Looking into the Future*

Using Timed Automata to Provide a Priori Advice about Timed Declarative Process Models

Michael Westergaard and Fabrizio Maria Maggi

Eindhoven University of Technology, The Netherlands
{m.westergaard,f.m.maggi}@tue.nl

Abstract. Many processes are characterized by high variability, making traditional process modeling languages cumbersome or even impossible to be used for their description. This is especially true in cooperative environments relying heavily on human knowledge. Declarative languages, like Declare, alleviate this issue by not describing what to do step by step but by defining a set of constraints between actions that must not be violated during the process execution. Furthermore, in modern cooperative business, time is of utmost importance. Therefore, declarative process models should be able to take this dimension into consideration. Timed Declare has already previously been introduced to monitor temporal constraints at runtime, but it has until now only been possible to provide an alert when a constraint has already been violated without the possibility of foreseeing and avoiding such violations. Moreover, the existing definitions of Timed Declare do not support the static detection of time-wise inconsistencies. In this paper, we introduce an extended version of Timed Declare with a formal timed semantics for the entire language. The semantics degenerates to the untimed semantics in the expected way. We also introduce a translation to timed automata, which allows us to detect inconsistencies in models prior to execution and to early detect that a certain task is *time sensitive*. This means that either the task cannot be executed after a deadline (or before a latency), or that constraints are violated unless it is executed before (or after) a certain time. This makes it possible to use declarative process models to provide a priori guidance instead of just a posteriori detecting that an execution is invalid.

Keywords: declarative process modeling, metric temporal logic, error detection, operational support, timed automata, Declare.

1 Introduction

Organizations work today in a dynamic, complex and interconnected world. Even in the heterogeneity of the environment where they operate, they need to execute

* 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.

R. Meersman et al. (Eds.): OTM 2012, Part I, LNCS 7565, pp. 250–267, 2012.

their processes in a trustworthy and correct manner. A compliance model is, in general, a set of *business constraints* that allow practitioners to discriminate whether a process instance behaves as expected or not.

During the execution of a business process it is often extremely important to meet deadlines and optimize response times, especially in cooperative environments where contracts among multiple parties need to be adhered to. To this aim, a compliance model can also include temporal constraints to guarantee the correct execution of a process in terms of *latencies* (related to events that cannot occur *before* a certain time, or must occur *after* a certain time) and *deadlines* (related to events that cannot occur *after* a certain time, or must occur *before* a certain time).

Nevertheless, the declarative nature of business constraints makes it difficult to use procedural languages to describe compliance models. First, the integration of diverse and heterogeneous constraints would quickly make models extremely complex and tricky. Second, business constraints often target uncontrollable aspects, such as activities carried out by internal autonomous actors (e.g., a doctor in a health-care process) or even by external independent entities (e.g., a web service in a service choreography). Representing this variability through a procedural model would require the explicit specification in the same model of multiple alternatives. Again, this would make models completely unreadable.

For this reason, in this paper, we represent business constraints using *Declare* [9,10,1,13]. Declare is a declarative language that combines a formal semantics grounded in Linear Temporal Logic (LTL) with a graphical representation for users. Differently from procedural models, a Declare model describes a process as a list of constraints that must be satisfied during the process execution. A Declare model is an "open world" where everything is allowed unless it is explicitly forbidden. In this way, Declare is very suitable for describing compliance models.

Nevertheless, the standard LTL semantics of Declare is not sufficient to represent metric temporal constraints, i.e., constraints that specify latencies and deadlines on the execution of the activities of a business process. Therefore, in order to monitor such a kind of constraints in Declare, it is necessary to represent them through a more expressive formal semantics. For example, in [8,5], the authors use the Event Calculus (EC). However, this approach allows users to identify a violation only *after* it has occurred and it is not possible to prevent violations from taking place. Moreover, by using the EC, it is not possible to detect violations that cannot be ascribed to an individual constraint but are determined by the interplay of two or more constraints.

To address these issues, the approach presented in this paper uses timed automata [2] instead of the EC to evaluate the compliance of a process instance w.r.t. a (timed) Declare model at runtime. In particular, to express metric temporal constraints in Declare, we extend the original LTL semantics of Declare with MTL (Metric Temporal Logic) [7,3], a real-time extension of LTL with quantitative temporal operators. MTL reasons over infinite traces. In contrast, traces in a business process are supposed to finish sooner or later. Therefore,

we use a variant of MTL for finite traces first introduced in [11]. This semantics produces as output a boolean value representing whether the current (finite) trace complies with the monitored property or not. In addition, we extend MTL for finite traces with the four valued semantics RV-MTL (Runtime Verification MTL), in order to respect the fact that it is not always possible to produce at runtime a definitive answer about compliance.

We monitor RV-MTL rules through timed automata. However, we show with some counterexamples that RV-MTL is undecidable, i.e., it is not possible to translate every RV-MTL rule to timed automata. For this reason, we restrict our perspective to the set of rules that we use to formally represent the semantics of Timed Declare. For this (limited) set of rules, we present automata to monitor them at runtime and check models a priori.

While evaluating the compliance of a running process instance w.r.t. a Declare model, users are allowed, using timed automata, to "look into the future" from two different perspectives. First of all, using timed automata, it is possible to generate a (red) alert to warn users that a constraint (or a combination of constraints) is going to be violated. In this way, they are advised to undertake specific actions within a specific lapse of time *before* the violation has occurred so that the violation can be avoided. We can also generate alerts with a lower severity (yellow or orange), i.e., alerts to warn users that a specific activity can currently be executed but, in the future, it will be (temporarily or permanently) forbidden. Secondly, our approach allows *early detection* of violations. In fact, using timed automata, it is possible to detect *non-local violations* when still none of the individual constraints in the compliance model has been violated. A non-local violation is a violation that cannot be ascribed to an individual constraint but is determined by the interplay of two or more constraints and indicates that (at least) one of them will be violated in the future.

