Bounded Model Checking for Timed Systems

G. Audemard[1], A. Cimatti[1], A. Kornilowicz[1], and R. Sebastiani[1,2]

[1] ITC-IRST, via Sommarive 16, 38050 Povo, Trento, Italy
{audemard,cimatti,kornilow}@itc.it
[2] DIT, Università di Trento, via Sommarive 14, 38050 Povo, Trento, Italy
rseba@science.unitn.it

Abstract. Enormous progress has been achieved in the last decade in the verification of timed systems, making it possible to verify significant real-world protocols. An open challenge is the identification of fully symbolic verification techniques, able to deal effectively with the finite state component as well as with the timing aspects.
In this paper we propose a new, symbolic verification technique that extends the Bounded Model Checking (BMC) approach for the verification of timed systems. The approach is based on the following ingredients. First, a BMC problem for timed systems is reduced to the satisfiability of a math-formula, i.e., a boolean combination of propositional variables and linear mathematical relations over real variables (used to represent clocks). Then, an appropriate solver, called MathSAT, is used to check the satisfiability of the math-formula. The solver is based on the integration of SAT techniques with some specialized decision procedures for linear mathematical constraints, and requires polynomial memory. Our methods allow for handling expressive properties in a fully-symbolic way. A preliminary experimental evaluation confirms the potential of the approach.

1 Introduction

The verification of timed systems is a very important and challenging problem. In the last decade, it has being devoted a lot of interest, and significant results have been achieved, making it possible to verify real protocols with limited computational resources (see e.g. [LPY95,DY95]). The verification of timed systems combines the challenge of finite-state variables with the problems related to time: a state can be seen as an assignment to propositional variables and to real variables, called clocks.

The verification of timed systems is traditionally based on the use of Difference Bound Matrices (DBMs) [Dil89], that compactly represent a region associated with an assignment to the clocks that are compatible with a specific assignment to propositional variables. Despite their efficiency, such techniques are basically explicit-state: a complete assignment to propositional variables is associated with a complete region, and the amount of required memory is the most limiting factor in the verification process. Recently proposed techniques, such as DDD [MLAH01], CDDs [LWYP98], and RED [Wan00], provide symbolic representations for the search space. Also in this case, however, the memory requirements can be substantial, because of the inheritance of the properties of Binary Decision Diagrams.

In this paper, we propose a new symbolic technique for the verification of timed systems. The approach is a generalization of Bounded Model Checking (BMC) [BCCZ99], that is gaining increasing interest for the verification of finite state systems [Sht00,CFG$^+$01a]. The approach consists on encoding the BMC problem of timed systems into the problem of deciding the satisfiability of math-formulas, i.e. boolean combinations of boolean variables and linear (in)equalities over real variables, representing clocks. The resulting problems are then tackled with MATHSAT, a solver combining an efficient procedure for propositional satisfiability (SAT) [DLL62] with mathematical constraint solvers of increasing deductive power.

The approach is rather general, since it allows to express specifications in full LTL, such as fairness properties. Furthermore, the approach is fully symbolic: it allows us to tackle the digital component of timed systems with symbolic technologies as in the untimed case, while the timed component is tackled by means of specialized mathematical constraint solvers. Finally, the math-formulas generated are polynomial w.r.t. the size of the representation of the input system and the maximum path length k, and are solved by a solver requiring a polynomial amount of memory. Although preliminary, our experimental analysis confirms the potentials of the proposed approach.

The paper is organized as follows. In Section 2 we give some background: we recall the basics on timed automata [Alu99], and we describe the MATHSAT problem [ABC$^+$02]. In Section 3 we describe the idea of BMC for timed automata, and we show how to reduce the problem to the satisfiability of a math-formula. In Section 4 we discuss some optimizations to the encoding. In Section 5 we discuss some related approaches. In Section 6 we present some preliminary empirical results. In Section 7 we draw some conclusions, and outline the future directions.

2 Background

2.1 Model Checking Timed Automata

In this section we briefly recall timed automata [Alu99]. An *atomic clock constraint* is any expression in the form $(x \bowtie c)$, $\bowtie \in \{\leq, \geq, <, >\}$, x being a clock variable with values in $\mathbb{R}$ and $c \in \mathbb{Z}$ being a constant; a *clock constraint* is any conjunction of atomic clock constraints. (To this extent, notice that every clock constraint is convex.)

A *timed automaton* A_1 is a tuple $\langle L_1, L_1^0, \Sigma_1, X_1, I_1, E_1 \rangle$ where L_1 is a finite set of locations, $L_1^0 \subseteq L_1$ is the set of initial locations, Σ_1 is a finite set of labels for the possible *events*, X_1 is a finite set of clock variables, I_1 is a map labeling every location $s \in L_1$ with a clock constraint on X_1, and E_1 is the set of switches. Every switch $T = \langle s_i, a, \varphi, \lambda, s_j \rangle$ is characterized by its source and target locations $s_i, s_j \in L_1$, an event $a \in \Sigma_1$, a clock constraint φ on X_1, and a set of clock reset conditions λ in the form $(x = 0)$, $x \in X_1$. If $A_1 = \langle L_1, L_1^0, \Sigma_1, X_1, I_1, E_1 \rangle$ and $A_2 = \langle L_2, L_2^0, \Sigma_2, X_2, I_2, E_2 \rangle$, s.t. X_1 and X_2 are disjoint, then the *product* $A = A_1 || A_2$ is defined as $\langle L, L^0, \Sigma, X, I, E \rangle$, with $L = L_1 \times L_2$, $L^0 = L_1^0 \times L_2^0$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $X = X_1 \cup X_2$, I s.t. $I(s_1, s_2) = I(s_1) \wedge I(s_2)$, and E is defined as follows:

1. if $a \in \Sigma_1 \cap \Sigma_2$, $\langle s_{1i}, a, \varphi_1, \lambda_1, s_{1j} \rangle \in E_1$ and $\langle s_{2i}, a, \varphi_2, \lambda_2, s_{2j} \rangle \in E_2$, then $\langle (s_{1i}, s_{2i}), a, \varphi_1 \wedge \varphi_2, \lambda_1 \cup \lambda_2, (s_{1j}, s_{2j}) \rangle \in E$;
2. if $a \in \Sigma_1 \backslash \Sigma_2$, $\langle s_{1i}, a, \varphi_1, \lambda_1, s_{1j} \rangle \in E_1$ and $t \in L_2$, then $\langle (s_{1i}, t), a, \varphi_1, \lambda_1, (s_{1j}, t) \rangle \in E$;
3. if $a \in \Sigma_2 \backslash \Sigma_1$, $\langle s_{2i}, a, \varphi_2, \lambda_2, s_{2j} \rangle \in E_2$ and $t \in L_1$, then $\langle (t, s_{2i}), a, \varphi_2, \lambda_2, (t, s_{2j}) \rangle \in E$.

