N. Markey and Ph. Schnoebelen

Symbolic model checking of simply–timed systems

Research Report LSV–03–14, Oct. 2003

Laboratoire
Spécification
et
Vérification

CENTRE NATIONAL
DE LA RECHERCHE
SCIENTIFIQUE

Ecole Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France

Symbolic model checking of simply-timed systems*

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

[1] Département d'Informatique
Université Libre de Bruxelles
[2] Lab. Spécification & Vérification
ENS de Cachan & CNRS UMR 8643
email: {markey|phs}@lsv.ens-cachan.fr

Abstract. We propose 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. Model checkers, and in particular tools based on *symbolic model checking* techniques relying on OBDD technology [3, 15], are able to analyze larger and larger models of systems and check whether they satisfy some behavioral specifications [11, 1].

One prominent family of systems where symbolic model checking techniques are harder to apply successfully are *time-critical systems*. These are systems whose behavior is influenced by the timing of events, and where the correctness of the behavior critically depends on timing issues. A popular approach to these systems is based on *timed automata*, for which several model checking tools exist [17, 14]. Timed automata are very expressive, but they raise difficult problems for symbolic model checking. 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, TCTL model checking is not harder than CTL model checking [12]. (TCTL, or "Timed CTL", is the extension of CTL where numerical time bounds can be added to temporal modalities.)

* The first author is supported by an INRIA Postdoctoral Grant.

TCTL model checking on KS's adapts easily to symbolic OBDD-based model checking [4]. The corresponding algorithms have been implemented (e.g. they are available in nuSMV [10]) and perform quite efficiently.

The main advantages of this approach are simplicity and efficiency. Its limitation is the restricted expressive power (even disregarding the issue of discrete vs. dense time): while timed automata may use several clocks to keep track of several distinct past events, KS's have to encode such information in the states.

Our contribution. We extend symbolic TCTL 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 TCTL 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?*", as advocated in [8].

Our algorithms exploit the semantics of TKS's: we do not reduce a long transition to a sequence of unit steps. Then, for models that would naturally be described with long transitions, our algorithms perform better than the algorithms offered in nuSMV (see Section 7).

Because nuSMV is open source and has a very clean architecture, it was very easy to implement our algorithms on top of it. This readily provides a symbolic model checker for TKS's, called TSMV and available at `http://www.lsv.ens-cachan.fr/~markey/TSMV/`. We report on our experiments with TSMV: the gain in expressive power is appreciable and the running time is improved when we compare with the verification of "equivalent" unit-steps KS's.

Finally, TSMV improves the available set of solutions for model checking systems where the control part is too large for a timed-automata 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: They 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 [5, 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: They 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 TCTL formulae count the durations of the other, time-relevant, transitions. This can also be used in models where transitions carry some cost, for an additive notion of cost that is 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 the model of an elevator, one can wish to only count cabin moves, or door closings and openings, and use these in cost-constrained CTL formulae.

Comparison with other work. Campos, Clarke *et al.* have pioneered the use of OBDD-based symbolic model checking of timed properties on KS's, assuming unit-steps [8, 7]. In [9] they extended TCTL model checking (and the extremum delays algorithms) to deal with zero-length transitions. Finally, in [6] they provide symbolic methods for counting the extremum number of times a condition holds along paths that satisfy some linear-time constraint (assuming unit-steps): this is one other way of counting special events only but computing extremum delays is less general than allowing arbitrary TCTL properties.

Another approach to instantaneous transitions is to amalgamate consecutive zero-length steps and produce a KS with only unit-length transitions, as is done in Verus [5]. This approach aims as producing a smaller OBDD (partly sacrificing the branching-time information) but this can fail when we have a too liberal understanding of what are these instantaneous steps, e.g. when one wants to count special events only.

When it comes to long durations, the TTGs described at the end of [8] is exactly equivalent to our TKS's. But no real algorithm is given (or implemented) for TTGs. In their other works, Campos, Clarke *et al.* suggest modeling long durations by a sequence of unit steps. Our experiments suggest that this approach is unnecessarily costly.

nuSMV implements TCTL model checking for the unit steps semantics only [10]. We observe that nuSMV does not deal with right-open TCTL constraints.

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 SMV [3, 10]. These notions are available in several textbooks, e.g. [11, 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 new thing in a TKS is that $R \subseteq S \times \mathbb{N} \times S$ is a finite set of *transitions* labeled by a natural number, called the *duration* of the transition.[3] We require that R is total.

[3] We restrict to finite sets of durations between two states for simplicity. Some infinite sets, e.g. right-open intervals, could be accommodated at no extra cost [13].

Here is a simple example. Observe 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.

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* $\mathsf{D}\pi[n..m]$ 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.

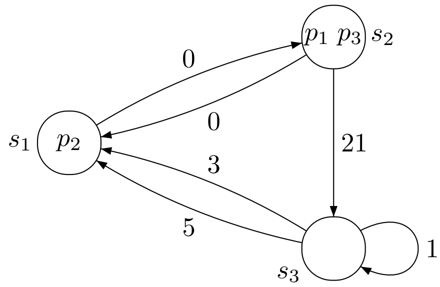

Fig. 1. A simple TKS

2.2 TCTL: a temporal logic with timing constraints

TCTL formulae are 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. Typically, all constraints of the form "$\leq k$", "$= k$" and "$\geq k$" for k a value in the relevant domain (here $\mathbb{N}$) are allowed [4]. Sometimes, slight extensions are allowed. For example, nuSMV allows *interval constraints* of the form "$\in [k_1..k_2]$" (but does not admit "$\geq k$" constraints). 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 TCTL and TKS's:

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

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

[4] For convenience we allow negative numbers appearing in constraints. The constraints still apply to positive durations. For instance $\mathsf{EF}_{[-3..5]}\varphi \equiv \mathsf{EF}_{[0..5]}\varphi$ and $\mathsf{EF}_{\leq -2}\varphi \equiv \bot$.

