SAT Based Bounded Model Checking with Partial Order Semantics for Timed Automata*

Janusz Malinowski and Peter Niebert

Laboratoire d'Informatique Fondamentale de Marseille
Université de Provence, 39 rue Joliot-Curie, Marseille, France
{peter.niebert,janusz.malinowski}@lif.univ-mrs.fr

Abstract. We study the model checking problem of timed automata based on SAT solving. Our work investigates alternative possibilities for coding the SAT reductions that are based on parallel executions of independent transitions.

While such an optimization has been studied for discrete systems, its transposition to timed automata poses the question of what it means for timed transitions to be executed "in parallel". The most obvious interpretation is that the transitions in parallel take place at the same time (synchronously). However, it is possible to relax this condition. On the whole, we define and analyse three different semantics of timed sequences with parallel transitions.

We prove the correctness of the proposed semantics and report experimental results with a prototype implementation.

1 Introduction

In this paper, we describe a SAT based model checking algorithm for timed concurrent systems that includes partial order concepts, as well as its implementation in the POEM model checker.

While symbolic state exploration with zones [10] as implemented in Uppaal [2] remains the most widely used algorithm for model checking timed automata, reductions to SAT solvers [15,13,1,17] have been studied with encouraging results. However, the situation is far from the dominance of SAT methods used to analyse synchronous circuits.

On the other hand, the zone based state exploration has seen several works investigating improvements based on partial order semantics [12,16,7,14]. In the development of timed automata, this investigation came late, maybe because the algorithms used defy the intuition of time as a total order. For instance, in [12] it is possible that the algorithm provides sequences where the time may go backwards between transitions (but these executions can nevertheless be reordered to represent real executions).

The basis of most so-called partial order approaches for model checking asynchronous systems is the structural observation that pairs of *independent transitions*, i.e. transitions that concern separate parts of a distributed system, may

* Partially supported by the ANR project ECSPER(ANR JC09_472677 ECSPER).

J. Esparza and R. Majumdar (Eds.): TACAS 2010, LNCS 6015, pp. 405–419, 2010.

be executed in any order with the same result. For timed automata, this was at first not obvious, since these transitions may reset clocks and the order of firing introduces a relation on the clock values. The cited works using partial order concepts for timed automata avoid in one way or another the introduction of this artificial relation and can therefore outperform the classical algorithms in many cases.

An obvious question occurs when combining two different methods (here SAT reductions and partial order semantics): will the performance improve on each method separately? For untimed asynchronous systems like safe Petri nets, a positive answer to this question was given in [9,11] and a few sequels. However, the answer given in those works was to improve the SAT reduction by allowing several independent transitions to actually occur in parallel, i.e. in one step. This concept was known before in Petri nets as *step semantics* [8], but it found an unexpected application. Intuitively, multisteps (transitions that are executed in parallel) allow to compress execution sequences leading to a state: while the overall number of executed transitions remains the same, the possibility of executing several of them in one parallel step means that there are less intermediate states to consider. Moreover, when coding reachability in SAT, differently from state exploration, the sub-formulae coding the possible execution of a transition are present for every step in the sequence anyway. From this perspective, requiring interleaving semantics can be perceived as nothing more than a restriction stating that in any multistep at most one transition takes place. In tight cases, the best SAT solver will have to try out every interleaving, i.e. every permutation of independent transitions. We cannot imagine a case where this interleaving requirement will have any benefit for the SAT approach, but relaxing it and allowing multisteps will very often give dramatic improvements.

The contribution of this work is to extend the reduction with *multisteps* to timed automata.

This being said, we invite the reader to consider what it means for several timed transitions to be executed "in parallel" or in the same multistep before reading on.

Indeed, the first idea that may come to ones mind is that these transitions should take place "at the same time", but this turns out to be just one of several options, which we call "synchronous". A more relaxed notion may require that each transition in a multistep has to be executed temporally before each transition of the following multistep, yet allowing the individual transitions to take place at different times, a notion we call "semi synchronous time progress". Based on notions from [12,14], the seemingly least restrictive sensible notion limits the time progress to transitions that are dependent, which we call "relaxed time progress". Based on previous work, we show that the three proposed notions of time progress are equivalent in the sense that the execution sequences of either semantics can be transformed into execution sequences of the other and into the classical notion of runs. However, they turn out not to be equivalent with respect

to performance: the more relaxed notions are more complex to code in SAT but can in some cases yield superior results.

