Symbolic model checking for simply-timed systems

N. Markey[1,2] and Ph. Schnoebelen[2]

[1] Département d'Informatique
Université Libre de Bruxelles
email: `nmarkey@ulb.ac.be`

[2] Lab. Spécification & Vérification
ENS de Cachan & CNRS UMR 8643
email: `(markey,phs)@lsv.ens-cachan.fr`

Abstract. We describe OBDD-based symbolic model checking algorithms for *simply-timed systems*, i.e. finite state graphs where transitions carry a duration. These durations can be arbitrary natural numbers. A simple and natural semantics for these systems opens the way for improved efficiency. Our algorithms have been implemented in NuSMV and perform well in practice (on standard case studies).

1 Introduction

Symbolic verification of timed systems. Formal verification tools play an ever-increasing role in the design of complex systems. Using OBDD technology, *symbolic model checking* techniques [3, 17], are able to analyze larger and larger models [12, 1]. Symbolic model checking techniques are not so successful with *time-critical systems.* A popular approach to these systems is based on *timed automata*, for which several model checkers exist (e.g. [22, 16]). Timed automata raise difficult problems: the existing symbolic representations for sets of clock valuations (including DBMs, CDDs and REDs) are not as simple and convenient as OBDDs and do not mix well with the OBDDs used for the control part. Finally fixpoint computations are notably tricky for timed automata [2].

State graphs as timed models. Time-critical systems do not always have to be modeled with timed automata. Indeed, labeled state graphs, a.k.a. *Kripke structures* (KS's), have been used for such purposes. If we assume that each transition in a KS takes one time unit, RTCTL model checking is not harder than CTL model checking [13] [3]. Symbolic OBDD-based model checking extends easily to RTCTL on KS's [6]. The corresponding algorithms have been implemented (*e.g.* in NuSMV [11], Verus [4]) and perform quite efficiently. This approach combines simplicity and efficiency. Its limitation is the restricted expressive power of the KS model, even disregarding the issue of discrete *vs.* dense time.

[3] RTCTL, or "Real Time CTL", is the extension of CTL where temporal modalities may carry numerical time bounds.

Our contribution. We extend symbolic RTCTL model checking to *timed Kripke structures* (TKS's), *i.e.* Kripke structures where each transition carries a *duration*, that can be any integer value (including zero). Thus the difference between KS's and TKS's is that we do not assume that each transition takes one and only one time unit. We provide algorithms for symbolic model checking on TKS's against RTCTL specifications. We also compute quantitative information, answering queries of the kind "what is the minimum (resp. maximum) delay between a state satisfying some *Start* property and a state satisfying some *Final* property?" [9].

Durations in TKS's are *atomic* and do not reduce to a sequence of unit steps, contrary to the model used in RAVEN [19], for instance. Our model checking algorithms take advantage of this semantics of time and, for models that would naturally be described with long transitions, they perform better than the algorithms offered in NuSMV (see Section 6). Thus, while our approach builds on earlier research by Campos, Clarke *et al.* [5–10], there are nevertheless some clear differences between the two approaches: differences in the models, in the algorithms, and in the running times. A more detailed comparison is provided in section 7.

We implemented our algorithms on top of NuSMV. This readily provides a symbolic model checker for TKS's, called TSMV and available at `http://www.lsv.ens-cachan.fr/~markey/TSMV/`. Our experiments indicate that our tool TSMV provides appreciable gain in expressive power and improved running time (when compared with the verification of "equivalent" unit-steps KS's).

Finally, TSMV improves the available set of solutions for model checking *simply-timed systems*, i.e. systems where the control part is too large for a timed-automaton approach, and where the timing aspects are simple enough to be described with durations on transitions.

What use are arbitrary durations? Compared to plain KS's, there are three main ways where TKS's provide increased expressive power:

Long steps: TKS's allow considering that some transitions take a long time, *e.g.* 1000 time units. Such transitions could be encoded in KS's by inserting 999 intermediate states (*e.g.* using a simple counter) as advocated in [7, p.106]. But this encoding is tedious, and leads to quite costly extra verification work. It is better to have algorithms that understand arbitrary durations.

Instantaneous steps: TKS's allow considering that some transitions have *zero duration*. This is very convenient in models where some steps are described indirectly, as a short succession of micro-steps.

Counting specific events only: Zero-length transitions are also a convenient way of abstracting away (for timing aspects only) from some internal transitions. Then the RTCTL formulae count the durations of the other, time-relevant, transitions. This can also be used in models where transitions carry some "cost", a notion more general than elapsed time. For example, when verifying a communication protocol, one can wish to only count the sendings and receivings of messages. When checking a model of an elevator system, one can wish to count cabin moves, or door closings and openings, and use these in cost-constrained CTL formulae.

2 Basic notions

We assume familiarity with CTL model checking on finite Kripke structures (KS's). Furthermore, we assume the reader understands the working of OBDD-based symbolic model checking, as implemented *e.g.* in (Nu)SMV [3, 17, 11]. These notions are available in several textbooks, for instance [12, 1].

2.1 Kripke structures with durations on transitions

Simply-timed Kripke Structures (TKS's) are an extension of KS's where each transition is labeled by a nonnegative integer. Formally, given a set $AP = \{p_1, p_2, \ldots\}$ of *atomic propositions*, a TKS is a tuple $M = \langle S, S_0, R, L\rangle$ where $S = \{s, s', \ldots\}$ is a finite set of states, $S_0 \subseteq S$ is a set of initial states, and $L : S \mapsto 2^{AP}$ is a labeling of states with atomic propositions. The difference with KS's is that here $R \subseteq S \times \mathbb{N} \times S$ is a finite set of *transitions* labeled by a natural number, called the *duration* of the transition. We require that the untimed relation $\overline{R}$, obtained from R by dropping durations, is total.