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). The most crucial difference is not the issue of discrete vs. dense time. We shall rather comment on Zeno behaviors and, most importantly, the atomicity of durations:

Zeno behaviors: We allow paths where the durations are eventually all zero. This simplifies the exposition, and allows using durations for counting things other than "elapsed time", e.g. counting some events of interest. If need be, Zeno behaviors can be ruled out by fairness constraints (see Section 6.3).

Atomicity of durations: When performing a transition $s \xrightarrow{10} s'$, the system moves from being in state s at some current time t to being in s' at $t + 10$: between t and $t + 10$, there is no state (or time) where the system is. In particular, the system in Fig. 1 has $s_1 \models \mathsf{AG}_{=2}\bot$ since no path starting from s_1 encounters a cumulative duration of 2.

Considering atomic durations makes TCTL model checking easier. We shall see why later, when we describe our algorithms. Observe that the reduction in complexity also occurs at a theoretical level: a semantics with implicit (integer-valued) intermediary points, *à la* timed automata, would make TCTL model checking PSPACE-complete, while it is Δ_2^p-complete with our semantics [13].

3 Symbolic computation of minimum delays

Assume $M = \langle S, S_0, R, L\rangle$ is a TKS. With $d_{\max}$ we denote the highest duration occurring as a label in R. For $d = 0, \ldots, d_{\max}$, we 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 two states that are connected by a transition with duration d and R'_d relates them if, additionally, they are not related by a "shorter" transition. Finally, we 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 $\{d'_1, \ldots, d'_m\}$ (in increasing order) of all durations for which R'_d is non-empty. Observe that *Dur* contains exactly the labels that appear in R (hence $d_l = d_{\max}$) and that $\mathit{Dur}' \subseteq \mathit{Dur}$.

The *minimum delay* between a set $\mathit{Start} \subseteq S$ of starting states and a set $\mathit{Final} \subseteq S$ 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 $d = d_1 + \cdots + d_n$ [8]. 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.

```
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

The procedure and its correctness proof. With Algorithm 1 we provide a backward chaining procedure computing minimum delays. Proving that this algorithm is correct can be done 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 $\mathtt{Incr[}-1\mathtt{]}$ 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 (proved in Appendix A) 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) $\exists d \in \{1, \ldots, n\}, \exists s', s'' \in S : \begin{cases} 1.\ s' \xrightarrow{d} s'', \text{ and} \\ 2.\ s'' \models \mathsf{EF}_{\leq n-d}\varphi \;\wedge\; \neg\mathsf{EF}_{<n-d}\varphi, \text{ and} \\ 3.\ s \models \mathsf{E}(\neg\mathsf{EF}_{<n}\varphi)\mathsf{U}_{=0}(s' \;\wedge\; \neg\mathsf{EF}_{<n}\varphi). \end{cases}$
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).
- The algorithm returns ∞ iff $\mathtt{Incr[n]} = \mathtt{Incr[n-}d_{\max}\mathtt{]}$ for some $\mathtt{n} \geq d_{\max}$ s.t. *Start* does not intersect `Incr[n]`. In such a case, all $\mathtt{Incr[}m\mathtt{]}$ for $m \geq \mathtt{n}$ are equal (stabilization has been reached at index `n` and detected at index $\mathtt{n} + d_{\max}$) 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. 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 we let the system nondeterministically pick any large enough duration for its steps, as in the `bridge` benchmark (see Section 7).

OBDD-based implementation. The code given in 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:

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

$$Dur' := \exists x_1, \ldots, x_m, x'_1, \ldots, x'_m R'. \tag{3}$$

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

$$R_k := \exists d(R \wedge d = k), \tag{4}$$

$$R'_k := \exists d(R' \wedge d = k). \tag{5}$$

Finally, the routines for $\mathsf{EF}_{=0}$ and $\mathsf{EU}_{=0}$ that are required in Algorithm 1 are obtained from the routines for EF and EU that exist in nuSMV (see Section 5.1).

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 computation of maximum delays

The *maximum delay* between *Start* and *Final* (two sets of states) is the longest time that paths from *Start* can avoid reaching *Final*. (The maximum delay is infinite if some path from *Start* avoids *Final* forever.) Compared to plain KS's, our TKS's make the computation of maximum delay trickier because of zero-length steps.

Our procedure for computing maximum delays shares similarities with the one for minimum delays. Here we iteratively compute the set of states satisfying $\mathsf{EG}_{\leq n}\neg Final$, (or, equivalently, not satisfying $\mathsf{AF}_{\leq n} Final$) for $n = 0, 1, 2, \ldots$

until it does not intersect with *Start* anymore, or until it reaches a fixed point (whichever happens first).

The procedure uses relations R''_d for $d = 0, \dots, d_{\max}$. These are defined like the R'_ds but with longest transitions, instead of shortest. Formally we let $(s, s') \in R''_d$ iff $s \xrightarrow{d} s'$ and there is no $s \xrightarrow{e} s'$ with $e > d$. Dur'' is the set of d s.t. R''_d is not empty. OBBDs for Dur'' and the R''_ds are readily obtained by something similar to Eq. (3) and (5).

The procedure also uses an auxiliary call to $\mathtt{EG}(F \wedge \mathtt{duration} = 0)$, where we compute the set of states from which there exists an infinite sequence of zero-length transitions that always remain inside set F. Like $\mathsf{EU}_{=0}$, this is just a variant of (untimed) CTL model checking where the transition relation is restricted to R_0 (see Section 5.1).

The procedure and its correctness proof. Algorithm 2 presents a backward chaining procedure computing maximum delays. Proving that this algorithm is correct