The paper is structured as follows. Section 2 introduces some background notions about automata, timed automata and MTL. In this section, we also present RV-MTL. In Sect. 3, we present the semantics of Timed Declare and automata to check individual constraints. In Sect. 5, we outline our prototype implementation of Timed Declare, and in Sect. 6, we sum up our conclusions and provide directions for future work.

2 Background

In this section, we introduce some background material. We first present standard Declare using a running example. Then, we introduce timed automata and MTL (Metric Temporal Logic), the temporal logic we use in this paper. We also present a runtime version of MTL, which extends MTL to a four-valued logic for handling ongoing traces.

2.1 Declare and Running Example

Declare is a workflow language and tool [13] for modeling workflows using a declarative approach. Instead of specifying what has to be done, constraints

between tasks are specified. In Fig. 1, we see a simple Declare model (ignore the intervals for now) of an ordering process in a web shop. Here we have five tasks, specified using rectangles (e.g., Order) and five constraints, specified either using arrows or as the house annotation above Discount. The constraints specify when certain tasks are allowed or required. For example, we have a *precedence* constraint from Order to Pay, indicating that we can only pay after placing an order (we do not have to pay after placing an order, as we can cancel it though this is not explicitly modeled). We have a *succession* constraint from Pay to Delivery, indicating that if an order has been paid, it must be delivered, and it can only be delivered after successful payment. Furthermore, we can only get a discount if we order something (as specified by the *precedence* constraint from Order to Discount), but if we get a discount, we have to sign up for subsequent advertisement, as indicated by the *response* constraint from Discount to Advertisement. The house above Discount restricts how many times this task can occur; in this case the restriction is that it must occur zero or more times, which does not mean anything for an untimed version of the model, but which shall become useful later.

2.2 Timed Automata

A timed automaton augments standard finite automata with a set of clocks. Clocks all run at the same rate and are typically denoted by c. While we cannot control the progression of clocks, we can observe and reset them. We can also perform actions depending on *clock constraints*, which compare the value of a clock with any integer:

Definition 1 (Clock Constraints). *Given a single clock c, the set of* ***clock constraints*** *over c are*

$$\mathcal{B}(c) = \{c \sim n \mid n \in \mathbf{N}, \sim \in \{\leq, <, =, >, \geq\}\}.$$

A timed automaton extends standard finite automata by adding *invariants* as clock constraints to states and adding *guards* as clock constraints to transitions:

Definition 2 (timed Automaton). *A* ***timed automaton*** *is a septuple:*

$$\mathcal{TA} = (S, AP, C, \delta, I, s_I, A)$$

where S is a finite set of ***states*** *(also called* ***locations****), AP is a finite set of* ***labels****, C is a set of clocks, $\delta \subseteq S \times 2^{\mathcal{B}(C)} \times AP \uplus \{\tau\} \times 2^{\{C\}} \times S$ is the* ***transition***

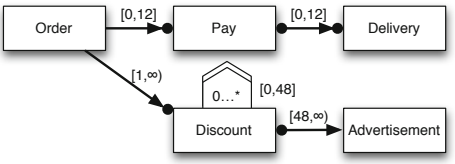

Fig. 1. Running example: Declare model consisting of four constraints

relation *where each transition has a set of clock constraints as* ***guards*** *and a label,* $I : S \to \mathcal{B}(C)$ *assigns* ***invariants*** *to states,* s_I *is the* ***initial state*** *and* $A \subseteq S$ *is the set of* ***accepting states****.*

The intuition is that time progresses at a constant and uncontrollable rate. We are only allowed to stay in a state as long as the state invariant holds and can only follow a transition if the guard constraint is true at the current time. The distinguished transition τ represents time passing and corresponds to an invisible transition, i.e., we can follow such a transition without consuming events as long as the guard constraint is true. When following a transition, we reset all clocks in the fourth component of the transition.

Often, we represent timed automata as directed graphs where nodes correspond to states and arcs to transitions. An example of a timed automaton is shown in Fig. 2. We have 4 labeled states and 8 transitions. We indicate the initial state using an unrooted arrow (s0) and accepting states using a double outline (s0 and s2). Invariants are shown in brackets below states (e.g., [xA <= a] below s1). Next to transitions, we show their labels, guards, and clocks to reset. For example, the transition from s2 to s0 has label τ (indicated by :tau), has guard yA > a, and resets no clocks. The transitition from s0 to s1 resets both xA and yA. We can have any number of guards and clock resets, including none (e.g., the transition from s0 to s3). The dashed state (s3) indicates that no accepting state is reachable from there.

We interpret timed automata over *timed sequences*, i.e., strings over $\mathbf{R}_+ \times AP$ where $\mathbf{R}_+ = \{x \in \mathbf{R} \mid x \geq 0\}$, denoted by $\sigma = (t_0, p_0)(t_1, p_1) \cdots (t_{k-1}, p_{k-1})$. We require that for $i < j$ we have $t_i \leq t_j$. The intuition is the same as for regular automata; we follow states from the initial state. However, we also introduce steps happening automatically due to time progressing.