Plan. The paper is structured as follows: in Section 2, we introduce the basic notions of timed systems on a certain specification level: multithreaded programs with shared variables. It is essential to use such a model to understand the SAT coding. We also introduce notions from timed automata, notably "clocks", clock conditions and resets. For the sake of readability, we do not introduce state invariants at this point. In Section 3, we recall notions of independence in the context of timed automata and introduce semantics with multisteps. The main formal tools are developed here, different notions of time progress are formally defined and their equivalence is shown. In Section 4, we show how these concepts integrate into a SAT reduction for systems of the kind described in Section 2. This description, although held informal where possible, aims to give a self-contained description of how such a reduction is constructed and how the notions of Section 3 integrate in the construction of a SAT problem instance. In Section 5, we informally discuss how state invariants, an important modelling concept in timed automata can be integrated with each of the three notions of time progress. In Section 6, we illustrate the potential of the algorithms by a few benchmarks in our prototype implementation. We conclude and discuss related work in Section 7.

2 Preliminaries

Let T_i with $1 \leq i \leq N$ denote a thread with $trans_i$ its set of transitions. Let $trans = \bigcup trans_i$ the set of all transitions. Let V_i be the set of local variables of T_i and let V_g the set of global variables. Then we introduce $V = V_g \cup \bigcup V_i$ the set of all variables. Each variable $v \in V$ takes its values in the domain D_v. Control locations of a thread T_i are represented by a local variable $pc_i \in V_i$ (program counter).

A state of a program is a valuation of local and global variables, formally $s : V \longrightarrow \bigcup D_v$ with $s(v) \in D_v$. The set of all states is denoted by $S = \prod D_v$. We moreover assume an *initial state* s_0, i.e. an initial valuation of variables.

Expressions over the variables are defined as usual (e.g. arithmetic expressions). Atoms are comparisons over pairs of expressions and conditions are boolean combinations of atoms.

Syntactically, a transition t of T_i is enabled by a *condition* (boolean combination of atoms) ranging over $V_i \cup V_g$, and it has as effect an *action* defined as a set of assignments (of expressions to variables), i.e. values of variables are *written*. For both actions and conditions, the variables appearing in the expressions are *read*. If t is a transition from the control location loc_1 to control location loc_2 then the condition of t includes $pc_i = loc_1$ and the action includes $pc_i := loc_2$.

For two states $s, s' \in S$, $s \xrightarrow{t} s'$ denotes a state transition enabled at s and transforming s to s' when applying the action of t. Let $s \xrightarrow{t_1} s_1 \ldots \xrightarrow{t_n} s_n = s'$ denote a sequence of transition executions.

2.1 Adding Time

We introduce real valued variables called *clocks* which differ from other variables:

- Their values increase synchronously and proportionally with time: if x has value ρ at time τ then it has value $\rho + \delta$ at time $\tau + \delta$.
- The only assignments allowed are resets (to zero).
- We only allow comparisons of clocks with integer constants (e.g. $x \leq 3$).
- At the initial state s_0, all clocks are valued 0.

Now, each transition execution t_k has an execution time $\tau_k \in \mathbb{R}^+$ also called a *timestamp*. Then, we denote a timed transition t_k executed at time τ_k as $s \xrightarrow{(t_k,\tau_k)} s'$ and *timed sequence* as $s_0 \xrightarrow{(t_1,\tau_1)} s_1 \ldots \xrightarrow{(t_n,\tau_n)} s_n = s'$.

A timed sequence $s_0 \xrightarrow{(t_1,\tau_1)} s_1 \ldots \xrightarrow{(t_n,\tau_n)} s_n = s'$ satisfies *normal time progress* iff for every pair $k < l$, we have $\tau_k \leq \tau_l$.

The *reachability problem for timed automata* can be understood in this setting as the existence of a timed sequence with normal time progress leading from s_0 to a state s' satisfying a desired property.

3 Concurrency

In this section, we will review standard notions from classical partial order methods. Then we will introduce the notion of *multisteps*, i.e. the execution of several transitions *in parallel* and we will see how to analyse timed systems using multisteps.

3.1 Independence Relation

A classical definition underlying concurrency analysis is reader-writer dependency as first introduced in [6]: two transitions t_1 and t_2 are said to be *dependent* if a variable read in t_1 (in the guard or in the action) is written in t_2 (or vice versa), or if the same variable is written by t_1 and t_2. Otherwise, they are *independent*.

A timed sequence $s_0 \xrightarrow{(t_1,\tau_1)} s_1 \ldots \xrightarrow{(t_n,\tau_n)} s_n = s'$ satisfies *relaxed time progress* if for every pair $k < l$ with t_k, t_l dependent, we also have $\tau_k \leq \tau_l$. Note, that normal time progress as defined in the previous section trivially implies relaxed time progress.

We define the Mazurkiewicz equivalence of timed sequences as the least equivalence relation $\equiv$ such that any timed sequence $s_0 \ldots s_{k-1} \xrightarrow{(t_k,\tau_k)} s_k \xrightarrow{(t_{k+1},\tau_{k+1})} s_{k+1} \ldots s'$ with t_k, t_{k+1} independent is equivalent to $s_0 \ldots s_{k-1} \xrightarrow{(t_{k+1},\tau_{k+1})} {s_k}' \xrightarrow{(t_k,\tau_k)} s_{k+1} \ldots s'$ for some state ${s_k}'$. In other words, two timed sequences are equivalent if one can be transformed into the other by a finite number of exchanges of adjacent independent transitions together with their execution times.

