Monitoring Business Constraints with Linear Temporal Logic: An Approach Based on Colored Automata

Fabrizio Maria Maggi[1,⋆], Marco Montali[2,⋆⋆], Michael Westergaard[1,⋆⋆⋆], and Wil 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. Today's information systems record real-time information about business processes. This enables the monitoring of business constraints at runtime. In this paper, we present a novel runtime verification framework based on linear temporal logic and colored automata. The framework continuously verifies compliance with respect to a predefined constraint model. Our approach is able to provide meaningful diagnostics even after a constraint is violated. This is important as in reality people and organizations will deviate and in many situations it is not desirable or even impossible to circumvent constraint violations. As demonstrated in this paper, there are several approaches to recover after the first constraint violation. Traditional approaches that simply check constraints are unable to recover after the first violation and still foresee (inevitable) future violations. The framework has been implemented in the process mining tool ProM.

Keywords: Runtime Verification, Monitoring, Linear Temporal Logic, Declare, Automata.

1 Introduction

Entities within an organization are supposed to operate within boundaries set by internal policies, norms, best practices, regulations, and laws. For example,

⋆ This research has been carried out as a 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.

⋆⋆ This research has been partially supported by the NWO "Visitor Travel Grant" initiative, and by the EU Project FP7-ICT ACSI (257593).

⋆⋆⋆ This research is supported by the Technology Foundation STW, applied science division of NWO and the technology program of the Dutch Ministry of Economic Affairs.

S. Rinderle-Ma, F. Toumani, and K. Wolf (Eds.): BPM 2011, LNCS 6896, pp. 132–147, 2011.

requests of a particular type need to be followed by a decision. Also in a cross-organizational setting, people and organizations need respect certain rules, e.g., a bill should be paid within 28 days. We use the generic term *business constraint* to refer a requirement imposed on the execution of an intra- or inter-organizational process [12,10]. Business constraints separate compliant behavior from non-compliant behavior.

Compliance has become an important topic in many organizations. Nevertheless, it is still difficult to operationalize compliance notions. Several authors developed techniques to ensure compliance by designing process models that enforce a set of business constraints [1,7]. Given a process model and a set of constraints, e.g., expressed in some temporal logic, one can use model checking [4] to see whether the model satisfies the constraints.

However, static verification techniques are not sufficient to tackle compliance problems in a comprehensive way. First of all, some aspects cannot be verified a priori as compliance may depend on the particular context and its participants. Second, it cannot be assumed that the behavior of all actors is known or can be controlled. Most processes involve autonomous actors, e.g., a specialist in a hospital may deviate to save lives and another organization may not be very responsive because of different objectives. Third, process designs are typically the outcome of a collaborative process where only some constraints are taken into account (to reduce complexity and increase flexibility). Due to the procedural nature of most process modeling languages [12], incorporating all constraints is unreasonable (especially in environments with a lot of variability): the model would become unreadable and difficult to maintain. Last but not least, violations do not always correspond to undesirable behavior. Often people deviate for good reasons. In unpredictable and dynamic settings, breaking the rules is sometimes justified by the inadequacy or incompleteness of rules.

All these issues call for *runtime verification facilities*, able to monitor the running cases of a process and to assess whether they comply with the business constraints of interest. Such facilities should provide meaningful information to the stakeholders involved. Roughly speaking, the purpose of monitoring is to check whether a running case satisfies or violates a correctness property, which in our setting is constituted by a set of business constraints. Since runtime verification and monitoring focus on the dynamics of an evolving system as time flows, LTL (Linear Temporal Logic) have been extensively proposed as a suitable framework for formalizing the properties to be monitored, providing at the same time effective verification procedures. However, the majority of current LTL runtime verification techniques limit themselves to produce as output only a truth value representing whether the current trace complies with the monitored property or not. Typically, a three or four-valued semantics is chosen for the LTL entailment, in order to reflect the fact that it is not always possible to produce a definitive answer about compliance (e.g., [2]). Furthermore, runtime verification usually halts as soon as a (permanent) violation is encountered: from now on, every possible extension of the current trace would still lead to a permanent violation.

In this paper, we present a novel framework for compliance evaluation at runtime. The framework offers the following:

1. *intuitive diagnostics*, to give fine-grained feedback to the end users (which constraints are violated and why);
2. *continuous support*, to provide verification capabilities even after a violation has taken place;
3. *recovery capabilities*, to realize different strategies for continuous support and accommodate sophisticated recovery mechanisms.

Our proposed approach is based on *colored automata*, i.e., automata whose states are associated to multiple relevant information ("colors"). Moreover, we adopt *Declare* [11] as a constraint language. Declare constraints have an intuitive graphical notation and LTL-based semantics.

Concerning the feedback returned to the monitor system, our approach does not only communicate if a running case is currently complying with the constraint model, but also computes the *state* of each constraint. We consider three possible states for constraints: *satisfied*, *possibly violated* and *permanently violated*. The first state attests that the monitored case is currently compliant with the constraint. The second state indicates that the constraint is currently violated, but it is possible to bring it back to a satisfied state by executing a sequence of events. The last state models the situation where it has become impossible to satisfy the constraint.

At runtime, two possible permanent violations may occur: (a) a forbidden event is executed, or (b) a state is reached such that two or more constraints become conflicting. The presence of a *conflict* means that there is no possible future continuation of the case such that all the involved constraints become satisfied. Furthermore, when the case is terminated all the possibly violated constraints become permanently violated, because no further event will be executed to satisfy them.