Formally, we consider triples $(t, s, V) \in \mathbf{R}_+ \times S \times C_+^{\mathbf{R}}$, where the first entry is the absolute timestamp, the second is the state and the third includes the values of the clocks. We allow 2 kinds of transitions:

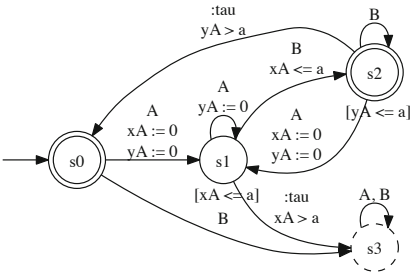

Fig. 2. Timed automaton for the $succession_{[0,a]}(A, B)$ constraint

$$(t,s,V) \to^{d} (t+d,s,V+d) \qquad \text{if } \forall t' \in [V,V+d], I(s)(t') = \text{true, and}$$
$$(t,s,V) \to^{a} (t,s',V') \qquad \text{if } (s,\gamma,a,C,s') \in \delta, \gamma(V) = \text{true, and}$$
$$I(s')(V') = \text{true with } V'(c) = \begin{cases} 0 & \text{if } c \in C, \text{ and} \\ V(c) & \text{otherwise.} \end{cases}$$

Note that the second case also includes invisible steps. If either of the two cases hold, we write $(t,s,V) \to (t',s',V')$. A *trace* $(t_0,s_0,v_0) \to (t_1,s_1,v_1) \to \cdots \to (t_{k-1},s_{k-1},v_{k-1})$ is *accepting* if $(t_0,s_0,V_0) = (0,s_I,\mathbf{0})$ and $s_{k-1} \in A$. A timed sequence $\sigma = (t_0,p_0)(t_1,p_1)\cdots(t_{k-1},p_{k-1})$ is accepting if there exists an accepting trace $T = (t'_0,s_0,V_0) \to (t'_1,s_1,V_1) \to \cdots \to (t'_{n-1},s_{n-1},V_{n-1})$ such that $\sigma = project(T)$ where *project* projects the trace onto the first two components and ignores τ steps.

2.3 Metric Temporal Logic

Metric Temporal Logic (MTL) is a logic talking about timed sequences of states. The idea is that we have a set of *atomic propositions*, denoted by $AP = \{p_0, p_1, \ldots, p_{n-1}\}$. For our variant of metric temporal logic, we look at timed sequences of *events* and assume that events fall in the set of atomic propositions. We deal with a fragment of MTL where all traces are finite. Therefore, we use the MTL semantics for dealing with finite timed sequences presented in [11].

To express MTL formulas, we use the syntax:

Definition 3 (MTL Syntax). *Formulas of MTL contain* ***atomic propositions*** *and are closed under negation, conjunction, disjunction, timed next operator, timed until operator, timed previous/yesterday operator and timed since operator, i.e., a* ***formula*** ψ *belongs to MTL if*

$$\psi ::= p \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \mathbf{X}_I\psi \mid \psi_1 \mathbf{U}_I \psi_2 \mid \mathbf{Y}_I\psi \mid \psi_1 \mathbf{S}_I \psi_2$$

where $p \in AP$, $\psi, \psi_1, \psi_2 \in MTL$, *and* $I \subseteq \mathbf{R}_+$ *is an interval.*

Let $\sigma = (t_0,p_0)(t_1,p_1)\cdots(t_{k-1},p_{k-1})$ be a finite timed sequence of states and let ψ be an MTL formula. We write $\sigma, i \models \psi$ to indicate that ψ holds at position i in σ. The semantics of $\sigma, i \models p$, $\sigma, i \models \neg\psi$, $\sigma, i \models \psi_1 \wedge \psi_2$ and $\sigma, i \models \psi_1 \vee \psi_2$ are as normally in propositional logic: p is true at position i in σ if $p = p_i$, $\neg\psi$ is true if ψ is not, $\psi_1 \wedge \psi_2$ is true if both ψ_1 and ψ_2 are and $\psi_1 \vee \psi_2$ if either is. We say that $\sigma, i \models \mathbf{X}_I\psi$, if (t_i,p_i) has a successor state ($i < k-1$) and $\sigma, i+1 \models \psi$ with $t_i + a \leq t_{i+1} \leq t_i + b$. Moreover, $\sigma, i \models \mathbf{Y}_I\psi$, if (t_i,p_i) has a predecessor state ($i > 0$) and $\sigma, i-1 \models \psi$ with $t_i - b \leq t_{i-1} \leq t_i - a$. We say that $\sigma, i \models \psi_1 \mathbf{U}_I \psi_2$, if $\sigma, j \models \psi_2$ for some $j \geq i$ with $t_i + a \leq t_j \leq t_i + b$ and $\sigma, l \models \psi_1$ for all $i \leq l < j$. Finally, $\sigma, i \models \psi_1 \mathbf{S}_I \psi_2$, if $\sigma, j \models \psi_2$ for some $j \leq i$ with $t_i - b \leq t_j \leq t_i - a$ and $\sigma, l \models \psi_1$ for all $j < l \leq i$. This semantics coincides with FLTL (LTL for finite traces) where only $I = \mathbf{R}_+$ is allowed.

We add syntactic sugar for the normal connectives, such as $\psi_1 \rightarrow \psi_2 \equiv (\neg\psi_1) \vee \psi_2$ and $\psi_1 \leftrightarrow \psi_2 \equiv (\psi_1 \rightarrow \psi_2) \wedge (\psi_2 \rightarrow \psi_1)$. We also add temporal syntactic sugar, $\mathbf{F}_I\psi \equiv \text{true}\mathbf{U}_I\psi$ (timed future operator), $\mathbf{G}_I\psi \equiv \neg(\mathbf{F}_I(\neg\psi))$ (timed globally operator), $\mathbf{O}_I\psi \equiv \text{true}\mathbf{S}_I\psi$ (timed once operator) and $\mathbf{H}_I\psi \equiv \neg(\mathbf{O}_I(\neg\psi))$ (timed historically operator). The intuition behind the future operator and the once operator is that ψ has to happen in the specified interval of time from now (in the future or in the past). The intuition behind the globally operator and the historically operator is that ψ has to hold for the entire interval (in the future or in the past).

2.4 RV-MTL: A Metric Temporal Logic for Runtime Verification

When focusing on runtime verification of MTL properties, reasoning is carried out on partial, ongoing traces, which describe a portion of the system's execution. Therefore, here, we extend MTL (for finite traces) with a four-valued semantics called *Runtime Verification Metric Temporal Logic* (RV-MTL). Differently from the original MTL semantics, which gives to the user only a boolean feedback (specifying whether a trace is compliant or not w.r.t. a given property), RV-MTL provides more sophisticated diagnostics.