Fig. 1 displays a simple example. Note that, between two states, there may exist several transitions with different durations. In graphical representations, such sets of transitions are sometimes depicted with several labels on one arrow.

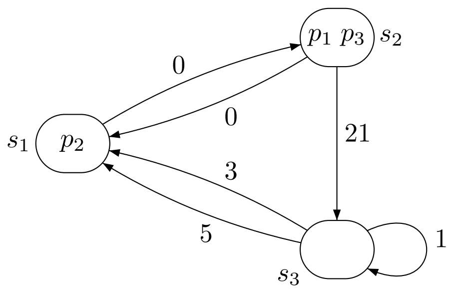

Fig. 1. A simple TKS

A path π in M is an infinite sequence $s_0 \xrightarrow{d_1} s_1 \xrightarrow{d_2} s_2 \xrightarrow{d_3} \cdots$ of linked transitions. For such a path, and for $n \in \mathbb{N}$, we let $\pi(n)$ denote the nth state s_n, and π^n denote the nth suffix $s_n \xrightarrow{d_{n+1}} s_{n+1} \xrightarrow{d_{n+2}} \cdots$. Finally, for $n \leq m \in \mathbb{N}$, we let $\pi[n..m]$ denotes the finite sequence $s_n \xrightarrow{d_{n+1}} s_{n+1} \xrightarrow{d_{n+2}} \cdots s_m$ with $m-n$ transitions and $m-n+1$ states. The (cumulative) *duration* $\mathtt{D}\pi[n..m]$ of such a finite sequence is $d_n + \cdots + d_{m-1}$ (hence 0 when $n = m$). We write $\Pi(s)$ for the set of paths that start from s: since R is total, $\Pi(s)$ is never empty.

2.2 RTCTL: a temporal logic with timing constraints

RTCTL formulae are the state formulae built according to the following syntax:

$$\varphi_{\mathrm{s}} ::= \neg\varphi_{\mathrm{s}} \mid \varphi_{\mathrm{s}} \wedge \psi_{\mathrm{s}} \mid \mathsf{E}\,\varphi_{\mathrm{p}} \mid p_1 \mid p_2 \mid \cdots \qquad \text{(state formulae)}$$
$$\varphi_{\mathrm{p}} ::= \neg\varphi_{\mathrm{p}} \mid \mathsf{X}\varphi_{\mathrm{s}} \mid \varphi_{\mathrm{s}}\mathsf{U}_{\alpha}\psi_{\mathrm{s}} \qquad \text{(path formulae)}$$

where α, called the *timing constraint*, is a predicate on durations. All constraints of the form "$\leq k$", "$= k$", "$\geq k$" and "$[k..l]$" for k and l values in the relevant domain (here $\mathbb{N}$) are allowed. We write $n \models \alpha$ when duration $n \in \mathbb{N}$ satisfies the constraint α (definition omitted). We write $s \models \varphi_\mathrm{s}$ and $\pi \models \varphi_\mathrm{p}$ when a state $s \in S$ satisfies a state formula φ_s (resp. a path π satisfies a path formula φ_p). The definition is as expected, and we only spell out the few cases that are specific to RTCTL and TKS's:

$$
\begin{aligned}
s \models \mathsf{E}\,\varphi_\mathrm{p} &\overset{\text{def}}{\Leftrightarrow} \text{there exists a } \pi \in \mathit{\Pi}(s) \text{ s.t. } \pi \models \varphi_\mathrm{p},\\
\pi \models \mathsf{X}\varphi_\mathrm{s} &\overset{\text{def}}{\Leftrightarrow} \pi(1) \models \varphi_\mathrm{s},\\
\pi \models \varphi_\mathrm{s}\mathsf{U}_\alpha\psi_\mathrm{s} &\overset{\text{def}}{\Leftrightarrow} \begin{cases} \text{there exists } i \in \mathbb{N} \text{ s.t. } \mathtt{D}\pi[0..i] \models \alpha,\\ \pi(i) \models \psi_\mathrm{s}, \text{ and } \pi(j) \models \varphi_\mathrm{s} \text{ for all } 0 \leq j < i. \end{cases}
\end{aligned}
$$

We use all the classical abbreviations: $\top$, $\bot$, $\varphi_\mathrm{s} \vee \psi_s$, $\varphi_\mathrm{s} \Rightarrow \psi_s$, $\mathsf{A}\,\varphi_p$ (for $\neg\mathsf{E}\neg\varphi_p$), $\mathsf{F}_\alpha\varphi_\mathrm{s}$ (for $\top\mathsf{U}_\alpha\varphi_\mathrm{s}$), and $\mathsf{G}_\alpha\varphi_\mathrm{s}$ (for $\neg\mathsf{F}_\alpha\neg\varphi_\mathrm{s}$). Then $\mathsf{EF}_\alpha\varphi$ means that it is possible to reach a state satisfying φ via a finite path whose cumulative duration satisfies α, and $\mathsf{EG}_\alpha\varphi$ means that there is a path along which φ holds at all states visited after a cumulative duration satisfying α.

2.3 Some comments on semantics

Our semantics for TKS's is the simplest and most natural way of reading state graphs like the one in Fig. 1. However, it differs from the semantics commonly found with other models (like timed automata). Disregarding the well-known issue of discrete vs. dense time, the important differences are:

Zeno behaviors: we admit infinite paths where time stays forever bounded. This simplifies the exposition, and lets us use durations for counting things other than elapsed time. It is however possible to rule our Zeno behaviors with fairness constraints (see Section 5.3).