The approach has been implemented using ProM and Declare. Declare [11] is a flexible workflow system based on the Declare language. ProM[1] is a pluggable framework for process mining providing a wide variety of analysis techniques (discovery, conformance, verification, performance analysis, etc.). In the context of ProM, we have developed a generic *Operational Support* (OS) environment [14,17] that allows ProM to interact with systems like Declare at runtime. Our monitoring framework has been implemented as an OS provider.

The remainder of this paper is organized as follows. Section 2 presents some preliminaries: Declare as a specification language, RV-LTL as finite trace LTL semantics, and a translation of LTL into automata to build the monitors. Section 3 explains how *colored automata* can be used to check compliance at runtime and provide meaningful diagnostics. Section 4 presents three strategies for dealing with violations. Section 5 shows that the Declare model can also be modified at runtime, e.g., frequent violations may trigger an update of the model. Related work is discussed in Sect. 6. Section 7 concludes the paper.

[1] ProM and the runtime verification facilities described in this paper can be downloaded from `www.processmining.org`.

2 Background

In this section, we introduce some background material illustrating the basic components of our framework. Using a running example, we introduce Declare (Sect. 2.1). In Sect. 2.2, we present RV-LTL, an LTL semantics for finite traces. In Sect. 2.3, we introduce an approach to translate a Declare constraint model to a set of automata for runtime verification.

2.1 Running Example Using Declare

Declare is both a language and a system based on constraints [11]. The language is grounded in LTL but is equipped with a graphical and intuitive language. The Declare system is a full-fledged workflow system offering much more flexibility than traditional workflow systems.

Figure 1 shows a fictive Declare model that we will use as a running example throughout this paper. This example models two different strategies of investment: bonds and stocks. When an investor receives an amount of money, she becomes in charge of eventually investing it in bonds or in stocks and she cannot receive money anymore before the investment (*alternate response*). If the investor chooses for a low risk investment, she must buy bonds afterwards (*response*). Moreover, the investor can receive a high yield only if she has bought stocks before (*precedence*). Finally, the investor cannot receive a high yield and buy bonds in the same trace (*not coexistence*). The figure shows four constraints. Each constraint is automatically translated into LTL. In particular, the *response* constraint can be modeled as $\varphi_r = \Box(Low\_Risk \Rightarrow \Diamond Bonds)$; the *not coexistence* corresponds to $\varphi_n = \Diamond Bonds \Rightarrow \neg\Diamond High\_Yield$; the *alternate response* is translated into $\varphi_a = \Box(Money \Rightarrow \bigcirc(\neg Money \sqcup (Bonds \vee Stocks)))$; and the *precedence* constraint corresponds to $\varphi_p = \Diamond High\_Yield \Rightarrow (\neg High\_Yield \sqcup Stocks)$.

Unlike procedural languages, a Declare model allows for everything that is not explicitly forbidden. Removing constraints yields more behavior. The *not coexistence* constraint in Fig. 1 is difficult or even impossible to model in procedural languages. Mapping this constraint onto a procedural language forces the modeler to introduce a choice between *Bonds* and *High_Yield* (or both). Who makes this choice? When is this choice made? How many times will this choice be made? In a procedural language all these questions need to be answered, resulting in a complex or over-restrictive model.

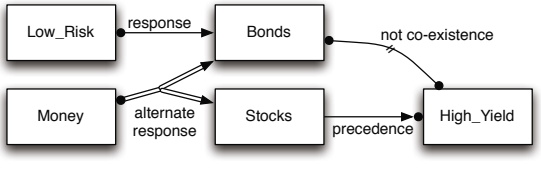


Fig. 1. Reference Model

2.2 LTL Semantics for Runtime Verification

Classically, LTL is defined for infinite traces. However, when focusing on runtime verification, reasoning is carried out on partial, ongoing traces, which describe a finite portion of the system's execution. Therefore, several authors defined alternative finite-trace semantics for LTL. Here, we use the four-valued semantics proposed in [2], called *Runtime Verification Linear Temporal Logic* (RV-LTL). Differently from the original RV-LTL semantics, which focuses on trace suffixes of infinite length, we limit ourselves to possible finite continuations. 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.

Let us denote with $u \models \varphi$ the truth value of an LTL formula φ in a partial finite trace u (according to FLTL [9], a variant of the standard semantics for dealing with finite traces). The semantics of $[u \models \varphi]_{RV}$ is defined as follows:

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

Note that when monitoring a business process using LTL, it rarely happens that a constraint is *permanently satisfied*. For the most part, business constraints are *possibly satisfied* but can be violated in the future. For this reason, in this paper, we make no difference between *permanently satisfied* and *possibly satisfied* constraints but we refer to both of them as *satisfied*. The following example explains how the above semantics can be used in practice to monitor a running process case.

Example 1. Let us consider the Declare model represented in Fig. 1. Figure 2 shows a graphical representation of the constraints' evolution: events are displayed on the horizontal axis. The vertical axis shows the four constraints.