Let $\sigma = (t_0, p_0)(t_1, p_1) \cdots (t_{k-1}, p_{k-1})$ be a finite timed sequence of states and let ψ be an MTL formula. The semantics of $[\sigma \models \psi]_{RV}$ is defined as follows:

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

3 Timed Declare

In this section we introduce a timed version of Declare. The version is similar to the one in [8], but we allow time on more constraints and instead give a semantics which collapse to the standard LTL semantics when removing time. We also introduce new constraints that are useful when dealing with time.

Returning to the running example in Fig. 1, we see that the constraints all have intervals next to them. This represents *when* things have to occur. For example, we indicate that payment has to be performed withing 12 time units after the initial order (for example, because orders without payment are purged after 12 time units), and shipment has to take place within 12 time units after payment. Processing a discount cannot occur earlier than 1 time unit after the order. Sending out advertisements is only performed 48 time units after the discount. Finally, we have a new constraint, *exclusive allowance*, which states that the discount can only be applied in the first 48 time units of the process.

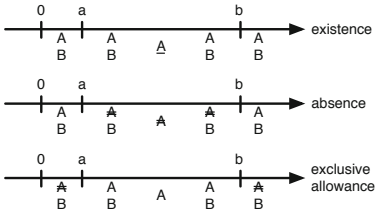

Fig. 3. Existence, Absence, and Exclusive Allowance

In Fig. 3, we give a graphical representation of the semantics for the timed existence and absence, and a new constraint which only makes sense in the timed version, exclusive allowance. The *timed existence* indicates that, starting from the beginning of a process instance, A must occur at least once (indicated by underline) at some point $t \in I$ where I is some interval from a to b (either of which may be included and b may also be ∞). A is allowed outside this interval (as is any other event, indicated by B in the figure. The *timed absence* specifies that A must not occur in the interval I (indicated by a double strikeout). A is allowed outside this interval. Where existence forces something to happen, it may also be useful to just allow something to happen in a specific interval, i.e., consider the conjunction of absence in the intervals before and after. This yields the *exclusive allowance*, which specifies that A is only allowed inside the interval I (e.g., *exclusive allowance*$_{[2,7]}(A) \equiv \wedge$*absence*$_{[0,2)}(A) \wedge$ *absence*$_{(7,\infty]}(A)$). In our example in Fig. 1, we use exclusive allowance to only allow for a discount within the first 48 time units (though here it is equivalent to absence after this interval).

Figure 4 shows a graphical representation of the semantics for the *timed responded existence*. This constraint indicates that, if A occurs at time t_1, B must occur at some point $t_0 \in [t_1 - b, t_1 - a]$ *or* $t_2 \in [t_1 + a, t_1 + b]$ (assuming $I = [a, b]$; if the interval is semi-open the intervals for t_0 and t_2 need to be updated accordingly). In the interval $[t_0, t_2]$ another A or another B can occur and, also, any event different from A and B (indicated by C in the figure). For the sake of readability, in this representation, we do not specify the behavior outside the interval $[t_0, t_2]$ where any event can occur. This semantics must be valid *for each* A in a process instance. The *timed co-existence* (which is not shown) is the conjunction of the timed responded existence with parameters (A, B) and the timed responded existence with parameters (B, A).

Figure 5 shows the semantics for the timed response, the timed alternate response and the timed chain response. The *timed response* indicates that, if A

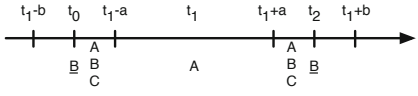

Fig. 4. Responded Existence

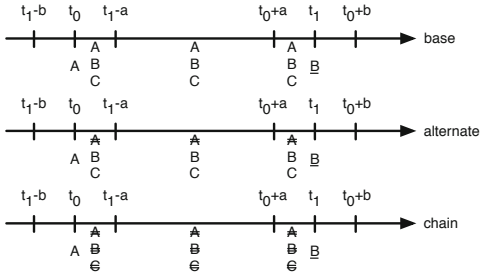

Fig. 5. Response, Alternate Response, Chain Response

occurs at time t_0, B must occur at some point $t_1 \in I$. Any event can occur inside this interval. In all representations in Fig. 5 (and also in the ones in Fig. 6) we do not specify the behavior outside the interval $[t_0, t_1]$, because outside this interval any event can occur. The *timed alternate response* specifies that if A occurs at time t_0, B must occur at some point $t_1 \in I$. A is not allowed in the interval $[t_0, t_1]$. Any event different from A is allowed. The *timed chain response* indicates that, if A occurs at time t_0, B must occur next at some point $t_1 \in [t_0 + a, t_0 + b]$. Nothing is allowed between A and B. Each of these constraints must hold for each A.

In Fig. 6, we give a graphical representation of the semantics for the timed precedence, the timed alternate precedence and the timed chain precedence. The *timed precedence* indicates that, if B occurs at time t_1, A must occur at some point $t_0 \in I$. Any event can occur between A and B. The *timed alternate precedence* specifies that, if B occurs at time t_1, an A must occur at some point $t_0 \in [t_1 - b, t_1 - a]$. B is not allowed in the interval $[t_0, t_1]$. Any event different from B is allowed. The *timed chain precedence* indicates that, if B occurs at time t_1, A must occur immediately before at some point $t_0 \in [t_1 - b, t_1 - a]$. Other events are not allowed in between.

