \\
\delta("\langle \rho \rangle\varphi", \Pi) &= \begin{cases} 
\delta("\varphi", \Pi) & \text{if } \rho \text{ is test-only} \\
\delta("\varphi", \Pi) \lor \delta("\langle \rho \rangle\varphi", \Pi) & \text{o/w}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\delta("\langle \rho \rangle\varphi", \Pi) &= \begin{cases} 
\text{true} & \text{if } \Pi \not\models \phi \\
\text{false} & \text{if } \Pi \models \phi 
\end{cases} \quad \text{(}\phi\text{ propositional)}
\end{align*}
\]
1: algorithm LDL_{f}2NFA()
2: input LTL_{f} formula \( \varphi \)
3: output NFA \( A_{\varphi} = (2^{\mathcal{P}}, \mathcal{S}, \{ s_0 \}, q, \{ s_f \}) \)
4: \( s_0 \leftarrow \{ "\varphi^n" \} \quad \text{▷ single initial state} \\
5: s_f \leftarrow \emptyset \quad \text{▷ single final state} \\
6: \mathcal{S} \leftarrow \{ s_0, s_f \}, q \leftarrow \emptyset \\
7: \textbf{while} (\mathcal{S} \text{ or } q \text{ change}) \textbf{do} \\
8: \quad \textbf{if} (q \in \mathcal{S} \text{ and } q' = \bigwedge_{(\psi^n, \Theta) \in q} \delta("\psi^n", \Theta)) \text{ then} \\
9: \quad \quad \mathcal{S} \leftarrow \mathcal{S} \cup \{ q' \} \\
10: \quad q \leftarrow q \cup \{ (q, \Theta, q') \} \quad \text{▷ update transition relation} \\

\textbf{Fig. 1.} NFA construction

\[
\begin{align*}
\delta("\psi?\varphi^n", \Pi) &= \delta("\text{nnf}(\neg\psi)^n", \Pi) \lor \delta("\varphi^n", \Pi) \\
\delta("[\rho_1 + \rho_2]\varphi^n", \Pi) &= \delta("[\rho_1]\varphi^n", \Pi) \land \delta("[\rho_2]\varphi^n", \Pi) \\
\delta("[\rho_1; \rho_2]\varphi^n", \Pi) &= \delta("[\rho_1]\varphi^n", \Pi) \lor \delta("[\rho_2]\varphi^n", \Pi)
\end{align*}
\]

where \( \delta("\varphi^n", \epsilon) \), i.e., the interpretation of LDL_{f} formula in the case the (remaining fragment of the) trace is empty, is defined as follows:

\[
\begin{align*}
\delta("tt^n", \epsilon) &= \text{true} \\
\delta("ff^n", \epsilon) &= \text{false} \quad (\phi \text{ propositional}) \\
\delta("\varphi^n", \epsilon) &= \text{false} \\
\delta("\varphi_1 \land \varphi_2^n", \epsilon) &= \delta("\varphi_1^n", \epsilon) \land \delta("\varphi_2^n", \epsilon) \\
\delta("\varphi_1 \lor \varphi_2^n", \epsilon) &= \delta("\varphi_1^n", \epsilon) \lor \delta("\varphi_2^n", \epsilon) \\
\delta("\langle \phi \rangle\varphi^n", \epsilon) &= \text{false} \quad (\phi \text{ propositional}) \\
\delta("\langle \psi?\varphi^n, \epsilon \rangle = \delta("\psi^n, \epsilon \land \delta("\varphi^n", \epsilon) \\
\delta("[\rho_1 + \rho_2]\varphi^n", \epsilon) &= \delta("(\rho_1]\varphi^n, \epsilon) \lor \delta("(\rho_2]\varphi^n, \epsilon) \\
\delta("[\rho_1; \rho_2]\varphi^n", \epsilon) &= \delta("[\rho_1]\varphi^n, \epsilon) \lor \delta("[\rho_2]\varphi^n, \epsilon)
\end{align*}
\]

Notice also that for \( \phi \) propositional, \( \delta("\langle \phi \rangle\varphi^n", \Pi) = \delta("\langle \phi \rangle tt^n", \Pi) \) and \( \delta("\varphi^n", \epsilon) = \delta("\langle \phi \rangle tt^n", \epsilon) \), as a consequence of the equivalence \( \phi \equiv \langle \phi \rangle tt \).

