An automaton-based approach to the verification of timed workflow schemas

Elisabetta De Maria, Angelo Montanari, Marco Zantoni
University of Udine, Italy
{demaria,montana,zantoni}@dimi.uniud.it

Abstract

Nowadays, the ability of providing an automated support to the management of business processes is commonly recognized as a main competitive factor for companies. One of the most critical resources to deal with is time, but, unfortunately, the time management support offered by most workflow systems is rather limited. In this paper we focus our attention on the modeling and verification of workflows extended with time constraints. We propose timed automata *as an effective tool to specify timed workflow schemas and to check their consistency.*

1. Introduction

Nowadays, workflow system ability of providing an automated support to the management of business processes is widely recognized as a main competitive factor for companies [17]. There exist sophisticated tools for the specification and validation of workflow schemas, that allow one, for instance, to detect inconsistencies in process constraints and to identify process bottlenecks, as well as tools for monitoring and managing process execution. One of the most critical resources to deal with is time. Unfortunately, the time management support offered by most workflow systems is rather limited. The few existing approaches are all based on graph-theoretic techniques [4, 6, 11, 13, 14, 15], that basically extend workflow graphs by embedding time-related information about activities into graph activity nodes and/or by adding time edges that express time constraints between activities.

One of the most elaborated approaches to timed workflow schema is the one proposed by Eder et al. in a series of papers [6, 7, 8] (a similar approach has been followed by Sadiq et al. [15]). It exploits an extension of the Critical Path Method, commonly used in project planning, to associate a range of possible execution durations with every activity. Moreover, different scenarios are taken into account by introducing conditional and optional activities. For every scenario, the algorithm computes the earliest time at which a given activity can terminate and the latest time at which it can terminate to guarantee minimal execution time for the whole process. Times associated with the last activity to be executed (without loss of generality, we may assume that there exists a single final activity) give information about the duration of the whole process. By comparing time information associated with the internal activity nodes with information about the duration of the whole process one can detect execution paths that may lead to time violations. The authors outline a number of techniques to reason about timed workflow graphs at build time, instantiation time, and execution time. In particular, they present an algorithm for consistency checking at build time which looks for a workflow execution that satisfies all time constraints. A different solution to the problem of checking the consistency (schedulability) and boundedness of workflow schemas extended with time constraints has been recently proposed by Li et al. [11]. To deal with timed workflows, they refine Petri-nets into Timing Constraint WorkFlow nets (TCWF nets) and they solve the schedulability and boundedness problems for a meaningful class of TCWF nets (acyclic and free-choice). As a matter of fact, the only external time constraints they consider are those between events that terminate activities and events that initiate their successor activities.

In this paper we focus our attention on the modeling and verification of workflows extended with time constraints. We propose *timed automata* as an effective tool to specify workflow schemas with time constraints and to check their consistency [12]. More precisely, we reduce the consistency problem for these workflow schemas to the emptiness problem for timed automata, making it possible to exploit the machinery developed to solve the latter to address the former. Furthermore, we take advantage of such a reduction to solve other relevant problems, such as, for instance, the problem of checking whether there exists an execution of a consistent workflow that satisfies some specific conditions (e.g., passing through some specific subpaths). From a technical point of view, we first show how the basic workflow constructs for activity composition can be rendered in terms of the automata operations of concatenation, union, product, and intersection. Then, we show how the time constraints of a timed workflow schema can be encoded into constraints on the finite set of real-valued clocks of a timed automaton. Putting together these two ingredients, we define a translation algorithm that maps a large class of timed workflow schemas into a decidable class of timed automata. Finally, we apply the proposed approach to a concrete example. Possible extensions to the proposed solution are briefly discussed in the conclusions.

2. Workflow schemas with time constraints

A workflow is a collection of activities, agents, and dependencies between activities [6, 7, 8]. Activities correspond to individual steps in a business process, agents (software systems or humans) are responsible for the enactment of activities, and dependencies determine the execution sequence of activities and the data flow between them. Activities can be executed sequentially, repeatedly in a loop, or in parallel. Parallel executions can be unconditional (all activities are executed), conditional (only activities that satisfy a certain condition are executed) or alternative (only one activity among several alternative ones is executed). In addition, workflows may contain optional activities (activities that may be executed or not). Workflows can naturally be modeled as *graphs*, whose nodes represent activities and whose edges capture dependencies between activities. An *and-split* denotes an activity with two or more immediate successor activities, all of which are executed in parallel. A *conditional split* denotes an activity whose immediate successor activities to be executed (one or more) are determined by evaluating some Boolean expression. An *or-split* denotes an activity whose immediate successor activity to be executed (exactly one) is chosen according to workflow policies. An example of a workflow graph is given in Figure 1. First, activity A takes place; then either B or C are executed (or-split). D is followed by E, F, or both E and F (conditional split). E is followed by L, and G is possibly executed in between E and L (G is optional). F is followed by H and I, which are executed in parallel once F has been completed (and-split). When both H and I terminate, M starts its execution. N is the final activity. Then, the whole process is repeated.

