Runtime Verification of LTL-Based Declarative Process Models

F.M. Maggi[1*], M. Westergaard[1†], M. Montali[2‡], and W.M.P. van der Aalst[1]

[1] Eindhoven University of Technology, The Netherlands.
{f.m.maggi, m.westergaard, w.m.p.v.d.aalst}@tue.nl
[2] KRDB Research Centre, Free University of Bozen-Bolzano, Italy.
montali@inf.unibz.it

Abstract. Linear Temporal Logic (LTL) on finite traces has proven to be a good basis for the analysis and enactment of flexible constraint-based business processes. The *Declare* language and system benefit from this basis. Moreover, LTL-based languages like Declare can also be used for runtime verification. As there are often many interacting constraints, it is important to keep track of *individual constraints* and *combinations of potentially conflicting constraints*. In this paper, we operationalize the notion of conflicting constraints and demonstrate how innovative automata-based techniques can be applied to monitor running process instances. Conflicting constraints are detected immediately and our toolset (realized using Declare and ProM) provides meaningful diagnostics.
Keywords: Monitoring, Linear Temporal Logic, Finite State Automata, Declarative Business Processes, Operational Support, Process Mining

1 Introduction

Linear Temporal Logic (LTL) provides a solid basis for design-time verification and model checking. Moreover, LTL has also been used for the *runtime verification* of dynamic, event-based systems. In this latter setting, desired properties are expressed in terms of LTL. These properties and/or their conjunction are translated to a monitor which can be used to dynamically evaluate whether the current trace, representing an evolving run of the system, complies with the desired behavior or not.

Traditionally, LTL-based approaches were mainly used to verify or monitor running programs. However, the need for flexibility and a more declarative view on work processes, fueled the interest in the Business Process Management (BPM) field. The *Declare* language and system [11] show that it is possible to

* Research carried out as part of the Poseidon project at Thales under the responsibilities of the Embedded Systems Institute (ESI). The project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.
† Research supported by the Technology Foundation STW, applied science division of NWO and the technology program of the Dutch Ministry of Economic Affairs.
‡ Research supported by the NWO "Visitor Travel Grant" initiative and by the EU Project FP7-ICT ACSI (257593).

model LTL constraints graphically such that end user can understand them, while a workflow engine can enact the corresponding process. Constraints may be enforced by the Declare system or are monitored while the process unfolds.

Each graphical constraint in Declare is represented as an LTL formula, and the global process model is formalized as the conjunction of all such "local" formulas. Hence, there are two levels: (a) *individual constraints* well-understood by the end-user and (b) *global constraints* resulting from the interaction of local constraints. Runtime verification must provide *intuitive diagnostics* for every individual constraint, tracking its state as the monitored process instance evolves, but at the same time also provide diagnostics for the overall process model, giving a meaningful feedback obtained from the combination of different constraints.

In [6], we have investigated automata-based techniques for the runtime verification of LTL-based process models. In particular, we proposed *colored automata* to provide intuitive diagnostics for singular constraints and ways to continue verification even after a violation has taken place. Intuitively, a colored automaton is a finite state automaton built for the whole set of constraints composing a process model, where each state contains specific information (*colors*) indicating the state of individual constraints.

Here, we again use colored automata for runtime verification. However, now we focus on the *interplay of constraints*, i.e., we immediately detect violations which cannot be attributed to a single constraint in isolation, but result from combinations of conflicting constraints. To do so, we extend a variant of the RV-LTL semantics [2] with the notion of *conflicting constraint set*. Given the current trace of a system's instance, a set of constraints is conflicting if for any possible continuation of the instance at least one of such constraints will be eventually violated. Hence, our approach is able to *detect* constraint violations as early as possible. We show how to compute *minimal* conflicting sets, i.e., conflicting sets where the conflict disappears if one of the constraints is removed.

Our approach has been implemented in the context of the Declare system[3] and ProM[4]. We provide diagnostics that assist end-users in understanding the nature of deviations and suggest recovery strategies focusing on the constraints that are truly causing the problem.

The remainder of this paper is organized as follows. Section 2 presents some background material, and in Sect. 3 we introduce our runtime verification framework. Section 4 explains the core algorithms used in our approach. We have been applying our approach to various real-world case studies. In Sect. 5, we report on the monitoring of Declare constraints in the context of maritime safety and security. Section 6 concludes the paper.

2 Background

In this section, we introduce some background material illustrating the basic components of our framework. Using a running example, we introduce Declare,

[3] www.win.tue.nl/declare/
[4] www.processmining.org

present RV-FLTL, an LTL semantics for finite traces, and an approach to translate a Declare model to a set of automata for runtime verification.

2.1 Declare and Running Example

Declare is a declarative process modeling language and a workflow system based on constraints [9]. The language is grounded in LTL but has an intuitive graphical representation. Differently from imperative models that are "closed", Declare models are "open", i.e., they specify undesired behavior and allow everything that is not explicitly forbidden. The Declare system is a full-fledged workflow management system that, being based on a declarative language, offers more flexibility than traditional workflow systems.