Using the auxiliary function \( \delta \) we can build the NFA \( A_{\varphi} \) of an LDL_{f} formula \( \varphi \) in a forward fashion as described in Figure 1, where: states of \( A_{\varphi} \) are sets of atoms (recall that each atom is quoted \( \varphi \) subformulas) to be interpreted as a conjunction; the empty conjunction \( \emptyset \) stands for \text{true}; \( \Theta \) is either a propositional interpretation.
Π over \( \mathcal{P} \) or the empty trace \( \epsilon \) (this gives rise to epsilon transition either to true or false) and \( q' \) is a set of quoted subformulas of \( \varphi \) that denotes a minimal interpretation such that \( q' \models \bigwedge_{\psi \in q} \delta(\psi, \Theta) \). (Note: we do not need to get all \( q \) such that \( q' \models \bigwedge_{\psi \in q} \delta(\psi, \Theta) \), but only the minimal ones.) Notice that trivially we have \( (\emptyset, a, \emptyset) \in \varrho \) for every \( a \in \Sigma \).

The algorithm LDLf2NFA terminates in at most exponential number of steps, and generates a set of states \( S \) whose size is at most exponential in the size of \( \varphi \).

**Theorem 1.** Let \( \varphi \) be an LDLf formula and \( A_\varphi \) the NFA constructed as above. Then \( \pi \models \varphi \) iff \( \pi \in L(A_\varphi) \) for every finite trace \( \pi \).

**Proof (sketch).** Given a LDLf formula \( \varphi \), \( \delta \) grounded on the subformulas of \( \varphi \) becomes the transition function of the AFW, with initial state "\( \varphi \)" and no final states, corresponding to \( \varphi \) [4]. Then LDLf2NFA essentially transforms the AFW into a NFA.

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

\[
(q, \Pi', q') \in q' \text{ iff } (q, \Pi', q') \in \varrho \text{ or } (q, \Pi' \cup \{\text{last}\}, \text{true}) \in \varrho \text{ and } q' = \text{ended}.
\]

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 LDLf formula can be checked in PSPACE by nonemptiness of \( A_\varphi \) (or \( A_\varphi' \)).

Considering that it is known that satisfiability in LDLf 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 LDLf (see [4] for details).

### 4 Run-time Monitoring

From an high-level perspective, the monitoring problem amounts to observe an evolving system execution and 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 [9], which is basically the finite-trace variant of the RV semantics in [2]: Given a LTLf or LDLf formula \( \varphi \), each time the system evolves, the monitor returns one among the following truth values:

- \( [\varphi]_{\text{RV}} = \text{temp.true} \), meaning that the current execution trace temporarily satisfies \( \varphi \), i.e., it is currently compliant with \( \varphi \), but there is a possible system future prosecution which may lead to falsify \( \varphi \);
– $[\varphi]_{RV} = \text{temp}_\text{false}$, meaning that the current trace temporarily falsify $\varphi$, i.e., $\varphi$ is not current compliant with $\varphi$, but there is a possible system future prosecution which may lead to satisfy $\varphi$;

– $[\varphi]_{RV} = \text{true}$, meaning that the current trace satisfies $\varphi$ and it will always do, no matter how it proceeds;

– $[\varphi]_{RV} = \text{false}$, meaning that the current trace falsifies $\varphi$ 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 $\varphi$, 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 $\varphi$ in the future. We call a prefix possibly good for an LDL$_f$ formula $\varphi$ if there exists an extension of it which satisfies $\varphi$. More precisely, given an LDL$_f$ formula $\varphi$, we define the set of possibly good prefixes for $L(\varphi)$ as the set

$$L_{\text{poss,good}}(\varphi) = \{ \pi \mid \exists \pi'. \pi\pi' \in L(\varphi) \}$$