The control structure of the workflow implicitly defines a number of (qualitative) time constraints, called *structural time constraints*, which constrain an activity to start only when its predecessor activities have been completed. *Explicit* (qualitative or quantitative) *time constraints* can be added to take into account time restrictions on activities imposed by organizational rules, laws, and commitments. Explicit time constraints can be associated with the start or the end of an activity A_2 to constrain the time elapsed from the start or the end of an activity A_1. As an example, if we assume that A_1 takes place before A_2, by constraining the time elapsed from the end of A_1 when A_2 starts, we can impose a condition on the delay between A_1 and A_2. If we take $A_1 = A_2$, by constraining the time elapsed from the start of A_1 when A_1 ends, we can impose a condition on the duration of A_1. Explicit time constrains can also be used to constrain the start of an activity A_3 to the temporal order or temporal distance between the start or the end of two (other) activities A_1, A_2. To express these constraints, we associate two events, called *start* and *end* events, with an activity, which respectively initiate and terminate the activity. Following Eder et al. [7], we assume the duration of an activity to be deterministic (it can be viewed as the expected execution duration) and expressed in some basic time unit, and we consider only constraints between the occurrence times of *end* events. In such a way, the time elapsing between the execution of two activities and the variations in the duration of an activity can be dealt with in a uniform way. Both the case in which an activity is delayed/anticipated with respect to its scheduled time and the case in which it takes longer/shorter than its expected duration can indeed be modeled by constraining the temporal relations between its end time and the end times of the preceding activities.

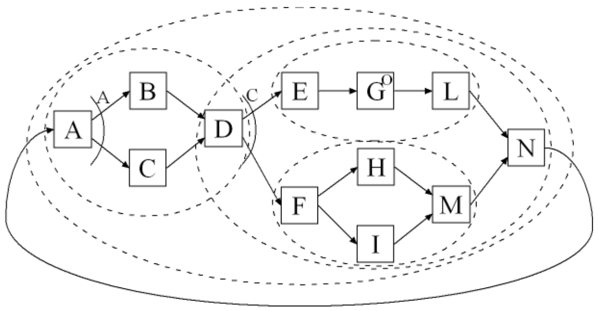

Figure 1. A workflow graph

We distinguish two basic kinds of explicit time constraints that link a source event *s* (the end of an activity) to a destination event *d* (the end of another activity), namely, the *lower-bound* (resp., *upper-bound*) *constraint lbc(s,d,δ)* (resp., (*ubc(s,d,δ)*)). The lower-bound (resp., upper-bound) constraint states that the time distance between *s* and *d* must be greater (resp., smaller) than or equal to δ time units. Explicit time constraints can also be used to express the condition of a conditional (parallel) composition of two or more activities. As an example, they allow one to select the activities to be executed depending on whether a given activity started its execution before another activity or not.

3. Timed automata

Timed automata are one of the most widely used formalisms for the specification and verification of real-time systems. They have been introduced by Alur and Dill in [2], where decidability results for some meaningful classes of them are provided. Most contributions focus on timed automata over infinite words, but all results can be easily transferred to the finite case. In the following, we summarize the main features of timed automata (a detailed presentation can be found in [2]).

Definition 1 *A* time sequence $\tau = \tau_1\tau_2\cdots$ *is a sequence of time values* $\tau_i \in \mathbb{R}^+$ *such that* (i) τ *increases strictly monotonically, that is, for all* $i \geq 1$ $\tau_i < \tau_{i+1}$ (monotonicity) *and* (ii) *for every* $t \in \mathbb{R}^+$*, there is some* $i \geq 1$ *such that* $\tau_i \geq t$ (progress). *A* timed word *over an alphabet* Σ *is a pair* (σ, τ)*, where* $\sigma = \sigma_1\sigma_2\cdots$ *is a word over* Σ *and* τ *is a time sequence over* $\mathbb{R}^+$*. A* timed language *over* Σ *is a set of timed words over* Σ*.*

The timed word (σ, τ) can be viewed as an input to an automaton, which presents the symbol σ_i (an event occurrence) at time τ_i (its occurrence time). Language-theoretic operations, such as intersection, union, and complementation, are defined as usual. In addition, we define an *Untime* operation which projects a time word (σ, τ) on its first component.

Timed automata are obtained from classical ones by adding a finite set of real-valued clock variables (*clocks* for short). The vertices of the graph are called *locations* and the edges are called *switches*. While switches are instantaneous, time can elapse in a location. All the clocks increase at a uniform rate counting time with respect to a fixed global time. When transitions take place, clocks can possibly be reset, that is, their values can be set to zero. The value of a clock at any instant thus equals the time elapsed since the last time it was reset. We associate a clock constraint with each switch and we require that the switch may be taken only if the current values of the clocks satisfy the constraint. We also associate a clock constraint with each location, called its *invariant*, and we require that time can elapse in a location only as long as its invariant stays true.

Definition 2 *For a given set* X *of clock variables, the set* $\Phi(X)$ *of* diagonal-free clock constraints δ *is defined inductively by* $\delta ::= x \sim c \mid \neg\delta \mid \delta_1 \wedge \delta_2$*, where* x *is a clock in* X*,* c *is a constant in* $\mathbb{Q}$*, and* $\sim \in \{<, \leq, =, \neq, \geq, >\}$*.*

A *clock interpretation* ν for a set X of clocks assigns a real value to each clock. We say that a clock interpretation ν for X satisfies a clock constraint δ over X iff δ evaluates to true under the interpretation ν. For $t \in \mathbb{R}$, $t + \nu$ denotes the clock interpretation which maps every clock x to the value $t + \nu(x)$, while the clock interpretation $t \cdot \nu$ assigns to each clock x the value $t \cdot \nu(x)$. For $Y \subseteq X$, $[Y \mapsto t]\nu$ denotes the clock interpretation for X which assigns t to each $x \in Y$ and agrees with ν over $X \setminus Y$.