Atomicity of durations: during a transition $s \xrightarrow{10} s'$, the system moves from "s at some time t" to "s' at $t + 10$": between t and $t + 10$, there is no state (or time) where the system is in. Hence $s_1 \models \mathsf{AG}_{=2}\bot$ in Fig. 1 since no path starting from s_1 encounters a cumulative duration of 2. See section 7 for a comparison with other semantics of durations.

3 Symbolic computation of minimum and maximum delays

In this section, we present a symbolic algorithm for computing the minimum delay between two sets of states. An algorithm for maximum delay would be similar. As we explain below, this algorithm improves on earlier proposals in its strategy for handling the durations that appear in the model.

The *minimum delay* between a set *Start* $\subseteq S$ of starting states and a set *Final* $\subseteq S$ of final states is the minimum d s.t. there is a finite path $s_0 \xrightarrow{d_1} s_1 \ldots \xrightarrow{d_n} s_n$ in S with $s_0 \in \mathit{Start}$, $s_n \in \mathit{Final}$, and $d = d_1 + \cdots + d_n$ [9]. The minimum is ∞ if no such path exists. It is 0 when $\mathit{Start} \cap \mathit{Final}$ is non-empty, or when *Final* can be reached from *Start* using only zero-transitions.

Assume $M = \langle S, S_0, R, L\rangle$ is a TKS. Let $d_{\max}$ denote the highest duration occurring as a label in R. For $d = 0, \ldots, d_{\max}$, write R_d for the relation $\{(s, s') \in S^2 \mid s \xrightarrow{d} s'\}$ and R'_d for $R_d \setminus \bigcup_{e<d} R_e$. Hence R_d relates states that are connected by a transition with duration d and R'_d also relates these states provided they are not related by a "shorter" transition. Finally, write Dur for the set $\{d_1, \ldots, d_l\}$ (in increasing order) of all durations d with non-empty R_d and Dur' for the set $\{d'_1, \ldots, d'_m\}$ of all durations for which R'_d is non-empty. Note that Dur contains exactly the labels appearing in R (thus $d_l = d_{\max}$) and that $Dur' \subseteq Dur$.

The procedure and its correctness proof. Algorithm 1 is a backward chaining procedure computing minimum delays (Algorithms for modalities $\mathtt{EF}_{=0}$ and $\mathtt{EU}_{=0}$

```
function min_delay(Start,Final)
n ← 0;
Incr[0], Diff[0] ← EF=0( Final );
repeat {
    if ( Start ∩ Diff[n] ≠ ∅ ) { return(n); }
    n++;
    aux ← ⋃{Pre[R'_d](Diff[n-d]) | d ∈ Dur' ∧ 0 < d ≤n};
    Diff[n] ← EU=0( ¬Incr[n-1], ¬Incr[n-1] ∩ aux );
    Incr[n] ← Incr[n-1] ∪ Diff[n];
} until (n≥ d_max ∧ Incr[n] = Incr[n-d_max]);
return(∞);
```

Algorithm 1: Algorithm for computing minimum delay

are discussed later, in section 4.1.) We prove correctness by showing that the following two invariants hold whenever we enter the `repeat` loop:

$$0 \leq i \leq \mathtt{n} \;\Rightarrow\; \Big[s \in \mathtt{Incr[}i\mathtt{]} \text{ iff } s \models \mathsf{EF}_{\leq i} \mathit{Final}\Big] \tag{Inv1}$$

$$0 \leq i \leq \mathtt{n} \;\Rightarrow\; \Big[s \in \mathtt{Diff[}i\mathtt{]} \text{ iff } s \in (\mathtt{Incr[}i\mathtt{]} \setminus \mathtt{Incr[}i-1\mathtt{]})\Big] \tag{Inv2}$$

- (Inv1) and (Inv2) clearly hold the first time the loop is entered: `n` is 0 and both `Incr[0]` and `Diff[0]` have been initialized correctly (by convention `Incr[`-1`]` is empty).
- Once we are inside the loop, `n` is incremented. The last three lines of the loop compute `Incr[n]` and `Diff[n]` for that new value. The following key Lemma ensures that (Inv1) and (Inv2) are maintained.

Lemma 3.1. *Assume $n > 0$. The following are equivalent:*
(a) $s \models \mathsf{EF}_{\leq n}\varphi \;\wedge\; \neg\mathsf{EF}_{<n}\varphi$.
(b) $\exists d \in \{1, \ldots, n\}, \exists s', s'' \in S :$ $\begin{cases} 1.\ s' \xrightarrow{d} s'', \textit{ and} \\ 2.\ s'' \models \mathsf{EF}_{\leq n-d}\varphi \;\wedge\; \neg\mathsf{EF}_{<n-d}\varphi, \textit{ and} \\ 3.\ s \models \mathsf{E}(\neg\mathsf{EF}_{<n}\varphi)\mathsf{U}_{=0}(s' \;\wedge\; \neg\mathsf{EF}_{<n}\varphi). \end{cases}$

Additionally, (b) implies that there is no $s' \xrightarrow{d'} s''$ with $d' < d$.

- Once the invariants are established, we can check the return values. The algorithm returns a number `n` iff it is the smallest value s.t. *Start* intersects `Diff[n]`. This is correct by (Inv1) and (Inv2).
- The algorithm returns ∞ iff `Incr[n]` $=$ `Incr[n-`$d_{\max}$`]` for some `n` $\geq d_{\max}$ s.t. *Start* does not intersect `Incr[n]`. In such a case, all `Incr[`m`]` for $m \geq$ `n` are equal (stabilization has been reached at index `n` and detected at index `n` $+ d_{\max}$) and *Final* is not reachable from *Start*. Hence the returned value is correct.
- Finally, the algorithm must terminate since, as can be seen from (Inv1), the `Incr[n]`'s are increasing.