The dynamics of the timed automaton A_1 is given by means of a transition system S_{A_1}. A state of S_{A_1} is a pair (s, ν) s.t. $s \in L_1$ and $\nu : X_1 \longmapsto \mathbb{R}^+$ is a *clock evaluation* satisfying $I(s)$. (s, ν) is an initial state iff $s \in L_1^0$ and $\nu(x) = 0$, $\forall x \in X_1$. S_{A_1} can evolve into two different ways. With **time elapse**, if $\delta \geq 0$, then $(s, \nu) \stackrel{\delta}{\longmapsto} (s, \nu + \delta)$, $\nu + \delta$ being the evaluation such that $(\nu + \delta)(x) = \nu(x) + \delta$, $\forall x \in X_1$, and $\nu + \delta'$ satisfies I, $\forall \delta'$ s.t. $0 \leq \delta' \leq \delta$. With **switch**, if $\langle s_i, a, \varphi, \lambda, s_j \rangle \in E_1$ and ν satisfies φ, then $(s_i, \nu) \stackrel{a}{\longmapsto} (s_j, \nu[\lambda = 0])$, $\nu[\lambda = 0]$ being the evaluation assigning $x = 0$ $\forall x \in \lambda$ and agreeing with ν for the other clocks in X_1.

Let $A = \langle L, L^0, \Sigma, X, I, E \rangle$, and let $\mathbf{\Phi}(X)$ and $\mathbf{\Lambda}(X)$ be the set of all possible atomic clock constraints and clock reset conditions on X. Let $\mathbf{\Phi}_A(X)$ be the set of all atomic clock constraints on X occurring in the automata A. We see as atomic propositions the locations $s \in L$ (meaning "A is in location s"), the elements of $\mathbf{\Phi}_A(X)$ and $\mathbf{\Lambda}(X)$. The timed automaton A induces a Kripke structure $\langle S, S^0, R, \mathcal{L} \rangle$, with a finite set of states S, a set of initial states $S^0 \subset S$, a transition relation $R \in S \times S$ and a labeling function $\mathcal{L} : S \longmapsto \mathcal{P}ow(L \cup \mathbf{\Phi}_A(X) \cup \mathbf{\Lambda}(X))$. $\mathcal{L}$ is such that, for every $\sigma \in S$, exactly one $s \in L$ is in $\mathcal{L}(\sigma)$ —we denote s as $location(\sigma)$— and the elements of $\mathbf{\Phi}_A(X) \cup \mathbf{\Lambda}(X)$ in $\mathcal{L}(\sigma)$ have an evaluation in $(\mathbb{R}^+)^{|X|}$. S^0 is the set of $\sigma \in S$ s.t., for every $x \in X$, $(x = 0) \in \mathcal{L}(\sigma)$, $location(\sigma) \in L^0$ and $I(location(\sigma))[X := 0] \subseteq \mathcal{L}(\sigma)$. R is such that, for every $\sigma_i, \sigma_j \in S$, $R(\sigma_i, \sigma_j)$ holds if and only if either (i) one switch $\langle s_i, a, \varphi, \lambda, s_j \rangle$ in Σ is such that $s_i = location(\sigma_i)$, $I(s_i) \subseteq \mathcal{L}(\sigma_i)$, $s_j = location(\sigma_j)$, $I(s_j) \subseteq \mathcal{L}(\sigma_j)$, $\varphi \subseteq \mathcal{L}(\sigma_i)$, $\lambda \subseteq \mathcal{L}(\sigma_j)$ for some $a \in E$; (ii) for some $\delta \geq 0$, $s := location(\sigma_i) = location(\sigma_j)$ is such that $I(s) \subseteq \mathcal{L}(\sigma_i)$, $I(s)[X := X + \delta'] \subseteq \mathcal{L}(\sigma_j)$ for every $\delta' \in [0, \delta]$.

We use Linear Temporal Logic (LTL) with its standard semantics [Eme90] to specify properties of timed automata. Basic propositions are atomic propositions, atomic clock constraint in $\mathbf{\Phi}_A(X)$, and clock reset conditions in $\mathbf{\Lambda}(X)$; a propositional literal (i.e., a basic proposition or its negation) is a LTL formula; if h and g are LTL formulas, then $h \wedge g$, $h \vee g$, $h \leftrightarrow g$, $\mathbf{X}g$, $\mathbf{G}g$, $\mathbf{F}g$, $h\mathbf{U}g$ and $h\mathbf{R}g$ are LTL formulas, $\mathbf{X}$, $\mathbf{G}$, $\mathbf{F}$, $\mathbf{U}$ and $\mathbf{R}$ being the standard "next", "globally", "eventually", "until" and "releases" temporal operators respectively. In order to encompass the case of a property f including atomic clock constraints not in $\mathbf{\Phi}_A(X)$, it is possible to extend the labeling function of the Kripke structure to $\mathcal{L} : S \longmapsto \mathcal{P}ow(L \cup \mathbf{\Phi}_A(X) \cup \mathbf{\Phi}_f(X) \cup \mathbf{\Lambda}(X))$, where $\mathbf{\Phi}_f(X)$ is the set of atomic clock constraints in f.

2.2 Satisfiability of math-formulae

We call *math-formula* a boolean combination of boolean variables and linear constraints over numerical variables. We call an *interpretation* a map $\mathcal{I}$ which assigns real and boolean values to real and boolean variables respectively and preserves constant values, arithmetical and boolean operators. For instance, $\mathcal{I}((x - y \geq 4) \wedge (\neg A_1 \vee (x = y))) = (\mathcal{I}(x) - \mathcal{I}(y) \geq 4) \wedge (\neg \mathcal{I}(A_1) \vee (\mathcal{I}(x) = \mathcal{I}(y)))$. We say that $\mathcal{I}$ *satisfies* a math-formula ϕ, written $\mathcal{I} \models \phi$, iff $\mathcal{I}(\phi)$ evaluates to true. We call *MATHSAT* the problem of checking the satisfiability of a math-formula. It is easy to see that the general MATHSAT problem is NP-complete.

We call a *truth assignment* for a math-formula ϕ a truth value assignment μ to (a subset of) the atoms of ϕ. We say that *μ propositionally satisfies ϕ*, written $\mu \models_p \phi$, iff it makes ϕ evaluate to true. We represent truth assignments as sets of literals, with the intended meaning that positive and negative literals represent atoms assigned to true and to false respectively. $\mathcal{I}$ satisfies μ iff it satisfies all its elements. For instance, the

```
boolean MathSAT(formula φ, interpretation & I)
    μ = ∅;
    return MathDPLL(φ, μ, I);

boolean MathDPLL(formula φ, assignment & μ, interpretation & I)
    if (φ == ⊤) {                                   /* base         */
        I = MathSolve(μ) ;
        return (I ≠ Null) ; }
    if (φ == ⊥)                                     /* backtrack */
        return False;
    if {a literal l occurs in φ as a unit clause}   /* unit propagation    */
        return MathDPLL(assign(l, φ), μ ∪ {l}, I);
    l = choose-literal(φ);                          /* split        */
    return   (MathDPLL(assign(l, φ), μ ∪ {l}, I)   or
              MathDPLL(assign(¬l, φ), μ ∪ {¬l}, I) );
```

Fig. 1. Pseudo-code of the basic version of the MathSAT procedure.

assignment $\{(x - y \geq 4), \neg A_1\}$ propositionally satisfies $(x - y \geq 4) \wedge (\neg A_1 \vee (x = y))$, and it is satisfied by $\mathcal{I}$ s.t. $\mathcal{I}(x) = 6$, $\mathcal{I}(x) = 1$, $\mathcal{I}(A_1) = \perp$.