Initially, all four constraints are satisfied. Let $u_0 = \varepsilon$ denote the initial (empty) trace:

$$[u_0 \models \varphi_r]_{RV} = \top^p \quad [u_0 \models \varphi_n]_{RV} = \top^p \quad [u_0 \models \varphi_a]_{RV} = \top^p \quad [u_0 \models \varphi_p]_{RV} = \top^p$$

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

$$[u_1 \models \varphi_r]_{RV} = \top^p \quad [u_1 \models \varphi_n]_{RV} = \top^p \quad [u_1 \models \varphi_a]_{RV} = \bot^p \quad [u_1 \models \varphi_p]_{RV} = \top^p$$

Note that $[u_1 \models \varphi_a]_{RV} = \bot^p$ because the *alternate response* constraint becomes possibly violated after the occurrence of *Money*. The constraint is waiting for the occurrence of another event (execution of *Bonds* or *Stocks*) to become satisfied

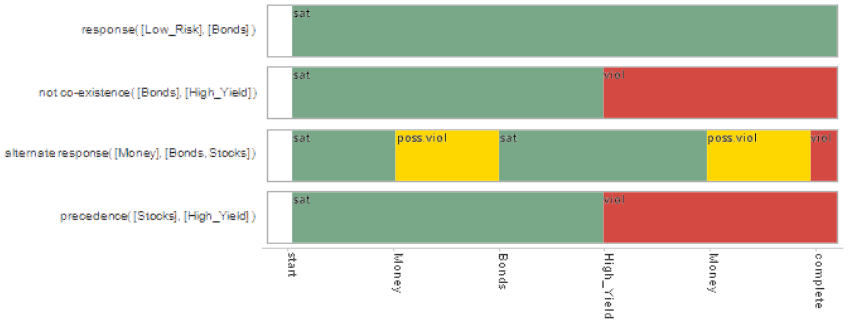


Fig. 2. One of the views provided by our monitoring system. The colors show the state of each four constraints while the process instance evolves; red refers to $\bot$, yellow refers to $\bot^p$, and green refers to $\top$ or $\top^p$.

again. Then, *Bonds* is executed ($u_2 = Money, Bonds$), leading to a situation in which constraint φ_a is satisfied again:

$$[u_2 \models \varphi_r]_{RV} = \top^p \quad [u_2 \models \varphi_n]_{RV} = \top^p \quad [u_2 \models \varphi_a]_{RV} = \top^p \quad [u_2 \models \varphi_p]_{RV} = \top^p$$

The next event is *High_Yield* ($u_3 = Money, Bonds, High\_Yield$), resulting in:

$$[u_3 \models \varphi_r]_{RV} = \top^p \quad [u_3 \models \varphi_n]_{RV} = \bot \quad [u_3 \models \varphi_a]_{RV} = \top^p \quad [u_3 \models \varphi_p]_{RV} = \bot$$

φ_n and φ_p become permanently violated because *Bonds* and *High_Yield* cannot coexist in the same trace. Moreover, the *precedence* constraint requires that *High_Yield* is *always* preceded by *Stocks* and this is not the case for trace u_3.

After reporting the violation, the monitoring system should continue to monitor the process. Suppose that the framework is able to provide continuous support and uses the strategy to simply ignore the violated constraints. If *Money* is executed again, i.e., $u_4 = Money$, *Bonds*, *High_Yield*, *Money*, φ_a becomes possibly violated again:

$$[u_4 \models \varphi_r]_{RV} = \top^p \quad [u_4 \models \varphi_n]_{RV} = \bot \quad [u_4 \models \varphi_a]_{RV} = \bot^p \quad [u_4 \models \varphi_p]_{RV} = \bot$$

However, this time the case completes its execution. We suppose that this is communicated to the runtime verifier by means of a special *complete* event. Using u_f to denote the resulting total trace, we obtain:

$$[u_f \models \varphi_r]_{RV} = \top \quad [u_f \models \varphi_n]_{RV} = \bot \quad [u_f \models \varphi_a]_{RV} = \bot \quad [u_f \models \varphi_p]_{RV} = \bot$$

Note that constraint φ_a that is the possibly violated when the case completes becomes permanently violated (because it cannot become satisfied anymore).

2.3 Translation of a Declare Constraint Model to Automata

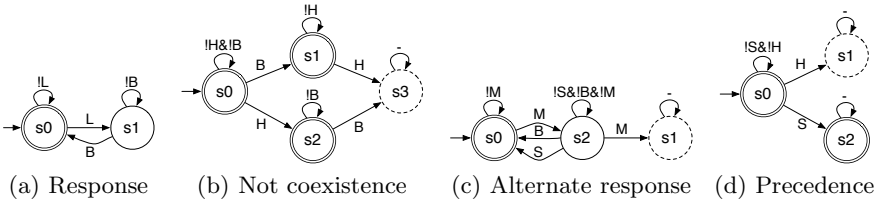
To automatically determine the state of each constraint of a Declare model during runtime, we construct a *deterministic finite state automaton* (we will

simply refer to such an automaton as "automaton"). The automaton accepts a trace if and only if it does not (permanently or possibly) violate the modeled constraint. We assume that constraints are specified in LTL (with a finite trace semantics). We use the translation in [6] for constructing the automaton.

For the constraints in the model in Fig. 1, we obtain the automata depicted in Fig. 3. In all cases, state 0 is the initial state and accepting states are indicated using a double outline. A transition is labeled with the initial letter of the event triggering it (e.g., we use the label L to indicate that the Low_Risk event occurs). For example, the *response* constraint automaton starts in state 0, which is accepting. Seeing an L (Low_Risk) we transition to state 1, which is not accepting. Only upon seeing a B ($Bonds$) do we transition back to state 0. This models our understanding of the constraint: when we execute Low_Risk we have to subsequently execute $Bonds$. As well as transitions labeled with a single letter, we also have transitions labeled with one or more negated letters (e.g., $!L$ for state 0 of the *response* constraint automaton and $!H\&!B$ for state 0 of the *not coexistence* automaton). This indicates that we can follow the transition for any event not mentioned (e.g., we can execute the event $High\_Yield$ from state 0 of the *response* automaton and remain in the same state). This allows us to use the same automaton regardless of the input language.