Some comments on the algorithm. When comparing this algorithm to the standard one (assuming durations are encoded as sequences of unit-length transitions between intermediate states), it could be noted that we compute more `Pre`'s: The standard algorithm computes one `Pre` and one union of BDDs per iteration of the loop computing `min`, while Algorithm 1 computes (at most) $d_{\max}$ `Pre's` and as many unions per iteration. However, our computations use much smaller BDDs:

- When considering unit-length transitions, the sets of states computed by the standard algorithm contain some "fake" intermediate states, leading to larger BDDs and unnecessary computations that are not performed with our algorithm;
- We split the transition relation into one transition BDD for each duration occuring in the model.

Also, a specific feature of our algorithm is that it uses the R'_d relations instead of the R_d's. This is correct in situations where shorter transitions have no disadvantages (*e.g.* when looking for minimum delays, or for evaluating $\mathsf{EU}_{\leq k}$ modalities). In practice, using R'_d results in using smaller relations and a smaller Dur' set. The difference between R_d and R'_d can be dramatic in examples where the system can nondeterministically pick any large enough duration for its steps, as in the `bridge` benchmark (see Section 6);

OBDD-based implementation. Algorithm 1 leads naturally to an OBDD-based implementation. As usual, we assume a state is given by the values of state variables $x_1, \ldots, x_m$. The transition relation is an OBDD-encoded predicate $R(x_1, \ldots, x_m, d, x'_1, \ldots, x'_m)$. Our implementation precomputes Dur' and the R'_k relations for $k \in Dur'$. This only requires simple Boolean operations on OBDDs. An OBDD for R' is obtained via

$$R' := \forall d'[R \wedge (d \leq d' \vee \neg R[d/d'])]. \tag{1}$$

(Here $R[d/d']$ is R where d' has been substituted for d.) Then Dur and Dur' (in predicate form) are obtained with:

$$Dur := \exists x_1, \ldots, x_m, x'_1, \ldots, x'_m R, \quad Dur' := \exists x_1, \ldots, x_m, x'_1, \ldots, x'_m R'. \tag{2}$$

And, for $k \in Dur$ (resp. in Dur'), the relations R_k and R'_k are obtained with:

$$R_k := \exists d(R \wedge d = k), \qquad R'_k := \exists d(R' \wedge d = k). \tag{3}$$

Some other implementation details are not so relevant. For example, observe that it is not necessary to store all previous `Incr[`i`]` and `Diff[`i`]`: Our implementation only stores the last $1 + d_{\max}$ values.

4 Symbolic model checking of RTCTL properties

We describe algorithms for the main RTCTL modalities. As is usual in symbolic model checking, these algorithms accept sets of states (corresponding to the subformulae that are combined by the modality) and return a set of states, where the compound formula holds.

4.1 Zero-duration conditions

Modalities with "=0" constraints play a special role, as seen in Section 3.

$\mathsf{EU}_{=0}$: Formula $\mathsf{E}\varphi\mathsf{U}_{=0}\psi$ holds in all states where it is possible to reach a state satisfying ψ all the while visiting only states satisfying φ *and only using zero-length steps*. Our procedure for the $\mathsf{EU}_{=0}$ modality mimics the standard NuSMV routine for the (untimed) CTL modality EU, with the difference that R_0 rather than R is taken as the underlying transition relation.

$\mathsf{EG}_{=0}$: This modality can be seen as a weak until: $\mathsf{EG}_{=0}\varphi$ means that φ must hold until we (possibly) reach a state by firing a non-zero duration transition. We use the modality $\mathsf{EU}_{=0}$ above for the strong until case, and add states from which there exists a path verifying $\mathsf{G}\varphi$ and only using zero-steps.

$\mathsf{AU}_{=0}$: Formula $\mathsf{A}\varphi\mathsf{U}_{=0}\psi$ is equivalent to $\mathsf{AF}_{=0}\psi \wedge \mathsf{A}\varphi\mathsf{U}\psi$, and $\mathsf{AF}_{=0}\psi$ is equivalent to $\neg\mathsf{EG}_{=0}\neg\psi$.

4.2 RTCTL modalities with $\leq k$ constraints

The procedures for the $\mathsf{EU}_{\leq k}$ and the $\mathsf{EG}_{\leq k}$ modalities work by computing iteratively a sequence `Incr[0]`, `Incr[1]`, ..., `Incr[k]` of sets associated with $\mathsf{E}\varphi\mathsf{U}_{\leq i}\psi$ (resp. $\mathsf{EG}_{\leq i}\varphi$) for $i = 0, 1, \ldots, k$, and returning `Incr[k]` Other modalities with $\leq k$ constraints are readily obtained from the above ones.

We do not describe these procedures in more detail here since they are very similar to the functions that compute minimum delays: Compare the specification for the sequence `Incr[0]`, `Incr[1]`, ... and the invariant (Inv1).

4.3 RTCTL modalities with interval constraints

The $\mathsf{EU}_{[k..l]}$ modality. For $\mathsf{EU}_{[k..l]}$ we use Algorithm 2. Write w for the *width* $l - k + 1$ of the interval $[k..l]$ (and assume that both w and l are nonnegative). Algorithm 2 performs $l+1$ steps, computing intermediary results. The same comments as for the minimum delay algorithm could be issued here: this algorithm computes more `Pre`'s and unions than an equivalent algorithm with unit-length transition, but they involve much smaller BDDs.