Proposition 1. *If a timed sequence satisfies relaxed time progress, then so do its equivalent sequences. Each timed sequence satisfying relaxed time progress is equivalent to a timed sequence satisfying normal time progress*

Proof. This was originally shown in [12]. Indeed, the order of dependent transition executions is preserved by exchanges and hence so is relaxed time progress. For the second part, it is possible to transform a timed sequence with relaxed time progress by applying a "bubble sort" transformation: suppose that two adjacent transitions are in bad order with respect to their timestamps, then relaxed time progress implies that they are independent and it is possible to exchange them. The result follows by applying this reasoning in an induction. □

3.2 Concurrently Enabled Transitions

The notions proposed in the following have first come up in the context of (untimed) Petri nets under the name *step semantics* [8], generalized here for our purposes to timed automata. Let $MT = \{(t_1, \tau_1), \ldots, (t_n, \tau_n)\}$ a set of pairwise independent transitions with timestamps: it is *concurrently enabled* at global state s iff each transition is enabled at state s and at time τ_i. Note, that implicitly if $(t, \tau_a), (t, \tau_b) \in MT$ then $\tau_a = \tau_b$ because a transition is always dependent with itself. By definition of independence, all possible executions using all these transitions and beginning at s are equivalent, and lead to the same state s'. Then we say that they can be executed in parallel and we write $s \xrightarrow{MT} s'$.

A *multistep timed sequence* is a sequence $s_0 \xrightarrow{MT_1} s_1 \ldots \xrightarrow{MT_n} s_n = s'$. The interest here of multisteps is immediate: because several transitions are executed at each multistep, the execution can be shorter (i.e. a lower number of multisteps may be executed) to reach a certain state than with interleaving semantics.

3.3 Time Progress in Multistep Sequences

A *multistep timed sequence* $s_0 \xrightarrow{MT_1} s_1 \ldots \xrightarrow{MT_n} s_n = s'$ satisfies *relaxed time progress* if for every pair $k < l$ with $(t_1, \tau_1) \in MT_k, (t_2, \tau_2) \in MT_l$, t_1, t_2 dependent, we also have $\tau_1 \leq \tau_2$.

Lemma 1. *Let* $s_0 \xrightarrow{MT_1} s_1 \ldots \xrightarrow{MT_n} s_n = s'$ *be a multistep timed sequence that satisfies relaxed time progress, then there exists a timed sequence* $s_0 \xrightarrow{(t_1,\tau_1)} s_1 \ldots \xrightarrow{(t_m,\tau_m)} s_m = s'$ *with* $m = \sum |MT_i|$ *(the total number of single transition) that satisfies relaxed time progress.*

Proof. We build a timed sequence $s \xrightarrow{(t_1,\tau_1)} \ldots \xrightarrow{(t_m,\tau_m)} s_m = s'$ by extracting each (t_i, τ_i) from each MT_k respecting the order of the MT_k (the order of the (t_i, τ_i) extracted from the same MT_k is not important because they are independent). □

We now introduce alternative representations of time progress of a multistep timed sequence $s_0 \xrightarrow{MT_1} s_1 \ldots \xrightarrow{MT_n} s_n = s'$:

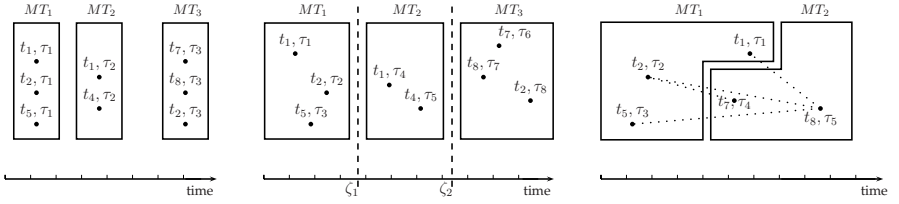


Fig. 1. Synchronous, semi synchronous and relaxed time progress in multisteps

- *synchronous time progress*: for all multisteps MT_k and for all pairs of transitions $(t_1, \tau_1), (t_2, \tau_2) \in MT_k$ it holds that $\tau_1 = \tau_2$ and for $k < l$ and any $(t_a, \tau_a) \in T_k$, $(t_b, \tau_b) \in MT_l$ it holds that $\tau_a \leq \tau_b$, i.e. all transitions in the same multistep are executed at the same time, and time progresses between multisteps.
- *semi synchronous time progress*: for all $k < l$ and any $(t_a, \tau_a) \in MT_k$, $(t_b, \tau_b) \in MT_l$ it holds that $\tau_a \leq \tau_b$, i.e. all transitions of a later multistep are executed at a later time than the transitions of an earlier multistep (but transitions of a same multistep may be executed at different times).