```
function max_delay(Start,Final)
aux0 ← EG(¬Final ∧ duration = 0);
n ← 0;
Incr[0] ← EG_{=0}( ¬Final );
repeat {
    if ( Start ∩ Incr[n] = ∅ ) { return(n); }
    n++;
    aux1 ← ⋃{Pre[R''_d](Incr[n-d]) | d ∈ Dur'' ∧ 0 < d ≤ n};
    aux2 ← aux1 ∪ ⋃{Pre[R''_d](⊤) | d ∈ Dur'' ∧ d > n};
    aux3 ← EU_{=0}( ¬Final, ¬Final ∩ aux2 );
    Incr[n] ← aux3 ∪ aux0
} until (n≥ d_max ∧ Incr[n] = Incr[n-d_max]);
return(∞);
```

Algorithm 2: Algorithm for computing maximum delay

can be done by showing that the following invariant holds whenever we enter the `repeat` loop:

$$0 \leq i \leq \mathtt{n} \;\Rightarrow\; \Big[s \in \mathtt{Incr[}i\mathtt{]} \text{ iff } s \models \mathsf{EG}_{\leq i} \neg \mathit{Final} \Big]. \tag{Inv3}$$

– (Inv3) clearly holds the first time the loop is entered: `n` is 0 and `Incr[0]` has been initialized correctly.
– Once we are inside the loop, `n` is incremented. The last three lines compute `Incr[n]` for that new value. The following easy Lemma (proof omitted) ensures that (Inv3) is maintained.

Lemma 4.1. *Assume $n > 0$, $s_0 \in S$, and $F \subseteq S$. Then $s_0 \models \mathsf{EG}_{\leq n} F$ iff*
(a) there exists an infinite path $s_0 \xrightarrow{0} s_1 \xrightarrow{0} \dots$ of zero-length steps with

$s_0, s_1, \ldots \in F$, *or*
(b) there exists a path $s_0 \xrightarrow{0} s_1 \xrightarrow{0} \ldots \xrightarrow{0} s_i \xrightarrow{d} s_{i+1}$ *with* $s_0, s_1, \ldots, s_i \in F$, $d > 0$ *and* $s_{i+1} \models \mathsf{EG}_{\leq n-d} F$.

Inside the loop, `aux1` computes the set of potential s_i's from case *(b)* when it is assumed that $d \leq$ `n`. `aux2` further adds s_i's for $d >$ `n` (in this case $s_{i+1} \models \mathsf{EG}_{<0} F$ necessarily holds). Then `aux3` collects all s_0's that satisfy case *(b)*. There only remains to add `aux0`, the (fixed) set of states that satisfy case *(a)*.
Finally, it is clear that using R'' instead of R is correct: the Lemma remains true if in case *(b)* we further assume that $(s_i, s_{i+1}) \in R''_d$.

– Once we know the invariant holds we can check the return values. To begin with, (Inv3) implies that the sequence of `Incr[n]`'s decreases. The procedure returns some `n` iff $s \models \mathsf{AF}_{\leq n} \mathit{Final}$ for all $s \in \mathit{Start}$ but there is some $s' \in \mathit{Start}$ with $s' \not\models \mathsf{AF}_{<n} \mathit{Final}$. Hence `n` is indeed the maximum delay between *Start* and *Final*.
– The procedure returns ∞ if after the `n`th loop it holds that `Incr[n]` = `Incr[n-`$d_{\max}$`]` (and `n` $\geq d_{\max}$). This implies equality of all `Incr[`i`]`'s for $i \geq$ `n`. Thus there is some $s \in \mathit{Start}$ that satisfies all $\mathsf{EG}_{\leq i} \neg \mathit{Final}$, i.e. $s \models \mathsf{EG} \neg \mathit{Final}$ and the maximum delay is indeed ∞.

5 Symbolic model checking of TCTL properties

We describe algorithms for the main TCTL 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.

5.1 Zero-duration conditions

Modalities with "=0" constraints play a special role, as we already saw in Sections 3 and 4.

$\mathsf{EU}_{=0}$: This modality is used in Algorithms 1 and 2. $\mathsf{E}\varphi\mathsf{U}_{=0}\psi$ 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 is used in Algorithm 2. It can be seen as a weak until: $\mathsf{EG}_{=0}\varphi$ means that φ must hold until (possibly) a non-zero duration is fired. 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-length steps.

$\mathsf{A}\varphi\mathsf{U}_{=0}\psi$: 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 $\neg\mathsf{EG}_{=0}\neg\psi$.

5.2 TCTL modalities with $\leq k$ constraints

The procedures for the $\mathsf{EU}_{\leq k}$ and the $\mathsf{AU}_{\leq k}$ 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. $\mathsf{A}\varphi\mathsf{U}_{\leq i}\psi$) for $i = 0, 1, \ldots, k$, and returning `Incr[k]`

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

5.3 TCTL modalities with interval constraints

Dealing with a modality featuring an interval constraint like $\mathsf{EU}_{[k..l]}$ combines problems raised by the $\mathsf{EU}_{\geq k}$ and $\mathsf{EU}_{\leq l}$ cases.

The $\mathsf{EU}_{[k..l]}$ modality. For $\mathsf{EU}_{[k..l]}$ we use Algorithm 3. Write w for the *width*

```
function EUint(S₁, S₂, 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( S₁, S₂ ∩ 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( S₁, S₂ );