The *timed succession*, the *timed alternate succession* and the *timed chain succession* (which are not shown in the table for the sake of readability) can be

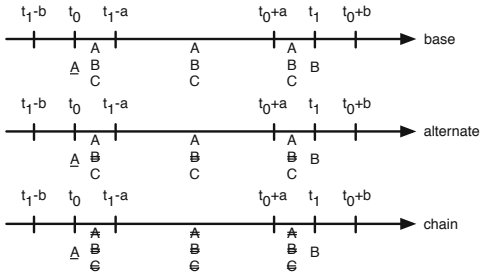

Fig. 6. Precedence, Alternate Precedence, Chain Precedence

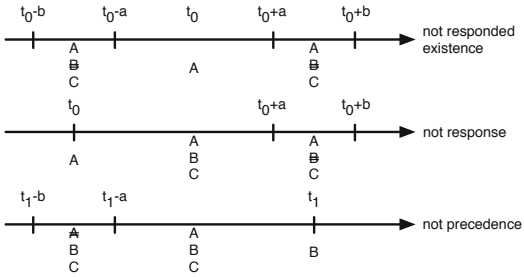

Fig. 7. Not Co-existence, Not Succession and Not Chain Succession

defined as the conjunction of the appropriate timed precedence and the timed response.

Figure 7 gives a graphical representation of the semantics for negations of constraints. The constraint *timed not responded existence* specifies that, whenever A occurs, B is forbidden in the specified interval before and after. Again, the *timed not co-existence* is not shown, but remains the conjunction of the timed not responded existence with parameters (A, B) and the timed not responded existence with parameters (B, A). The *timed not response* indicates that, whenever A occurs, B is forbidden in the time interval $t \in I$. The *timed not precedence* indicates that, whenever B occurs, A is forbidden in the time interval $t \in [t_B - b, t_B - a]$. The *timed not succession* is the conjunction of these last two constraints. The chain versions of the precedence and response constraints (not shown) allow B/A to occur inside the interval if any action occurs between them.

In Table 1, we summarize the timed semantics for each constraint. The semantics for the untimed constraints is the same as in [9], except we allow for the use of past operators, which makes specifying some constraints simpler. The timed versions are in most cases the same as for the untimed version except we add time. The (negated) responded existence is a bit more complicated to make the timing correct, but it is easy to see that this formula is equivalent to the corresponding untimed formula if time is removed.

3.1 Timed Automata for Declare

MTL is undecidable. It is not always possible to translate an MTL formula to a timed automaton. We show this with a counterexample. Consider, for instance, the timed semantics for the response constraint $\mathbf{G}(A \rightarrow \mathbf{F}_I B)$. We cannot express this semantics with a fixed number of clocks. Intuitively, we need to start a new timer for each A to make sure we that can see which (if any) of the As are satisfied by a given B. We have chosen to use MTL to specify Timed Declare anyway as we need the power to express the full semantics of Timed Declare. Other timed logics similar to LTL cannot express [4] the semantics for, e.g., timed responded existence and response.

Table 1. Semantics for some Declare constraints

Constraint	Untimed semantics	Timed semantics
existence	$\mathbf{F}A$	$\mathbf{F}_I A$
absence	$\neg\mathbf{F}A$	$\neg\mathbf{F}_I A$
exclusive allowance	–	$\neg\mathbf{F}_{[0,a]}A \wedge \neg\mathbf{F}_{[b,\infty]}A$
responded existence	$\mathbf{F}A \rightarrow \mathbf{F}B$	$\mathbf{G}(A \rightarrow (\mathbf{O}_I B \vee \mathbf{F}_I B))$
response	$\mathbf{G}(A \rightarrow \mathbf{F}B)$	$\mathbf{G}(A \rightarrow \mathbf{F}_I B)$
alternate response	$\mathbf{G}(A \rightarrow \mathbf{X}(\neg A\mathbf{U}B))$	$\mathbf{G}(A \rightarrow \mathbf{X}(\neg A\mathbf{U}_I B))$
chain response	$\mathbf{G}(A \rightarrow \mathbf{X}B)$	$\mathbf{G}(A \rightarrow \mathbf{X}_I B)$
precedence	$\mathbf{G}(B \rightarrow \mathbf{O}A)$	$\mathbf{G}(B \rightarrow \mathbf{O}_I A)$
alternate precedence	$\mathbf{G}(B \rightarrow \mathbf{Y}(\neg B\mathbf{S}A))$	$\mathbf{G}(B \rightarrow \mathbf{Y}(\neg B\mathbf{S}_I A))$
chain precedence	$\mathbf{G}(B \rightarrow \mathbf{Y}A)$	$\mathbf{G}(B \rightarrow \mathbf{Y}_I A)$
not responded existence	$\mathbf{F}A \rightarrow \neg\mathbf{F}B$	$\mathbf{G}(A \rightarrow (\neg\mathbf{O}_I B \wedge \mathbf{F}_I B))$
not response	$\mathbf{G}(A \rightarrow \neg(\mathbf{F}B))$	$\mathbf{G}(A \rightarrow \neg(\mathbf{F}_I B))$
not precedence	$\mathbf{G}(B \rightarrow \neg(\mathbf{O}A))$	$\mathbf{G}(B \rightarrow \neg(\mathbf{O}_I A))$
not chain response	$\mathbf{G}(A \rightarrow \neg(\mathbf{X}B))$	$\mathbf{G}(A \rightarrow \neg(\mathbf{X}_I B))$
not chain precedence	$\mathbf{G}(B \rightarrow \neg(\mathbf{Y}A))$	$\mathbf{G}(B \rightarrow \neg(\mathbf{Y}_I A))$

The uncomputability is only present if we wish to translate a formula to an automaton for static analysis. For on-line monitoring, we can easily instantiate a timer every time we activate a constraint. We can do this in terms of automata or as in [8] in terms of the Event Calculus. The desire to provide a meaningful timed semantics for Declare, even though we lose some analytical power is what prompts us to go with the undecidable logic.