Definition 3 *A* timed automaton $\mathcal{A}$ *is a tuple* $\langle S, S_0, \Sigma, C, I, E, F\rangle$*, where* S *is a finite set of locations,* $S_0 \subseteq S$ *is a set of initial locations,* Σ *is an input alphabet,* C *is a finite set of clocks,* I *is a mapping from* S *to a set of clock constraints* $\Phi(C)$*,* $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$ *is a set of switches, and* $F \subseteq S$ *is a set of final locations.*

For any location s, I specifies the time constraints that must be satisfied to remain in s (invariant set). As soon as an invariant is violated due to the elapse of time, we must exit the location. A switch $\langle s, s', a, \lambda, \delta\rangle$ represents a transition from location s to location s' on input symbol a; the set $\lambda \subseteq C$ specifies the clocks to be reset by the transition, while δ is a clock constraint over C that specifies when the switch is enabled.

Definition 4 *A* run r *of a timed automaton* $\mathcal{A} = \langle S, S_0, \Sigma, C, I, E, F\rangle$ *over a timed word* (σ, τ)*, denoted by* $(\overline{s}, \overline{\nu})$*, is a sequence of the form*

$$r : \langle s_0, \nu_0\rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, \nu_1\rangle \xrightarrow[\tau_2]{\sigma_2} \langle s_2, \nu_2\rangle \xrightarrow[\tau_3]{\sigma_3} \cdots ,$$

where for all $i \geq 0$*,* $s_i \in S$*, and the clock interpretation* ν_i *is a mapping from* C *to* $\mathbb{R}$ *such that* (i) $s_0 \in S_0$ *and* $\nu_0(x) = 0$ *for all* $x \in C$ (initiation) *and* (ii) *for all* $i \geq 1$*, there is an edge* $(s_{i-1}, s_i, \sigma_i, \lambda_i, \delta_i)$ *in* E *such that* $\nu_{i-1} + (\tau_i - \tau_{i-1})$ *satisfies* δ_i *and* ν_i *equals* $[\lambda_i \mapsto 0](\nu_{i-1} + (\tau_i - \tau_{i-1}))$ (consecution).

A run r over an infinite timed word is *accepting* iff $Inf(r) \cap F \neq \varnothing$, where $Inf(r)$ is the set of locations through which the computation passes infinite times. In the finite case, r is accepting if it starts at an initial location and it ends in a final one. A timed word w is accepted by $\mathcal{A}$ is there exists an accepting run of $\mathcal{A}$ on w. The language of $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of accepted timed words.

Theorem 1 *The class of timed automata / languages is closed under concatenation, union, intersection, and product (the locations of the product automaton are obtained by pairing locations of the component automata, the location invariants are the conjunction of the invariants of the component locations, and the switches are obtained by properly merging pairs of switches of the component automata).*

We restrict our attention to *deterministic* timed automata which have one run over a given timed word. The emptiness problem for a deterministic timed automaton $\mathcal{A}$ is the problem of checking whether $\mathcal{L}(\mathcal{A})$ is empty or not. The next theorem proves that such a problem is decidable [1].

Theorem 2 *Let* $\mathcal{A}$ *be a deterministic timed automaton, whose clock constraints only compare clocks with constants. The emptiness problem for* $\mathcal{L}(\mathcal{A})$ *is PSPACE-complete.*

In [3], Bouyer et al. propose a natural extension of Alur and Dill's deterministic timed automata that allows one to update the clock value in a more general way than simply reset it to 0 and it generalizes the set of admissible clock constraints according to the following definition.

Definition 5 *Given set X of clock variables, the set $\Phi(X)$ of* generic clock constraints *δ is defined inductively by $\delta ::= x \sim c \mid x - y \sim c \mid \neg\delta \mid \delta_1 \wedge \delta_2$, where x, y are clocks in X, c is a constant in $\mathbb{Q}$, and $\sim \in \{<, \leq, =, \neq, \geq, >\}$.*

They prove that the addition of simple resets, such as $x := c$ and $x := y$, with $c \in Q^+$ and $x, y \in X$, and generic clock constraints preserves PSPACE-completeness. In the next section, we will take advantage of generic clock constraints to express the conditions of conditional (parallel) compositions.

4. From workflow schemas to timed automata

In this section we show how to reduce the problem of consistency checking for timed workflow schemas (as well as other relevant problems) to the emptiness problem for deterministic timed automata with generic clock constraints: given a timed workflow schema, we map it into a timed automaton such that workflow constraints are satisfiable if and only if the language recognized by the automaton is not empty. Details of the reduction are given in [12]. As a preliminary step, we introduce some basic notions.