/* 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( S₁, S₁ ∩ Pre[R_j](Res) ) ;
  }
  Res ←  Incr[1];
  if (i<w) { Res ← Res ∪ aux0 }
}
return(Res);
```

Algorithm 3: Algorithm for $\mathsf{EU}_{[k..l]}$

$l - k + 1$ of the interval $[k..l]$ (and assume that both w and l are nonnegative). Algorithm 3 performs $l+1$ steps, computing intermediary results. The correctness of the algorithm relies on the following Lemma:

Lemma 5.1. *After step i, the following invariants hold for all $j = 1, \ldots, d_{\max} + 1$:*

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

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

The proof of these invariants is given in Appendix B.

5.4 Other modalities

The algorithm and proof of correctness for the modality $\mathsf{EG}_{[k..l]}$ are given in appendix C.

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$0"-constraints are not described in the paper, but they are very similar to the above ones, except that we modify the initial assignments and the bounds of the loop. Moreover, these computations use the maximal duration relation R'' defined in section 4.

6 TSMV: embedding our algorithms on top of nuSMV

We now describe how we implemented our algorithms on top of nuSMV [10]. The resulting model checker is called TSMV and is available at `http://www.lsv.ens-cachan.fr/~markey/TSMV/`, together with all the examples we mention in section 7.

6.1 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

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

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 7 and in Appendix D.

Using this scheme has a number of benefits. It lets us reuse all the nuSMV machinery for building complex models by assembling and instantiating modules. Furthermore, it makes it easy to describe transition relations where the steps between some state s and some state s' admit many different durations, something that is very useful when we synchronize several modules and have them "wait" for the slowest module when performing a joint step (see e.g. the `bridge` example).

Once nuSMV has combined the modules and 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.

6.2 Querying the TKS

TSMV uses exactly the same syntax that nuSMV uses for TCTL formulae. Additionally, we support TCTL modalities with right-open intervals. Computing minimum and maximum delays reuse the nuSMV syntax too. For example, one would write:

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

6.3 Extra features

As much as possible, TSMV reuses the 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 [5].

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

[5] This only involves slight modifications in the algorithms we described, where one sometimes has to check that given states have at least one fair path starting from them. Existing nuSMV primitives perform this test for us.

7 Experimental results

7.1 The bridge-crossing problem

The problem. The bridge-crossing problem is a famous mathematical puzzle with time critical aspects [16]. A group of four persons, called P1, P2, P3 and P4, have to cross a bridge at night. It is dark and they can only cross the bridge if they carry 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, bringing back the lamp with him for the remaining people. 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. See Appendix D.1. 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. See Appendix D.1.

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. The bridge example readily gives rise to many variants with more than four persons, with different crossing times, and where a larger number of persons can cross in the same step.

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 nuSMV. The same example can be treated with nuSMV. This is easy since the example does not use zero-length steps. For long steps, we use the method advocated in [5, p.106] and introduce a counter forcing several t.u. between actual system moves. See Appendix D.2 for the nuSMV variant. It can be argued that this model is slightly more cumbersome.

The answer 60 t.u. for `COMPUTE MIN[initial, safe]` is obtained a little less quickly 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 models are different. In fact, the nuSMV encoding describes a different TKS, *where all intermediate positions exists along what should be long steps.* The AG modality quantifies over these intermediate positions too.

The intended semantics can be recovered by adapting the TCTL formula. The new formula relies on a new `inbetween` proposition that has to be added: it labels the intermediate configurations that should not exist in the TKS. See nuSMV code in Appendix D.2. The resulting formula

```
SPEC  AG (!inbetween -> EBF 0..85 (safe & !inbetween))
```

is satisfied.

We conclude that the encoding of TKS's in normal KS's does not only compromise model checking efficiency. It also compromises the semantics subtly.

7.2 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). The algorithms used in TSMV are designed in such a way that only useful durations are considered (we were influenced by the complexity analysis in [13]). As a result, TSMV is mostly insensitive to scaling up the durations. Assume that 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 [6].

This must be contrasted with nuSMV behavior. When we scale up the durations, the computation time increases dramatically. In fact, there is no way to avoid this: nuSMV does not know about TKS's and is 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.

The following table shows how running time and memory requirements grow when we scale up the timing parameters of the bridge problem of TSMV:

[6] Observe that the TKS defined with `bridge x 10` is not exactly a scaling up of `bridge` since we use conditions of the form "`duration >= 100`" that allow all values $100, 101, 102, \ldots, d_{\max}$

	nuSMV		TSMV	
	time	memory	time	memory
bridge	0.03 sec.	1 280 KB	0.02 sec.	1 280 KB
bridge × 10	0.12 sec.	8 716 KB	0.04 sec.	1 280 KB
bridge × 20	0.47 sec.	8 948 KB	0.06 sec.	1 280 KB
bridge × 50	3.24 sec.	9 816 KB	0.12 sec.	8 760 KB
bridge × 100	12.65 sec.	11 260 KB	0.22 sec.	9 180 KB
bridge × 200	50.75 sec.	14 240 KB	0.54 sec.	10 132 KB

Table 1. Scale up (in)sensitivity

7.3 Verifying the PCI local bus

A model of the PCI local bus was analyzed by Campos *et al.* in [7], computing minimum and maximum delays with symbolic model checking techniques. This model is a standard example that comes with the nuSMV distribution [7]. We refer to [7] 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. 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;
INIT  duration=0
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 [7]).

It is also possible to count specific kinds of events. For instance, we compute the number of transactions issued on the PCI bus between a request and a grant of each of the masters. This is achieved by the following lines:

```
DEFINE start_tr := processor.start_transaction |
                    vga_ctrl.start_transaction |
                     scsi_ctrl.start_transaction |
```

[7] (Essentially) two different models for the PCI local bus are available at `http://nusmv.irst.itc.it/examples/examples.html`. We use `pci4p.smv` here, which specifies the Round-Robin priority policy.

```
                    isa_bridge.start_transaction ;
   VAR  duration: 0..1;
   INIT  duration=0
   TRANS  next(duration)=case                                      (E2)
              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 ]
```

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 [7], where they were obtained with a special condition counting algorithm.

We also counted the amount of data the processor can transfer between a request and its corresponding grant, with the following lines:

```
(in module bus_master)
   VAR transmitting: boolean;
   ASSIGN init(transmitting) := FALSE;
          next(transmitting) := (count>next(count));