It is obvious that synchronous time progress implies semi synchronous time progress which in term implies relaxed time progress.

Theorem 1. *Let $s, s' \in S$, then the following elements can be transformed into each other:*

1) *A timed sequence $s \xrightarrow{(t_1,\tau_1)} \ldots \xrightarrow{(t_n,\tau_n)} s'$ with normal time progress.*
2) *A timed sequence $s \xrightarrow{(t_1,\tau_1)} \ldots \xrightarrow{(t_n,\tau_n)} s'$ with relaxed time progress.*
3) *A multistep timed sequence $s \xrightarrow{MT_1} \ldots \xrightarrow{MT_k} s'$ with synchronous time progress.*
4) *A multistep timed sequence $s \xrightarrow{MT_1} \ldots \xrightarrow{MT_l} s'$ with semi synchronous time progress.*
5) *A multistep timed sequence $s \xrightarrow{MT_1} \ldots \xrightarrow{MT_m} s'$ with relaxed time progress.*

Proof.
- $1 \Rightarrow 2$: by definition
- $2 \Rightarrow 1$: see Proposition 1
- $2 \Rightarrow 3$: we build a multistep timed sequence $s \xrightarrow{MT_1} \ldots \xrightarrow{MT_n} s_n = s'$ where each MT_k is the singleton $\{(t_k, \tau_k)\}$; it is trivially a multistep timed sequence with synchronous time progress
- $3 \Rightarrow 4$: by definition
- $4 \Rightarrow 5$: by definition
- $5 \Rightarrow 2$: see Lemma 1

□

4 SAT Reduction

4.1 Context

We have implemented the SAT coding outlined below in the tool POEM (Partial Order Environment of Marseille).

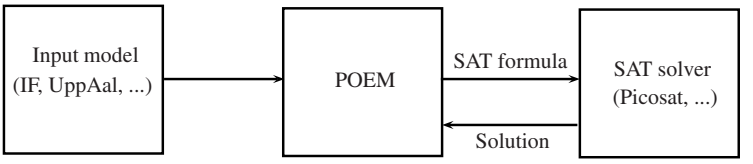


Fig. 2. A diagram of POEM with a SAT BACKEND

POEM is a modular model checker written in OCAML. Its main executable is composed of three parts, a frontend (syntactic analysis), a core with static analysis and model transformation, and an analysis oriented backend. The frontend part reads the model written in a specification language (currently Uppaal [2], IF2 [5]) and transforms it into a common format (GDS) on which type verification, transformations and other aspects of static analysis (notably for dependency analysis) are applied.

The backends currently use as input the declaration of variables and processes and a list of transitions much like the one described in Section 2. In particular, for each transition the sets of written and read variables can be determined statically (an over approximation) or dynamically (context dependent) where the latter is close in practice to notions of dynamic dependency relations.

Previously, there was only a state exploration based backend with the underlying algorithms described in [12,14]. In this section, we will describe the way we coded the SAT backend, which is used to perform a Bounded Model Checking (BMC): given a multi-threaded program and a reachability property, we construct a SAT formula Φ that is satisfiable iff a state with the property can be reached by an execution of the program with up to K multisteps. This construction involves several aspects described below.

4.2 Coding Variables

Each *variable* $v \in V$ of the input model is transformed into a vector of boolean variables of size $\log_2 |D_v|$. As an example, let's examine the following declaration in an IF2 input model:

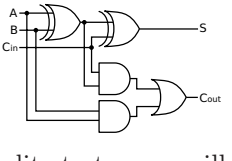
```
var x range 0..3;
```

This command declares a variable x taking its values over the domain [0..3] or a total of 4 possible values. Then we need a boolean vector of size 2: $\{x_1, x_2\}$.

The *program counter* pc_i for each thread T_i is coded as a normal variable, i.e. as a boolean vector, its length depending on the number of states in the thread.

subsectionCoding expressions

Expressions are coded as digital circuits (like the simple adder below), where each port is coded as a small set of clauses concerning input and output variables and the auxiliary *cables* of the circuits (that are neither inputs or outputs, 3 for the adder circuit) are coded using additional boolean variables.



Boolean vectors are manipulated bit-wise, e.g. if x and y are two variables over the same domain [0..3], then the equality test $x = y$ will

be coded as $z = ((x_1 = y_1) \wedge (x_2 = y_2))$. Again, the boolean variable z is an implicit variable added to simplify the formula and to allow subformula sharing (when the expression $x = y$ appears in some context, it will be replaced by z).