Definition 6 (Workflow schema) *Given a set of activities Act, a* workflow schema *W on Act consists of a directed graph $\langle N, E\rangle$, where $N \subseteq$ Act and $E \subseteq N \times N$, and a set of temporal constraints $\mathcal{C}$. N identifies the set of relevant activities and it includes an initial activity I and a final one F; E defines the dependencies between activities. W can be inductively defined as follows:*

- Base case. *$W = \langle\{A\}, \emptyset\rangle$, where $A \in$ Act (single activity) and $I = F = A$.*

- Inductive step *(in the following, we denote by I_i and F_i the initial activity and the final activity of a workflow schema W_i).*

  - Sequential composition. *$W = ;(I, W_1, W_2, F)$ is the concatenation of the workflow schemas W_1 and W_2, where $I = I_1$, $F = F_2$, and if $F_1 \neq I_2$, then there exists an edge from F_1 to I_2.*
  - Parallel composition. *$W = ||(I, W_1, W_2, F)$ is the parallel composition of the workflow schemas W_1 and W_2, where there exists an edge from I to I_1 (resp. I_2) and there exists an edge from F_1 (resp. F_2) to F.*
  - Alternative composition. *$W = |(I, W_1, W_2, F)$ is the alternative composition of the workflow schemas W_1 and W_2, where there exists an edge from I to I_1 (resp. I_2) and there exists an edge from F_1 (resp. F_2) to F. The edges which exit from I are marked as alternative ones.*
  - Conditional (parallel) composition. *$W = ||_C(I, W_1, W_2, F)$ is the conditional (parallel) composition of the workflow schemas W_1 and W_2, where C is a Boolean condition, there exists an edge from I to I_1 (resp. I_2), and there exists an edge from F_1 (resp. F_2) to F. The edges which exit from I are marked by the condition C. Conditions can be either temporal or atemporal. Temporal conditions allow one to check the temporal order or the temporal distance between the start/end of pairs of activities.*
  - Optional workflow. *$W = \mathcal{O}(W_1)$, where W_1 is a workflow schema.*
  - Repetition. *$W = *(I, W_1, F)$ is the repetition of the workflow schema W_1 (0 or more times), where $I = I_1$, $F = F_1$, and there exists an edge from F to I.*

$\mathcal{C}$ specifies a (deterministic) duration for every activity and a set of lower/upper bound constraints between the ends of pairs of activities.

Note that, without loss of generality, we defined composition operators on pair of activities (to obtain more compact definitions of workflow schemas, we can generalize them to cope with more than two activities) and we assumed the existence of exactly one initial activity and one final activity. The workflow schema in Figure 1 can be encoded by the expression $W = *(A, ;(A, |(A, B, C, D), ||_C(D, ;(E, E, \mathcal{O}(G), L, L), ||(F, H, I, M), N), N), N)$.

Definition 7 (Workflow schema consistency) *Given a workflow schema W with activity set N, a workflow instance $\mathtt{I}$ of W is a list of pairs (A_i, e_i), where, for every $1 \leq i \leq |N|$, $A_i \in N$ and e_i is its ending time with respect to the workflow beginning time. Pairs in $\mathtt{I}$ are first ordered by topological order and then by their second component. A timed workflow schema W is* consistent *if and only if there exists an instance of W which satisfies all temporal constraints.*

Let $\mathcal{W}$ be the set of timed workflow schemas built up according to Definition 6 and let $\mathbb{C}_{\mathcal{A}}$ be the set of deterministic timed automata with generic clock constraints. The algorithm *Translation(W, $\mathcal{A}$)*, with $W \in \mathcal{W}$ and $\mathcal{A} \in \mathbb{C}_{\mathcal{A}}$, defines a mapping $\tau : \mathcal{W} \rightarrow \mathbb{C}_{\mathcal{A}}$ such that, for each $W \in \mathcal{W}$, W is consistent iff $\mathcal{L}(\tau(W)) \neq \varnothing$ (see Theorem 3 below).

We assume the names of both workflow activities and automaton locations to be pairwise disjoint. Furthermore, for all $A \in N$, we denote by $A.d$ the duration of A. Finally, we introduce the following notion of macro-label.

Definition 8 (Macro-label) *A* macro-label *is a set of atomic labels of the forms a_S (the beginning of activity A), a_E (the end of A), C_S (the beginning of a conditional composition with condition C), and C_E (the end of a conditional composition with condition C).*

Translation($W, \mathcal{A}$) consists of two main procedures. The first one, *Automation(W,S)*, maps W into a pseudo-automaton S (devoid of initial and final locations), which features (at least) one location labeled by $\mathtt{A}$ for every activity A, and models temporal delays by means of clock constraints and additional *waiting* locations. It works inductively and calls the procedure $Constrain(T, S)$ in order to encode lbc and ubc. The second one, $Completion(S, \mathcal{A})$, turns the pseudo-automaton S into a timed automaton $\mathcal{A}$ by defining initial and final locations.