Figure 1 shows a simple Declare model used within the maritime safety and security field. We use this example to explain the main concepts. It involves four *events* (depicted as rectangles, e.g., Under way using engine) and three *constraints* (shown as arcs between the events, e.g., not coexistence). In our example, a vessel can be Under way, either using an engine or sailing but not both, as indicated by the not coexistence between the two events. A vessel can be Constrained by her draught, but only after being Under way sailing (as a vessel with an engine cannot be constrained by draught and a sailing vessel cannot be constrained before it is under way). This is indicated by the precedence constraint. Due to harbor policy, only vessels with an engine can be Moored (sailing ships are instead anchored). This is indicated by the responded existence, which says that if Moored occurs, Under way using engine has to occur before or after.

Each individual Declare constraint can be formalized as an LTL formula talking about the connected events. Let us consider, for example, Fig. 1, naming the LTL formulas formalizing its different constraints as follows: φ_n is the not coexistence constraint, φ_p is the precedence constraint and φ_r is the responded existence constraint. Using M, S, E and C to respectively denote Moored, Under way sailing, Under way using engine and Constrained by her draught, we then have

$$\varphi_n = (\Diamond \mathsf{E}) \Rightarrow (\neg \Diamond \mathsf{S}) \qquad \varphi_p = (\Diamond \mathsf{C}) \Rightarrow (\neg \mathsf{C} \sqcup \mathsf{S}) \qquad \varphi_r = (\Diamond \mathsf{M}) \Rightarrow (\Diamond \mathsf{E})$$

The semantics of the whole model is determined by the conjunction of these formulas.

2.2 LTL Semantics for Constraint-Based Business Processes

Traditionally, LTL is used to reason over infinite traces. When focusing on runtime verification, reasoning is carried out on partial, ongoing traces, which describe a finite portion of the system's execution. Among the possible LTL semantics on finite traces, we use a variant of *Runtime Verification Linear Temporal Logic* (RV-LTL), a four-valued semantics proposed in [2]. Indeed, the four values used by RV-LTL capture in an intuitive way the possible states in which Declare constraints can be during the execution. Differently from the original RV-LTL semantics, which focuses on trace suffixes of infinite length, we limit ourselves

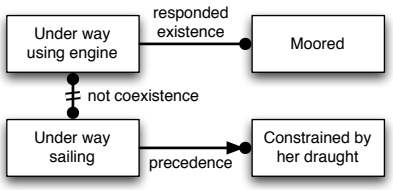

Fig. 1. Example Declare model.

to possible finite continuations (RV-FLTL). This choice is motivated by the fact that we consider process instances that need to complete eventually. This has considerable impact on the corresponding verification technique: reasoning on Declare models is tackled with finite state automata.

We denote with $u \models \varphi$ the truth value of an LTL formula φ in a finite trace u, according to FLTL [5], a standard LTL semantics for dealing with finite traces.

Definition 1 (RV-FLTL). *The semantics of* $[u \models \varphi]_{RV}$ *is defined as follows:*

- $[u \models \varphi]_{RV} = \top$ *(φ* permanently satisfied *by u) if for each possible finite continuation σ of u: $u\sigma \models \varphi$;*
- $[u \models \varphi]_{RV} = \bot$ *(φ* permanently violated *by u) if for each possible finite continuation σ of u: $u\sigma \not\models \varphi$;*
- $[u \models \varphi]_{RV} = \top^p$ *(φ* possibly satisfied *by u) if $u \models \varphi$ but there is a possible finite continuation σ of u such that $u\sigma \not\models \varphi$;*
- $[u \models \varphi]_{RV} = \bot^p$ *(φ* possibly violated *by u) if $u \not\models \varphi$ but there is a possible finite continuation σ of u such that $u\sigma \models \varphi$.*

We denote $\mathbb{B}_4 = \{\top, \bot, \top^p, \bot^p\}$ *and assume an order* $\bot \prec \bot^p \prec \top^p \prec \top$.

We say a formula is satisfied or violated if it is either permanently or possible violated or satisfied.

As we have seen for Declare, we do not look at specifications that consist of a single formula, but rather at specifications including sets of formulas. We generalize this aspect by defining an LTL process model as a set of (finite trace) LTL formulas, each capturing a specific business constraint.

Definition 2 (LTL process model). *An LTL process model is a finite set of LTL constraints* $\Phi = \{\varphi_1, \ldots, \varphi_m\}$.

One way to verify at runtime an LTL process model $\Phi = \{\varphi_1, \ldots, \varphi_m\}$ is to test the truth value $[u \models \Phi]_{RV} = [u \models \bigwedge_{i=1,\ldots,m} \varphi_i]_{RV}$. This approach, however, does not give any information about the truth value of each member of Φ in isolation. A solution for that is to test the truth values $[u \models \varphi_i]_{RV}, i = 1, \ldots, m$ separately. This is, however, still not enough. Let us consider, for example, the Declare model represented in Fig. 1. After executing the trace Moored, Under way sailing, the conjunction $\varphi_n \wedge \varphi_p \wedge \varphi_r$ is permanently violated but each member of the conjunction is not (φ_n is possibly satisfied, φ_p is permanently satisfied,

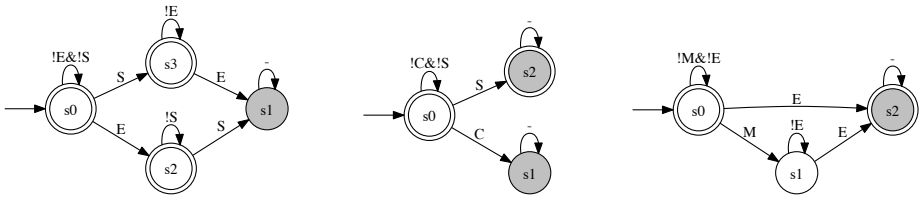

Fig. 2. Local automata for φ_n, φ_p, and φ_r from the example in Fig. 1.

and φ_r is possibly violated). Therefore, to give insights about the state of each constraint of an LTL process model and still detect non-local violations, we need to check both global and local formulas.