The correctness of the algorithm relies on the following Lemma:

```
function EUint(S_1, S_2, k, l)
w ← l - k + 1; i ← 0;
if (w≤0 || l<0) { return(FALSE) }
/* Initialization (step i=0)*/
for (j←1; j ≤min(w, d_max); j++) {
  Incr[j] ← EU_{=0}( S_1, S_2 ∩ EF_{=0}( ⋃{Pre[R_d](⊤) | d ≥j} ) )
}
for (j← w+1; j≤d_max; j++) { Incr[j] ← ∅ }
Incr[d_max + 1] ← ∅
aux0, Res ← EU_{=0}( S_1, S_2 );
/* Loop (steps i=1..l)*/
for (i←1; i≤ l; i++) {
  for (j←1; j≤ d_max; j++) {
    Incr[j] ← Incr[j+1] ∪ EU_{=0}( S_1, S_1 ∩ Pre[R_j](Res) ) ;
  }
  Res ← Incr[1];
  if (i<w) { Res ← Res ∪ aux0 }
}
return(Res);
```

Algorithm 2: Algorithm for $\mathsf{EU}_{[k..l]}$

Lemma 4.1. *After step i, the following invariants hold for all j between 1 and $d_{\max}+1$:*

$$s \in \texttt{Incr[}j\texttt{]} \textit{ iff } \begin{bmatrix} \textit{there is a path } \pi = s \xrightarrow{0} \cdots \xrightarrow{0} s' \xrightarrow{d} s'' \cdots \textit{ s.t.} \\ (a)\ d \geq j, \textit{ and} \\ (b)\ \pi \models S_1 \mathsf{U}_{[i+j-w..i+j-1]} S_2. \end{bmatrix} \tag{Inv4}$$

$$s \in \texttt{Res} \textit{ iff } s \models \mathsf{E} S_1 \mathsf{U}_{[i+1-w..i]} S_2. \tag{Inv5}$$

Other modalities The algorithm for modality $\mathsf{EG}_{[k..l]}$ is similar to the previous one, and also has to consider "border case" paths never reaching an interesting cumulative duration.

The modality $\mathsf{AU}_{[k..l]}$ is handled using previous algorithms: We use the following equivalence, which holds whenever $k > 0$:

$$\mathsf{A}(\varphi \mathsf{U}_{[k..l]} \psi) \equiv \mathsf{AF}_{[k..l]} \psi \ \wedge\ \mathsf{AG}_{<k}(\varphi \ \wedge\ \mathsf{AX}(\mathsf{A}\varphi \mathsf{U} \psi))$$

Algorithms for modalities for "$\geq$ k"-constraints are not described in the paper, but they are very similar to the above ones.

5 TSMV: embedding our algorithms on top of NuSMV

We now describe how we implemented our algorithms on top of NuSMV [11]. The resulting model checker is called TSMV and is available at `http://www.lsv.ens-cachan.fr/~markey/TSMV/`, together with all the examples of Section 6.

Extending NuSMV was quite easy and, essentially, we added about 3.500 lines of `C` code in the model checking section of the tool. These implement the procedures described in earlier sections (Algorithms 1, 2 and other timed modalities described in section 4), and the computations of sets Dur, Dur', Dur'' and relations R_d, R'_d described in section 3.

5.1 Defining TKS's

For describing TKS's, we opted for the simplest and most practical solution. When describing a model, we use normal NuSMV syntax and reserve an additional state variable, called `duration`, for specifying the duration of steps. Hence, instead of specifying durational TKS steps of the form

$$\langle v_1, \ldots, v_m \rangle \xrightarrow{d} \langle v'_1, \ldots, v'_m \rangle \qquad \text{(TKS step)}$$

we describe simple KS steps of the form

$$\langle v_1, \ldots, v_m, \star \rangle \to \langle v'_1, \ldots, v'_m, d \rangle \qquad \text{(extended KS step)}$$

For example, the TKS from Fig. 1 could be defined with

```
VAR
  state: 1..3;
  duration: 0..21;
TRANS
   (state = 1 & next(state)=2 & next(duration) = 0)
 | (state = 2 & next(state)=1 & next(duration) = 0)
 | (state = 2 & next(state)=3 & next(duration) = 21)
...
```

More realistic examples can be found in Section 6.

Using this scheme has a number of benefits. It lets us reuse all the NuSMV machinery for assembling and instantiating modules. Furthermore, it allows using symbolic constraints when defining sets of possible durations between states (see e.g. the `bridge` example).

Once NuSMV has computed an OBDD for the transition relation of the extended KS, it is easy to extract the relations of the underlying TKS, with the projections and other Boolean operations described in Section 3.

5.2 Querying the TKS

TSMV reuses NuSMV syntax for RTCTL formulae. Additionally, we support RTCTL modalities with right-open intervals. Computing minimum and maximum delays reuses NuSMV syntax too. For example, one would write:

```
SPEC AG>=4 (p2 -> AF<=21 ! p1)
COMPUTE MAX[p1 | p2, state = 2]
```

5.3 Extra features

As much as possible, TSMV reuses features from NuSMV. For example:

Fairness: For simplicity, our exposition in Section 2 did not mention fairness issues. But fairness constraints *à la* SMV are allowed in TSMV, and only fair paths are considered. Fairness constraints can involve the `duration` variable, thus TSMV can express the so-called "non-Zeno" behavior.

Counter-example generation: In some cases, it is possible to reuse NuSMV counter-example generation features. This could be done e.g. with "=0" constraints, that can easily be expressed in pure RTCTL (using the subformula `duration=0`). For other modalities, our algorithms do not rely enough on NuSMV temporal primitives and we are developping our own counter-example generation functions for these cases.