(1)

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

$$L_{\text{nec,good}}(\varphi) = \{ \pi \mid \forall \pi'. \pi\pi' \in L(\varphi) \}.$$  

(2)

The set of necessarily bad prefixes $L_{\text{nec,bad}}(\varphi)$ can be defined analogously as

$$L_{\text{nec,bad}}(\varphi) = \{ \pi \mid \forall \pi'. \pi\pi' \notin L(\varphi) \}.$$  

(3)

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

Using this language theoretic notions, we can provide a precise characterization of the semantics four standard monitoring evaluation functions [9].

**Proposition 1.** Let $\varphi$ be an LDL$_f$ formula and $\pi$ a trace. Then:

- $\pi \models [\varphi]_{RV} = \text{temp}_\text{true}$ iff $\pi \in L(\varphi) \setminus L_{\text{nec,good}}(\varphi)$;
- $\pi \models [\varphi]_{RV} = \text{temp}_\text{false}$ iff $\pi \in L(\neg \varphi) \setminus L_{\text{nec,bad}}(\varphi)$;
- $\pi \models [\varphi]_{RV} = \text{true}$ iff $\pi \in L_{\text{nec,good}}(\varphi)$;
- $\pi \models [\varphi]_{RV} = \text{false}$ iff $\pi \in L_{\text{nec,bad}}(\varphi)$.
Proof (sketch). Immediate from the definitions in [9] 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 $2^P$ 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 $\mathcal{L}(\varphi) \subseteq \mathcal{L}_{\text{possgood}}(\varphi)$ and $\mathcal{L}(\neg \varphi) \subseteq \mathcal{L}_{\text{possgood}}(\neg \varphi)$, we also have that $\mathcal{L}_{\text{possgood}}(\varphi) \cup \mathcal{L}_{\text{possgood}}(\neg \varphi) = (2^P)^*$. Also from the definition it is easy to see that $\mathcal{L}_{\text{possgood}}(\varphi) \cap \mathcal{L}_{\text{possgood}}(\neg \varphi) = \{ \pi | \exists \pi'. \pi \pi' \in \mathcal{L}(\varphi) \land \exists \pi''. \pi \pi'' \in \mathcal{L}(\neg \varphi) \}$ 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 an intersection are paths that can be extended to satisfy $\varphi$ but can also be extended to satisfy $\neg \varphi$. It is also easy to see that $\mathcal{L}(\varphi) = \mathcal{L}_{\text{possgood}}(\varphi) \setminus \mathcal{L}(\neg \varphi)$.

Turning to necessarily good prefixes and necessarily bad prefixes, it is easy to see that $\mathcal{L}_{\text{necgood}}(\varphi) = \mathcal{L}_{\text{possgood}}(\varphi) \setminus \mathcal{L}_{\text{possgood}}(\neg \varphi)$, that $\mathcal{L}_{\text{necbad}}(\varphi) = \mathcal{L}_{\text{possgood}}(\neg \varphi) \setminus \mathcal{L}_{\text{possgood}}(\varphi)$, and also that $\subseteq \mathcal{L}(\varphi)$ and $\mathcal{L}_{\text{necgood}}(\varphi) \subseteq \mathcal{L}(\neg \varphi)$.

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

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

\[L_{\text{necgood}}(\varphi) \quad L_{\text{possgood}}(\varphi) \cap L_{\text{possgood}}(\neg \varphi) \quad L_{\text{necbad}}(\varphi)\]

such that $L_{\text{necgood}}(\varphi) \cup (L_{\text{possgood}}(\varphi) \cap L_{\text{possgood}}(\neg \varphi)) \cup L_{\text{necbad}}(\varphi) = (2^P)^*$

$L_{\text{necgood}}(\varphi) \cap (L_{\text{possgood}}(\varphi) \cap L_{\text{possgood}}(\neg \varphi)) \cap L_{\text{necbad}}(\varphi) = \emptyset$.