2.3 Translation of an LTL Process Model to Automata

Taking advantage of finiteness of traces in the RV-FLTL semantics, we construct a *deterministic finite state automaton* showing the state of each constraint given a prefix (we simply refer to such an automaton as "automaton"). An automaton accepts a trace if and only if it does not violate the constraint, and is constructed by using the translation in [3].

For the constraints in the model in Fig. 1, we obtain the automata depicted in Fig. 2. In all cases, state 0 is the initial state and accepting states are indicated using a double outline. A gray background indicates that the state is permanent (for both satisfied and violated). As well as transitions labeled with a single letter (repesenting an event), we also have transitions labeled with one or more negated letters; they indicate that we can follow the transition for any event not mentioned. This allows us to use the same automaton regardless of the exact input language. When we replay a trace on an automaton, we know that if we are in an accepting state, the constraint is satisfied, and when we are in a non-accepting state, it is violated. We can distinguish between the possible/permanent cases by the background; states with a gray background indicate that the state is permanent.

We can use these *local automata* directly to monitor each constraint, but to detect non-local violations we also need a *global automaton*. This can be constructed as the automaton product of the local automata or equivalently as the automaton of the conjunction of the individual constraints [12].

The global automaton for our example is shown in Fig. 3. We use state numbers from each of the automata from Fig. 2 as state names, so state 202 corresponds to constraint not coexistence being in state 2, constraint precedence being in state 0, and constraint responded existence being in state 2. These names are for readability only and do not indicate we can infer the states of local automata from the global states. To not clutter the diagram, we do not show self loops. These can be derived: every state also has a self-loop transition for any transition not otherwise explicitly listed. Accepting states in the global

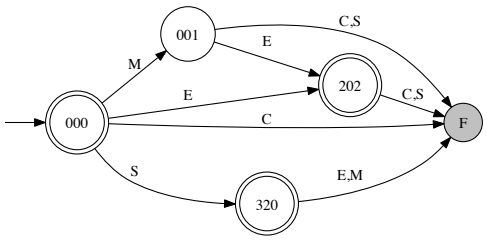

Fig. 3. Global automaton for our example.

automaton correspond to states where all constraints are satisfied. In a non-accepting state, at least one constraint is possibly violated. State F corresponds to all situations where it is no longer possible to satisfy all constraints. We note that state 321 is not present in Fig. 3 even though none of the local automata is in a permanently violated state and it is in principle reachable from state 001 via a S. The reason is that from this state it is never possible to reach a state where all constraints together are satisfied. Indeed, by executing the trace Moored, Under way sailing, Under way with engine, for instance, we obtain the trace $000 \rightarrow^M 001 \rightarrow^S F \rightarrow^E F$. Hence, we correctly identify that after the first event, we possibly violate some constraints, and after Under way sailing there is a non-local violation and we cannot satisfy all constraints together anymore.

The global automaton in Fig. 3 allows us to detect the state of the entire system, but not for individual constraints. In [6], we introduced a more elaborate automaton, the *colored automaton*. This automaton is also the product of the individual local automata, but now we include information about the acceptance state for each individual constraint. The colored automaton for our example is shown in Fig. 4. We retain the state numbering strategy, but add a second line describing which constraints are satisfied. In this case, each state of the colored automaton really contains indications about the acceptance state for each individual constraint. If a constraint is satisfied in a state, we add the first letter of the name of the constraint in uppercase (e.g., R indicating that the constraint responded existence is permanently satisfied in state 202). If a constraint is only possibly satisfied, we put parentheses around the letter (e.g., (R) in state 320). If a constraint is possibly violated in a state, we add the letter in lowercase (e.g., r in state 001), and if a constraint is permanently violated, we omit it entirely (e.g., precedence is permanently violated in state 011). Executing the trace Moored, Under way sailing, Under way using engine on the colored automaton, we obtain the trace $000 \rightarrow^M 001 \rightarrow^S 321 \rightarrow^E 122$. We can see in state 122 that we have permanently violated the constraint not coexistence and permanently satisfied the others (PR). Note that the presence of an undesired situation, attesting an unavoidable future violation, is already detected in state 321. However, in 321, the problem cannot be attributed to a single constraint. The problem is non-local and is caused by the interplay between not coexistence and responded existence (the first forbidding and the other requiring the presence of event Under way using engine). We capture this kind of situation by introducing the notion of *conflicting* constraint sets.