Algorithm 1 $Automation(W, S)$

```
1: CASE OF
2: $W \in N \rightarrow Base(W, T)$;     //structural constraints
3: $W = \circ(W_1, \ldots, W_n) \rightarrow Cseq(W_1, \ldots, W_n, T)$;
4: $W = ||(W_1, \ldots, W_n) \rightarrow Cpar(W_1, \ldots, W_n, T)$;
5: $W = |(W_1, \ldots, W_n) \rightarrow Calt(W_1, \ldots, W_n, T)$;
6: $W = |_C(W_1, \ldots, W_n) \rightarrow Ccond(W_1, \ldots, W_n, C, T)$;
7: $W = *(W_1) \rightarrow Rep(W_1, T)$.
8: ENDCASE
9: Constrain($T, S$).     //explicit constraints
```

As a preliminary step, we rewrite the workflow schema expression in a more compact way. Such a rewriting will simplify the synthesis of the algorithm $Translation$. The alternative notation we propose uses '$\circ$' to denote a generalized sequential composition, '$||$' for the parallel composition (of two or more branches), '$|$' for the alternative composition (of two or more branches), '$||_C$' for the conditional (parallel) composition (of two or more branches), '$\mathcal{O}$' for an optional activity, and '$*$' for repetitions. A generalized sequential composition can be either a sequential composition of two or more activities (previously denoted by ;), or the connection of an activity to two or more immediate successor activities (split) or the connection of two or more activities to an immediate successor one (join). As an example, the workflow schema of Figure 1 can be expressed as follows: $W = *(\circ(A, |(B, C), D, ||_C(\circ(E, \mathcal{O}(G), L), \circ(F, ||(H, I), M)), N))$.

Activity. The procedure $Base(W, T)$ deals with the case of a single activity A. Given an activity A, with duration $A.d$, we constrain the pseudo-automaton T to remain in the location $\mathtt{A}$ exactly $A.d$ time units.

Sequential composition. The sequential composition of two workflow schemas is dealt with by inserting a waiting location between the corresponding automata, which accounts for the temporal delay between the end of the first process and the beginning of the second one. The generalization to n workflows is immediate. The treatment of sequential composition is exemplified in Figure 2.

Parallel composition. The parallel composition of two (resp. n) workflow schemas is captured by the product of the corresponding automata. To avoid undesired clashes, we constrain the sets of clocks of the two (resp. n) component automata to be disjoint. Given $\mathcal{A}_i = \langle S_i, S_i^0, \Sigma_i, C_i, I_i, E_i, F_i \rangle$, where, for $1 \leq i \leq n$, Σ_i is a set of *macro-labels*, the product automaton $\|\mathcal{A}_i$ is defined as follows:

$$\|\mathcal{A}_i = \left(\prod_{i=1}^{n} S_i, \prod_{i=1}^{n} S_i^0, \prod_{i=1}^{n} \Sigma_i, \bigcup_{i=1}^{n} C_i, I, E, \prod_{i=1}^{n} F_i \right), \text{with}$$

- $I(s_1, \ldots, s_n) = \bigwedge_{i=1}^{n} I_i(s_i)$;
- each switch $e \in E$ is labeled by a set of macro-labels $\{A_{j_1}, \ldots, A_{j_k}\} \subseteq \{A_1, \ldots, A_n\} \in \prod_{i=1}^{n} \Sigma_i$;
- if $e = ((l_1, \ldots, l_n), (m_1, \ldots, m_n), \{A_{j_1}, \ldots, A_{j_k}\}, \lambda, \phi)$ is a switch exiting from $(l_1, \ldots, l_n)$, then
  - if $A_i \in \{A_{j_1}, \ldots, A_{j_k}\}$, then there exists a switch from l_i to m_i in $\mathcal{A}_i$ labeled by A_i, otherwise $m_i = l_i$;
  - λ is $\bigcup_{A_{j_i} \in \{A_{j_1}, \ldots, A_{j_k}\}} \lambda_{j_i}$, where λ_{j_i} is the set of clocks to be reset in correspondence of the switch from l_{j_i} to m_{j_i} labeled by A_{j_i} in $\mathcal{A}_{j_i}$;
  - ϕ is $\bigwedge_{A_{j_i} \in \{A_{j_1}, \ldots, A_{j_k}\}} \phi_{j_i}$, where ϕ_{j_i} is the clock constraint associated with the switch from l_{j_i} to m_{j_i} labeled by A_{j_i} in $\mathcal{A}_{j_i}$.

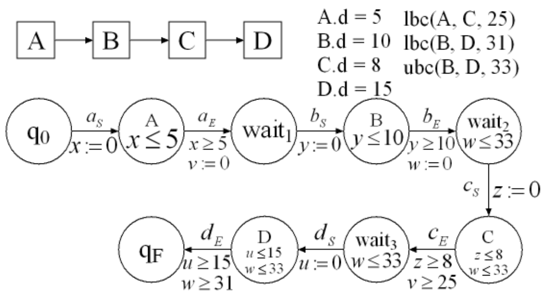

Figure 2. The sequential case

The resulting product automaton captures all possible parallel executions of the different branches.

Figure 4 contains a workflow schema with a parallel structure and the corresponding product automaton. Observe that, in the product automaton, for each location of the form (l_1, l_2), where both l_1 and l_2 are not final locations, there are three switches exiting from it: the switch labelled by a symbol belonging to the first automaton, the one labelled by a symbol belonging to the second automaton, and the one containing a couple of symbols.

Alternative composition and optional activity. If the workflow schema contains an alternative composition and we are only interested in the existence of a consistent execution, without minding which alternative is chosen, we can proceed as follows. The algorithm first generates an automaton for each alternative branch; then, it introduces a waiting location and connects it to the initial locations of the automata; finally, it introduces another location and connects the final locations of the automata to it. In such a way, the problem of establishing the existence of a consistent alternative in the given workflow can be reduced to the emptiness problem for the resulting automaton. The treatment of alternative composition is exemplified in Figure 3. Later, we will show how to solve the problem of checking whether a specific alternative is consistent (see forcing automata below). The case of optional activities can be easily reduced to the case of alternative composition. It suffices, as a preliminary step, to replace the branch including the optional activity by two alternative branches, one including the optional activity, the other excluding it, and then to proceed as in the case of alternative composition.