To solve the MATHSAT problem, we have implemented MathSAT [Seb01,ABC+02], a solver based on a variant of the DPLL SAT procedure [DLL62]. The basic schema of such a procedure is reported in Figure 1. MathSAT takes as input a math-formula φ, expressed in CNF, and (by reference) an empty interpretation $\mathcal{I}$, and returns a truth value asserting whether φ is satisfiable or not, $\mathcal{I}$ being respectively an interpretation satisfying φ or $Null$. MathSAT invokes MathDPLL passing as arguments φ and (by reference) an empty assignment μ and the interpretation $\mathcal{I}$. MathDPLL tries to find a truth assignment μ propositionally satisfying φ which is satisfiable from the mathematical viewpoint.

Basically, MathDPLL is a variant of DPLL, modified to work as an enumerator of truth assignments, whose satisfiability is recursively checked by MathSolve. (The function *assign*(l, φ) assigns l to $\top$ in φ and propositionally simplifies the result.) The key difference w.r.t. standard DPLL is in the "base" step. Standard DPLL needs finding only one satisfying assignment μ, and thus simply returns $True$. MathDPLL instead also needs checking the satisfiability of μ, and thus it invokes MathSolve(μ). Then it returns $True$ if a non-null interpretation satisfying μ is found, it returns $False$ and backtracks otherwise.

MathSolve takes as input an assignment μ and returns either an interpretation $\mathcal{I}$ satisfying μ or $Null$ if there is none. In our implementation, MathSolve first performs all the substitutions allowed by the equalities in μ. Then, if only inequalities with two-variable are left, then a variant of Bellman-Ford minimal path algorithm is invoked, a linear programming procedure otherwise.

Notice that MathSAT works in polynomial space. In [Seb01,ABC+02], the proofs of correctness and completeness of MathSAT are reported, as well as the description of some improvements on the procedure of Figure 1 (e.g. preprocessing and sorting, intermediate assignment checking, triggering, math-driven backjumping and learning) which we have implemented.

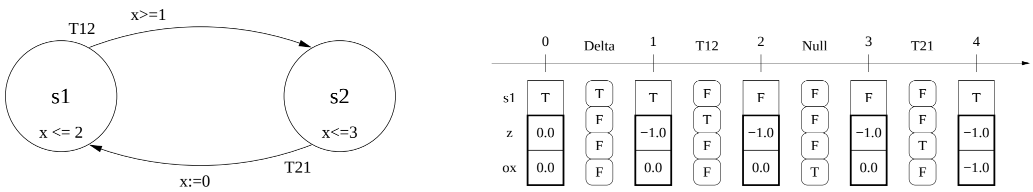

Fig. 2. A simple automaton example

3 Bounded Model Checking for Timed Automata

Bounded Model Checking (BMC) is a recent approach to symbolic model checking [BCCZ99]. The starting point is an existential model checking problem $M \models \mathbf{E}f$, for an LTL formula f, and a Kripke structure M. The idea is to solve the problem by looking for a witness to the property that can be presented within a bound of k steps. Given k, the problem is reduced to the satisfiability of a propositional formula $[[M, f]]_k$. If $[[M, f]]_k$ is satisfiable, the propositional model provides a witness of k steps to f. The method is not complete: if $[[M, f]]_k$ is unsatisfiable, then nothing can be said about the existence of solutions for $M \models f$ models with higher bound. Thus, the typical technique is to generate and solve $[[M, f]]_k$ for increasing values of k, until either a counter-example is found, or a given time-out is reached. (Completeness can be in principle achieved when k reaches the *diameter* of the problem. Unfortunately, such value is typically hard to compute, and very big. A somewhat partial solution is the use of inductive techniques [BCC+99].)

Despite this limitation, BMC is being increasingly accepted as an effective and practical technique, in particular in the process of falsification, i.e. bug finding. The problem is tackled by refutation, looking for witnesses of bound k to the negation of the property being analyzed. The technique relies on the use of efficient SAT solvers (e.g. based on DPLL procedures) for checking the propositional satisfiability of $[[M, f]]_k$. As shown in [CFG+01b], BMC avoids the blow-up in memory that can occur with model checking based on Binary Decision Diagrams, and is therefore able to tackle much larger circuits. Furthermore, SAT-based techniques appear to require less tuning to be effective, and are therefore more amenable to the introduction in industrial settings.

In this paper, we address the problem of BMC for $M \models f$ for the case of timed systems, where M is a Kripke structure induced by a timed automaton, and f is an LTL formula. The encoding $[[M, f]]_k$ is a math-formula, where real variables are used to represent the temporal part of the state space and its evolution. The encoding is a combination of a characterization of the paths of the automaton (described in section 3.1) with a characterization of the paths that satisfy the specification (described in section 3.2).

3.1 Encoding Paths and Loops of Timed Automata

In the following, we assume that A_1 and A_2, with $A_i = \langle L_i, L_i^0, \Sigma_i, X_i, I_i, E_i \rangle$, and $A = A_1 || A_2 = \langle L, L^0, \Sigma, X, I, E \rangle$, are given. For explanatory purposes, we use the simple automaton depicted in Figure 2.

Boolean Variables In order to represent the status and the evolution of the system A_1, we introduce the following propositional variables. For locations, we introduce an array $\underline{s}$ of $\lceil log_2(|L_1|) \rceil$ boolean variables. The intended meaning is that

$$\bigvee_{s_i \in L_1^0} \underline{s_i} \qquad\qquad\qquad \bigwedge_{x \in X_1} (x = z) \tag{1}$$

$$\bigwedge_{s_i \in L_1} (\underline{s_i} \rightarrow \bigwedge_{\psi \in I(s_i)} \psi), \tag{2}$$

Fig. 3. Encoding Initial Conditions and Invariants for A_1

$\underline{s_i}$ holds if and only if the system is in the location s_i. By "$(\underline{s_i} = \underline{s_j})$" we mean "$\bigwedge_n(\underline{s_i}[n] \leftrightarrow \underline{s_j}[n])$", $n \in \{1, ..., \lceil log_2(|L_1|)\rceil\}$. To represent each event $a \in \Sigma_1$, we introduce a boolean variable $\underline{a}$, with the intended meaning that $\underline{a}$ holds if and only if the system executes a switch of event a. For each switch $\langle s_i, a, \varphi, \lambda, s_j\rangle \in E_1$ we introduce a single boolean variable (e.g., T), with the intended meaning that T holds if and only if the system executes the corresponding switch. Finally, we introduce two boolean variables T_δ and T^1_{null}, with the intended meaning that T_δ holds if and only if time elapses by some $\delta > 0$, and that T^1_{null} holds if and only A_1 does nothing, respectively. For instance, in the example of Figure 2, we have the variable $\underline{s}$ for the state, while for transitions we have the variables $T_\delta, T_{12}, T_{21}, T_{null}$. (Since both transitions are labeled by the same event, no variables for events are needed.)