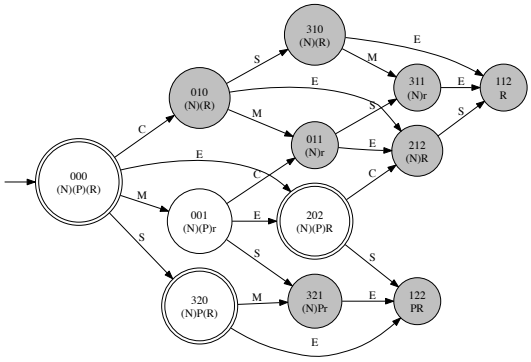

Fig. 4. Colored automaton for the example in Fig. 1.

3 Conflicting Constraint Sets

We aim at diagnostics explaining the core reason of a non-local violation. Therefore, it is important to identify the smallest parts of the original LTL process model that cause the problem. We tackle this issue by characterizing the relationship between the overall state of the system and the one of individual constraints. In particular, we show that the global state can be determined from the local states only when an explicit notion of *conflicting set* is defined and included in the semantics. In this respect, we first look at the truth value of subsets of the original specification.

Definition 3 (Monitoring evaluation). *Given an LTL process model Φ and a finite trace u, we define the sets $P_s(\Phi, u) = \{\Psi \subseteq \Phi \mid [u \models \Psi]_{RV} = s\}$ for $s \in \mathbb{B}_4$. The monitoring evaluation of trace u w.r.t. Φ is then $\mathcal{M}(\Phi, u) = (P_\bot(\Phi, u), P_{\bot^p}(\Phi, u), P_{\top^p}(\Phi, u), P_\top(\Phi, u))$.*

As our goal is to deduce the global state of a system from the states of individual constraints, we need to analyze the structure of elements of $\mathcal{M}(\Phi, u)$. It can be observed that $\mathcal{M}(\Phi, u)$ is a partition of the powerset of Φ:

Property 1 (Partitioning). Given an LTL process model Φ and a finite trace u, $\mathcal{M}(\Phi, u)$ is a partition of the powerset of Φ, i.e., $\bigcup_{s \in \mathbb{B}_4} P_s(\Phi, u) = 2^\Phi$ and for $s, s' \in \mathbb{B}_4$ with $s \neq s'$: $P_s(\Phi, u) \cap P_{s'}(\Phi, u) = \emptyset$.

This is realized by observing that every subset of Φ has exactly one assigned truth value. Second, for two subsets of Φ, $\Psi' \subseteq \Psi'' \subseteq \Phi$, the larger one is not easier to satisfy:

Property 2 (Inclusion). Given an LTL process model Φ and a finite trace u, then for $\Psi' \subseteq \Psi'' \subseteq \Phi$ and an $s \in \mathbb{B}_4$, if $\Psi' \in P_s(\Phi, u)$ then $\Psi'' \in P_{s'}(\Phi, u)$ for some $s' \in \mathbb{B}_4$ with $s' \preceq s$.

This stems from monotonicity of truth values of conjunctions. Third, permanently satisfied constraints do not change the truth value of sets of constraints:

Property 3 (Effect of permanently satisfied constraints). Given an LTL process model Φ and a finite trace u, and a $\psi \in \Psi$ such that $[u \models \psi]_{RV} = \top$, if $\Psi \in P_s(\Phi, u)$ for some $s \in \mathbb{B}_4$, then $\Psi \setminus \{\psi\} \in P_s(\Phi, u)$.

This stems from the fact that for any extension, the permanently satisfied one reduces to true and can be removed using identity $\top \wedge \psi = \psi$ for any constraint ψ. This allows us to characterize the structure of sets with a given truth value:

Property 4 (Structure of global states). Given an LTL process model Φ and a finite trace u, for a subset of constraints $\Psi \subseteq \Phi$

1. $\Psi \in P_\top(\Phi, u)$ if and only if $\forall \psi \in \Psi$, $[u \models \psi]_{RV} = \top$,
2. $\Psi \in P_{\top^p}(\Phi, u)$ if and only if $\forall \psi \in \Psi$, $[u \models \psi]_{RV} \in \{\top, \top^p\}$ and $\exists \psi \in \Psi$ such that $[u \models \psi]_{RV} = \top^p$,
3. if $\Psi \in P_{\bot^p}(\Phi, u)$, then $\forall \psi \in \Psi$, $[u \models \psi]_{RV} \in \{\top, \top^p, \bot^p\}$ and $\exists \psi \in \Psi$ such that $[u \models \psi]_{RV} = \bot^p$, and
4. if $\Psi \in P_\bot(\Phi, u)$ and $\forall \psi \in \Psi$, $[u \models \psi]_{RV} \neq \bot$, then $\exists \psi \in \Psi$ such that $[u \models \psi]_{RV} = \bot^p$.

The first item is seen by assuming that some constraint exists in Φ that is not permanently satisfied for u. Equivalently, there exists a finite continuation of u where this constraint is not satisfied and the conjunction of all constraints in Φ is not satisfied for u. The second and third are seen by similar arguments. The last one is seen by observing that if a set has only possibly or permanently satisfied members, it is itself possibly or permanently satisfied.