As not every MTL formula can be translated to an automaton and not every Timed Declare constraint can be represented using a timed automaton, we need to restrict what we allow if we are to do analysis. If we restrict all intervals to either include 0 or go to ∞ it turns out that all constraints can be represented as an automaton. The reason is that it now becomes enough to remember the first and last time we saw each event for each constraint. Therefore, for an event, A, we introduce two clocks xA and yA; xA keeps track of the first outstanding A and yA keeps track of the last. Note that $\mathit{succession}_{[0,b]}(A,B) \wedge \mathit{succession}_{[a,\infty)}(A,B) \neq \mathit{succession}_{[a,b]}(A,B)$, so this does not contradict that we cannot construct the automaton for the right side of the expression. The left side of the expression can be satisfied using two As or two Bs, each in one interval but not in the other, but the right side does not admit this.

The automaton in Fig. 2 checks the $\mathit{succession}_{[0,a]}(A,B)$ constraint. From the initial state s0, we can take an A to s1 and a B to s3. State s3 is an inescapable non-accepting state and can be thought of as a failure (indicated by a dashed outline). This makes sense: if we see a B before and A, we have violated the constraint. For all other events, we just remain in s0. When we see an A from s0, we reset the clocks xA and yA, indicating when we first and last saw an A (namely now). We can stay in s1 as long as xA<=a, i.e., until it is long enough ago we saw an A that executing a B is mandatory; if we do not progress before that, we are forced to follow the τ transition to s3. Intuitively s1 means "we have seen As that are not followed by Bs", s2 means "we have seen As, all obligations are satisfied, and the last A is close enough that we may still execute Bs", and

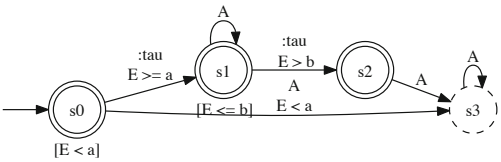

Fig. 8. Timed automaton for the *exclusive allowance*$_{[a,b]}(A)$ constraint

s0 means "we have no outstanding As and no A is close enough that we may execute Bs". Thus, we reset when we last saw an A in s1 and transition to s2 if we see a B. We may only stay in s2 as long as the last A is close enough (the invariant on s2, yA <= a); if we see an A we have a new outstanding A and move to s1 resetting both clocks. We allow for executing Bs, but when the invariant no longer is satisfied, we transition to s0 and start from scratch.

We employ a similar technique for all precedence, response, succession, and response constraints. The existence, absence, and exclusive allowance can trivially be checked in their full generality using a single clock for each constraint (see, e.g., Fig. 8 where the automaton for exclusive allowance is shown and uses only clock E). Precedence and response constraints can be checked using a single clock (we only need to keep track of either the first or last occurrence of A depending on whether the interval includes 0 or ∞; see Figs. 9 and 10 for examples). We can also represent the alternate response and the three chain constraints in their full generality. The intuition is that we can no longer execute more As between A and B (cf., Figs. 5 and 6).

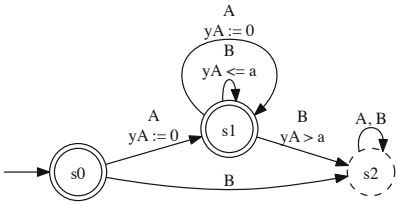

Fig. 9. Timed automaton for the *precedence*$_{[0,a]}(A, B)$ constraint

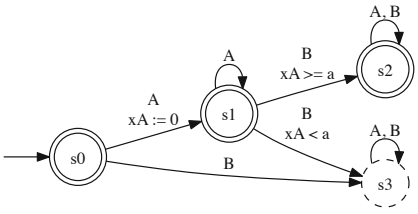

Fig. 10. Timed automaton for the *precedence*$_{[a,\infty)}(A, B)$ constraint

4 Analysis

In this section we show how to use automaton-based Timed Declare for analysis purposes to provide alerts when tasks are time-sensitive and to implement an a priori check of whether it is possible to meet deadlines.

4.1 Colored Alerts to Provide a Priori Advice

To monitor a Timed Declare model, we translate the model into a timed automaton. We simply instantiate our timed automata for each constraint. It is possible to compute the product of timed automata efficiently [2] so we do that in a way similar to how we construct colored automata for untimed Declare models. Such automata contains information about acceptance for each constraint in isolation and also about the acceptance of the conjunction. In the untimed version, this allows us to discover an inevitable violation even though no violation has yet taken place. The goal is to extend this to temporal properties as well.

Using a timed automaton we allow users to have relevant feedback during the process execution. During the execution of the process, this automaton can be used to give advice to the users about the action to undertake to obey the latencies and the deadlines specified by the compliance model, thus preventing possible violations from taking place. This advice is given through alerts that can be associated to different colors: *yellow* for alerts with low severity, *orange* for alerts with medium severity and *red* for alerts with high severity.

A *red alert* is generated when the automaton is in a consistent state, but letting time pass will unavoidable lead to violating one of the constraints. For instance, in the automaton in Fig. 2, modeling the succession constraint (in our example think of the succession from Order to Pay), if A (Pay) is executed (moving to state s1), a red alert is generated for the execution of B (Delivery) within 12 time units (as a$=12$). If Delivery is not executed, the constraint is violated (delivery has not taken place on time). Such an alert is generated for a state with an invariant where the only τ transition leads to a failure (dashed) state.

An *orange alert* is generated when the automaton is in a state where an activity can currently be executed but, after a certain number of time units, it cannot be executed anymore (and never in the future). For instance, in the automaton in Fig. 8 modeling the exclusive allowance constrains (in our model think of the Discount which is only available for the first 48 time units), when monitoring starts, we immediately transition to s1 (as a$=0$). Then, an orange alert is generated for A (Discount, because the customer missed out on the limited-time discount). The alert is generated because in state s1 it is possible to execute A, but s1 has an invariant and a τ transition to s2 from which executing A will always lead to a failure state. Alternatively, we generate such an alert if we are in a state where an action is guarded and can no longer be executed in any successor state and the clocks in the constraints cannot be reset.