(in module main)
   VAR  duration: 0..1;
   INIT  duration=0
   TRANS  next(duration)=case                                      (E3)
              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 ]
```

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 he 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;
   INIT  duration=0
   TRANS  next(duration)=case                                      (E4)
              processor.transmitting : 1;
              1                      : 0;
```

```
      esac

SPEC
  AG ((processor.transmitting & abort) -> !E [ (!req4) U>=2 TRUE ] )
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 these computations:

	nuSMV		TSMV	
	time	memory	time	memory
(E_1)	213.71 sec.	21192KB	188.90 sec.	33328KB
(E_2)			69.59 sec.	24340KB
(E_3)			306.74 sec.	31780KB
(E_4)			13.56 sec.	33900KB

Table 2. Verification of the PCI local bus

8 Conclusion

We proposed OBDD-based symbolic verification procedures for model checking TCTL 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 TCTL 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. On the contrary, we split the transition relation R in several subrelations R_d's, one for each duration appearing in the model. Then the model checking procedures manages arrays of sets, where each duration corresponds to some array offset. Other optimizations are introduced for left-open or right-open constraints.

The benefits of this approach are not only greater efficiency. Another outcome is a form of insensitivity to scaling up of durations, as demonstrated in Section 7.2. The reduction of long steps to many unit steps advocated in [5, p.106] does not enjoy this property: when it analyses the resulting model, a model checker has to do a lot of costly extra work for computing intermediary sets of states.

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. Untameable timed automata! In *Proc. 20th Ann. Symp. Theoretical Aspects of Computer Science (STACS'2003), Berlin, Germany, Feb. 2003*, volume 2607 of *Lecture Notes in Computer Science*, pages 620–631. Springer, 2003.
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. *Information and Computation*, 98(2):142–170, 1992.
4. S. Campos and E. M. Clarke. Analysis and verification of real-time systems using quantitative symbolic algorithms. *Journal of Software Tools for Technology Transfer*, 2(3):260–269, 1999.
5. S. Campos and E. M. Clarke. The Verus language: representing time efficiently with BDDs. *Theoretical Computer Science*, 253(1):95–118, 2001.
6. S. Campos, E. M. Clarke, and O. Grumberg. Selective quantitative analysis and interval model checking: Verifying different facets of a system. *Formal Methods in System Design*, 17(2):163–192, 2000.
7. 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*, pages 72–78. IEEE Comp. Soc. Press, 1995.
8. 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*, pages 266–270. IEEE Comp. Soc. Press, 1994.
9. 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*, volume 23(2) of *Electronic Notes in Theor. Comp. Sci.* Elsevier Science, 1999.
10. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Journal of Software Tools for Technology Transfer*, 2(4):410–425, 2000.
11. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
12. E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. *Real-Time Systems*, 4(4):331–352, 1992.
13. 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*, volume 2303 of *Lecture Notes in Computer Science*, pages 264–279. Springer, 2002.
14. K. G. Larsen, P. Pettersson, and Wang Yi. UPPAAL in a nutshell. *Journal of Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.
15. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.
16. G. Rote. Crossing the bridge at night. *EATCS Bull.*, 78:241–246, 2002.
17. S. Yovine. Kronos: A verification tool for real-time systems. *Journal of Software Tools for Technology Transfer*, 1(1–2):123–133, 1997.

A Proof of Lemma 3.1

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 $d' < d$.

(a) $\Rightarrow$ (b) : We pick a finite path π of the form $s(= s_0) \xrightarrow{d_1} s_1 \cdots \xrightarrow{d_l} s_l$ with $s_l \models \varphi$ and minimum $d_1 + \cdots + d_l$. From *(a)* we know that $d_1 + \cdots + d_l = n$ Let d_i be the first non-zero duration along this path. Write d for d_i, s' for s_{i-1}, and s'' for s_i, so that we already have $s' \xrightarrow{d} s''$ (b.1). Clearly $s_i \models \mathsf{EF}_{\leq n-d}\varphi$, as witnessed by π^i. But s_i cannot satisfy $\mathsf{EF}_{<n-d}\varphi$ otherwise π would not have minimum duration. Hence (b.2) holds. Finally, none of the states $s_0,\ldots,s_{i-1}$ satisfies $\mathsf{EF}_{<n}\varphi$, otherwise π would not have minimum duration. Hence π further witnesses (b.3).

(b) $\Rightarrow$ (a) : Combining (b.1), (b.2) and (b.3) we obtain a path $s \xrightarrow{0} \ldots \xrightarrow{0} s' \xrightarrow{d} s''$ with $s'' \models \mathsf{EF}_{\leq n-d}\varphi$. Hence $s \models \mathsf{EF}_{\leq n}\varphi$. Furthermore, from (b.3) we deduce that $s \models \neg\mathsf{EF}_{<n}\varphi$.

Finally, assuming *(b)* and $s' \xrightarrow{d'} s''$ means that $s' \models \mathsf{EF}_{\leq n-d'}\varphi$. Since $s \not\models \mathsf{EF}_{<n-d}\varphi$ by (b.3), we deduce $d' \geq d$. $\square$

