Appeared at CAV'03

Unbounded, Fully Symbolic Model Checking of Timed Automata using Boolean Methods

Sanjit A. Seshia and Randal E. Bryant

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA
{Sanjit.Seshia, Randy.Bryant}@cs.cmu.edu

Abstract. We present a new approach to unbounded, fully symbolic model checking of timed automata that is based on an efficient translation of quantified separation logic to quantified Boolean logic. Our technique preserves the interpretation of clocks over the reals and can check any property in timed computation tree logic. The core operations of eliminating quantifiers over real variables and deciding the validity of separation logic formulas are respectively translated to eliminating quantifiers on Boolean variables and checking Boolean satisfiability (SAT). We can thus leverage well-known techniques for Boolean formulas, including Binary Decision Diagrams (BDDs) and recent advances in SAT and SAT-based quantifier elimination. We present preliminary empirical results for a BDD-based implementation of our method.

1 Introduction

Timed automata [2] have proved to be a useful formalism for modeling real-time systems. A timed automaton is a generalization of a finite automaton with a set of real-valued clock variables. The state space of a timed automaton thus has a finite component (over Boolean state variables) and an infinite component (over clock variables). Several model checking techniques for timed automata have been proposed over the past decade. These can be classified, on the one hand, as being either *symbolic* or *fully symbolic*, and on the other, as being *bounded* or *unbounded*. Symbolic techniques use a symbolic representation for the infinite component of the state space, and either symbolic or explicit representations for the finite component. In contrast, fully symbolic methods employ a single symbolic representation for both finite and infinite components of the state space. Bounded model checking techniques work by unfolding the transition relation d times, finding counterexamples of length up to d, if they exist. As in the untimed case, these methods suffer from the limitation that, unless a bound on the length of counterexamples is known, they cannot verify the property of interest. Unbounded methods, on the other hand, can produce a guarantee of correctness.

The theoretical foundation for unbounded, fully symbolic model checking of timed automata was laid by Henzinger et al. [9]. The characteristic function of a set of states is a formula in *Separation Logic*, a quantifier-free fragment of first-order logic. Formulas in Separation Logic (SL) are Boolean combinations of

Boolean variables and predicates of the form $x_i \bowtie x_j + c$ where $\bowtie \in \{>, \geq\}$, x_i and x_j are real-valued variables, and c is a constant. *Quantified Separation Logic* (QSL) is an extension of SL with quantifiers over real and Boolean variables. The most important model checking operations involve deciding the validity of SL formulas and eliminating quantifiers on real variables from QSL formulas.

In this paper, we present the first approach to unbounded, fully symbolic model checking of timed automata that is based on a Boolean encoding of SL formulas and that preserves the interpretation of clocks over the reals. Unlike many other fully symbolic techniques, our method can be used to model check any property in Timed Computation Tree Logic (TCTL), a generalization of CTL. The main theoretical contribution of this paper is a new technique for transforming the problem of eliminating quantifiers on real variables to one of eliminating quantifiers on Boolean variables. In some cases, we can avoid introducing Boolean quantification altogether. These techniques, in conjunction with previous work on deciding SL formulas via a translation to Boolean satisfiability (SAT) [16], allow us to leverage well-known techniques for manipulating quantified Boolean formulas, including Binary Decision Diagrams (BDDs) and recent work on SAT and SAT-based quantifier elimination [11].

Related Work. The work that is most closely related to ours is the approach based on representing SL formulas using Difference Decision Diagrams (DDDs) [12]. A DDD is a BDD-like data structure, where the node labels are generalized to be separation predicates rather than just Boolean variables, with the ordering of predicates induced by an ordering of clock variables. This predicate ordering permits the use of local reduction operations, such as eliminating inconsistent combinations of two predicates that involve the same pair of clock variables. Deciding a SL formula represented as a DDD is done by eliminating all inconsistent paths in the DDD. This is done by enumerating all paths in the DDD and checking the satisfiability of the conjunction of predicates on each path using a constraint solver based on the Bellman-Ford shortest path algorithm. Note that each path can be viewed as a disjunct in the Disjunctive Normal Form (DNF) representation of the DDD, and in the worst case there can be exponentially many calls to the constraint solver. Quantifier elimination is performed by the Fourier-Motzkin technique [8], which also requires enumerating all possible paths. In contrast, our Boolean encoding method is general in that any representation of Boolean functions may be used. Our decision procedure and quantifier elimination scheme use a direct translation to SAT and Boolean quantification respectively, avoiding the need to explicitly enumerate each DNF term. In theory, the use of DDDs permits unbounded, fully symbolic model checking of TCTL; however, the DDD-based model checker [12] can only check reachability properties (these can express safety and bounded-liveness properties [1]).