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 $\text{LDL}_f$

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 for 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 $\varphi$.

$\text{LDL}_f$ 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 prove how use them for the monitoring problem.

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

**Lemma 1.** Given a $\text{LDL}_f$ formula $\varphi$, there exists a regular expression $\text{pref}_\varphi$ such that $\mathcal{L}(\text{pref}_\varphi) = \mathcal{L}_{\text{possgood}}(\varphi)$. 

Proof (sketch). The proof is constructive. We can build the NFA $A$ for $\varphi$ following the procedure in [4]. We then set as final all states of $A$ from which there exists a non-zero length path to a final state. This new finite state machine $A_{\text{poss_good}}(\varphi)$ is such that $L(A_{\text{poss_good}}(\varphi)) = L_{\text{poss_good}}(\varphi)$. Since NFA are exactly as expressive as regular expressions, we can translate $A_{\text{poss_good}}(\varphi)$ to a regular expression $\text{pref}_\varphi$.

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

**Theorem 3.** Given a $\text{LDL}_f$ formula $\varphi$,

\[
\pi \in L_{\text{poss_good}}(\varphi) \iff \pi \models \langle \text{pref}_\varphi \rangle \text{end}
\]

\[
\pi \in L_{\text{nec_good}}(\varphi) \iff \pi \models \langle \text{pref}_\varphi \rangle \text{end} \land \neg \langle \text{pref}_\neg \varphi \rangle \text{end}
\]

Proof (sketch). Any regular expression $\rho$, and hence any regular language, can be captured in $\text{LDL}_f$ as $\langle \rho \rangle \text{end}$. Hence the language $L_{\text{nec_good}}(\varphi)$ can be captured by $\langle \text{pref}_\varphi \rangle \text{end}$ and the language $L_{\text{nec_good}}(\neg \varphi)$ which is equivalent $L_{\text{poss_good}}(\varphi) \setminus L_{\text{poss_good}}(\neg \varphi)$ can be captured by captured by $\langle \text{pref}_\varphi \rangle \text{end} \land \neg \langle \text{pref}_\neg \varphi \rangle \text{end}$.\hfill $\square$

In other words, given a $\text{LDL}_f$ formula $\varphi$, formula $\varphi' = \langle \text{pref}_\varphi \rangle \text{end}$ is a $\text{LDL}_f$ formula such that $L(\varphi') = L_{\text{poss_good}}(\varphi)$. Similarly for $L_{\text{nec_good}}(\varphi)$.

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

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

- $\pi \models [\varphi]_{\text{RV}} = \text{temp_true}$ iff $\pi \models \varphi \land \langle \text{pref}_\varphi \rangle \text{end}$;
- $\pi \models [\varphi]_{\text{RV}} = \text{temp_false}$ iff $\pi \models \neg \varphi \land \langle \text{pref}_\neg \varphi \rangle \text{end}$;
- $\pi \models [\varphi]_{\text{RV}} = \text{true}$ iff $\langle \text{pref}_\varphi \rangle \text{end} \land \neg \langle \text{pref}_\neg \varphi \rangle \text{end}$;
- $\pi \models [\varphi]_{\text{RV}} = \text{false}$ iff $\langle \text{pref}_\neg \varphi \rangle \text{end} \land \neg \langle \text{pref}_\varphi \rangle \text{end}$.

Proof (sketch). Follows from Proposition 1 and Theorem 3 using the language theoretic equivalences discussed in Section 4.\hfill $\square$

6 Monitoring Declare Constraints and Metaconstraints

We now ground our monitoring approach to the case of DECLARE monitoring. DECLARE 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 [16][12]. 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.