Real Variables The clocks in the automaton are represented by means of real variables in the encoding, as follows. We introduce a real variable z, called "absolute time reference", whose negated value represents the time elapsed from the beginning of the path being analyzed. Then, for each clock x in the automaton, we introduce an "offset" variable ox whose negated value is the absolute time when the clock was last reset. In general, the value of a clock x is obtained as $ox - z$. In the following, we write r' for the value of the real variable r after a transition of the automaton. In this setting, when the automaton performs a delta transition, and time advances, we have that $z' < z$. Otherwise, time does not elapse, and $z' = z$. Every condition or operation over a clock x can be encoded by means of the difference between the absolute time z and ox. This trivially applies to state constraints and transition constraints of the form $(x \bowtie c)$, with $\bowtie \in \{\leq, \geq, <, >\}$, with $c \in \mathbb{Z}$ being a constant, that are reduced to $(ox' - z' \bowtie c)$. We encode the fact that transition resets a clock by means of the constraint $ox' = z'$. Similarly we impose that clocks have non-negative values by means of the constraint $ox' \leq z'$. In the example of Figure 2, the clock x in the automaton on the left is represented by the difference between ox and z on the right. When clear from context, in the following we write x for ox.

Paths and Loops A path of length k in the (Kripke structure associated with the) automaton is encoded by replicating the propositional and real variables from 0 to k, and the label and transition variables from 0 to $k-1$. (We use the suffix $^{(i)}$ to indicate the step index of a variable: e.g., $T_\delta{}^{(i)}$ and $T^{j\ (i)}_{null}$ indicate T_δ and T^j_{null} at step i respectively.) A path is obtained by constraining the values of the state vectors at $i, i+1$ via the transition relation of the automaton. Notice that, $z^{(i+1)} \leq z^{(i)}$, $ox^{(i)} \geq z^{(i)}$ and, by induction, $ox^{(i+1)} \leq ox^{(i)}$, as x can either be reset or keep its value. For the example automaton, we depict in Figure 2, on the right, the variables needed to encode a path of bound $k = 4$. The vertical squares represent the state vector at the different steps, where thick squares enclose values for real variables,

$$\bigwedge_{\substack{T = \langle s_i, a, \varphi, \lambda, s_j \rangle \\ T \in E_1}} T \rightarrow \left(\underline{s_i} \wedge \underline{a} \wedge \varphi \wedge \underline{s_j}' \wedge \bigwedge_{x \in \lambda} (x' = z') \wedge \bigwedge_{x \notin \lambda} (x' = x) \wedge (z' = z) \right) \tag{3}$$

$$T_\delta \rightarrow \left((z' - z < 0) \wedge (\underline{s}' = \underline{s}) \wedge \bigwedge_{x \in X_1} (x' = x) \wedge \bigwedge_{a \in \Sigma_1} \neg \underline{a} \right) \tag{4}$$

$$T^1_{null} \rightarrow \left((z' = z) \wedge (\underline{s}' = \underline{s}) \wedge \bigwedge_{x \in X_1} (x' = x) \wedge \bigwedge_{a \in \Sigma_1} \neg \underline{a} \right) \tag{5}$$

$$T^1_{null} \vee T_\delta \vee \bigvee_{T \in E_1} T \tag{6}$$

$$\bigwedge_{\substack{a_k, a_r \in \Sigma_1 \\ a_k \neq a_r}} (\neg \underline{a}_k \vee \neg \underline{a}_r) \qquad \bigwedge_{\substack{T_k = \langle s_i, a, \varphi_1, \lambda_1, s_j \rangle, \\ T_r = \langle s_i, a, \varphi_2, \lambda_2, s_j \rangle, \\ T_k \neq T_r,\ T_k, T_r \in E_1}} (\neg T_k \vee \neg T_r) \tag{7}$$

Fig. 4. Encoding Transitions for A_1

while the corner-rounded squares represent the propositional values of transitions (from top to bottom $T_\delta, T_{12}, T_{21}, T_{null}$).

A set of implicitly conjoined constraints is needed to make sure that the assignments to the variables represent a legal path of the automaton. The **initial conditions**, holding over the first state vector, state that the system can be only in one of the initial locations (Figure 3, Eq. 1, left), and that the clocks are all zero (right). The **invariants** (Figure 3, Eq. 2), state that if the system is in a location s_i, then all the associated clock constraints must hold, and must be replicated for all state vectors. The constraints in Figure 4 describe the effect of switches, delta and null transitions, and must be replicated from steps 0 to $k-1$. At step i, the current and state vectors are substituted with the state vector at step i and $i+1$, respectively.

Equation 3 encodes **switches** $\langle s_i, a, \varphi, \lambda, s_j \rangle \in E_1$. Intuitively, if $\langle s_i, a, \varphi, \lambda, s_j \rangle$ is being fired, then: (i) before the switch, the automaton is in location s_i, the event a occurs and the constraints φ are verified; (ii) after the switch, the automaton is in location s_j, all clocks in λ are reset, and the values of the other clocks are the same as before the transition; (iii) as no time elapse can occur when switching, the value of z is the same before and after the transition.

The formula (4) encodes **delta** transitions: (i) time elapse must be strictly greater than 0, (ii) in the next state the system must be in the same location as in the current state, (iii) the values of all clocks must be identical to those of the current state, and (iv) no event in Σ_1 can occur together with time elapsing. (The invariants $I(s)$ are conjunctions of linear inequalities, and represent convex regions; thus, if $x' = x$, $z' < z$ and both $x - z$ and $x' - z'$ verify I for every $x \in X_1$, then $x'' - z''$ s.t. $z'' \in [z', z]$ and $x'' = x$ verifies I for every $x \in X_1$.)

The formula (5) encodes the **null** transition, enforcing that (i) time elapse must be equal to zero, (ii) in the next state the system must be in the same location as in the current state, (iii) the values of all clocks must be identical to those of the current state, and (iv) no event in Σ_1 can occur.

The remaining formulae (6-7) express the relation between the different transitions. Formula (6) states that at least one variable among the variables T in E_1, T_δ and