4.3 Coding Time

Clock values increase with time, which is not easy to code in directly. Instead, we introduce new real valued variables $last_x$ to store the time of the last reset of clock x, e.g. if we have the following transition $s \xrightarrow[x:=0]{t,\tau} s'$, then the action $x := 0$ will be coded as $last_x := \tau$. A clock comparison in the transition $s \xrightarrow[x \bowtie c]{t,\tau} s'$ with $\bowtie \in \{\leq, <, =, >, \geq\}$ and c a constant will be coded as $\tau - last_x \bowtie c$. Hence, it is possible to substitute a clock x by the corresponding variable $last_x$ with assignments and conditions as above: in this practical coding, variables do not change *between* transition occurrences. The variables $last_x$ have the same expressive power as clocks.

As seen above, real valued variables are used to manipulate time such as $last_x$ and timestamps τ. However, as analysed in [15,18], it is possible to restrict time stamps to a bounded interval and fixed point numbers (a certain number of variables for the bits of the integer part and the bits of the fractional part), where both the size of the integer part and the precision of the fractional part depend on the length of the searched sequence (more precicely, the number of transition executions).

Alternatively, the coding could be applied for an SMT-solver as in [16,1], where all variables except the timestamps are coded with booleans but the timestamps are coded as real valued variables.

4.4 Duplication of Variables

Because the formula Φ must represent an execution of depth K, we need to add a copy of all variables for each step. We denote $v^i \in V^i$ with $1 \leq i \leq K$ the copy of $v \in V$ at step i. Then if $K = 5$ and $v \in [0..3]$, we have to allocate the following boolean vectors $\{v_1^1, v_2^1\}$, $\{v_1^2, v_2^2\}$, $\{v_1^3, v_2^3\}$, $\{v_1^4, v_2^4\}$, $\{v_1^5, v_2^5\}$.

The result is that the assignment $x := y + 3$ at step k will be coded as $x^{k+1} = y^k + 3$, i.e. an assignment becomes a relation between the value x^{k+1} of x after the current step and the value y^k of y before the current step. Note, that this transition reads y and writes x.

4.5 Transitions and Multisteps

Each execution of transition t in a multistep MT_k is coded with one boolean variable t^k indicating if the transition is executed or not.

If the transition $s_k \xrightarrow[cond,action]{t,\tau} s_{k+1}$ is executed, then its condition *cond* is true at s_k (at time τ) and the assignments of the action are performed (where *action* is coded as constraints between the variables at s_k and at s_{k+1} as indicated above. Formally it is coded as

$$t^{k+1} \rightarrow cond_t(s_k, \tau) \wedge action_t(s_k, s_{k+1})$$

At this point, if s_k is determined and the set of executed transitions includes a transition t that writes v, then $action_t(s_k, s_{k+1})$ also determines the value of v at s_{k+1}. If however, v is written by no executed transition, then its value must be maintained. Suppose that the set of transitions that writes v is $\{t_a, t_b, t_c\}$, then this requirement is coded by the clause

$$t_a^{k+1} \vee t_b^{k+1} \vee t_c^{k+1} \vee v_k = v_{k+1}$$

As for dependency, the condition of pairwise independence of transition executions in a multistep can be coded by a conjunction of constraints $(\neg t_a^k \vee \neg t_b^k)$ for dependent pairs t_a, t_b.

The combination of the action related clauses, dependency related clauses and the clauses for the keeping of values ensure the consistency of successor states. In practice, the constraints concerning writing and reading of variables and those for dependency are coded together, allowing for a more compact coding with sharing. Still, the conflict clauses constitute a significant part of the overall formula.

4.6 Coding Time Progress

- Synchronous: all transitions of a multistep MT_k are executed at the same moment τ_i, i.e. only one timestamp is needed for each step i. We get the following constraints :

$$\bigwedge_{i=1..K-1} \tau_i \leq \tau_{i+1}$$

- Semi synchronous: all transitions of multistep MT_k are executed before some moment ζ_k (additional variable) and all transitions of multistep MT_{k+1} are executed after ζ_k . Each transition t has its own timestamp τ_t, resulting to the following constraints

$$\bigwedge_{\substack{i=1..K-1 \\ t,t' \in \text{trans}}} (\tau_{t^i} \leq \zeta^i) \wedge (\zeta^i \leq \tau_{t'^{i+1}})$$

- Relaxed: it is not straight forward to code relaxed time progress since the condition given in Section 3.3 is not local to two adjacent multi steps. A trick can be used to make occurrence times of previous multisteps locally accessible: for transitions that are not executed, the timestamp τ_t^k has no meaning. We then use it to represent the *last execution time* of t before or including the current multi step. This leads to two cases : if the transition t, τ_t is not executed at step i then the value of τ_t must be maintained at step $i+1$ and if it is executed we add constraints $\leq$ with timestamps of the last executions of dependent transitions:

$$\bigwedge_{k=1..K-1} \bigwedge_{t \in \text{trans}} \neg t^k \rightarrow (\tau_t^k = \tau_t^{k-1}) \wedge$$

$$\bigwedge_{k=1..K-1} \bigwedge_{t_a \in \text{trans}} \bigwedge_{\substack{t_b \in \text{trans} \\ t_a D t_b}} t_a^k \rightarrow (\tau_{t_b}^{k-1} \leq \tau_{t_a}^k)$$

In this formula $t_a \; D \; t_b$ denotes that t_a and t_b are dependent transitions.