When we replay a trace on the 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 possibly violated. To support also the case where the constraint is permanently violated, we extend the original automaton (in fact, Fig. 3 already shows the extended version of the automaton) by connecting all the "illegal" transitions of the original automaton to a new state represented using a dashed outline (e.g., state 3 in the *not coexistence* constraint automaton). When we reach a state with a dashed outline during the execution of the automaton, we know that the constraint is permanently violated.

We can use these *local automata* directly to monitor each constraint, but we can also construct a single automaton for monitoring the entire system. We call such an automaton a *global automaton*. The global automaton is needed to deal with conflicting constraints. Conflicting constraints are constraints for which there is no possible continuation that satisfies them all. Note that even when all individual local automata indicate that the constraint is not permanently violated, there can still be conflicting constraints.



(a) Response (b) Not coexistence (c) Alternate response (d) Precedence

Fig. 3. Finite automata accepting traces satisfying (a) φ_r, (b) φ_n, (c) φ_a, and (d) φ_p

The global automaton can be constructed in different ways. The simplest way just constructs it as the automaton product of the local automata (or, equivalently, as the automaton modeling the conjunction of the individual constraints). [16] describes how to construct it efficiently.

The global automaton for the system under study is shown in Fig. 4. We use as state names the state numbers from each of the automata from Fig. 3, so state 1020 corresponds to constraint *response* being in state 1, constraint *not coexistence* being in state 0, and so on. 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. State $fail$ corresponds to all situations where it is no longer possible to satisfy all constraints. Note that state 1202 is not present in Fig. 4 even though none of the local automata is in a permanently violated state and it is in principle reachable from state 0202 via a L. The reason is that from this state it is never possible to ever reach a state where both *response* and *not coexistence* are satisfied, i.e., the two constraints are conflicting (in order to satisfy the first, we have to execute B which would move the latter to state 3). Executing the trace from Example 1 ($Money, Bonds, High\_Yield, Money, complete$), we obtain the trace $0000 \rightarrow^{M} 0020 \rightarrow^{B} 0100 \rightarrow^{H} fail \rightarrow^{M} fail$. Hence, we correctly identify that after the first two events all constraints are satisfied, but after executing $High\_Yield$ we permanently violate a constraint.

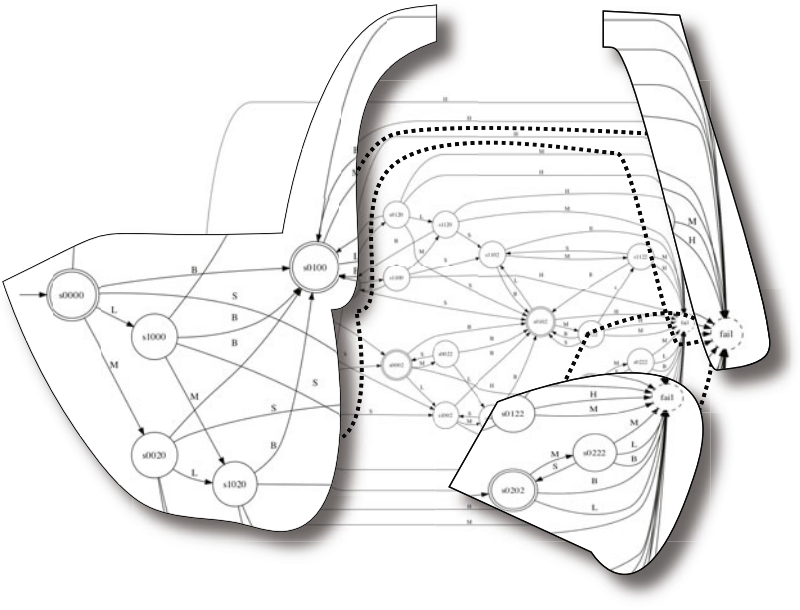


Fig. 4. Global automaton for the system in Fig. 1

3 Colored Automata for Runtime Verification

The global automaton in Fig. 4 allows us to detect the state of the entire system, but not for individual constraints. In order to combine local and global information in one single automaton, we use a *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. In effect, we color the states with a unique color for each constraint, assigning the color to the state if the constraint is satisfied. Figure 5 shows the colored automaton for our running example. We have indicated that a constraint is satisfied by writing the first letter of its name in upper case (e.g., in state 0000 we have colors RNAP and all constraints are satisfied) and that a constraint can be eventually satisfied by writing the first letter of its name in lower case (e.g., in state 1202 we have colors rNAP where constraint *response* is not satisfied, but it can be satisfied by executing *Bonds* and transitioning to state 0302). If the first letter of the name of a constraint does not appear at all, the constraint is permanently violated. Figure 2 already used such a coloring, i.e., red refers to $\bot$, yellow refers to $\bot^p$, and green refers to $\top$ or $\top^p$.