UPPAAL2K and KRONOS are unbounded, symbolic model checkers that explicitly enumerate the discrete component of the state space. KRONOS uses Difference Bound Matrices (DBMs) as the symbolic representation [18] of the infinite component. UPPAAL2K uses, in addition, Clock Difference Diagrams (CDDs) to symbolically represent unions of convex clock regions [4]. In a CDD, a node is labeled by the difference of a pair of clock variables, and each outgoing edge from a node is labeled with an interval bounding that difference. While KRONOS can check arbitrary TCTL formulas, UPPAAL2K is limited to checking reachability properties and very restricted liveness properties such as the CTL formula $\mathbf{AF}p$.

RED is an unbounded, fully symbolic model checker based on a data structure called the Clock Restriction Diagram (CRD) [17]. The CRD is similar to a CDD, labeling each node with the difference between two clock variables. However, each outgoing edge from a node is labeled with an upper bound, instead of an interval. RED represents separation formulas by a combined BDD-CRD structure, and can model check TCTL formulas.

A fully symbolic version of KRONOS using BDDs has been developed by interpreting clock variables over integers [6]; however, this approach is restricted to checking reachability for the subclass of closed timed automata[1], and the encoding blows up with the size of the integer constants. Rabbit [5] is a tool based on this approach that additionally exploits compositional methods to find good BDD variable orderings. In comparison, our technique applies to all timed automata and its efficiency is far less sensitive to the size of constants. Also, the variable ordering methods used in Rabbit could be used in a BDD-based implementation of our technique.

Many fully symbolic, but bounded model checking methods based on SAT have been developed recently (e.g., [3, 13]). These algorithms cannot be directly extended to perform unbounded model checking.

The rest of the paper is organized as follows. We define notation and present background material in Section 2. We describe our new contributions in Sections 3 and 4. We conclude in Section 5 with experimental results and ongoing work. Details including proofs of theorems stated in the paper can be found in our technical report [14].

2 Background

We begin with a brief presentation of background material, based on papers by Alur [2] and Henzinger et al. [9]. We refer the reader to these papers for details.

2.1 Separation Logic

Separation logic (SL), also known as *difference logic*, is a quantifier-free fragment of first-order logic. A formula ϕ in separation logic is a Boolean combination of Boolean variables and *separation predicates* (also known as *difference bound constraints*) involving real-valued variables, as given by the following grammar:

$$\phi ::= \mathbf{true} \mid \mathbf{false} \mid b \mid \neg\phi \mid \phi \wedge \phi \mid x_i \geq x_j + c \mid x_i > x_j + c$$

We use a special variable x_0 to denote the constant 0; this allows us to express bounds of the form $x \geq c$. We will however use both $x \bowtie c$ and $x \bowtie x_0 + c$, where

[1] Clock constraints in a closed timed automaton do not contain strict inequalities.

$\bowtie \in \{>, \geq\}$, as suits the context. We will denote Boolean variables by $b, b_1, b_2, \ldots$, real variables by $x, x_1, x_2, \ldots$, and SL formulas by $\phi, \phi_1, \phi_2, \ldots$. Note that the relations $>$ and $\geq$ suffice to represent equalities and other inequalities.

Characteristic functions of sets of states of timed automata are SL formulas. Deciding the satisfiability of a SL formula is NP-complete [9].