4.7 The Global Formula

We resume all the steps for the construction of the global formula Φ which states the existence of a timed multistep sequence:

- Allocate boolean vectors $v_1^k, \ldots, v_n^k$ for all $v \in V$ and for all $1 \leq k \leq K$
- Initialise Φ with initial assignments (constraints) for each variable v^0
- For each step $1 \leq k \leq K-1$
  - $\Phi := \Phi \wedge$ transitions coding
  - $\Phi := \Phi \wedge$ dependency coding
  - $\Phi := \Phi \wedge$ value maintaining
  - $\Phi := \Phi \wedge$ time progress coding
- Add to Φ constraints to ensure the desired path property. For reachability, this can be achieved by stating that the last state satisfies the desired property.

5 Integrating State Invariants

The reader familiar with timed automata will have noticed that we have not dealt with "state invariants" in our reduction. In modelling frameworks like Uppaal, a state invariant is a condition on clocks that is attached to a state of a thread (a value of the local program counter in terms of Section 2) and which is intuitively a "residence permit": the condition $pc_3 = loc_1 \rightarrow x < 5$ states that the state 1 of thread 3 has to be left by a transition before clock x reaches 5. To avoid this violation, either a transition of this thread leaving the state could be executed or a transition of another thread could reset c, thus effectively extending the residence permit for this state.

More generally, state invariants are of the form $pc_i = loc_k \Rightarrow \bigwedge x_j \leq c_j$, i.e. a state value implies a conjunction of upper bounds for clocks. For systems with just one thread, a state invariant has the same effect as adding the constraints to each outgoing transition of the state; they add nothing to the expressiveness of the formalism. For parallel systems, the invariants also imply additional constraints for the outgoing transitions (which must therefore be considered to be reading the corresponding clocks), however, they have a more global effect: the entire system is forced to execute some transition before the expiring of the invariants of each thread. This is very useful in modeling the coupling of subsystems by time, e.g. for modeling timeouts.

It is possible to extend the framework we have developed so far to include state invariants, but technically, this integration depends on the notion of time progress and it is quite complex for relaxed timed progress.

Interleaving Semantics and Synchronous Time Progress. Consider a timed sequence $s_0 \xrightarrow{(t_1,\tau_1)} s_1 \ldots \xrightarrow{(t_n,\tau_n)} s_n = s'$. For standard interleaving semantics, i.e. one transition at a time, the conjunction of all (thread local) state

invariants at global state s_i must be satisfied at time τ_{i+1} (the execution time of the next transition after s_i). Obviously, since $\tau_{i+1} \leq \tau_{i+2}$ the invariant holding *after* the execution of the transition also (already) holds at τ_{i+1}.

For synchronous time progress, the same coding as for interleaving semantics is valid! Although we execute several transitions at time τ_{i+1} we do not execute any transition before that date and hence the state invariant must be satisfied at τ_{i+1}. Since intereavings of timed multisteps with synchronous time progress do not let time pass between the transitions of a multistep, it is easy to see that the condition is necessary and sufficient.

Semi Synchronous Time Progress. For semi synchronous time progress, a similar reasoning as for the synchronous case helps to understand why it is sufficient to require that for each execution time τ_{i+1}^k of some transition in the multistep the invariant must be satisfied.

This can be very efficiently coded by requiring the state invariant to hold at ζ_{i+1}, the separating variable introduced in Section 4.7 for coding semi synchronous time progres: if all τ_{i+1}^k satisfy the invariant then so does their maximum. ζ_{i+1}, by the time progress condition situated anywhere between the timestamps of MT_{i+1} and those of MT_{i+2} can be chosen minimal, i.e. the maximum timestamp of MT_{i+1}. Requiring ζ_{i+1} to respect the state invariant of s_i is thus equivalent to requiring this of every timestamp of MT_{i+1}.

One might argue that this condition, while sufficient, need not be necessary and that a more relaxed condition, while still sufficient, might allow shorter timed multi step sequences. However, then a technique as indicated below for relaxed time progress must be applied. We feel that the technique above is the natural way of dealing with state invariants in the semi synchronous setting.