Comparing figures 4 and 5 shows that we now have many states with a dashed outline, i.e., states from which we cannot reach a state where all constraints are satisfied. This reflects our desire to continue processing after permanently

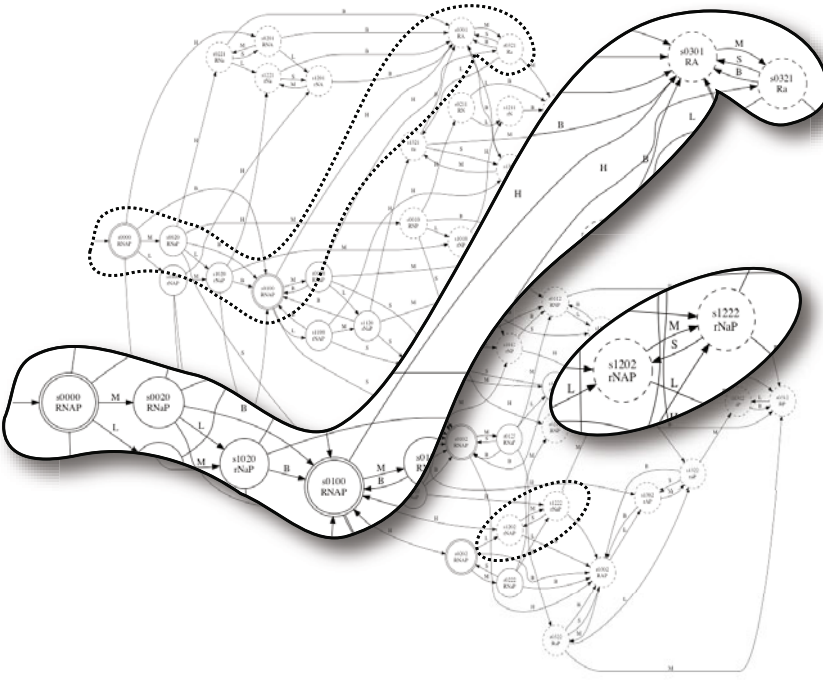


Fig. 5. Colored global automaton for the system in Fig. 1

violating a constraint. In fact, by folding all states with a dashed outline in Fig. 5, we obtain the original global automaton of Fig. 4. Note that states 1202 and 1222 have a dashed outline even though all constraints are satisfied or at least can be satisfied in successor states. This is because it is not possible to reach a state where all constraints are satisfied at the same time (we have basically asked for low risk, requiring investment in bonds as well as asked for high yield, which requires investment in stocks only). Executing the trace from Example 1 ($Money, Bonds, High\_Yield, Money, complete$), we obtain $0000 \rightarrow^{M} 0020 \rightarrow^{B} 0100 \rightarrow^{H} 0301 \rightarrow^{M} 0321$. The last state is colored with Ra, indicating that the *response* constraint is satisfied, the *alternate response* constraint is possibly violated, and the two remaining constraints are permanently violated.

Once the colored global automaton is constructed, runtime monitoring can be supported in an efficient manner. The state of an instance can be monitored in constant time, independent of the number of constraints and their complexity. According to [16], the time to construct the global or colored automaton is 5-10 seconds for random models with 30-50 constraints. For models larger than this, automata can no longer routinely be constructed due to lack of memory, even on machines with 4-8 GiB RAM.

4 Strategies for Continuous Support

As discussed in Sect. 1, the monitoring system should continue to provide meaningful diagnostics after a violation takes place. This section presents three ways of recovering from a violation. These have been implemented in ProM.

4.1 Recovery by Ignoring Violated Constraints

The first recovery strategy simply ignores a constraint after it is permanently violated. This was the approach we used for the trace discussed in Example 1. Figure 2 illustrates the desired behavior when this strategy is used for $Money, Bonds, High\_Yield, Money, complete$. The colored automaton directly supports this strategy. In Fig. 5, there are multiple states with a dashed outline to be able to monitor the state of all constraints after a violation.

4.2 Recovery by Resetting Violated Constraints

The second recovery strategy resets a constraint when it is permanently violated. Constraints that are not violated progress as before. Consider, for instance, trace $Money$, $Bonds$, $Money$, $Stocks$, $High\_Yield$, $Money$, $Stocks$, $High\_Yield$, $Money$, $Stocks$, $High\_Yield$, $complete$. The first four events can be executed without encountering any problem: $0000 \rightarrow^{M} 0020 \rightarrow^{B} 0100 \rightarrow^{M} 0120 \rightarrow^{S} 0102$. Executing $High\_Yield$ results in a failed state in the colored automaton: $0102 \rightarrow^{H} 0302$. The automaton modeling the *not coexistence* constraint is in state 3. Resetting the constraint results in global state 0002. The remaining events can be executed without any problem: $0002 \rightarrow^{M} 0022 \rightarrow^{S} 0002 \rightarrow^{H} 0202 \rightarrow^{M} 0222 \rightarrow^{S} 0202 \rightarrow^{H} 0202$.

Figure 6 shows the monitor in ProM using the reset strategy for this trace. After the first violation (corresponding to the first occurrence of $High\_Yield$), the *not coexistence* constraint is reset and no violations are detected anymore even when $High\_Yield$ occurs again (by resetting the constraint we do not preserve information about the previous occurrence of $Bonds$).

We can provide support for the reset strategy in two different ways: (a) by retaining the local automata for the system and translating back and forth between the global and local automata when an error is encountered or (b) by making an automaton specifically tailored to handle this strategy.