A *yellow alert* is generated when the automaton is in a state where an activity can currently be executed but, after a certain number of time units, it cannot be

executed anymore. However, there is still the possibility to execute the activity somewhere in the future. For instance, in the automaton in Fig. 9 modeling the precedence constraint (in our example, we have a precedence of this type from Order to Pay). If A (Order) is executed, a yellow alert is generated to execute B (Pay) within 12 time units (as a= 12). This is because in state s1 we can execute Pay but if we let time pass, we can no longer satisfy the guard and there is no τ transition to a state where we can. We also generate such an alert if we are in a state with an invariant and a τ transition to a state where the event is not enabled.

4.2 Constraint Interaction

Constraint interaction can be checked by computing the strongly connected components (SCCs) of the automaton, marking failure states (dashed states in the examples), and for each state compute which events lead to non-failure states, called the *enabled events*. We compute the union of these sets for each strongly connected component, and propagate them backwards in the SCC graph. We call these the *possible events*. Now, as we traverse the automaton, we can identify each of the three kinds of alerts and notice that after payment, delivering is very important (red alert) to avoid violating a constraint, applying for the discount is of medium importance (orange alert) as it becomes unavailable after some time, and after ordering, payment is important to avoid erasure of the order.

When we just start the process, we may not realize that we are on the clock. In our example, we actually have to hurry with our order, because if we do not order within 47 time units, we cannot wait one time unit more and apply for the discount before it becomes unavailable at time 48. We thus wish for an orange alert for Order in the initial state even though Order itself does not become permanently unavailable. We cannot see this from automata from individual constraints, but the product automaton is needed. In Fig. 11, we see the product of the automata for *precedence*$_{[1,\infty)}$(Order, Discount) and *exclusive allowance*$_{[0,48]}$(Discount). We have hidden the failure state for legibility; anytime Order or Discount is not explicitly possible, they lead to the failure

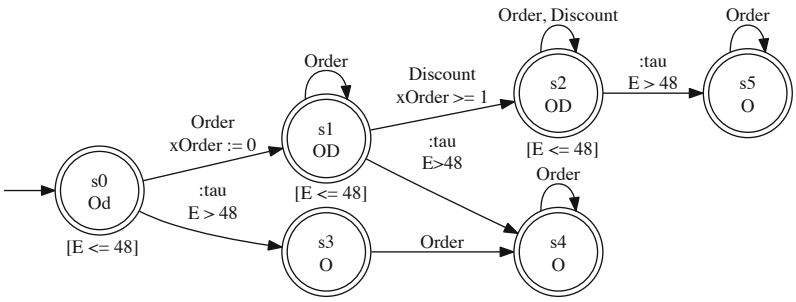

Fig. 11. Timed automaton for the conjunction of the *exclusive allowance*$_{[0,48]}$(Order) and *precedence*$_{[1,\infty)}$(Order, Discount) constraints

state. We have also annotated the states with the enabled and possible events. An O means that Order is enabled, a D means that Discount is enabled, and d means that Discount is possible. We see that in the initial state, Discount is possible, but the state has an invariant and a τ transition to a state where Discount no longer is possible. We can see that while Discount is possible, it is not enabled, so producing an orange alert would be of little use. Instead, we produce an orange alert for Order as we can see it leads us to another strongly connected component where Discount still is possible and even enabled in this case. The orange alert now moves to Discount where it rightly belongs.

4.3 Detection of Inconsistencies

If we modify the model in Fig. 1, adding a *not succession* constraint from Order to Deliver with a time limit of 36 time units, we model that we may not deliver goods within 36 time units from the order(e.g., due to local tax laws). We can see that Pay has to be executed no more than 12 time units from Order and Deliver no more than 12 time units from Pay, forcing delivery within 24 time units of Order. This of course conflicts with the new constraint, and we get an automaton accepting the empty language. We can detect this and point out the conflict between the 3 constraints, and let the user alleviate it by removing one or loosening one of the temporal constraints.

5 Implementation

In this section, we briefly describe our prototype implementation of Timed Declare in UppAal [12], a tool for analysis of timed automata.

UppAal makes it possible to design a model by defining process templates. We have designed a process template for each of the (most commonly used) constraints in Timed Declare. One such an example is shown in Fig. 12. In this example, we have a template in the field to the left for each constraint. The model shown is the UppAal implementation of the automaton from Fig. 2 testing the $succession_{[0,a]}(A, B)$ constraint. UppAal implements automata slightly differently from what we want. Most importantly, it does not have a notion of accepting states nor of synchronization. This is possible to get around, though.

To get around lack of synchronization, we use *broadcast channels*, which make it possible for a single sender to synchronize with multiple recipients. We then have a single driver (see Fig. 13) acting as sender and all the templates act as receivers and hence progress as the driver dictates. The driver (Fig. 13 (left)) has two states, an initial state and a termination state. The driver is a real flower-model, which allows for any (here of four) actions looping in the initial state. Each time, we transmit on the corresponding broadcast channel (e.g., a!). At some point, the driver decides to terminate, and communicates on done. If we look back at the implementation of Fig. 2 in Fig. 12, we see that many events receive on channels (e.g., A?). This will be synchronized with other automata listening to the same channel. Furthermore, channels cannot advance without receiving, so they just stay put for events that do not affect them.

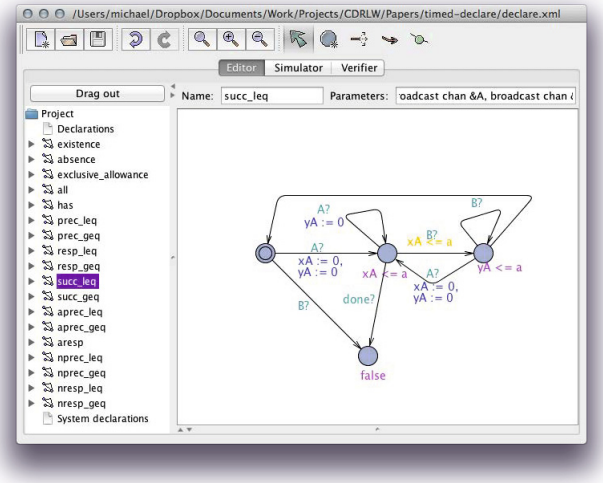

Fig. 12. Our prototype in UppAal

We handle non-accepting states by indicating that they forbid communicating on done. In our example, we see this construction from the middle state in Fig. 12. It has a transition receiving on done but leading to a state with the invariant false underneath it. This means that it is never a legal move to go there, so if the automaton in Fig. 12 is in the middle state, the driver is prevented from sending on done and hence terminating.