6 Experimental results

6.1 The bridge-crossing problem

The problem. The bridge-crossing problem is a famous mathematical puzzle with time critical aspects [21, 18]. A group of four persons, called P1, P2, P3 and P4, cross a bridge at night. It is dark and one can only cross the bridge with a lamp. Only one lamp is available and at most two persons can cross at the same time. Therefore any solution requires that, after the first two persons cross the bridge, one of them returns, carrying back the lamp for the remaining persons. The four persons have different maximal speeds: here P1 crosses in 5 time units (t.u.), P2 in 10 t.u., P3 in 20 t.u. and P4 in 25 t.u. When a pair crosses the bridge, they move at the speed of the slowest person in the pair. Now, how much time is required before the whole group is on the other side?

Bridge-crossing in TSMV. A person is described as an SMV module with his crossing time as a parameter. His possible steps are to stay where he is, or move to the other side. He can only cross when the lamp is on his side (and then the lamp crosses with him). When he crosses, the transition takes at least his crossing time. This way, when four persons are synchronized, the crossing time is any integer greater or equal to the maximum crossing time of the crossing persons. The complete system is obtained by combining four persons (four instances of the same `person` module, with different crossing times) with a Boolean `lamp` value keeping track of the position of the lamp, and adding a further constraint (an `INVAR` in SMV) telling that at most two persons cross in one move. The system is further labeled with two propositions: `initial` for the initial configuration, and `safe` for the configurations where everyone is on the other side of the bridge.

We can ask how much time is required for crossing:

```
COMPUTE MIN[initial, safe]
```

The answer (60 t.u.) is obtained in a few milliseconds.

It turns out that 60 t.u. is the best total crossing time *from the initial configuration*. But if we let people move freely and at some time ask them to cross quickly, can it happen that they are in a configuration where more than 60 t.u. are required? We ask

```
SPEC  AG EF<=60 safe
```

and get a negative answer: in some reachable configurations, more than 60 t.u. are required.

Indeed, if we start after just one person has crossed, that person will have to come back before we can implement the 60 t.u. solution. Hence 85 t.u. (=25+60) are sometimes required. We check that this is indeed the worst case with

```
SPEC  (AG EF<=85 safe) & !(AG EF<=84 safe)
```

Comparison with related tools. The same example can be treated with NuSMV, Verus or RAVEN. Since NuSMV and Verus only handle unit steps, we use the method advocated in [7, p.106] and introduce a counter forcing several t.u. between actual system moves. It can be argued that these models are slightly more cumbersome. With RAVEN, we can directly specify duration intervals for each transitions, even though it internally considers the semi-continuous semantics.

The command `COMPUTE MIN[initial, safe]` produces the (correct) answer but takes significantly more time with NuSMV, Verus and RAVEN than with TSMV (see Table 1).

When verifying `SPEC (AG EF<=85 safe) & !(AG EF<=84 safe)` we obtain a negative answer! This is because the semantical models are different. With all three tools, the semi-continuous semantics introduces intermediary positions along what are long steps in our TKS's, and the AG modality quantifies over these intermediate positions too.

It can be argued that the semi-continuous semantics is "better", or closer to specifier's purposes. However, it makes verification more costly. Furthermore, it is slightly less general: our TKS's generalize unit-steps KS's, while TKS semantics can only be simulated akwardly in KS's. The previous example shows that a counter-based encoding is not enough: one further has to adapt the RTCTL formulae. For example, in the bridge problem, we can introduce a new proposition, `crossing`, labeling intermediate configurations that should not exist in the TKS. Then the following formula is satisfied:

```
SPEC  AG (!crossing -> EF<= 85 (safe & !crossing)).
```

6.2 TSMV and sensitivity to scaling up the durations

The bridge problem is an example of a system where durations are mostly "long" (greater than 1). Our algorithms are designed in such a way that only useful durations are considered. As a result, TSMV is mostly insensitive to scaling up the durations. When we define a model "`bridge x 10`" by replacing 5, 10, 20 and 25 with (resp.) 50, 100, 200, and 250, TSMV computes the minimum delay of 600 t.u. in more or less the same time it needed for the initial problem [4].

[4] The TKS defined with `bridge x 10` is not *exactly* a scaling up of `bridge`. It uses conditions of the form "`duration >= 100`" that allow all values $100, 101, 102, \ldots, d_{\max}$.

	TSMV		NuSMV		Verus		RAVEN	
	time	mem	time	mem	time	mem	time	mem
bridge	0.02 s	1.3 M	0.21 s	9.5 M	7.14 s	16.5 M	4.01 s	5.9 M
bridge×10	0.04 s	1.3 M	23.24 s	18.5 M	259.54 s	39.2 M	3 098 s	371 M
bridge×20	0.07 s	1.3 M	120.13 s	18.5 M	573.05 s	44.1 M		
bridge×50	0.22 s	9.0 M	2 209 s	35.0 M	3 626 s	55.0 M		
bridge×100	0.45 s	11.0 M	14 296 s	65.0 M	17 870 s	59.2 M		

Table 1. Scale up (in)sensitivity

By contrast, the computation time for NuSMV, Verus and RAVEN increases dramatically when we scale up the durations. In fact, there is no way to avoid this: These tools do not know about TKS's and are bound to compute all sets associated with different values of the counter for intermediate states. Computing these sets is a tedious and mostly repetitive task that cannot be avoided unless a notion of TKS is introduced.

Table 1 shows how running time and memory requirements grow when we scale up the timing parameters of the bridge problem.