f	$[[f]]_k^i$	${}_l[[f]]_k^i$
p	$p^{(i)}$	$p^{(i)}$
$\neg p$	$\neg p^{(i)}$	$\neg p^{(i)}$
$h \wedge g$	$[[h]]_k^i \wedge [[g]]_k^i$	${}_l[[h]]_k^i \wedge {}_l[[g]]_k^i$
$h \vee g$	$[[h]]_k^i \vee [[g]]_k^i$	${}_l[[h]]_k^i \vee {}_l[[g]]_k^i$
$\mathbf{X}g$	$[[g]]_k^{i+1}$ $if\ i < k$ $\perp$ $otherwise.$	${}_l[[g]]_k^{i+1}$ $if\ i < k$ ${}_l[[g]]_k^l$ $otherwise.$
$\mathbf{G}g$	$\perp$	$\bigwedge_{j=min(i,l)}^k {}_l[[g]]_k^j$
$\mathbf{F}g$	$\bigvee_{j=i}^k [[g]]_k^j$	$\bigvee_{j=min(i,l)}^k {}_l[[g]]_k^j$
$h\mathbf{U}g$	$\bigvee_{j=i}^k \left([[g]]_k^j \wedge \bigwedge_{n=i}^{j-1} [[h]]_k^n \right)$	$\bigvee_{j=i}^k \left({}_l[[g]]_k^j \wedge \bigwedge_{n=i}^{j-1} {}_l[[h]]_k^n \right) \vee$ $\bigvee_{j=l}^{i-1} \left({}_l[[g]]_k^j \wedge \bigwedge_{n=i}^{k} {}_l[[h]]_k^n \wedge \bigwedge_{n=l}^{j-1} {}_l[[h]]_k^n \right)$
$h\mathbf{R}g$	$\bigvee_{j=i}^k \left([[h]]_k^j \wedge \bigwedge_{n=i}^{j} [[g]]_k^n \right)$	$\bigwedge_{j=min(i,l)}^k {}_l[[g]]_k^j \vee$ $\bigvee_{j=i}^k \left({}_l[[h]]_k^j \wedge \bigwedge_{n=i}^{j} {}_l[[g]]_k^n \right) \vee$ $\bigvee_{j=l}^{i-1} \left({}_l[[h]]_k^j \wedge \bigwedge_{n=i}^{k} {}_l[[g]]_k^n \wedge \bigwedge_{n=l}^{j} {}_l[[g]]_k^n \right)$

Table 1. Recursive definition of $[[f]]_k^i$ and ${}_l[[f]]_k^i$.

T_{null}^1 must hold, i.e. in system A_1 either a switch shoots, time passes, or stuttering occurs. The formula (7) states mutual exclusion between events, that is, two different events in Σ_1 cannot occur at the same time. The formulas (7) state the mutual exclusion between two switches in the (rare) case they share the same event, source and target locations. In every other case, the mutual exclusion between two switches is a consequence of (3) and the mutual exclusion of states and of events (7).

An infinite, cyclic behaviour can be encoded as a path of length k with a loop back at l, with $0 \leq l < k$. For the propositional part of the state vector, this can be expressed by imposing that each variable has the same value at l and k. For each clock x, we impose that $(ox^{(k)} - z^{(k)} = ox^{(l)} - z^{(l)})$. Notice that it is not possible to express this condition simply by means of equalities between $(ox^{(k)} = ox^{(l)})$, since ox decreases monotonically. Given M and an integer $k \geq 0$, we write $[[M]]_k$ for (the math-formula representing) a path of bound k for the Kripke structure M, ${}_lL_k$ for the loopback condition from k to l.

Product construction The encoding for the product automaton $A = A_1||A_2$ follows almost directly from the encodings of A_1 and A_2, by conjunction. (Notice that T_δ is common to all systems A_i.) The only addition is the following constraint:

$$\bigwedge_{\substack{a_1 \in \Sigma_1 \setminus \Sigma_2 \\ a_2 \in \Sigma_2 \setminus \Sigma_1}} (\neg \underline{a}_1 \vee \neg \underline{a}_2). \tag{8}$$

needed to prevent two different events $a_1 \in \Sigma_1 \setminus \Sigma_2$ and $a_2 \in \Sigma_2 \setminus \Sigma_1$ to occur at the same time. It is clear that our approach meets the requirement that the combinatorial explosion due to the product construction is not present when generating the encoding. Rather, it is deferred to search time, where it can be tackled by mean of symbolic techniques.

3.2 Encoding LTL specifications of Timed Automata

The existential BMC problem $M \models_k \mathbf{E}f$, read "there exist an execution path of M of bound k satisfying the LTL property f", is equivalent to the satisfiability problem

$$\bigwedge_{x\in X_1} (\neg(x=z) \leftrightarrow (x-z>0)) \tag{9}$$

$$\neg(z'=z) \leftrightarrow (z'-z<0) \tag{10}$$

$$\bigwedge_{x\in X_1} \neg(x'=x) \leftrightarrow (x'-x<0). \tag{11}$$

$$((x=z) \wedge (x'=x) \wedge (z'=z)) \rightarrow (x'=z') \tag{12}$$

$$(\neg(x=z) \wedge (x'=x) \wedge (z'=z)) \rightarrow \neg(x'=z') \tag{13}$$

$$((x=z) \wedge \neg(x'=x) \wedge (z'=z)) \rightarrow \neg(x'=z') \tag{14}$$

$$((x=z) \wedge (x'=x) \wedge \neg(z'=z)) \rightarrow \neg(x'=z') \tag{15}$$

$$((x'=x) \wedge (z'-z<0) \wedge (x-z>0)) \rightarrow (x'-z'>0) \tag{16}$$

$$((z'=z) \wedge \neg(x-z>0) \wedge (x'-x<0)) \rightarrow \neg(x'-z'>0) \tag{17}$$

$$((x-z \bowtie c) \wedge (x'=x) \wedge (z'=z)) \rightarrow (x'-z' \bowtie c) \tag{18}$$

$$(\neg(x-z \bowtie c) \wedge (x'=x) \wedge (z'=z)) \rightarrow \neg(x'-z' \bowtie c) \tag{19}$$

Fig. 5. Formulas for the boolean propagation of mathematical constraints.

for the math-formula $[[M,f]]_k$ defined as $[[M]]_k \wedge [[f]]_k$, where $[[f]]_k$ is $[[f]]_k^0 \vee \bigvee_{l=0}^{k-1}({}_lL_k \wedge {}_{l+1}[[f]]_k^0)$ [CPRS02]. Table 1 describes, in the cases without loopback ($[[f]]_k^i$) and with loopback (${}_l[[f]]_k^i$), the bounded tableau, depending on the structure of the formula. (Without loss of generality we assume that f is in extended negative normal form.) The table encodes the necessary conditions for the existence of a path satisfying the formula. For instance, in the case of an eventuality formula $\mathbf{F}g$, it requires that g must hold in at least one of the steps within in the bound. Notice that, in the case of a globally formula $\mathbf{G}g$, an infinite behaviour is required in order to be able to provide a witness. The encoding of LTL formulae is very similar to the encoding proposed in [BCCZ99], apart from two differences. First, in the specification, constraints over clocks are also possible as atomic propositions, so that $[[M,f]]_k$ is a math-formula. Second, we define loopback as an equivalence between the state vectors at l and k, while in [BCCZ99] a transition from step k to step l is required.

4 Improvements and extensions to the encodings

The encoding described in previous section can be improved and extended in various ways, in order to best exploit the features of the solver. Here we consider some optimizations, and show how they can be implemented in our approach. (Care must be taken since some of them may change the semantics of "next state", and consequently the validity of some LTL properties, in particular those with "**X**" operators.)