B Proof of Lemma 5.1

Lemma 5.1. *After step i, the following invariants hold for all $j = 1,\ldots,d_{\max}+1$:*

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

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

We simultaneously prove (Inv4) and (Inv5) by induction on i.

When $i = 0$, for each $1 \leq j \leq \min(w, d_{\max})$, a state s is in $\mathtt{Incr}[j]$ iff there exists two states s' and s'', and an integer $d \geq j$, s.t. $s' \xrightarrow{d} s''$ and s satisfies $E[S_1\ U_{=0}(S_2 \wedge \mathsf{EF}_{=0}s')]$. This entails the existence of a path π of the form

$$\underbrace{s \xrightarrow{0} \cdots}_{\in S_1} \xrightarrow{0} \underset{S_2}{\underset{\Cap}{t}} \xrightarrow{0} \cdots \xrightarrow{0} s' \xrightarrow{d} s'' \cdots \tag{6}$$

That path fulfills (a). Since it fulfills $S_1U_{=0}S_2$, thus $S_1U_{[j-w..j-1]}S_2$, (b) also holds. Conversely, consider a state s and a path π starting in s and satisfying contidions (a) and (b). Then $\pi \models S_1 \mathsf{U}_{[j-w,j-1]} S_2$, and the first transition with non-zero duration along π is $s' \xrightarrow{d} s''$, with duration $d \geq j$. Thus necessarily, $\pi \models S_1 \mathsf{U}_{=0} S_2$, and π has the form of the path (6). This entails that $s \in \texttt{Incr[j]}$.

For possible values of j between $w+1$ and $d_{\max}$, $\texttt{Incr[}j\texttt{]}$ is empty on the one hand, and no path can satisfy both (a) and (b) on the other hand. Indeed, condition (b) requires that there is a prefix of the path whose total duration lies between $j-w$ and $j-1$. This is impossible because the first non-zero duration is greater than j.

Last, when $j = d_{\max}+1$, $\texttt{Incr[}d_{\max}+1\texttt{]}$ is empty on the one hand, and there can be no transition with duration $d \geq d_{\max}+1$ on the other hand.

We now prove (Inv5) for $i=0$. If $w \leq 0$, $\texttt{Res}$ is empty (since $\texttt{Incr[1]}$ is), and the invariant holds since no path can satisfy $S_1 \mathsf{U}_{[1-w..0]} S_2$. If $w > 0$, the equivalence is obvious.

We tackle the inductive step now. The case where $j = d_{\max}+1$ is obvious, for the same reason as above. We and assume both invariant hold at step $i-1$. We pick a state $s \in \texttt{Incr[j]}$ at step i, for some $1 \leq j \leq d_{\max}$, if $\texttt{Incr[j]}$ is not empty. By construction of $\texttt{Incr[j]}$, either $s \in \texttt{Incr[j+1]}$ *at step* $i-1$, or $s \in \mathsf{EU}_{=0}(S_1, S_2 \cap \texttt{Pre}[R_j](\texttt{Res}))$. In the first case, by i.h., there exists a path $\pi = s \xrightarrow{0} \cdots \xrightarrow{0} s' \xrightarrow{d} s'' \cdots$ s.t. $d \geq j+1$ and $\pi \models S_1 \mathsf{U}_{(i-1)+(j+1)-w..(i-1)+(j+1)-1} S_2$. The first condition implies (a), the second one is exactly (b). Now consider the second case above, where $s \in \mathsf{EU}_{=0}(S_1, S_2 \cap \texttt{Pre}[R_j](\texttt{Res}))$. This means that there exists two state s' and s'', and a path π s.t.

$$\underbrace{s \xrightarrow{0} \cdots \xrightarrow{0} s'}_{\in S_1} \xrightarrow{j} \underset{\underset{\texttt{Res}}{\Cap}}{s''} \cdots$$

Since $s'' \in \texttt{Res}$ *at step* $i-1$, by i.h., there exists a path π' starting at state s'' and satisfying $S_1 \mathsf{U}_{[i-1+1-w..i-1]} S_2$. By branching π' after state s'' in π, we get a path whose first non-zero duration equals j (hence satisfying (a)) and verifying $S_1 \mathsf{U}_{[i+j-w..i+j-1]} S_2$ (hence satisfying (b)).

Conversely, pick a state s, if any, s.t. there exists a path $\pi = s \xrightarrow{0} \cdots \xrightarrow{0} s' \xrightarrow{d} s'' \cdots$ satisfying both conditions (a) and (b) at step i for some j. Observe that condition (b) at step i for some j is equivalent to condition (b) at step $i-1$ for $j+1$. Thus if $d \geq j+1$, $s \in \texttt{Incr[j+1]}$ at step $i-1$, and, by construction of $\texttt{Incr[}j\texttt{]}$, $s \in \texttt{Incr[j]}$ at step i. Otherwise $d = j$, and the suffix of π starting at s'' satisfies $S_1 \mathsf{U}_{[i-w..i-1]} S_2$, since the prefix we removed has total duration j. By i.h., $s'' \in \texttt{Res}$ at step $i-1$, then $s \models \mathsf{E}(S_1 \mathsf{U}_{=0}(S_1 \wedge s''))$, and $s \in \texttt{Incr[j]}$.

We end the proof by showing the second invariant. Pick a state $s \in \texttt{Res}$. If $s \in \texttt{Incr[1]}$, then it satisfies $\mathsf{E} S_1 \mathsf{U}_{[i+1-w..i]} S_2$. If $s \in \mathsf{EU}_{=0}(S_1, S_2)$, then $i < w$, that is $i+1-w \leq 0$, and $s \models \mathsf{E} S_1 \mathsf{U}_{[0..i]} S_2$. Conversely, consider a state $s \models \mathsf{E} S_1 \mathsf{U}_{[i+1-w..i]} S_2$, and a path π witnessing that formula. If π contains one transition with non-zero duration, then it is a witness for $s \in \texttt{Incr[1]} \subseteq \texttt{Res}$. Otherwise, π contains only zero-duration tansitions. Since it satisfies $S_1 \mathsf{U}_{[i+1-w..i]} S_2$, necessarily $i+1-w \leq 0$, and $\pi \models S_1 \mathsf{U}_{=0} S_2$, witnessing $s \in \texttt{Res}$. $\square$

C The $\mathsf{EG}_{[k..l]}$ modality.

C.1 The algorithm

We use Algorithm 4 for the $\mathsf{EG}_{[k..l]}$ modality. That algorithm is similar to the one