6.3 Verifying the PCI local bus

A model of the PCI local bus was analyzed by Campos *et al.* in [8], computing minimum and maximum delays with symbolic model checking techniques. This model is a standard example that comes with the NuSMV distribution. We refer to [8] for more explanations on this case study. The reader only needs to know that it is a large example where NuSMV can show that a complete transaction might take up to 244 steps, from the request of the bus to the end of the transaction.

First of all, we added duration 1 to all transitions. This allows a direct comparison with NuSMV. In that case, we were able to prove the aforementioned maximal time for a complete transaction, by adding the following lines to the `main` module of the PCI model:

```
VAR  duration: 0..1;
TRANS  next(duration)=1
COMPUTE MAX[ req0, isa_bridge.end_transaction ]
COMPUTE MAX[ req1, scsi_ctrl.end_transaction ]
COMPUTE MAX[ req2, vga_ctrl.end_transaction ]
COMPUTE MAX[ req4, processor.end_transaction ]
```

(E_1)

For the `isa_bridge` and `scsi_ctrl` components, the result is 244. It is 130 for the other two components (see explanations for this difference in [8]).

Counting specific kinds of events is possible. For instance, one can use durations to count the number of transactions issued on the PCI bus between a request and a grant of each of the masters.

```
   DEFINE start_tr := processor.start_transaction |
                       vga_ctrl.start_transaction |
                        scsi_ctrl.start_transaction |
                       isa_bridge.start_transaction ;
   VAR  duration: 0..1;
   TRANS  next(duration)=case
                start_tr: 1;
                1        : 0;
          esac
   COMPUTE MAX[ req0, grant=0 ]
   COMPUTE MAX[ req1, grant=1 ]
   COMPUTE MAX[ req2, grant=2 ]
   COMPUTE MAX[ req4, grant=4 ]
```

(E_2)

TSMV answers that up to 5 transactions might start between request and grant for both the `isa_bridge` and the `scsi_ctrl` components, and 2 for the other two masters. These are the values given in [8], where they were obtained with a special condition counting Algorithm.

With the following lines, we count the amount of data the processor can transfer between a request and its corresponding grant:

```
(in module bus_master)
   VAR transmitting: boolean;
   ASSIGN init(transmitting) := FALSE;
          next(transmitting) := (count>next(count));
(in module main)
   VAR  duration: 0..1;
   TRANS  next(duration)=case
                processor.transmitting : 1;
                1                      : 0;
          esac
   COMPUTE MAX[ req0, grant=0 ]
   COMPUTE MAX[ req1, grant=1 ]
   COMPUTE MAX[ req2, grant=2 ]
   COMPUTE MAX[ req4, grant=4 ]
```

(E_3)

The result is 30 for the `isa_bridge`, the `scsi_ctrl` and the `vga_ctrl`, and 15 for the `processor` (the total amount of data that a master can transmit in one "session" is limited to 15 in this model).

Last, we verify that, when a transaction is aborted, the component has to request for the bus before being able to transmit anew. We verify this property for the processor by counting only data transfers of the processor, and specifying that, after abortion, he can transmit at most one bit of data if it does not assert a new grant. This is achieved with the following additions to `pci4p.smv`:

```
(in module bus_master)
   VAR transmitting: boolean;
   ASSIGN init(transmitting) := FALSE;
          next(transmitting) := (count>next(count));
(in module main)
   VAR  duration: 0..1;
   TRANS  next(duration)=case
                processor.transmitting : 1;
                1                      : 0;
          esac
 SPEC AG ((processor.transmitting & abort) -> !E [ (!req4) U>=2 TRUE ] )
```

(E_4)

```
SPEC EF ((processor.transmitting & abort) ->  E [ (!req4) U= 1 TRUE ] )
SPEC EF ((processor.transmitting & abort) ->  E [ (!req4) U= 0 TRUE ] )
```

The table below summarizes the total time and memory consumption for the four examples above. Since NuSMV does not handle zero-duration transitions, it could only be applied in the first example:

	NuSMV		TSMV	
	time	memory	time	memory
(E_1)	178.37 sec.	21 228KB	186.34 sec.	26 932KB
(E_2)			69.59 sec.	24 340KB
(E_3)			306.74 sec.	31 780KB
(E_4)			13.56 sec.	33 900KB

Table 2. Verification of the PCI local bus

7 Comparison with other work

On the semantics of time. Emerson *et al.* [13] pioneered the investigation of RTCTL model checking on KS's with unit-length transitions. Campos, Clarke *et al.* [5, 10] and Kropf, Ruf *et al.* [14, 20] considered more general Timed Transition Systems (TTGs) where durations can be arbitrary (sets of) natural numbers. The difference between these models and our TKS's is that time in TTGs elapses "semi-continuously", *i.e.* using intermediate states, while TKS's transitions are atomic (see section 2.3). We chose atomic durations because they lead to more efficient model checking algorithms [15]. When it comes to modeling actual systems, the semi-continuous semantics is perhaps more natural (but we did not feel hampered by atomic durations in our case studies). Observe that it is easy to simulate semi-continuous durations in atomic durations (by adding intermediate states) while designing a simulation the other way around is more involved.

Another major issue is that consecutive zero-length transitions in TKS's are not amalgamated (which would make intermediate states disappear). Amalgamating these can make verification more efficient when zero-length transitions are just internal micro-steps, as in [7]. But when zero-length transitions are used in condition-counting applications, amalgamating them is costly (sometimes infeasible). Additionally, this loses a lot of the branching-time aspects that *CTL* talks about. Finally, these two different ways of treating zero-length transitions have very different applications.