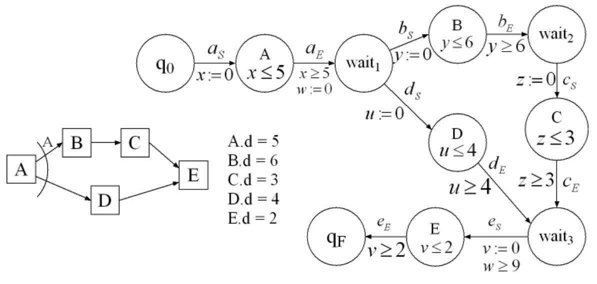

Figure 3. The alternative case

Conditional composition. In the case of conditional composition, the choice of the branches to follow depends on the current state of the specific workflow instance at run time. Let ϕ be the Boolean condition whose truth value determines the set of branches to follow. At build time, we partition the set of conditional branches into two sets: a set $\mathcal{B}_1$, including all branches to execute in case ϕ is true, and a set $\mathcal{B}_2$, including all branches to execute if it is false. Next, we build an automaton for every branch in $\mathcal{B}_1$ and then we combine them into a product automaton $P_{\mathcal{B}_1}$. We do the same with $\mathcal{B}_2$. Finally, we merge the product automata $P_{\mathcal{B}_1}$ and $P_{\mathcal{B}_2}$, as in the case of alternative composition, and we add the constraint ϕ (resp. $\neg\phi$) to the switch entering the initial location of $P_{\mathcal{B}_1}$ (resp. $P_{\mathcal{B}_2}$). In addition, we add the label C_S (start of conditional) to the switches entering the first location of $P_{\mathcal{B}_1}$ and $P_{\mathcal{B}_2}$ and the label C_E (end of conditional) to the switches exiting from the last location of $P_{\mathcal{B}_1}$ and $P_{\mathcal{B}_2}$. Condition ϕ can be of temporal nature. As an example, consider a workflow schema containing a conditional composition whose condition ϕ checks whether the activity A started its execution before the activity B or not (in the schema both A and B precede the conditional composition). If x (resp. y) is the clock which has been reset at the beginning of activity A (resp. B), the temporal constraint to add to the switch entering the initial location of $P_{\mathcal{B}_1}$ (resp. $P_{\mathcal{B}_2}$) is $x > y$ (resp. $x \leq y$). As in the case of alternative composition, forcing automata can be used to check the existence of a consistent execution in case ϕ turns out to be true (resp. false).

Repetition. The repetition of a workflow schema is dealt with by connecting the last location of the corresponding automaton to the first one and inserting a waiting location between them in order to let time elapse between an execution and the following one.
It is possible to prove that (the proof is by induction on the structure of the workflow schema):

Theorem 3 *For all $W \in \mathcal{W}$, W is consistent iff $\mathcal{L}(\tau(W)) \neq \varnothing$.*

To check the existence of workflow instances passing through a specific branch, we can take advantage of an auxiliary automaton, that we call *forcing automaton*. (Notice that, to force one workflow instance to pass through a given alternative or conditional branch, it suffices to force it to pass through the first activity of that branch.) Given a workflow schema W, we first build the corresponding timed automaton $\mathcal{A}$ by means of *Translation($W, \mathcal{A}$)*. Then, we build an auxiliary automaton $\mathcal{B}$ that only recognizes workflow instances passing through the desired branch. Finally, we intersect $\mathcal{A}$ and $\mathcal{B}$. In such a way, we reduce the considered problem to the emptiness problem for the resulting intersection automaton. As an example, consider the workflow schema of Figure 3 and suppose that we are interested in the existence of a consistent execution in case we choose the first alternative branch. To this end, it suffices to intersect the timed automaton of Figure 3 with a simple timed automaton that forces the passing through B (such an automaton consists of 2 states, an initial state and a final one, both provided with a self loop, labelled by all symbols, and a switch from the initial to the final state, labelled by b_S).

5. A concrete example

We illustrate the potentialities of the proposed approach by applying it to a concrete example. The workflow schema of Figure 5 is a variant of a schema originally proposed by van der Aalst to model the process of insurance claim handling [16]. The deterministic duration of each activity is denoted by a number on the top right of the corresponding node. Its basic structure can be described as follows. First, every claim is classified according to its complexity by the *Classify* activity. On the basis of the result of the classification (condition C_1), the appropriate conditional branch is chosen. The first alternative (*Open_simple*) consists in the parallel execution of two activities, while the second one