```
function EGint(S, k, l)
w ← l − k + 1; i ← 0;
if (w≤0 || l≤0) { return(TRUE) }
for (j←1; j≤min(w,d_max); j++) {
  Incr[j] ← EU=0( S, S ∩ ⋃{Pre[R_d](⊤) | d ≥j} )
}
for (j← w+1; j≤d_max; j++) {
   Incr[j] ← EF=0( ⋃{Pre[R_d](⊤) | d ≥j} )
}
Incr[d_max + 1] ← ∅ ;
aux0, Res ← EG=0(S) ;

/* Loop (step 1 to k)*/
for (i←1; i≤ l; i++) {
  for (j←1; j≤d_max; j++) {
    if (i+j-w≤0) {
      Incr[j] ← Incr[j+1] ∪ EU=0( S, S ∩ Pre[R_j](Res) ) ;
    } else {
      Incr[j] ← Incr[j+1] ∪ EF=0( Pre[R_j](Res) ) ;
    }
  }
  Res ←  Incr[1];
  if (i<w) { Res ← Res ∪ aux0 }
}
Res ← Res ∪ EF≤k−1( EG( duration=0 ) )
return(Res);
```

Algorithm 4: Algorithm for $\mathsf{EG}_{[k..l]}$

we designed for $\mathsf{EU}_{[k..l]}$. The correctness proof relies on the following invariants (still assuming that w and l are nonnegative):

Lemma C.1. *After step i, the following invariants hold for all $j = 1, \ldots, d_{\max} + 1$:*

$$s \in \mathtt{Incr[}j\mathtt{]} \text{ iff } \left[\begin{array}{l} \text{there is a path } \pi = s \xrightarrow{0} \cdots \xrightarrow{0} s' \xrightarrow{d} s'' \cdots \text{ s.t.} \\ (a)\ \mathtt{D}\pi \geq i + j - w, \text{ and} \\ (b)\ d \geq j, \text{ and} \\ (c)\ \pi \models \mathsf{G}_{[i+j-w..i+j-1]} S. \end{array}\right] \tag{Inv6}$$

$$s \in \mathtt{Res} \text{ iff } \left[\begin{array}{l} \text{there is a path } \pi \text{ starting in } s \text{ s.t.} \\ (d)\ \mathtt{D}\pi \geq i + 1 - w, \text{ and} \\ (e)\ \pi \models \mathsf{G}_{[i+1-w..i]} S. \end{array}\right] \tag{Inv7}$$

The proof for this Lemma is given below. At step l, a state s is in `Res` iff there is a path π starting in s, whose total length is greater than k, and satisfying $\mathsf{G}_{[k..l]}S$. To complete the computation, we add states from where there exists a path whose total duration is always lower than $k-1$ (see last line of the algorithm).

C.2 Proof of the algorithm for $\mathsf{EG}_{[k..l]}$

We prove both invariant simultaneously, by induction on i.

When $i=0$, for each $1 \le j \le \min(w, d_{\max})$, a state s is in `Incr[`j`]` iff there exists two states s' and s'', a duration $d \ge j$, and a path π of the form

$$\underbrace{s \xrightarrow{0} \cdots \xrightarrow{0} s'}_{\in S} \xrightarrow{d} s'' \cdots$$

Condition (a), (b) and (c) hold in that case. Conversely, if a state π, starting at state s, satisfies conditions (a) to (c), necessarily is of the form (C.2).

When $w < j \le d_{\max}$, $s \in$ `Incr[j]` is equivalent to (b). Conditions (a) follows from (b). Condition (c) holds vacuously since the timing constraint never holds. The case where $j = d_{\max}+1$ is obvious.

To finish with this base case, `Res` initialy contains all states satisfying $\mathsf{EG}_{=0}(S)$, which entails conditions (e). Condition (d) holds vacuously. The converse is obvious too.

Now suppose that both invariants hold at some step $i-1$. Pick some state s (if any) in `Incr[`j`]` at step i. If $s \in$ `Incr[j+1]` *at step* $i-1$, then conditions (a) to (c) hold. Otherwise, two cases may arise:

- If $i+j-w \le 0$, then $s \in \mathsf{EU}_{=0}(S, S \cap \mathtt{Pre}[R_j](\mathtt{Res}))$. This means that there exists two states s' and s'', and a path π, s.t.

$$\pi = \underbrace{s \xrightarrow{0} \cdots \xrightarrow{0} s'}_{\in S} \xrightarrow{j} \underset{\mathtt{Res}}{\underset{\Cap}{s''}} \cdots$$

  By i.h., since $s'' \in$ `Res`, there exists a path π' starting from s'' satisfying conditions (d) and (e) at step $i-1$. Moreover, since $i+j-w \le 0$, condition (e) implies that $\pi' \mathsf{G}_{\le i-1} S$. Merging π' at position s'' in π provides us with a path verifying conditions (a) to (c) at step i;
- If $i+j-w > 0$, then $s \in \mathsf{EF}_{=0}(\mathtt{Pre}[R_j](\mathtt{Res}))$. *i.e.* there exists two states s' ans s'', and a path π, s.t.

$$\pi = s \xrightarrow{0} \cdots \xrightarrow{0} s' \xrightarrow{j} \underset{\mathtt{Res}}{\underset{\Cap}{s''}} \cdots$$

  By i.h., there exists a path π' starting from s'' and verifying conditions (d) and (e) at step $i-1$. As before, merging π' into π at state s'' gives rise to a path π'' verifying conditions (a) and (b) at step i. Condition (c) also holds:

it is obvious if $(i-1)+1-w > 0$. If $(i-1)+1-w \leq 0$, then the path π'' has the following form:

$$\pi'' = s \xrightarrow{0} \cdots \xrightarrow{0} s' \xrightarrow{j} \underbrace{s'' \to \cdots \to s'''}_{S} \to \cdots$$

It satisfies $\mathsf{G}_{[j..i-1+j]}S$, but also all $\mathsf{G}_{[k..i-1+j]}S$ for all k between 1 and j (included). Since $i+j-w > 0$, $\pi'' \models \mathsf{G}_{[i+j-w..i+j-1]}S$.

Conversely, pick a state (if any) satisfying conditions (a) to (c) at step i for some j. If $d \geq j+1$, then by i.h., $s \in$ `Incr[j+1]` at step $i-1$. Otherwise, $d = j$.

– If $i+j-w \leq 0$, then we have the following situation:

$$\pi = \underbrace{s \xrightarrow{0} \cdots \xrightarrow{0} s' \xrightarrow{j} s'' \to \cdots \to s'''}_{S} \to \cdots$$

The prefix of π starting at s'' then satisfies conditions (d) and (e) at step $i-1$, and $s \in \mathsf{EU}_{=0}(S, S \cap \mathtt{Pre}[R_j](\mathtt{Res}))$.

– If $i+j-w > 0$, we distinguish between two cases: If $i-w > 0$, we have the following situation:

$$\pi = s \xrightarrow{0} \cdots \xrightarrow{0} s' \xrightarrow{j} s'' \to \cdots \to \underbrace{s''' \cdots \to s^{(4)}}_{S} \to \cdots$$

Then $s'' \in \mathtt{Res}$ at step $i-1$. If $i-w \leq 0$, we have the following situation:

$$\pi = s \xrightarrow{0} \cdots \xrightarrow{0} s' \xrightarrow{j} \underbrace{s'' \to \cdots \to s'''}_{S} \to \cdots$$

since $i+j-w > 0$. Here again, the suffix of π starting in s' satisfies (d) and (e).

We now have to prove (Inv7). Pick $s \in \mathtt{Res}$ if any, at step i. If $s \in$ `Incr[1]`, then condition (d) and (e) hold. Otherwise, we are in the case where $i < w$, and $s \in \mathsf{EG}_{=0}(S)$. Since $i-w+1 \leq 0$, both conditions (d) and (e) hold. Conversely, we pick a state s for which there exists a path π starting in s and satisfying (d) and (e). If π has at least one positive duration, then π satisfies all conditions (a) to (c) for $j=1$ at step i, and $s \in$ `Incr[1]`. Otherwise, π has only zero-duration transitions. Condition (d) implies that $i < w$, and since $\pi \models \mathsf{G}_{[0..i]}S$, it satisfies $\mathsf{G}_{=0}S$, and $s \in \mathtt{aux0} \subseteq \mathtt{Res}$. □

D The bridge Crossing Problem

D.1 The Bridge Crossing Problem in TSMV