On algorithmics. It may seem that our symbolic algorithms are not very different from earlier algorithms for RTCTL model checking: these procedures all compute fixed-points iteratively. However, an important difference exists: all our algorithms use information on what durations really appear in the model. In connection with the atomicity of durations, this leads to more efficient algorithms. In particular, it makes model checking mostly insensitive to scaling up the durations, as illustrated in section 6.2. Additionally, the algorithms we designed for the "$\geq k$" and "$\leq k$" constraints benefit from considering a derived transition relation where only maximal (resp. minimal) durations appear (see Section 3).

On implementations. Several symbolic algorithms for discrete-time models have been published, and some of them consider models with arbitrary durations on transitions. However, not all published algorithms have been implemented. NuSMV and Verus only support unit-length durations. RAVEN uses the semi-continuous semantics and does not allow null durations. TSMV is, to our knowledge, the first tool dealing with (general) integer durations in discrete time models. Similarly, "$\geq k$" constraints are not available in the other tools.

8 Conclusion

We proposed OBDD-based symbolic verification procedures for model checking RTCTL properties and for computing extremum delays. The underlying models are (symbolic descriptions of) TKS's, or state graphs where transitions carry an arbitrary duration.

For the analysis of timed systems, this extends the verification facilities that are offered in NuSMV (based on earlier work by Campos, Clarke *et al.*). Our algorithms are implemented on top of NuSMV, and allow all combinations of long, unit and zero-length steps. We also deal with an enlarged set of RTCTL modalities, and offer procedures for computing extremum delays.

The procedures we propose take advantage of the TKS model: we do not reduce a long step (e.g. 10 time units) to an implicit sequence of 10 unit steps. A consequence is that the behavior of our model checking algorithm enjoys a kind of insensitivity to scaling up of durations (a feature of tools based on timed automata). This suggests that there exist ways of bridging the gap that still exists between approaches *à la* SMV where clocks are considered like any other discrete variable, and approaches based on timed automata, à la UPPAAL and Kronos, still in need of efficient symbolic representations handling combinatorial control and clock valuations.

References

1. B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools.* Springer, 2001.
2. P. Bouyer. Forward analysis of updatable timed automata. *Formal Methods in System Design*, 24(3):281–320, 2004.
3. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. & Comp.*, 98(2):142–170, 1992.
4. S. Campos. *Verus 0.9 – Reference Manual.* Pittsburgh, PA, USA, March 1997.
5. S. Campos and E. M. Clarke. Real-time symbolic model checking for discrete time models. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Development*, vol. 2 of *AMAST Series in Computing*, pp. 129–145. World Scientific, 1995.

6. S. Campos and E. M. Clarke. Analysis and verification of real-time systems using quantitative symbolic algorithms. *J. Software Tools for Technology Transfer*, 2(3):260–269, 1999.
7. S. Campos and E. M. Clarke. The Verus language: representing time efficiently with BDDs. *Theor. Comp. Sci.*, 253(1):95–118, 2001.
8. S. Campos, E. M. Clarke, W. R. Marrero, and M. Minea. Verifying the performance of the PCI local bus using symbolic techniques. In *Proc. 1995 Int. Conf. on Computer Design (ICCD '95), VLSI in Computers and Processors, Austin, TX, USA, Oct. 1995*, pp. 72–78. IEEE Comp. Soc. Press, 1995.
9. S. Campos, E. M. Clarke, W. R. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *Proc. 15th IEEE Real-Time Systems Symposium (RTSS'94), San Juan, Puerto Rico, Dec. 1994*, pp. 266–270. IEEE Comp. Soc. Press, 1994.
10. S. Campos, M. Teixeira, M. Minea, A. Kuehlmann, and E. M. Clarke. Model checking semi-continuous time models using BDDs. In *Proc. 1st Int. Workshop on Symbolic Model Checking (SMC'99), Trento, Italy, July 1999*, vol. 23(2) of *Electronic Notes Theor. Comp. Sci.* Elsevier Science, 1999.
11. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *J. Software Tools for Technology Transfer*, 2(4):410–425, 2000.
12. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
13. E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. *Real-Time Systems*, 4(4):331–352, 1992.
14. J. Frößl, J. Gerlach, and Th. Kropf. An efficient algorithm for real-time symbolic model checking. In *Proc. European Design and Test Conference (ED&TC'96), Paris, France, Mar. 1996*, pp. 15–21. IEEE Comp. Soc. Press, 1996.
15. F. Laroussinie, N. Markey, and Ph. Schnoebelen. On model checking durational Kripke structures (extended abstract). In *Proc. 5th Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'2002), Grenoble, France, Apr. 2002*, vol. 2303 of *Lect. Notes Comp. Sci.*, pp. 264–279. Springer, 2002.
16. K. G. Larsen, P. Pettersson, and Wang Yi. UPPAAL in a nutshell. *J. Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.
17. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.
18. G. Rote. Crossing the bridge at night. *EATCS Bull.*, 78:241–246, 2002.
19. J. Ruf. RAVEN: Real-time analyzing and verification environment. *J. Universal Comp. Sci.*, 7(1):89–104, 2001.
20. J. Ruf and Th. Kropf. A new algorithm for discrete timed symbolic model checking. In *Proc. Int. Workshop Hybrid and Real-Time Systems (HART'97), Grenoble, France, Mar. 1997*, vol. 1201 of *Lect. Notes Comp. Sci.*, pp. 18–32. Springer, 1997.
21. T. C. Ruys and E. Brinksma. Experience with literate programming in the modelling and validation of systems. In *Proc. 4th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), Lisbon, Portugal, Mar. 1998*, vol. 1384 of *Lect. Notes Comp. Sci.*, pp. 393–407. Springer, 1998.
22. S. Yovine. Kronos: A verification tool for real-time systems. *J. Software Tools for Technology Transfer*, 1(1–2):123–133, 1997.