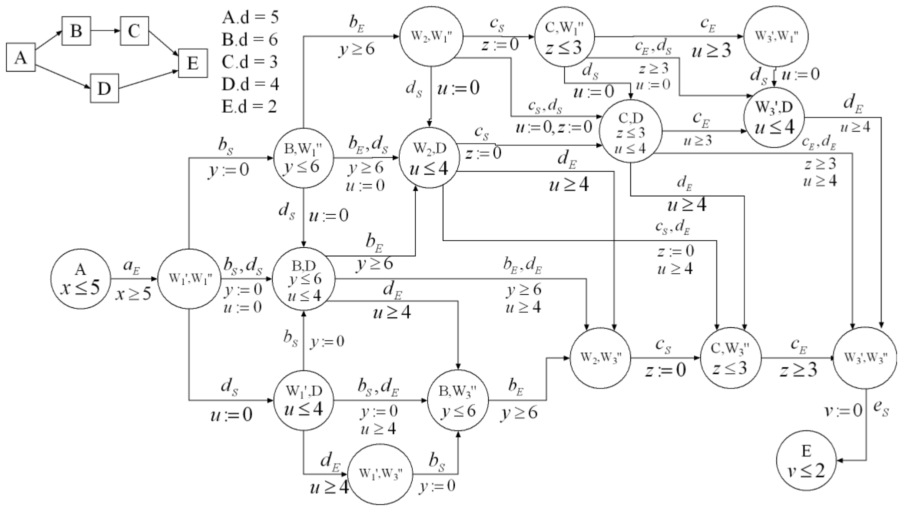

Figure 4. The parallel case

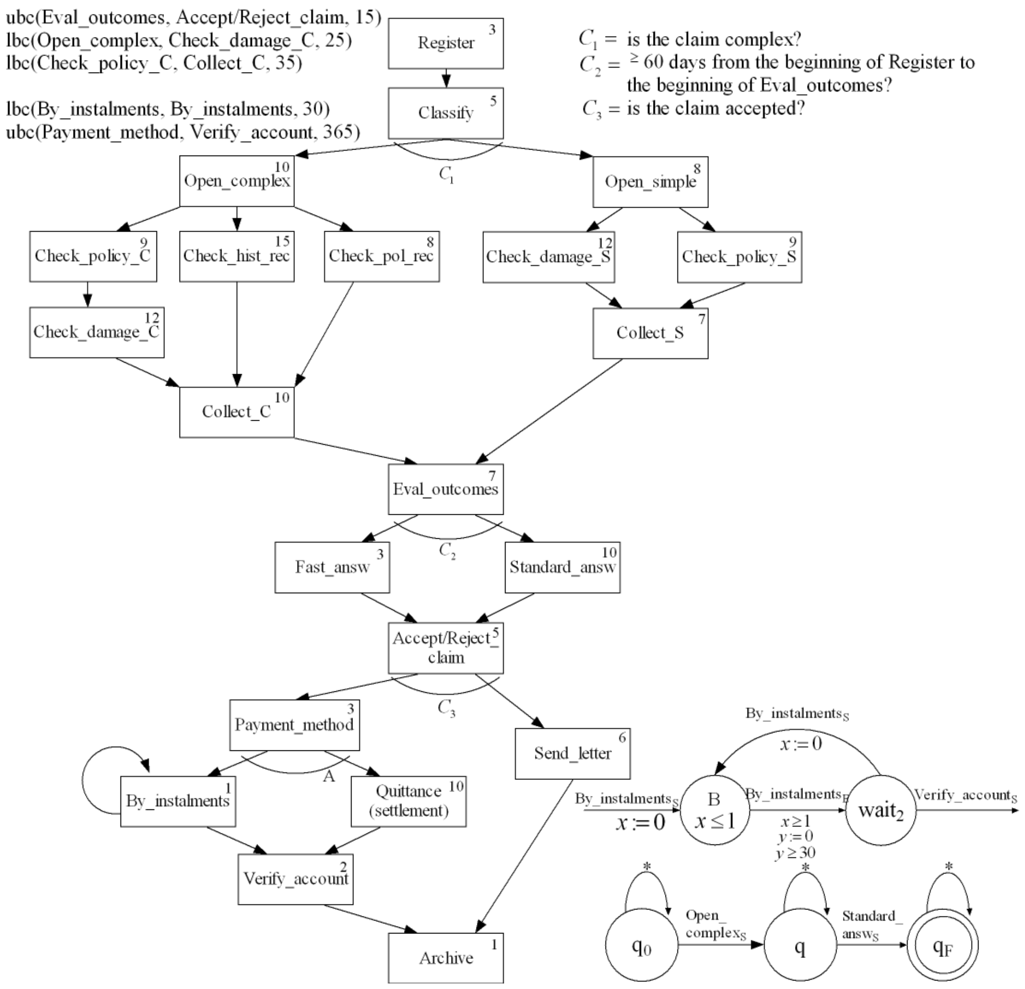
ubc(Eval_outcomes, Accept/Reject_claim, 15)
lbc(Open_complex, Check_damage_C, 25)
lbc(Check_policy_C, Collect_C, 35)
lbc(By_instalments, By_instalments, 30)
ubc(Payment_method, Verify_account, 365)
C1 = is the claim complex?
C2 = ≥ 60 days from the beginning of Register to the beginning of Eval_outcomes?
C3 = is the claim accepted?
Register
Classify
Open_complex
Open_simple
Check_policy_C
Check_hist_rec
Check_pol_rec
Check_damage_S
Check_policy_S
Check_damage_C
Collect_S
Collect_C
Eval_outcomes
Fast_answ
Standard_answ
Accept/Reject_claim
Payment_method
Send_letter
By_instalments
Quittance (settlement)
Verify_account
Archive
wait2