The first approach requires a mapping from states of the colored automaton to states of each of the local automata and vice versa. We can do this using a hash mapping, which provides constant lookup for this table. When we encounter a transition that would lead us to a state from which we can no longer reach a state where all constraints are satisfied (a dashed state in Fig. 5), we translate the state to states of the local automata. For instance, transition $0100 \rightarrow^{H} 0301$ during the trace $Money, Bonds, High\_Yield, Money$ from Example 1 results in a dashed state 0301. Two of the four local automata are in a permanently violated state. These automata are reset resulting in state 0000.

The second approach creates a dedicated *recovery automaton*. Figure 7 shows the recovery automaton for our running example. In this automaton, we take the colored automaton and replace any transition to an error (dashed) state with the state we would recover to, effectively precomputing recovery. We do this by translating any transition to a dashed state to a transition to the correct recovery state. In Fig. 7, we removed all dashed states, and introduced new states not previously reachable (here 0200, 1200, 0220, and 1220). We have handled recovery in states 1202 and 1222 by retaining both of the two conflicting (but not yet violated) constraints *response* and *not coexistence* and handling the conflict when the violation occurs.

From a performance point of view, a dedicated recovery automaton is preferable. Each step takes constant time regardless of the size of the original model. A possible disadvantage is its rigidity; the recovery strategy needs to be determined beforehand and the only way to switch to another recovery strategy is to generate a new recovery automaton.

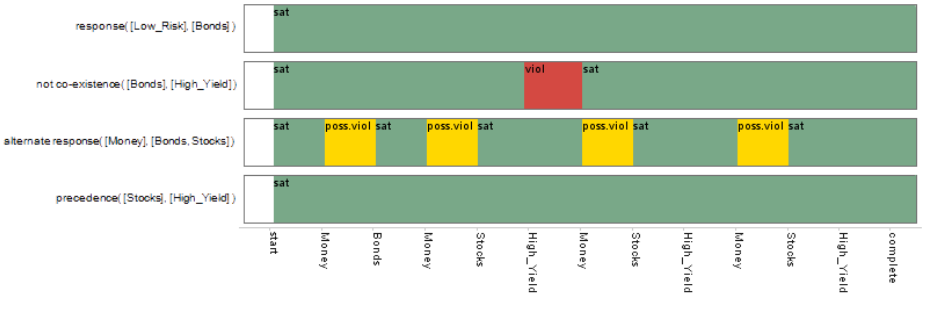


Fig. 6. Recovery by resetting violated constraints

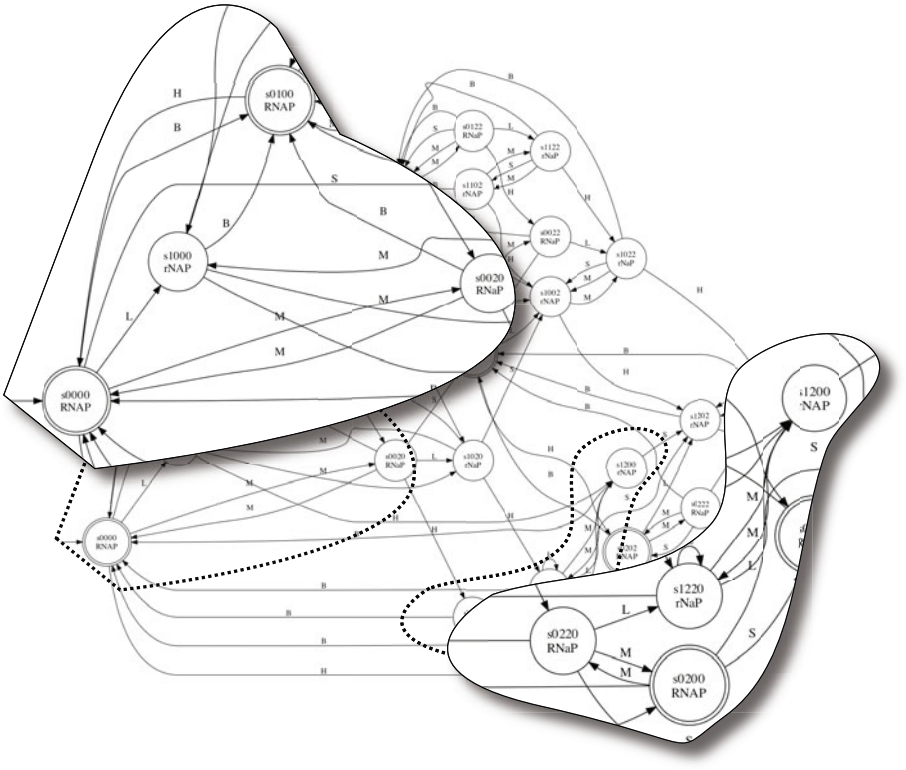


Fig. 7. Recovery automaton for the system in Fig. 1 using recovery strategy reset and retaining states for conflicting constraints

4.3 Recovery by Skipping Events for Violated Constraints

The third recovery strategy skips events for permanently violated constraints (but still executing it for non-violated constraints). Consider trace $Money$, $Bonds$, $Money$, $Stocks$, $High\_Yield$, $Money$, $Stocks$, $High\_Yield$, $Money$, $Stocks$, $High\_Yield$, *complete*. When $High\_Yield$ occurs for the first time, the *not coexistence* constraint is permanently violated. Under the *skip* strategy, this constraint is made again active, by bringing it back to the last state before the violation, i.e., the *not coexistence* constraint effectively ignores the occurrence of $High\_Yield$. In this way, when $High\_Yield$ occurs for the second time, the constraint is violated again and we have a third violation when $High\_Yield$ occurs for the third time. Figure 8 shows the monitor in ProM using the skip strategy for this trace. Figures 6 and 8 illustrate that the reset and skip strategies may produce different results; the former detects one violation whereas the latter detects three violations.