```
-- -------------------------------------------------------
--            Bridge crossing at night                 --
-- -------------------------------------------------------
```

```
MODULE person(time,lamp,duration)
-- knows its time for crossing, the position of the lamp,
-- and may impose the value for duration

VAR
  state: boolean;               -- true = left of the bridge, false = right
  changed: boolean;             -- true if the person just crossed the bridge

INIT
  !state & !changed

TRANS
  -- we have the lamp. We can cross...
  ((!state & !lamp) -> ((next(state) & next(duration)>=time & next(changed)) |
  -- ... or not
                         (next(!state) & next(!changed))))
&
  ((state & lamp) -> ((next(!state) & next(duration)>=time & next(changed)) |
                          (next(state) & next(!changed))))
&
  -- we don't have the lamp... Just seat and wait here!
  ((!state & lamp) -> (next(!state) & next(!changed)))
&
  ((state & !lamp) -> (next(state) & next(!changed)))

MODULE main
-- We define the persons here,
-- and control the moves of the lamp.

VAR
  v1: person(5,lamp,duration);
  v2: person(10,lamp,duration);
  v3: person(20,lamp,duration);
  v4: person(25,lamp,duration);
  lamp: boolean;                -- true = left of the bridge, false = right
  duration: 0..25;              -- the duration variable

DEFINE
  bridge:=v1.changed+v2.changed+v3.changed+v4.changed;
  safe:=(v1.state)&(v2.state)&(v3.state)&(v4.state);
  initial:=(!v1.state)&(!v2.state)&(!v3.state)&(!v4.state);

INIT
  !lamp & duration=0

INVAR
  bridge<=2;

TRANS
 ( (v1.state=next(v1.state) &
    v2.state=next(v2.state) &
    v3.state=next(v3.state) &
    v4.state=next(v4.state)  ) <-> (next(lamp)=lamp) )
 &
  next(duration mod 5 = 0)

COMPUTE MIN[initial, safe]

SPEC  EBF 60..60 safe
SPEC  EF<= 59 safe
SPEC  (AG EF<=85 safe) & !(AG EF<=84 safe)
```

D.2 The Bridge Crossing Problem in NuSMV

```
-- ------------------------------------------------------------
--        Bridge crossing at night ("untimed" version)       --
-- ------------------------------------------------------------

MODULE person(time, lamp, go)
-- knows its time to cross, where is the lamp,
-- and the value of the start signal

VAR
  position: 0..3;       -- 0 = left, 1 = left hand side of bridge
                        -- 2 = right hand side of bridge, 3 = right
```

```
  wait: 0..25;          -- a "timer"
  changed: boolean;     -- if the person just crossed the bridge

INIT
  position=0 & !changed & wait=0;

DEFINE
  arrived:= (wait>=time);

TRANS
  ( -- the way is free...
   (go) -> ( -- we have the lamp. We can go or wait here...
             ( (position=0 & !lamp) -> (
                        (next(position)=1 & next(wait)=1) |
                        (next(position)=0 & next(wait)=0) ) ) &
             ( (position=3 & lamp) -> (
                        (next(position)=2 & next(wait)=1) |
                        (next(position)=3 & next(wait)=0) ) ) &
             -- we don't have the lamp... and have to wait
             ( (position=0 & lamp) -> (
                        (next(position)=0 & next(wait)=0) ) ) &
             ( (position=3 & !lamp) -> (
                        (next(position)=3 & next(wait)=0) ) ) ) )
&
  ( -- We can't go, since someone is already on the bridge with the lamp
   (!go) -> ( -- if we are not on the bridge, we have to stay here
              ( (wait=0) -> (next(wait) = 0 & next(position)=position) ) &
              -- we are crossing... Let's continue...
              -- Time increase is 5 = gcd(5, 10, 20, 25)
              ( ((wait!=0) & (wait!=time - 5)) ->
                     (next(wait) = wait+5 & next(position)=position) ) &
               -- We are arriving on the other side...
              ( (wait=time - 5) ->
                     (next(wait)=time & next(position) = (6-3*position)) ) ) )
&
  next(changed) = (position!=next(position))


MODULE main

VAR
  v1: person(5, lamp, go);
  v2: person(10, lamp, go);
  v3: person(20, lamp, go);
  v4: person(25, lamp, go);
  lamp: boolean;                -- false: left, true: right

DEFINE
  -- we can start crossing if noone is on the bridge
  go:= (v1.position=0|v1.position=3) &
       (v2.position=0|v2.position=3) &
       (v3.position=0|v3.position=3) &
       (v4.position=0|v4.position=3);
  bridge:= (v1.wait!=0) + (v2.wait!=0) + (v3.wait!=0) + (v4.wait!=0);
  initial:= v1.position=0 & v2.position=0 & v3.position=0 & v4.position=0;
  safe:= v1.position=3 & v2.position=3 & v3.position=3 & v4.position=3;
  inbetween:= (bridge!=0);

INVAR
  (bridge<=2);

INIT
  !lamp;

TRANS
  (go & next(bridge)>0) <-> (next(lamp)=!lamp)

COMPUTE MIN[initial, safe];

SPEC  EBF 60..60 safe)
SPEC  EBF 0..59 safe
SPEC  (AG EBF 0..85 safe) & !(AG EBF 0..84 safe)
SPEC  AG ( (!inbetween) -> EBF 0..85 (safe & (!inbetween))
```