Figure 5. A concrete example

(*Open_complex*) results into the parallel execution of three branches. In both cases, the analysis of the claim terminates with an evaluation of the outcomes (*Eval_outcomes*). If 60 days, or more, have passed from the beginning of the process to the beginning of *Eval_outcomes*(condition C_2), *Fast_answ* is chosen, otherwise *Standard_answ* is chosen (if x_1 is the clock reset at the beginning of the workflow and x_2 the clock reset at the beginning of *Eval_outcomes*, the corresponding automaton encodes condition C_2 by the clock constraint $x_1 - x_2 \geq 60$). In case the claim is accepted (condition C_3), the payment phase begins. If the insurance company decides to pay by instalments, at least 30 days must elapse between an instalment and the next one. The iteration of the activity *By_instalments* is modeled by a loop (the component of the corresponding timed automaton dealing with the payment by instalments is reported at the right bottom of Figure 5).
An upper-bound constraint imposes that between the end of *Payment_method* and the end of *Verify_account* no more than 365 days can elapse. Timed automata can be used to check automaton satisfiability of such a workflow schema (cf. Section 3). Moreover, the forcing technique makes it possible to check the existence of a consistent instance which involves some specific activities. As an example, one may be interested in establishing whether there exists (at least) one instance where the answer to a complex claim (*Open_complex*) is managed in the standard way (*Standard_answ*). This can be done by the forcing automaton at the left bottom of Figure 5. It is possible to prove that the automaton obtained by intersecting this forcing automaton with the automaton obtained by applying the *Translation* algorithm to the given workflow schema recognizes the empty language. Since at least 62 days elapse from the beginning of the workflow to the beginning of *Eval_outcomes* whenever an instance passes through *Open_complex*, condition C_2 is always satisfied and *Fast_answ* must be taken.

6. Conclusions

In this paper we proposed timed automata as an effective tool to specify timed workflow schemas and to check their consistency. Such an approach allows one to obtain a compact and uniform representation of structural and explicit time constraints: all basic constructs of workflow schemas (sequential composition, and-split, or-split, etc.) can be expressed in terms of basic operations on automata (concatenation, product, union, etc.), while explicit time constraints (duration of activities, relative deadlines, upper bound and lower bound constraints) can be expressed in terms of suitable clock constraints. Moreover, to check the consistency of timed workflow schemas, one does not need to develop ad hoc algorithms, but he/she can take advantage of the existing machinery for timed automata (since the beginning of the '90s, many tools for the specification and verification of real-time systems based on timed automata have indeed been developed and successfully applied, including UPPAAL [10], COSPAN [9], and KRONOS [5]).

References

[1] R. Alur. Timed automata. *In Proc. 11th Int. Conference on Computer-Aided Verification*, LNCS 1633:8–22, 1999.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Compututer Science*, 126(2):183–235, 1994.

[3] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Updatable timed automata. *Theoretical Computer Science*, 321(2-3):291–345, 2004.

[4] S. J. Chinn and G. R. Madey. Temporal representation and reasoning for workflow in engineering design change review. *IEEE Transactions on Engineering Management*, 47(4):485–492, 2000.

[5] C. Daws, A. Olivero, S. Tripakis, and S. Jovine. The tool KRONOS. *In Hybrid Systems III: Verification and Control*, LNCS 1066:208–219, 1990.

[6] J. Eder. Workflow management systems (abstract). In *ADBIS*, page 210, 1997.

[7] J. Eder, W. Gruber, and E. Panagos. Temporal modeling of workflows with conditional execution paths. *In Proc. DEXA*, LNCS 1873:243–253, 2000.

[8] J. Eder, E. Panagos, and M. Rabinovich. Time constraints in workflow systems. *In Proc. CAiSE*, LNCS 1626:286–300, 1999.

[9] R. Hardin, Z. Har'El, and R. Kurshan. COSPAN. *In Proc. 8th Int. Conference on Computer-Aided Verification*, LNCS 1102:423–427, 1996.

[10] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1:134–152, 1997.

[11] J. Li, Y. Fan, and M. Zhou. Timing constraint workflow nets for workflow analysis. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 33(2):179–193, 2003.

[12] E. D. Maria, A. Montanari, and M. Zantoni. Checking workflow schemas with time constraints using timed automata. Technical Report UDMI/06/05, University of Udine, Mathematics and Computer Science Dept., 2005 (an extended abstract appeared in Proc. of the OTM Workshops 2005, LNCS 3762:1-2). http://www.dimi.uniud.it/zantoni/.

[13] O. Marjanovic and M. E. Orlowska. On modeling and verification of temporal constraints in production workflows. *Knowledge Information System*, 1(2):157–192, 1999.

[14] E. Panagos and M. Rabinovich. Predictive workflow management. In *Proc. 3rd Workshop NGITS*, pages 193–197, 1997.

[15] W. Sadiq, O. Marjanovic, and M. E. Orlowska. Managing change and time in dynamic workflow processes. *Int. J. Cooperative Inf. Syst.*, 9(1-2):93–116, 2000.

[16] W. van der Aalst. Process-oriented architectures for electronic commerce and interorganizational workflow. *Information Systems*, 24(8):639–671, 1999.

[17] W. van der Aalst and K. van Hee. *Workflow management models, methods and systems*. MIT Press, 2004.