Given an LTL process model Φ, a trace u, and a subset $\Psi \subseteq \Phi$, we can easily identify whether Ψ belongs to $P_\top(\Phi, u)$ or $P_{\top^p}(\Phi, u)$ by simple inspection of the state of individual constraints in the colored automaton mentioned earlier. For the first two items of Prop. 4, the states of the constraints in a node completely characterize, in this case, the global state of the system. However, we cannot determine whether a set belongs to $P_{\bot^p}(\Phi, u)$ or $P_\bot(\Phi, u)$ only by looking at the state of individual constraints: Prop. 4 only gives us implication in one direction in this case.

We introduce a fifth truth value of constraints $\bot^c$ that allows us to deduce the state of the entire system from the state of individual constraints. This reflects that a constraint is not permanently violated but is in conflict with others so the entire system cannot be satisfied again. To better characterize the problem when a permanent violation occurs, we minimize the sets originating the violation. Therefore, we look at minimal subsets $\Psi \in P_\bot(\Phi, u)$. A first group of these minimal subsets are singletons $\{\psi\}$ with $\psi \in P_\bot(\Phi, u)$. A second group consists of *conflicting sets*:

Definition 4 (Conflicting set). *Given an LTL process model Φ and a finite trace u, $\Psi \subseteq \Phi$ is a* conflicting set *of Φ w.r.t. u if:*

1. $\Psi \in P_\bot(\Phi, u)$,
2. $\forall \psi \in \Psi$, $[u \models \psi]_{RV} \neq \bot$, *and*
3. $\forall \psi \in \Psi$, $[u \models \Psi \setminus \{\psi\}]_{RV} \neq \bot$.

We extend the semantics of RV-FLTL to capture conflicting sets:

Definition 5 (RVc-FLTL). *The semantics of* $[u, \Phi \models \varphi]_{RVc}$ *is defined as*

$$[u, \Phi \models \varphi]_{RVc} = \begin{cases} \bot^c & \text{if there is a conflicting set } \Psi \subseteq \Phi \text{ s.t. } \varphi \in \Psi \\ [u \models \varphi]_{RV} & \text{otherwise.} \end{cases}$$

Therefore, we can introduce a variant of Prop. 4 allowing us to determine the global state solely using local values:

Theorem 1 (Structure of global states). *Given an LTL process model* Φ *and a finite trace* u*, then*

1. $[u \models \Phi]_{RV} = \top$*, if and only if* $\forall \psi \in \Phi$, $[u, \Phi \models \psi]_{RVc} = \top$,
2. $[u \models \Phi]_{RV} = \top^p$*, if and only if* $\forall \psi \in \Phi$, $[u, \Phi \models \psi]_{RVc} \in \{\top, \top^p\}$ *and* $\exists \psi \in \Phi$ *such that* $[u, \Phi \models \psi]_{RVc} = \top^p$,
3. $[u \models \Phi]_{RV} = \bot^p$*, if and only if* $\forall \psi \in \Phi$, $[u, \Phi \models \psi]_{RVc} \in \{\top, \top^p, \bot^p\}$ *and* $\exists \psi \in \Phi$ *such that* $[u, \Phi \models \psi]_{RVc} = \bot^p$,
4. $[u \models \Phi]_{RV} = \bot$ *if and only if* $\exists \psi \in \Phi$ *such that* $[u, \Phi \models \psi]_{RVc} \in \{\bot^c, \bot\}$.

In [6] we explain how to modify the original LTL process model on the fly in an efficient way when a violation occurs. Therefore, when a non-local violation is detected, it can be useful to identify minimal sets of constraints to be removed in the original LTL process model to recover from the violation. We capture this as a *recovery set*:

Definition 6 (Recovery set). *Given an LTL process model* Φ *and a finite trace* u *such that* $[u \models \Phi]_{RV} = \bot$*, then* $\Psi \subseteq \Phi$ *is a recovery set of* Φ' *w.r.t.* Φ *and* u *if*

1. $[u \models \Phi \setminus \Psi]_{RV} \neq \bot$
2. $\forall \psi \in \Psi$, $[u \models \Phi \setminus (\Psi \setminus \{\psi\})]_{RV} = \bot$

Intuitively, we must remove exactly one constraint from each conflicting set in Φ, but if two (or more) conflicting sets overlap, we can remove one from the intersection to make a smaller recovery set.

Let us consider the Declare model represented in Fig. 1. We name the LTL constraints of this model as specified in Sect. 2.1. Figure 5 shows a graphical representation of the constraints' evolution: events are displayed on the horizontal axis. The vertical axis shows the three constraints. Initially, all three constraints are possibly satisfied. Let $u_0 = \varepsilon$ denote the initial (empty) trace:

$$[u_0, \Phi \models \varphi_n]_{RVc} = \top^p \qquad [u_0, \Phi \models \varphi_p]_{RVc} = \top^p \qquad [u_0, \Phi \models \varphi_r]_{RVc} = \top^p$$

Event Moored is executed next ($u_1 = Moored$), we obtain:

$$[u_1, \Phi \models \varphi_n]_{RVc} = \top^p \qquad [u_1, \Phi \models \varphi_p]_{RVc} = \top^p \qquad [u_1, \Phi \models \varphi_r]_{RVc} = \bot^p$$