Given a set $P$ of tasks, a DECLARE model is a set $C$ of LTL$_f$ (and hence LDL$_f$) constraints over $P$, 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 [5]. 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 an absence constraint, visually and formally represented as:

\[
\varphi_{close} = \neg \Diamond (close \text{ order} \land \Diamond \Diamond close \text{ order})
\]

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:

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

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).

\[
\varphi_{pay} = (\neg pay \text{ suppl} \lor close \text{ order}) \lor \neg \Diamond close \text{ order}
\]

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 [9,10]. 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 \( \text{LDL}_f \).** Since \( \text{LDL}_f \) includes \( \text{LTL}_f \), DECLARE constraints can be directly encoded in \( \text{LDL}_f \) using their standard formalization \([17,15]\). 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 \( \text{LDL}_f \) 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 \( \varphi_{\text{close}} \) in Example 1. Following Table 1, its good prefix characterization is \( \text{pref}_{\varphi_{\text{close}}} = \sigma^* + (\sigma^*; \text{close order}; \sigma^*) \), where \( \sigma \) 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:

<table>
<thead>
<tr>
<th>0.3</th>
<th>start</th>
<th>do “close order”</th>
<th>do “pay suppl.”</th>
<th>do “close order”</th>
</tr>
</thead>
<tbody>
<tr>
<td>close order</td>
<td>temp_true</td>
<td>false</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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., for absence 2, which can never be evaluated to be true (since it is always possible to continue the execution so as to perform \( \mathbf{a} \) twice), nor to temp_false (the only way of violating the constraint is to perform \( \mathbf{a} \) 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 [9], 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 [9].

**Metaconstraints.** Thanks to the ability of \( \text{LDL}_f \) 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
Table 1. Some DECLARE constraints, together with their prefix characterization, minimal bad prefix characterization, and possible RV states; for each constraint, $o$ is a shortcut for “other tasks”, i.e., tasks not involved in the constraint itself.

<table>
<thead>
<tr>
<th>NAME</th>
<th>NOTATION</th>
<th>pref</th>
<th>POSSIBLE RV STATES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Existence</td>
<td>$a$</td>
<td>$(a + o)^*$</td>
<td>temp_false, true</td>
</tr>
<tr>
<td>Absence 2</td>
<td>$a$</td>
<td>$a^* + (a^<em>; a^</em>)$</td>
<td>temp_false, true</td>
</tr>
<tr>
<td>Choice</td>
<td>$a$</td>
<td>$(a + b + o)^*$</td>
<td>temp_false, true</td>
</tr>
<tr>
<td>Exclusive Choice</td>
<td>$a$</td>
<td>$(a + o)^* + (b + o)^*$</td>
<td>temp_false, true, false</td>
</tr>
<tr>
<td>Resp. existence</td>
<td>$a$</td>
<td>$(a + b + o)^*$</td>
<td>temp_true, temp_false, true</td>
</tr>
<tr>
<td>Coexistence</td>
<td>$a$</td>
<td>$(a + b + o)^*$</td>
<td>temp_true, temp_false, true</td>
</tr>
<tr>
<td>Response</td>
<td>$a$</td>
<td>$(a + b + o)^*$</td>
<td>temp_true, temp_false</td>
</tr>
<tr>
<td>Precedence</td>
<td>$a$</td>
<td>$a^<em>; (a + b + o)^</em>$</td>
<td>temp_true, true, false</td>
</tr>
<tr>
<td>Succession</td>
<td>$a$</td>
<td>$a^<em>; (a + b + o)^</em>$</td>
<td>temp_true, true, false</td>
</tr>
<tr>
<td>Not Coexistence</td>
<td>$a$</td>
<td>$(a + o)^* + (b + o)^*$</td>
<td>temp_true, false</td>
</tr>
<tr>
<td>Neg. Succession</td>
<td>$a$</td>
<td>$(b + o)^<em>; (a + o)^</em>$</td>
<td>temp_true, false</td>
</tr>
</tbody>
</table>

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 [9,10].