Relaxed Time Progress. For relaxed time progress, a technique developed for handling invariants in the context of state exploration with zones and partial order semantics [14] can be used. We refer the reader to that paper for technical details and the somewhat involved development of the correctness proof. But we can give a hint on the constraints that actually need to be coded for that approach. One has to distinguish between a local and a global view of invariants: locally, transitions leaving a state must satisfy the invariant, as discussed at the beginning of this section. Globally, a transition resetting a clock must satisfy all variants of the current states of other threads that mention that clock. Finally, the final state must satisfy the global invariant. These three types of constraints are not very difficult to code (and are included in our prototype), the condition for relaxed time progress itself is more complex than this addition.

6 Examples and Experiments

In this section, we present results obtained with a working version of POEM as used in [14], but with a new SAT backend, implementing all multistep algorithms and using Picosat [4] as SAT solver.

The tests were performed on a Mac Pro quad-core 2.66 Ghz, with 16 GB of memory (but a single core is used only and no more than 1GB is required in these computations). The **time** function of the Unix systems was used to get the timings. By default, they all use seconds except when a 'm' appears for minutes. Uppaal [2] times are given for reference.

The numbers for interleaving (only one transition for each step) and multi-steps columns are in the order: the time for the SAT solver to find a solution, the number of clauses in the formula and the number of (multi) steps of the solution, e.g. $2.4/164K/10$ indicates a solving time of $2.4s$ for a formula with approximatively 164000 clauses and 10 multisteps. The symbol '−' is used when no solution has been found within 20 minutes.

Circuit Analysis

We introduce a simple circuit problem: several NOT gates are connected one to the other as in figure 3. Each gate has a delay to propagate its input signal to its output. Initially each value is equal to zero. We want to know if the circuit can stabilize, i.e. if there exists a time t where input values and output values are coherent (and thus will no longer change). Of course, the circuit can stabilize only if there is an even number of gates. It turns out that the desired state is reachable with one (synchronous) multi step. As can be seen, the more complex encodings yield no advantage here.

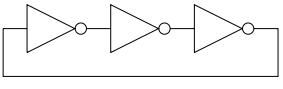


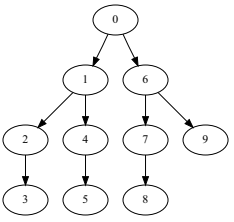
Fig. 3. A circuit with NOT gates

Table 1. Results for the circuit problem

nodes	UppAal	Interleaving	Multistep		
			synchronous	semi synchronous	relaxed
4	0	0/12K/2	0/8K/1	0/23K/1	0/53K/1
10	0.1	0.3/48K/5	0/22K/1	0.1/58K/1	0.9/139K/1
16	-	1.7/112K/8	0.5/38K/1	1.6/99K/1	1.2/242K/1
20	-	2.4/164K/10	0.8/48K/1	1.8/123K/1	1.3/301K/1
50	-	-	1.2/129K/1	2.3/327K/1	2.8/811K/1
100	-	-	3.1/279K/1	3.9/704K/1	5.6/1,7M/1
200	-	-	4.3/608K/1	6.1/1.5M/1	11.1/3,8M/1

Timed Network Protocols

We consider the following simplistic broadcast protocol: nodes arranged in a (non complete) binary tree can only send a signal to their children after receiving a signal from their parents. When a leaf receives a signal, it sends back an



acknowledgement. When an interior node receives acknowledgements from its two children, it sends one to its own parent, and so on. To resume, a signal starts from the root, is asynchronously propagated to the leaves and back to the root. A random delay for each transmission between a parent and its children is added. The model checker is asked to find a completed broadcast within a tight interval of time. For this series of examples, the semi synchronous coding allows significantly shorter multi step runs than the synchronous coding and sometimes the relaxed coding allows even shorter sequences. It turns out that shorter here means (much) faster, whereas at the same length, the relaxed coding comes with an overhead over the semi synchronous coding and is slower.

Table 2. Results for the network protocol problem

nodes	UppAal	Interleaving	Multistep		
			synchronous	semi synchronous	relaxed
5	0.0	0/11K/9	0.1/7K/6	0.2/19K/6	0.2/29K/6
10	0.1	2.9/49K/19	0.8/31K/12	0.3/59K/8	0.8/95K/8
15	20.0	19.0/110K/29	0.5/40K/10	0.5/90K/10	1.1/144K/8
20	-	4m46/196K/39	3.5/93K/18	3.5/175K/12	2.3/240K/10
50	-	-	18.2/292K/20	5.6/508K/12	8.2/846K/12
100	-	-	11m06/812K/28	19.2/1.2M/14	34.1/2M/14
200	-	-	-	3m09/2.7M/18	2m15/4.6M/16

7 Conclusions and Future Work

We have studied the problem of enhancing the SAT reductions of bounded model checking of timed automata with the help of multisteps. We have identified three different alternative semantics for coding and have given a few experiments to compare them.