Note that $[u_1 \models \varphi_r]_{RV} = \bot^p$ because the responded existence constraint becomes possibly violated after the occurrence of Moored. The constraint is waiting for

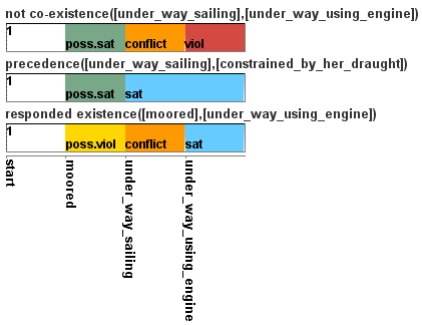

Fig. 5. One of the views provided by our monitoring system. Colors show the state constraints while the process instance evolves; *red* (*viol*) refers to $\bot$, *yellow* (*poss. viol*) to $\bot^p$, *green* (*poss. sat*) to $\top^p$, *blue* (*sat*) refers to $\top$, and *orange* (*conflict*) to $\bot^c$.

the occurrence of another event (execution of Under way using engine) to become satisfied again. Then, Under way sailing is executed ($u_2 = Moored, Under\ way\ sailing$), leading to a situation in which constraint precedence is permanently satisfied but not coexistence and responded existence are in conflict.

$$[u_2, \Phi \models \varphi_n]_{RVc} = \bot^c \qquad [u_2, \Phi \models \varphi_p]_{RVc} = \top \qquad [u_2, \Phi \models \varphi_r]_{RVc} = \bot^c$$

Note that we have exactly one conflicting set, $\{\varphi_n, \varphi_r\}$. Indeed, if we look at the automaton in Fig. 4, from 321 it is not possible to reach a state where both these constraints are satisfied. Moreover, no supersets can be a conflicting set (due to minimality). $\{\varphi_n, \varphi_p\}$ is not a conflicting set as they are both satisfied in 122, and $\{\varphi_n, \varphi_p\}$ is not a conflicting set as it is temporarily satisfied. The next event is Under way using engine ($u_3 = Moored, Under\ way\ sailing, Under\ way\ using\ engine$), resulting in:

$$[u_3, \Phi \models \varphi_n]_{RVc} = \bot \qquad [u_3, \Phi \models \varphi_p]_{RVc} = \top \qquad [u_3, \Phi \models \varphi_r]_{RVc} = \top$$

not coexistence becomes permanently violated because Under way using engine and Under way sailing cannot coexist in the same trace. Note that this violation has been detected as early as possible by our monitoring system; already when Under way sailing occurred, the conflicting set of constraints showed that it would be impossible to satisfy all constraints at the same time. However, it is still possible to see that the responded existence constraint becomes permanently satisfied by the Under way using engine event.

4 Deciding RVc-FLTL Using Automata

In this section, we give algorithms for detecting the state of sets of constraints. We start by giving algorithms for the extra information we have added to the automata in Sect. 2, and then focus on how to compute the information about conflicting sets contained in the colored automaton.

4.1 Local Automata

We get most of the information exhibited in the local automata in Fig. 2 from the standard translation in [3]. The only thing missing is the background color indicating whether a constraint is permanently/possibly satisfied or violated.

We get the background information by marking any state from which an accepting state is always/never reachable. We can do this efficiently using the strongly connected components (SCCs) of the automaton (this can be computed in linear time using Tarjan's algorithm [10]). We look at components with only outgoing arcs to components already processed (initially none), and we color a component gray only if i) it contains nodes that are all accepting/non-accepting and ii) all (if any) reachable components contain the same type states and are colored. This is also linear in the input automaton.

If the automaton we get is deterministic and minimal, we know that at most one accepting state will have gray background and at most one non-accepting state will have gray background. These can be identified as the (unique) accepting and non-accepting states with a self-loop allowing all events. All automata in Fig. 2 satisfy this, and we see they all have at most one gray state of each kind. Using these automata, we can decide the state of a constraint ($\top$, $\bot$, $\top^p$, or $\bot^p$) with respect to each trace, but we cannot detect non-local violations.

4.2 Global Automaton and its Combination with Local Automata

We can compute the global automaton directly using the same approach adopted for local automata (following [12] for better performance). This is the approach used for the automaton in Fig. 3. Using this automaton, we compute the state of the global system, but not for individual constraints. In this way, we can detect non-local violations but we cannot compute conflicting sets nor decide the state of individual constraints.

To infer the state of the entire system as well as of individual constraints, we can use at the same time the local and global automata. However, this forces us to replay the trace on many automata: the global one plus n local ones, where n is the number of constraints. Moreover, we cannot here detect exactly which constraints are conflicting, only that there are some, making this approach less useful for debugging.

4.3 Colored Automaton

To identify conflicting sets, we construct a colored automaton (like the one in Fig. 4) using the method described in [6]. We then post-process it to distinguish permanently/possibly satisfied or violated states (by computing SCCs, exactly like we did for the local automata).

To additionally compute conflicting sets, we notice that they are shared among states in an SCC (if a set of constraints cannot be satisfied in a state, it also cannot be satisfied in states reachable from it, and all states in an SCC are reachable from each other by definition). Furthermore, conflicting sets have to

be built using possibly satisfied and possibly violated constraints of an SCC. We can ignore permanently satisfied constraints because of Prop. 3 and item 3 of Def. 4. We can ignore permanently violated constraints due to item 2 of Def. 4. In an SCC, all states share permanently violated and satisfied constraints as they can all reach each other, so we can obtain all interesting constraints by looking at one of the states in isolation.