- Recovery mechanisms resembling contrary-to-duty obligations in legal reasoning [19], i.e., obligations that are put in place only when other obligations are not met. Technically, a generic form for metaconstraints is the pattern $\Phi_{\text{pre}} \rightarrow \Psi_{\text{exp}}$, where:
  - $\Phi_{\text{pre}}$ is a boolean formula, whose atoms are membership assertions of the involved constraints to the RV truth values;
  - $\Psi_{\text{exp}}$ is a boolean formula whose atoms are the constraints to be enforced when $\Phi_{\text{pre}}$ 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 $\text{LDL}_f$ formula. Hence, we can reapply Theorem 4 to it, getting four $\text{LDL}_f$ 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 $\varphi_{\text{canc}}$ 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 $\text{LDL}_f$ 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 $\varphi\text{dopay}$ denote the LTL$_f$ formalization of such a compensation constraint. Second, we capture the intended compensation behavior by using the following LDL$_f$ metaconstraint:

\[
\{[\varphi\text{canc}]RV = \text{false}\} \rightarrow \varphi\text{dopay}
\]

which, leveraging Theorem 4, corresponds to the standard LDL$_f$ formula:

\[
\langle \text{pref}\neg\varphi\text{canc} \rangle \text{end} \land \neg\langle \text{pref}\varphi\text{canc} \rangle \text{end} \rightarrow \varphi\text{dopay}
\]

A limitation of this form of metaconstraint is that the right-hand part $\Psi\text{exp}$ is monitored from the beginning of the trace. This is acceptable in many cases. E.g., in Example 3, it is ok if the user already paid a supplement before the order cancelation caused constraint $\varphi\text{canc}$ to be violated. In other situations, however, this is not satisfactory, because we would like to enforce the compensating behavior only after $\Phi\text{pre}$ 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: $\Phi\text{pre} \rightarrow [\rho]\Psi\text{exp}$, where $\rho$ is a regular expression denoting the paths after which $\Psi\text{exp}$ is expected to be enforced.

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

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

\[
\{[\varphi\text{canc}]RV = \text{false}\} \rightarrow [\text{re}\{[\varphi\text{canc}]RV = \text{false}\}]\varphi\text{dopay}
\]

where $\text{re}\{[\varphi\text{canc}] = \text{false}\}$ denotes the regular expression for the language $\mathcal{L}\{[\varphi\text{canc}] = \text{false}\} = \mathcal{L}\{[\text{pref}\neg\varphi\text{canc}] \land \neg[\text{pref}\varphi\text{canc}]\text{end}\}$. This regular expression describes all paths containing a violation for constraint $\varphi\text{canc}$.

7 Implementation

The entire approach has been implemented as an operational decision support (OS) provider for the PROM 6 process mining framework\footnote{http://www.promtools.org/prom6/}. PROM 6 provides a generic OS environment\cite{PROM6} that supports the interaction between an external workflow management systems 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\cite{DFA}. Figure 2 shows a graphical representation of the evolution of constraints described in Example 1 of Section 6 when monitored using our operational support provider. In the LDL$_f$ formulas, the literals $f$, $g$ and $d$ respectively stand for tasks close order, cancel order, and pay supplement.
8 Conclusion

We can see the approach proposed in this paper, as an extension of the declarative process specification approach, at the basis of DECLARE, to (i) more powerful specification logics ($LDL_f$, i.e., Monadic Second Order logics over finite traces, instead of $LTL_f$, i.e., First Order logic logics over finite traces), (ii) to monitoring constraints. Notably, this declarative approach to monitoring supports seamlessly monitoring 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 [9]. Furthermore, we intend to extend our approach to data-aware business constraints [13], mixing temporal operators with first-order queries over the data attached to the monitored events. This setting has been studied using the Event Calculus [13,14], 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”.  

Fig. 2. Screenshot of our operational support provider’s output.
References