Related work. While we are not aware of any work trying the combination we have considered here (SAT, partial order, timed automata), many aspects of this work find their origin in other works: The basic coding principles, including the variable transformation (using timestamps rather than clock values) are already present in previous works on BMC for timed automata, whether oriented towards pure boolean SAT or SMT (SAT modulo theories) [13,1,15,18]. "Relaxed semantics" with respect to time has been widely discussed in the context of symbolic state exploration of timed automata, e.g. in [3] whereas the idea of allowing timestamps to be commuted was first presented in [12]. When abstracting from time, the key idea of using step semantics in bounded model checking was stated in [9]. Beyond the context of bounded model checking, one work considers executing several transitions of timed automata in parallel in order to avoid zone splitting [19], but no indication towards SAT applications is found in that work.

Interpretation of experimental evidence. We have given two series of experiments that illustrate how the alternative semantics can dramatically improve the performance of the SAT approach to timed automata reachability. From these and non-documented experiments, our personal assessment is that "synchronous time progress" is always a good idea to start with (the smallest set of clauses at the same path length), and if it runs out of time with increasing path length to switch to semi synchronous time progress. We have not yet found examples where relaxed time progress yields an advantage in execution time (although sometimes shorter paths were found). Obviously, case studies on realistic examples are necessary for further evaluation.

Perspectives. We have not implemented a reduction to SMT, but, as outlined in Section 4, the coding would be the same except for the representation of time stamps by real valued variables and corresponding constraints. We believe that the improvement achieved for the current boolean only approach carry over seamlessly to the SMT case. We might explore an SMT variant of our implementation in the future.

References

1. Audemard, G., Cimatti, A., Kornilowicz, A., Sebastiani, R.: Sat-based bounded model checking for timed systems. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529. Springer, Heidelberg (2002)
2. Behrmann, G., David, A., Larsen, K.G., Hakansson, J., Petterson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: 3rd international conference on the Quantitative Evaluation of Systems QEST, Washington, DC, USA, pp. 125–126 (2006)
3. Bengtsson, J., Jonsson, B., Lilius, J., Yi, W.: Partial order reductions for timed systems. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 485–500. Springer, Heidelberg (1998)
4. Biere, A.: PicoSAT essentials. Journal on Satisfiability, Boolean Modeling and Computation 4, 75–97 (2008)
5. Bozga, M., Graf, S., Mounier, L.: If-2.0: A validation environment for component-based real-time systems. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 343. Springer, Heidelberg (2002)
6. Courtois, P.J., Heymans, F., Parnas, D.L.: Concurrent control with "readers" and "writers". Commun. ACM 14(10), 667–668 (1971)
7. Dams, D., Gerth, R., Knaack, B., Kuiper, R.: Partial-order reduction techniques for real-time model checking. Formal Aspects of Computing 10, 469–482 (1998)
8. Genrich, H.J., Lautenbach, K., Thiagarajan, P.S.: Elements of general net theory. In: Proceedings of the Advanced Course on General Net Theory of Processes and Systems, London, UK, pp. 21–163 (1980)
9. Heljanko, K.: Bounded reachability checking with process semantics. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 218–232. Springer, Heidelberg (2001)
10. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Information and Computation 111, 193–244 (1994)
11. Jussila, T., Niemelä, I.: Parallel program verification using BMC. In: ECAI 2002 Workshop on Model Checking and Artificial Intelligence, pp. 59–66 (2002)

12. Lugiez, D., Niebert, P., Zennou, S.: A partial order semantics approach to the clock explosion problem of timed automata. Theoretical Computer Science 345(1), 27–59 (2005)
13. Niebert, P., Mahfoudh, M., Asarin, E., Bozga, M., Jain, N., Maler, O.: Verification of timed automata via satisfiability checking. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 225–244. Springer, Heidelberg (2002)
14. Niebert, P., Qu, H.: Adding invariants to event zone automata. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 290–305. Springer, Heidelberg (2006)
15. Penczek, W., Wozna, B., Zbrzezny, A.: Towards bounded model checking for the universal fragment of TCTL. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 265–290. Springer, Heidelberg (2002)
16. Ben Salah, R., Bozga, M., Maler, O.: On interleaving in timed automata. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 465–476. Springer, Heidelberg (2006)
17. Sorea, M.: Bounded model checking for timed automata. Electronic Notes in Theoretical Computer Science 68(5) (2002), `http://www.elsevier.com/locate/entcs/volume68.html`
18. Zbrzezny, A.: Sat-based reachability checking for timed automata with diagonal constraints. Fundam. Inf. 67(1-3), 303–322 (2005)
19. Zhao, J., Xu, H., Xuandong, L., Tao, Z., Guoliang, Z.: Partial order path technique for checking parallel timed automata. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 417–431. Springer, Heidelberg (2002)