Due to item 1 for Def. 4 and Prop. 2, we only have to consider states that are permanently violated for computation of conflicting sets (gray states with single outline in Fig. 4). We notice that the conflicting sets of an SCC have to be super-sets of conflicting sets of all successor SCCs or contain a constraint that in a successor SCC is permanently violated. This is seen by a weaker version of the argument for members of SCCs sharing conflicting sets, as reachability is only true in one direction. The inclusion may be strict due to minimality of conflicting sets (item 3 of Def. 4).

We thus start in terminal SCCs (SCCs with no successors) and compute the conflicting sets. This is done by considering all subsets composed of possibly violated/satisfied constraints with more than one member and checking whether they are satisfiable in the component. This can be done by examining all states of the SCC and checking if there is one where all members of the considered subset are (possibly) satisfied. We can perform this bottom-up or top-down. The bottom-up approach starts with sets with two elements and adds elements until a set become unsatisfiable, exploiting minimality (item 3 of Def. 4) in that no superset of a conflicting set is a conflicting set. Alternatively, we can compute the sets top-down, starting with all possible violated/satisfied constraints and removing constraints until the set becomes satisfied, exploiting that subsets of a set of satisfied constraints do not need to be considered due to monotonicity. Which one is better depends on the size of the conflicting sets.

For each globally unsatisfiable SCC we recursively compute for all successors and then build conflicting set bottom-up, starting with all possible (minimal) unions of conflicting sets or singleton permanently violated properties of successors. For the example in Fig. 4, state 112 has, for instance, no conflicting sets, but two permanently violated constraints (φ_n and φ_p). Computing conflicting sets for 311 only needs to consider sets containing (at least) one of these, and as φ_p is permanently violated, we can ignore it. The only possibility, $\{\varphi_n, \varphi_r\}$, is indeed a conflicting set. For state 310 we need to consider unions of the conflicting sets and permanently violated constraints of successors of 311 and 112, i.e., $\{C_1 \cup C_2 \mid C_1 \in \{\{\varphi_n\}, \{\varphi_p\}\},\ C_2 \in \{\{\varphi_n, \varphi_r\}, \{\varphi_p\}\}\} = \{\{\varphi_n, \varphi_r\}, \{\varphi_n, \varphi_p\}, \{\varphi_p, \varphi_n, \varphi_r\}, \{\varphi_p\}\}$ which can be reduced by removing sets containing φ_p (which is permanently violated in 310) to $\{\{\varphi_n, \varphi_r\}\}$. We furthermore remove any supersets of contained sets (none in this case), and use the sets as basis for computing conflicting sets. As $\{\varphi_n, \varphi_r\}$ is satisfiable in 310, such constraints do not constitute a conflicting set, hence 310 has no conflicting sets.

Each SCC can have exponentially many conflicting sets in the number of constraints (assume we have n constraints and construct a SCC with all states possibly satisfying exactly $\frac{n}{2}$ constraints and possibly violating the remaining;

as all sets have the same size, none can be subsets of the others, and we have $\frac{n!}{\frac{n}{2}!} \in O(2^{\frac{n}{2}})$ such sets). In our initial experiments we have never seen examples with more than a few possibly violated/satisfied constraints, so in practice this is acceptable. Future work includes validating that this is also true for large real-life examples. If the pre-computation proves to be too expensive, we can also perform the algorithm at run-time, only computing conflicting sets when we reach a globally permanently violated state. By caching and sharing the results between instances (as well as intermediate results imposed by recursion), we should be able to provide acceptable runtime performance.

In our running example, executing the trace Moored, Under way sailing, we obtain the trace $000 \rightarrow^M 001 \rightarrow^S 321$. Using our algorithm to compute the conflicting sets, we see that in terminal SCC 122 in Fig. 4 we have no conflicting sets but a single permanently violated constraint φ_n. In state 321 we have exactly one conflicting set, $\{\varphi_n, \varphi_r\}$.

5 Case Study

We now present an application of our runtime verification framework to a real case study, focused on monitoring vessel behavior in the context of maritime safety and security. It has been provided by Thales, a global electronics company delivering mission-critical information systems and services for aerospace, defense, and security.

In our experiments, we use different logs describing the behavior of different types of vessels. These logs have been collected by a maritime Automatic Identification System (AIS) [4], which acts as a transponder that logs and sends events to an AIS receiver. Each log contains a set of process instances corresponding to the behavior of vessels of the same type (e.g., Passenger ship, Fishing boat, Dredger or Tanker). An event in a process instance is a change in the navigational state of the vessel (e.g., Moored, Under way using engine, At anchor, Under way sailing, or Restricted maneuverability). The logs are one-week excerpts of larger logs tracing the behavior of each vessel in the long term.

Starting from these logs, exploiting process mining techniques [1], we discover Declare models representing the behavior of each vessel type. A fragment of the discovered model for Dredger is shown in Fig. 6. The ultimate goal is to consequently use these models to monitor new vessel behaviors, using the colored automata-based approach outlined in this contribution.