Similar to recovery by resetting violated constraints, it is possible to construct a dedicated recovery automaton using the skipping events for violated constraints. As a result, monitoring can be done in an efficient manner.

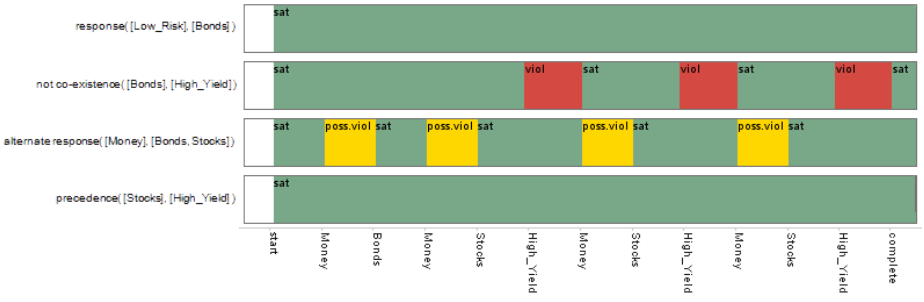


Fig. 8. Recovery by skipping events for violated constraints

In this section, we described three recovery strategies. There is no "best strategy" for continuous support. The choice strongly depends on the application domain and other contextual factors. Therefore, we have implemented all three approaches in ProM.

5 Runtime Modification

Thus far we assumed that the model does not change during monitoring. In workflow literature one can find many approaches supporting *flexibility by change* [15,13,5]. The basic idea is that the process model can be changed at runtime. This generates all kinds of complications (see for example the well-known "dynamic change bug" [13]). Models may change due to a variety of reasons, e.g., the implementation of new legislation or the need to reduce response times. This type of flexibility can easily be supported by Declare while avoiding problems such as the "dynamic change bug". Moreover, frequent violations of existing constraints may trigger model changes such as removing a constraint.

Consider for example a trace containing both *Bonds* and *High_Yield*, thus violating the *not coexistence* constraint, e.g., *Money*, *Bonds*, *Money*, *Stocks*, *High_Yield*, *Money*, *Low_Risk*, *Bonds*, *Money*, *Stocks*, *Bonds*, *complete*. Instead of applying one of the aforementioned recovery strategies, we could leverage on the possibility of modifying the model at runtime. In particular, we can perform runtime modification using the algorithms for dynamic modifications presented in [16]. The algorithms are able to update an automaton with changes (such as adding and removing constraints). This approach is much faster than regenerating the automaton from scratch.

In our example, we could remove the *not coexistence* constraint from the reference model adding at the same time a new *not succession* constraint, obtaining the Declare model shown in Fig. 9. After this modification, events *Bonds* and *High_Yield* can coexist but when *Low_Risk* occurs, *Stocks* cannot occur anymore. After the modification the trace is monitored w.r.t. the new model, leading to the result reported in Fig. 10. Note that in correspondence of the second violation (involving the *not succession* constraint), a strategy for continuous support is applied and the model does not change.

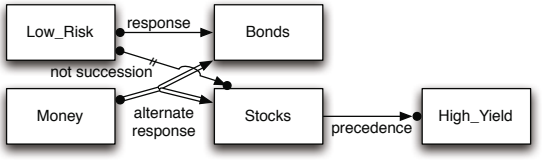


Fig. 9. Dynamic change of the model shown in Fig. 1

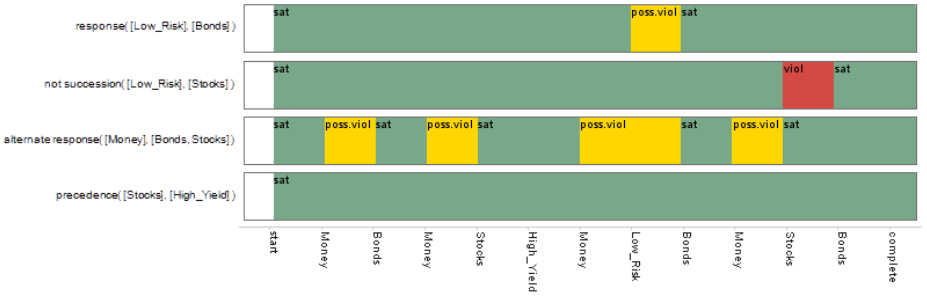


Fig. 10. Recovery by runtime change; at runtime the model shown in Fig. 1 is replaced by the model shown in Fig. 9

6 Related Work

Several BPM researchers have investigated compliance at the model level [1,7]. In this paper, we focus on runtime verification and monitoring based on the observed behavior rather than the modeled behavior. This has been a topic of ongoing research, not only in the BPM domain, but also in the context of software engineering and service oriented computing. Most authors propose to use temporal logics (e.g., LTL) and model checking [4]. We refer to the survey paper by Bauer et al. [2] for an overview of existing approaches. To implement runtime verification, classical automata-based model checking techniques must be adapted to reason upon *partial traces*. The monitored traces are *finite*, and also *subject to extensions* as new events happen, making it not always possible to draw a definitive conclusion about the property's satisfaction or violation. The approach presented in this paper is based on RV-LTL semantics [2] which is a version of LTL on finite strings tailored for runtime verification.