Deterministic propagations of real values. We exploit the fact that the truth value of some mathematical constraint may derive deterministically from those of other mathematical constraints. By making such information explicit in the encoding, the SAT solver deterministically propagates the truth values without investigating the mathematical constraint, and may avoid branching on the truth value of mathematical constraints. This is done by adding the formulas reported in Figure 5 to the encoding. The formulas (9), (10) and (11) make explicit the facts that $z' \leq z$, $x \geq z$ and $x' \leq x$, as observed in Section 3.1, and the fact that equalities and strict inequalities

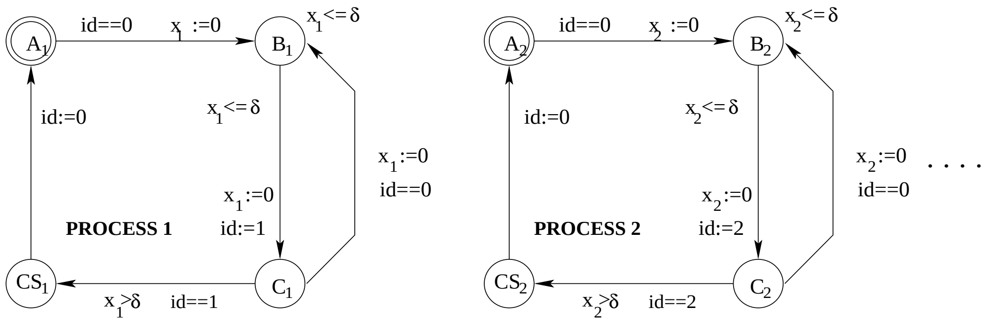

Fig. 6. Two processes in Fischer's mutual exclusion protocol.

are mutually incompatible. We also propagate the positive and negative values of equalities. If so, for every $x \in X_1$ we add the formulas (12), (13), (14), (15), (16) and (17). Moreover, we may propagate the positive and negative values of atomic clock constraints when time does not elapse and clocks are not reset, by adding the formulas (18) and (19).

Exploiting parallelism. The formula (8) imposes a mutual exclusion constraint between the two switches T_1 and T_2, since they are labeled by different events. Consider however that T_1 and T_2 do not interfere with each other, since they belong to different automata. Therefore we may want to release such a constraint, by dropping the formula (8) from the encoding, thus allowing them to shoot in parallel. This amounts to allowing the product system A —though not to forcing it— to collapse both sequences $(T_1 \wedge T^2_{null}) \cdot (T^1_{null} \wedge T_2)$ and $(T^1_{null} \wedge T_2) \cdot (T_1 \wedge T^2_{null})$ into one single transition $(T_1 \wedge T_2)$. This results into more compact formulas, and, more importantly, may significantly shorten the length of the minimum counterexample for a given formula, which is essential in the BMC approach. The resulting encoding of $A_1||A_2$ is exactly the conjunction of the encodings of A_1 and A_2.

Forcing System Activity and Compacting Time elapse. Assume that the system is given by the product of the automata $A_0, ..., A_{N-1}$. The encoding allows situations in which all the systems do nothing, that is, all systems A_i execute a transition T^i_{null}. (This corresponds to a time elapse with $\delta = 0$.) To prevent the search engine from considering this situation, we add the formula:

$$\bigvee_{j=0}^{N-1} \neg T^j_{null}. \tag{20}$$

Furthermore, we avoid two consequent time elapsing transitions to occur by collapsing $s \overset{\delta}{\longmapsto} s$, $s \overset{\delta'}{\longmapsto} s$ into $s \overset{\delta+\delta'}{\longmapsto} s$. To do this we add the constraint:

$$\neg T_\delta \vee \neg T'_\delta. \tag{21}$$

Adding global variables. Our encoding can be straightforwardly extended to handle global variables v on discrete domains. The intended meaning is that a switch T can be subject to a condition $\psi(\underline{v})$ on the variables v's, and can either assign to v a value n or maintain its value; T_δ maintain the value of v, and T_{null} impose no constraints on v. These facts are encoded respectively as

$$T \rightarrow \psi(\underline{v}), \quad T \rightarrow \underline{v}' = n, \quad T \rightarrow (\underline{v}' = \underline{v}), \quad T_\delta \rightarrow (\underline{v}' = \underline{v}), \tag{22}$$

$\underline{v}$ being a boolean representation of v preserving the mutual exclusion of its values. (Here we assume that (20) is part of the encoding, to make sure that at least one transition states the value of the global variable.)

Exploiting symmetries Assume that the system is the product of the automata $A_0, ..., A_{N-1}$. We say that the model checking problem is *symmetric* w.r.t. $A_0, ..., A_{N-1}$ if, for every permutation $\sigma = \{k_0 \rightarrow 0, ..., k_{N-1} \rightarrow N-1\}$ of the automata indexes, the permutation of a solution is a solution for the permuted problem. A sufficient condition for symmetry is that the automata $A_0, ..., A_{N-1}$ are identical and that both the initial conditions and the property to be verified are symmetric. If $A_0, ..., A_{N-1}$ are symmetric, we can simplify the search by imposing that, at step i, one of the processes with index $0 \leq j < i$ is forced to fire a transition. We substitute equation 20 with

$$\bigvee_{j=0}^{min(i,N-1)} \neg T_{null}^{j\ (i)} \tag{23}$$

that is, if $i < N-1$, then we drop the disjuncts $\neg T_{null}^{i+1\ (i)} \vee \ldots \vee \neg T_{null}^{N-1\ (i)}$. Notice that, while equation 20 is replicated "as is" for the different time steps, the equation above changes with the step i, so that the i-th instance constraints the possible transitions that can be fired at step i.

It is easy to see that the new encoding does not lose any interesting models. Indeed, let $[[M, f]]_k^{symm}$ be the formula obtained from $[[M, f]]_k$ by substituting equation 20 with equation 23, for every i. Every solution for $[[M, f]]_k^{symm}$ is trivially also a solution for $[[M, f]]_k$ and, for every solution μ for $[[M, f]]_k$, there exists a solution $\sigma\mu$ for $[[M, f]]_k^{symm}$ which is obtained by applying to μ a permutation σ on the index names of the automata $A_0, ..., A_{N-1}$. It is also clear that a significant amount of search is avoided. The idea is that exponentially-many symmetric executions are collapsed into one. In fact, using $[[M, f]]_k^{symm}$ rather than $[[M, f]]_k$ reduces the space of truth assignments for MathSAT of up to a $2^{N-1}2^{N-2}...2 = 2^{N(N-1)/2}$ factor.