More specifically, to construct the model in Fig. 6, we apply the Declare discovery technique described in [7]. We fix the not coexistence and response constraints as possible candidate constraints (the response indicating that if the source event occurs, then the target event must eventually occur). The miner identifies all the not coexistence and response constraints that are satisfied in all the traces of the log. However, when the log is an excerpt of a larger log, it is possible to make the discovery process more flexible by accepting a constraint also if it is possibly violated in some traces: being each execution trace incomplete, such a constraint could be satisfied in the continuation of the trace.

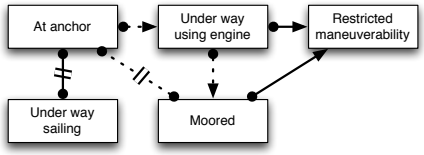

Fig. 6. Discovered model for vessel type Dredger; dashed constraints represent a conflicting set arising after the occurrence of At anchor

Even though the miner only identifies constraints that never give rise to a permanent violation by themselves, it is possible that conflicting sets of constraints exist in the discovered model. The conflicting sets are caused by the fact that, to extract the reference models from the logs, the miner checks each constraint separately while accepting possibly violated constraints. This makes the approach presented in this paper relevant in the prompt identification of an actual conflict during the monitoring process. For example, Fig. 6 contains a conflict when At anchor is executed; the conflicting constraints are depicted with dashed lines. Note that, in this specific case, each constraint of the conflicting set is a recovery set: the conflict is solved by removing any of them from the model.

6 Conclusion

We have introduced the runtime verification of flexible, constraint-based process models formalized in terms of LTL on finite traces, focusing on violations arising from interference of multiple constraints. A conflicting set provides a minimal set of constraints with no continuation where all constraints can be satisfied.

To do so, we have exploited in a novel way established results achieved in the field of temporal logics and runtime verification. In particular, we have considered a finite-trace variation of the RV-LTL semantics [2], following the finite state automata approach of [3] and the optimized algorithms proposed in [12] for the generation of automata. Such automata are employed to provide intuitive diagnostics about the business constraints during execution of a model. More specifically, we have shown how local and/or global information can be provided by combining the use of local automata and of a global automaton, or using a single colored automaton to provide full information.

All the techniques presented in this paper have been fully implemented in Declare and ProM. In particular, we have developed an Operational Support (OS) provider for ProM [1, 13], exploiting the recently introduced OS service. The OS service is the backbone for implementing process mining techniques that are not used in a post-mortem manner, i.e. on already completed process instances, but are instead meant to provide runtime support to running executions. Our provider takes in input a Declare model, and exploits the colored automata-based techniques presented here to track running instances and give intuitive

diagnostics to the end users, graphically showing the status of each constraint, as well as reporting local and non-local violations (see Fig. 5 for an example). In the latter case, recovery sets are computed, showing the minimal possible modifications that can be applied to the model to alleviate the detected conflict.

Monitoring business constraints can be also tackled by using the Event Calculus (EC) [8]. The two approaches are orthogonal to each other: the EC can only provide diagnostics about local violations but is easier to augment with other perspectives such as metric time constraints and data related aspects. We plan to investigate the incorporation of metric time aspects also in an automaton-based approach, relying on timed automata for verification.

References

1. W.M.P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.
2. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL Semantics for Runtime Verification. *Logic and Computation*, 20(3):651–674, 2010.
3. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proc. ASE*, pages 412–416. IEEE Computer Society, 2001.
4. International Telecommunications Union. *Technical characteristics for a universal shipborne Automatic Identification System using time division multiple access in the VHF maritime mobile band*, 2001. Recommendation ITU-R M.1371-1.
5. O. Lichtenstein, A. Pnueli, and L.D. Zuck. The Glory of the Past. In *Proc. of Logic of Programs*, pages 196–218. Springer, 1985.
6. 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 *Proc. of BPM*, 2011.
7. F.M. Maggi, A.J. Mooij, and W.M.P. van der Aalst. User-Guided Discovery of Declarative Process Models. In N. Chawla, I. King, and A. Sperduti, editors, *IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2011)*, pages 192–199, Paris, France, April 2011. IEEE.
8. M. Montali, F.M. Maggi, F. Chesani, P. Mello, and W.M.P. van der Aalst. Monitoring Business Constraints with the Event Calculus. Technical Report DEIS-LIA-002-11, University of Bologna (Italy), 2011. LIA Series no. 97, http://www.lia.deis.unibo.it/Research/TechReport/LIA-002-11.pdf.
9. M. Pesic, H. Schonenberg, and W.M.P. van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In *Proc. of EDOC*, pages 287–300. IEEE Computer Society, 2007.
10. R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
11. W.M.P. van der Aalst, M. Pesic, and H. Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science - R&D*, 23(2):99–113, 2009.
12. M. Westergaard. Better Algorithms for Analyzing and Enacting Declarative Workflow Languages Using LTL. In *Proc. of BPM*, 2011.
13. M. Westergaard and F.M. Maggi. Modelling and Verification of a Protocol for Operational Support using Coloured Petri Nets. In *Proc. of ATPN'11*, 2011.