Listing 1. System declarations for the model in Fig. 1

```
1  broadcast chan done, Order, Pay, Delivery, Discount, Advertisement;

3  c1 = prec_leq(Order, Pay, 12);
4  c2 = succ_leq(Pay, Delivery, 12);
5  c3 = prec_geq(Order, Discount, 1);
6  c4 = exclusive_absence(Discount, 0, 48);
7  c5 = response_geq(Discount, Advertisement, 48);
8  a = all(Order, Pay, Delivery, Discount, Advertisement);

10 system a, c1, c2, c3, c4, c5;
```

Finally, we need to tie our symbolic names (A, B, and a) in Fig. 12 to actual tasks. We notice near the top right of Fig. 12 that we have a field for parameters. We here state that A and B are broadcast channel input parameters, and we have declared a as an integer input parameter. We then tie the entire system together in the System declarations, (the description is as simple as Listing 1). We first set up a channel for each task (l. 1), then we instantiate the individual constraints (ll. 3–7) and the driver (l. 8), and finally we start the system (l. 10). Now the formal names are tied to actual names and we can run the model in the simulator

Fig. 13. Our driver (left) and test (right)

in UppAal. We can also perform analysis. We can check if the model is non-empty by checking if the driver (Fig. 13 (left)) can reach the finished state. We can also instantiate the test in Fig. 13 (right) for each task and, when the found state is reachable, indicating that it is possible to execute that event.

6 Conclusion

In this paper, we have introduced a timed version of Declare. Our version is similar to the one in [8], but allows the use of time for more Declare constraints. We give a semantics in terms of MTL, a timed version of LTL, and we represent these semantics through timed automata. We show how we can use these automata to not only identify when a constraint is violated like in [8], but even to provide a priori warnings that time constraints may be violated in the future or that certain actions may become unavailable if not executed swiftly. We can also detect that deadlines are impossible to meet prior to execution.

In this paper, we have considered tasks without duration taking place with time spans between them. We are very interested in looking into giving tasks duration. This can be done either by considering the start and completion of a task as separate events or by looking at tasks as signals instead of events. When we do so, it is obvious to start looking at the resource perspective as well, as it may be that a model cannot be executed by a single person (for example if two 14 time unit tasks have to be executed within a 24 time unit period). For these cases we can compute interesting statistics like how fast can a model be executed given infinite resources, how fast can it be executed (if at all) using a given amount resources, and how many resources are necessary to execute a model. We believe that this can be extended to also provide plans for individual resources, and we believe we can extend this to do planning for running multiple instances of multiple models. This is very similar to providing operational support (except where operational support tries to answer similar questions on-the-fly, we try to answer them before the fact).

Here, we have used timed automata because they make it possible and easy to do sophisticated analysis. It would also be very interesting to investigate how moving to more advanced automata admitting creating fresh clocks skews the balance between expressiveness and analysis.

We would also like to integrate the presented analysis facilities in Declare [13], preferably in a backwards compatible way. One way to do that is to integrate UppAal's command line tool, which may definitely be good for analysis, but less optimal for on-the-fly execution, as UppAal computes the product on-the-fly while checking properties. We can, therefore, not precompute the enabled and possible events, which is necessary to be able to provide orange and yellow alerts. Another possibility is to use UppAal's DBM library, which implements difference-bound matrices [6] (a very efficient data-structure to implement timed automata), and to leverage the automaton library already available in Declare.

References

1. van der Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative workflows: Balancing between flexibility and support. Computer Science - Research and Development 23, 99–113 (2009)
2. Alur, R., Dill, D.: A Theory of Timed Automata. Theoretical Computer Science 126(2), 183–235 (1994)
3. Alur, R., Henzinger, T.: Real-time logics: complexity and expressiveness. In: Proceedings of Fifth Annual IEEE Symposium on Logic in Computer Science, LICS 1990, pp. 390–401 (June 1990)
4. Bauer, A., Leucker, M., Schallhart, C.: Comparing ltl semantics for runtime verification. Logic and Computation, 651–674 (2010)
5. 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)
6. David, D.: Timing Assumptions and Verification of Finite-state Concurrent Systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
7. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems 2, 255–299 (1990), http://dx.doi.org/10.1007/BF01995674, 10.1007/BF01995674
8. Montali, M.: Specification and Verification of Declarative Open Interaction Models. LNBIP, vol. 56, pp. 1–383. Springer, Heidelberg (2010)
9. Pesic, M.: Constraint-Based Workflow Management Systems: Shifting Controls to Users. Ph.D. thesis, Beta Research School for Operations Management and Logistics, Eindhoven (2008)
10. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: Declare: Full support for loosely-structured processes. In: IEEE International EDOC Conference 2007, pp. 287–300 (2007)
11. Thati, P., Roşu, G.: Monitoring algorithms for metric temporal logic specifications. Electron. Notes Theor. Comput. Sci. 113, 145–162 (2005), http://dx.doi.org/10.1016/j.entcs.2004.01.029
12. UppAal webpage, http://www.uppaal.org
13. Westergaard, M., Maggi, F.: Declare: A Tool Suite for Declarative Workflow Modeling and Enactment. In: Ludwig, H., Reijers, H. (eds.) Business Process Management Demonstration Track (BPMDemos 2011). CEUR Workshop Proceedings, vol. 820. CEUR-WS.org (2011)