Given the compositional nature of our encoding, the idea can be generalized to the case in which only *subsets* of $\{A_0, ..., A_{N-1}\}$ are symmetric. For instance, if only $A_0, ..., A_{K-1}$ are symmetric, $K < N$, then we replace (20) with

$$\bigvee_{j=0}^{min(i,K-1)} \neg T_{null}^{j\ (i)} \vee \bigvee_{j=K}^{N-1} \neg T_{null}^{j}. \tag{24}$$

5 Related Work

Our work proposes a new way to tackle the verification of timed systems. In this field, several approaches have been proposed. In [HNSY94], a symbolic approach for formally checking whether a system modeled as a product of timed automata meets its requirements is presented. A product is built from components which can communicate each other through synchronization events. The associated tool Kronos, is able to perform both backward and forward exploration of the state space, with a symbolic representation combining DBMs ([Dil89]) and BDDs [BDM$^+$98]. UPPAAL ([LPY95]) is a tool for verification of real-time systems. It is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels and/or shared discrete variables. The description language is a non-deterministic guarded command language with simple data types (e.g. bounded integers, arrays, reals). The model checker is able to check invariants, reachability and some liveness properties by exploring the state space of a system in a symbolic way. To represent the state space, UPPAAL can use Clock Difference Diagrams (CDDs) [LWYP98], or DBMs. Difference Decision Diagrams (DDDs) [MLAH01] are BDD-like data structures to handle boolean formulas over inequalities of the form $x-y<c$ and $x-y\leq c$, and can be used to represent and explore sets of states of a timed system. The associated tool can verify a real time system modeled as a timed guarded command program in a fully symbolic way, by performing reachability analysis and model checking of TCTL formulas using standard fixed-point iteration algorithms. RED [Wan00] is a tool for the verification of real time systems modeled as a set of concurrent processes expressed as timed automata equipped with a clock, discrete variables and pointers. RED is based on the efficient Region Encoding Diagram (RED), a data structure which can be used for fully symbolic model checking of TCTL over timed systems. RED is able to exploit symmetries between processes, and is currently one of the most efficient verification tools in the field.

Our approach differs from the above techniques in several respects. On one side, it is limited to the bounded case. On the other side, it allows for the analysis of specifications expressed in LTL, and is therefore able to express general forms of fairness. Furthermore, our approach is based on (an extension of) propositional satisfiability techniques, that are increasingly accepted as an efficient and complementary alternative to the use of Decision Diagrams, for their limited memory requirements. Finally, we use specialized solvers that are able to deal efficiently with different classes of mathematical constraints: equalities (by substitution), binary inequalities (by Bellman-Ford), and arbitrary inequalities (by Symplex).

6 Some preliminary empirical results

In this section, we report some preliminary experimental results, where our approach is used to tackle a case study, and compared with the systems described in previous section. The evaluation is carried out on Fischer's mutual exclusion protocol [Lam87]. N identical processes (described in Figure 6) try to gain access to a critical section CS. The synchronization relies on a global variable id where each process P_i writes its identifier i when entering the waiting state C. Other processes P_j can enter the waiting state C in the same way. If after a certain delay δ still $id=i$, then P_i can enter the critical section CS, from which it eventually exits resetting id to 0; otherwise, as soon as id is reset, it can go back to B, from where it can subsequently retry. This protocol is interesting for many reasons: it is very simple to describe and understand, it contains several elements of interest (e.g. time advance, synchronization, mutual exclusions), it is scalable, so that we can increase its complexity at will by increasing N, and it is symmetric. Despite its simplicity, investigating non-obvious properties is non trivial even with small N's.

	MathSAT		MathSAT,Sym		DDD		Uppal		Kronos		Red		Red,Sym	
N	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size
3	0.05	2.9	0.04	2.9	0.11	106	0.01	1.7	0.01	0.8	0.23	2.0	0.19	2.0
4	0.09	3.0	0.08	3.0	0.14	106	0.02	1.9	0.02	2.2	1.00	2.1	0.70	2.1
5	0.20	3.2	0.16	3.2	0.24	106	0.21	1.9	0.09	19	3.70	2.2	2.00	2.4
6	0.60	3.7	0.23	3.7	0.47	106	3.44	6.7	0.39	236	12.00	2.7	5.20	3.1
7	3.20	4.2	0.36	4.2	1.30	106	153	54		-	38	4.0	12	4.7
8	29	4.9	0.52	4.9	3.96	106	-				121	7.6	26	7.8
9	343	5.9	0.75	5.9	14	106					416	16.6	49	13.3
10	3331	6.5	1.01	6.5	62	106					1382	39	90	23
11	-		1.39	7.0	691	106					-		157	38
12			1.89	7.5		-							266	63
13			2.44	8.2									439	100
14			3.24	8.9									709	155
15			4.11	9.7									1118	225
16			5.10	10.7									1717	342
17			6.30	11.7									2582	492
18			8.00	12.9									-	
19			9.50	14.2										

Table 2. Fischer's protocol – reachability (time in seconds, size in MB).

To analyze our example, for increasing values of N and increasing values of the bound $k = 1, 2, 3, \ldots$, we encoded the given problems into math-formulas and we tackled the resulting math-formulas with our implementation of MathSAT. The experiments were run under `Linux RedHat 7.1` on a 4-processor `PentiumIII` 700MHz machine with more than 6.5GB RAM. The time limit was fixed to 1 hour (only one processor is allowed for each run), while the memory was limited to 1GB for each run. All the math-formulas investigated here are available at `http://www.science.unitn.it/~rseba/Mathsat.html`, together with our implementation of MathSAT.

As a first example, we have considered the *reachability* problem "Is there a state in which all the processes are in the wait state C", formalized as:

$$M \models_k \mathbf{EF} \bigwedge_i P_i.C$$

For every N, the math-formulas are unsatisfiable for $k \leq N$ and satisfiable for $k > N$. In fact, the shortest solution path —all processes pass from A to B, and then from B to C one at a time— has length $N + 1$.

We have compared MathSAT —without and with the symmetry-exploiting encoding described in Section 4— with the DDD package, with Uppaal (version 3.2.4), with Kronos (version 2.4.4), and with Red (version 3.1) —without and with its own symmetry exploitation techniques. The results are collected in Table 2. MathSATwas run with the default splitting heuristic, i.e. the one of Satz [LA97]. Times are expressed in seconds and size in megabytes. "-" denotes that the system reached the time or size limit. For MathSAT, the reported times are *sums* of the times needed to analyze the (unsatisfiable) instances with bound $k = 1, \ldots, N$ and to the (satisfiable) instance $k = N + 1$.

Although the property is extremely simple, we see that the complexity of the problem blows up quickly with N. Without using the symmetry-exploiting encoding,

	MathSAT					MathSAT with Boehm heuristic				
$k \backslash N$	2	3	4	5	6	2	3	4	5	6
2	0.01	0.01	0.01	0.01	0.02	0.01	0.01	0.01	0.01	0.02
3	0.01	0.02	0.01	0.01	0.03	0.01	0.01	0.02	0.03	0.04
4	0.01	0.02	0.02	0.02	0.04	0.01	0.02	0.04	0.07	0.17
5	0.02	0.03	0.05	0.09	0.18	0.01	0.03	0.09	0.3	1.16
6	0.03	0.1	0.21	0.54	1.35	0.02	0.07	0.31	1.52	7.74
7	0.04	0.26	0.97	3.2	9.83	0.02	0.18	1.19	7.14	45
8		0.65	4.8	19.72	70.7		0.06	4.7	33.5	242
9			5.55	112.17	478			0.61	165.9	1348
10				303.17	3086				9.92	7824
11					5002					252
Σ	0.12	1.08	11.62	438.93	8648.15	0.07	0.37	6.98	218.4	9720.13

Table 3. Fischer's Protocol – fairness (time in seconds).

MathSAT is better than Uppaal and Kronos, slightly worse than Red, worse than DDD and much worse than Red with its symmetry-exploiting technique. With the symmetry-exploiting encoding, MathSAT runs dramatically better than DDD, Uppaal and Kronos, which have no symmetry-exploiting technique, and even better than Red with symmetry-exploiting technique. As memory consumption is concerned, we see that MathSAT behaves much better than all the other systems.