Our verification technique is inspired by [6], where the use of (a finite-trace version of) LTL is also considered to tackle runtime verification. In [6], a translation from arbitrary (next-free) LTL formulas is used to monitor any running system. The main difference with our approach is that we consider the monitor to be composed by several constraints, each of which can be violated, and we report and recover based on individual automata instead of the entire system.

Other logic-based approaches have been proposed to deal with runtime verification of running traces. The work closest to our approach is [3], where DecSerFlow (one of the constraint-based languages supported by Declare) is used to model service choreographies. Here a (reactive version) of the Event Calculus [8] is employed to provide the underlying formalization and monitoring capabilities. Unlike our approach, in [3], the interplay between constraints is not considered.

The approach presented in this paper has been implemented as an Operational Support (OS) provider in ProM. The OS framework in ProM can be used to detect, predict, and recommend at runtime. For example, [14] describes OS providers related to time. Based on a partial trace the remaining flow time is predicted and the action most likely to minimize the flow time is recommended.

7 Conclusion

Compliance has become an important topic in organizations that need to ensure the correct execution of their processes. Despite the desire to monitor and control processes, there are events that cannot be controlled. For example, it is impossible and also undesirable to control the actions of customers and professionals. Therefore, we propose a comprehensive set of techniques to monitor business constraints at runtime. These techniques are based on *colored automata*, i.e., automata containing both global information about the state of the entire system and local information about the states of all individual constraints. This automaton can be precomputed, thus making monitoring very efficient.

Since constraints may be permanently violated, it is important to recover after a violation (to still give meaningful diagnostics). We proposed and implemented three recovery approaches (ignore, reset, and skip). Moreover, we showed that it is possible to efficiently modify the constraint model while monitoring.

The approach has been applied in the Poseidon project where it has been used to monitor "system health" in the domain of maritime safety and security. Future work aims at more applications in typical BPM domains (banking, insurance, etc.). Moreover, we would like to further improve our diagnostics and include other perspectives (data, time, resources).

References

1. Awad, A., Decker, G., Weske, M.: Efficient Compliance Checking Using BPMN-Q and Temporal Logic. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 326–341. Springer, Heidelberg (2008)
2. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL Semantics for Runtime Verification. Logic and Computation 20(3), 651–674 (2010)
3. Chesani, F., Mello, P., Montali, M., Torroni, P.: Verification of Choreographies During Execution Using the Reactive Event Calculus. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 55–72. Springer, Heidelberg (2009)
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)

5. Dadam, P., Reichert, M.: The ADEPT project: a decade of research and development for robust and flexible process support. Computer Science - R&D 23(2), 81–97 (2009)
6. Giannakopoulou, D., Havelund, K.: Automata-Based Verification of Temporal Properties on Running Programs. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE 2001), pp. 412–416. IEEE Computer Society, Los Alamitos (2001)
7. Governatori, G., Milosevic, Z., Sadiq, S.W.: Compliance Checking Between Business Processes and Business Contracts. In: Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006), pp. 221–232. IEEE Computer Society, Los Alamitos (2006)
8. Kowalski, R.A., Sergot, M.J.: A Logic-Based Calculus of Events. New Generation Computing 4(1), 67–95 (1986)
9. Lichtenstein, O., Pnueli, A., Zuck, L.D.: The glory of the past. In: Proceedings of the Conference on Logic of Programs, London, UK, pp. 196–218. Springer, Heidelberg (1985)
10. Montali, M.: Specification and Verification of Declarative Open Interaction Models. LNBIP, vol. 56. Springer, Heidelberg (2010)
11. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: Full Support for Loosely-Structured Processes. In: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), pp. 287–300. IEEE Computer Society, Los Alamitos (2007)
12. Pesic, M., van der Aalst, W.M.P.: A Declarative Approach for Flexible Business Processes Management. In: Eder, J., Dustdar, S. (eds.) BPM Workshops 2006. LNCS, vol. 4103, pp. 169–180. Springer, Heidelberg (2006)
13. Schonenberg, H., Mans, R., Russell, N., Mulyar, N., van der Aalst, W.M.P.: Towards a Taxonomy of Process Flexibility. In: Bellahsene, Z., Woo, C., Hunt, E., Franch, X., Coletta, R. (eds.) Proceedings of the Forum at the CAiSE 2008 Conference. CEUR Workshop Proceedings, vol. 344, pp. 81–84 (2008)
14. van der Aalst, W.M.P., Pesic, M., Song, M.: Beyond process mining: From the past to present and future. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 38–52. Springer, Heidelberg (2010)
15. Weber, B., Rinderle, S., Reichert, M.: Change Patterns and Change Support Features in Process-Aware Information Systems. In: Krogstie, J., Opdahl, A.L., Sindre, G. (eds.) CAiSE 2007 and WES 2007. LNCS, vol. 4495, pp. 574–588. Springer, Heidelberg (2007)
16. Westergaard, M.: Better Algorithms for Analyzing and Enacting Declarative Workflow Languages Using LTL. In: Rinderle, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896. Springer, Heidelberg (2011)
17. Westergaard, M., Maggi, F.M.: Modelling and Verification of a Protocol for Operational Support using Coloured Petri Nets. In: Proc. of ATPN 2011 (2011)