As a second example, we have considered the following fairness property: "if the i-th process gets infinitely often in B, then it accesses infinitely often in the critical section CS". We look for a counterexample, i.e. we tackle the following problem:

$$M \models_k \mathbf{E}\neg(\mathbf{GF}P_i.B \rightarrow \mathbf{GF}P_i.CS).$$

For every N, the math-formulas are unsatisfiable for $k \leq N+4$ and satisfiable for $k > N+4$. In fact, the shortest solution path containing a loop has length $N+5$: all processes pass from A to B, and then from B to C one at a time; time elapses of a quantity greater than δ; then the last process arrived in C passes alone into CS and then into A; finally they all pass into B again.

To the best of our knowledge, there is no direct way to encode this problem in DDD, Uppaal, Kronos and Red. With MathSAT instead, we can encode it by forcing a loop from step k to each step $l < k$. (We report here only the "interesting" case where $l = 1$). Notice, that the property above is not symmetric, so that this time we cannot use the symmetry-exploiting encoding. The results are collected in Table 3. To emphasize the effects of different splitting heuristics, we run MathSAT not only with the Satz heuristics (left), but also with Boehm's (right). Thus, we notice that Satz heuristic gives better results with *unsatisfiable* instances, and worse results for satisfiable ones (last entries of each column).

The analysis is clearly preliminary in several respects. First, we are comparing a SAT-based *bounded* model checker with fixpoint-based *unbounded* model checkers, which were not specially conceived for a bounded search. However, as we are not aware of any other bounded model checker for timed system currently available, this is not a matter of choice. Second, the analysis can be biased by the fact that we are considering only one case study. Notice however that the case study was not tuned toward SAT-based techniques. (It would indeed be quite easy to find a problem where the data structures for the representation of regions blow up, simply because of the inheritance of the properties of BDDs.) On the contrary, we are tackling an asynchronous system, while so far SAT-based BMC methods have proved particularly effective for synchronous systems. It would be interesting to extend the comparison on synchronous applications such as real-time embedded controllers. It is also important to notice that we are comparing mature approaches, that have been optimized over the years, with a new encoding technique and solver. Having said this, the above results show that our approach as extremely promising. First, as CPU times are concerned, our approach can be comparable with other well-established ones. Second, MATHSAT has much more limited memory requirements. Third, even a very simple exploitation of symmetries can significantly reduce the verification times (and we believe that there are several directions of improvements). Fourth, although bounded, our technique can be used to falsify properties that can not be directly handled by the other approaches.

7 Conclusions and Future Work

In this paper, we have presented a new approach for symbolic model checking of timed systems. We have shown how to encode a BMC problem for timed systems into that of deciding the satisfiability of boolean combinations of boolean variables and atomic linear (in)equalities, which we can solve efficiently by the MATHSAT solver. The approach is fully symbolic, and is not limited to simple reachability, but allows for (Bounded) Model Checking of LTL formulas. Furthermore, the solver can be rather efficient in terms of run-times, even with its limited memory requirements. As BMC in the propositional case, our new technique is intended to be complementary rather than alternative to the current ones, in particular because of its ability of finding (counter)examples, of its expressiveness and of its reduced memory requirements.

In the future, we plan to extend and improve our work along the following directions. First, we want to improve and test new kinds of encodings. In particular, we will investigate alternative representations of locations, events and transitions; we will look for new propagation axioms and invariants to prune search, in particular taking into account more powerful forms of symmetry reduction. Then, we will investigate the customization of MATHSAT for encoded timed automata, by defining splitting heuristics that take into account the different semantics of the variables [GMS98,Sht00] and new mechanisms for propagating and exploiting equalities between real values. Finally, we want to perform an extensive experimental evaluation, also on synchronous domains, to identify the bottlenecks and the strengths of the approach.

References

[ABC+02] G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In *Proc. CADE'2002.*, 2002. To appear. Available at `http://www.dit.unitn.it/~rseba/publist.html`.

[Alu99] R. Alur. Timed Automata. In *Proc. 11th International Computer Aided Verification Conference*, pages 8–22, 1999.

[BCC+99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking Using SAT Procedures instead of BDDs. In *Proc. DAC'99*, pages 317–320, 1999.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Proc. TACAS'99*, pages 193–207, 1999.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In A. J. Hu and M. Y. Vardi, editors, *Proc. 10th International Conference on Computer Aided Verification, Vancouver, Canada*, volume 1427 of *LNCS*, pages 546–550. Springer-Verlag, 1998.

[CFG+01a] F. Copty, L. Fix, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi. Benefits of Bounded Model Checking at an Industrial Setting. In *Proc. CAV'2001*, LNCS. Springer, 2001.

[CFG+01b] F. Copty, L. Fix, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi. Benefits of bounded model checking at an industrial setting. In *Proceedings of CAV 2001*, pages 436–453, 2001.

[CPRS02] A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani. Improving the Encoding of LTL Model Checking into SAT. In *Proc. 3rd International Workshop on Verification, Model Checking, and Abstract Interpretation*, number 2294 in Lecture Notes in Computer Science. Springer Verlag, 2002.

[Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, June 1989.

[DLL62] M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with kronos. In *Proc. 16th IEEE Real-Time Systems Symposium*, pages 66–75, 1995.

[Eme90] E.A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publisher B.V., 1990.

[GMS98] E. Giunchiglia, A. Massarotto, and R. Sebastiani. Act, and the Rest Will Follow: Exploiting Determinism in Planning as Satisfiability. In *Proc. AAAI'98*, pages 948–953, 1998.

[HNSY94] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[LA97] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 366–371, 1997.

[Lam87] L. Lamport. A Fast Mutual-exclusion Algorithm. *ACM Transactions on Computer Systems*, 5(1), 1987.

[LPY95] K. G. Larsen, P. Pettersson, and W. Yi. Model-Checking for Real-Time Systems. In *Fundamentals of Computation Theory*, pages 62–88, 1995.

[LWYP98] K.G. Larsen, C. Weise, W. Yi, and J. Pearson. Clock difference diagrams. Technical Report 98/99, DoCS, Uppsala University, Sweden, 1998.

[MLAH01] J. Moeller, J. Lichtenberg, H. Andersen, and H. Hulgaard. Fully Symbolic Model Checking of Timed Systems using Difference Decision Diagrams. In *Electronic Notes in Theoretical Computer Science*, volume 23. Elsevier Science, 2001.

[Seb01] R. Sebastiani. Integrating SAT Solvers with Math Reasoners: Foundations and Basic Algorithms. Technical Report 0111-22, ITC-IRST, November 2001. Available at `http://www.science.unitn.it/~rseba/publist.html`.

[Sht00] Ofer Shtrichmann. Tuning SAT Checkers for Bounded Model Checking. In *Proc. CAV'2000*, volume 1855 of *LNCS*. Springer, 2000.

[Wan00] Farn Wang. Efficient data structure for fully symbolic verification of real-time software systems. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 157–171, 2000.