Quantified Separation Logic. Separation logic can be generalized by the addition of quantifiers over both Boolean and real variables. This yields *quantified separation logic* (QSL). The satisfiability problem for QSL is PSPACE-complete [10]. We will denote QSL formulas by $\omega, \omega_1, \omega_2, \ldots$.

2.2 Timed Automata

A *timed automaton* $\mathcal{T}$ is a tuple $\langle \mathcal{L}, \mathcal{L}_0, \Sigma, \mathcal{X}, \mathcal{I}, \mathcal{E} \rangle$, where $\mathcal{L}$ is a finite set of locations, $\mathcal{L}_0 \subseteq \mathcal{L}$ is a finite set of initial locations, Σ is a finite set of labels used for product construction, $\mathcal{X}$ is a finite set of non-negative real-valued clock variables, $\mathcal{I}$ is a function mapping a location to a SL formula (called a *location invariant*), and $\mathcal{E}$ is the transition relation, a subset of $\mathcal{L} \times \Psi \times \mathcal{R} \times \Sigma \times \mathcal{L}$, where Ψ is a set of SL formulas that form enabling *guard* conditions for each transition, and $\mathcal{R}$ is a set of *clock reset assignments.* A location invariant is the condition under which the system can stay in that location. A clock reset assignment is of the form $x_i := x_0 + c$ or $x_i := x_j$, where $x_i, x_j \in \mathcal{X}$ and c is a non-negative rational constant,[2] and indicates that the clock variable on the left-hand side of the assignment is reset to the value of the expression on the right-hand side. We will denote guards by $\psi, \psi_1, \ldots$. The invariant $\mathcal{I}_\mathcal{T}$ for the timed automaton $\mathcal{T}$ is defined as $\mathcal{I}_\mathcal{T} = \bigwedge_{l \in \mathcal{L}} [enc(l) \implies \mathcal{I}(l)]$, where $enc(l)$ denotes the Boolean encoding of location l. We will also denote a transition $t \in \mathcal{E}$ as $\psi \implies A$, where ψ is a guard condition over both Boolean state variables (used to encode locations) and clock variables of the system, and A is a set of assignments to clock and Boolean state variables.

Two timed automata are composed by synchronizing over common labels. We refer the reader to Alur's paper [2] for a formal definition of product construction. Note that in contrast to the definition of timed automata given by Alur [2], we allow location invariants and guards to be arbitrary SL formulas, rather than simply conjunctions over separation predicates involving clock variables.

2.3 Fully Symbolic Model Checking

Properties of timed automata can be expressed in a generalization of the μ calculus called the *timed* μ ($\mathbf{T}\mu$) calculus. Henzinger et al. [9] showed that the $\mathbf{T}\mu$ calculus can express TCTL, the dense-real-time version of CTL. They have given a fully symbolic model checking algorithm that verifies if a timed automaton $\mathcal{T}$ satisfies a specification given as a $\mathbf{T}\mu$ formula φ. The algorithm terminates, generating a SL formula $|\varphi|$, such that, if $\mathcal{T}$ is *non-zeno* (i.e., time can diverge

[2] The assignment $x_i := c$ is represented as $x_i := x_0 + c$. Wherever we use x_i to denote a clock variable, $i > 0$.

from any state), then $|\varphi|$ is equivalent to $\mathcal{I}_{\mathcal{T}}$ iff $\mathcal{T}$ satisfies φ. For lack of space, we omit details of the $\mathbf{T}\mu$ calculus, TCTL, and the model checking algorithm; these can be found in our technical report [14] and in the original paper [9].

Our work is based on Henzinger et al.'s model checking algorithm. It performs backward exploration of the state space and relies on three fundamental operations:

1. **Time Elapse:** $\phi_1 \rightsquigarrow \phi_2$ denotes the set of all states that can reach the state set ϕ_2 by allowing time to elapse, while staying in state set ϕ_1 at all times in between. Formally,

$$\phi_1 \rightsquigarrow \phi_2 \doteq \exists\delta\{\delta \geq x_0 \wedge \phi_2 + \delta \wedge \forall\epsilon[x_0 \leq \epsilon \leq \delta \implies \phi_1 + \epsilon]\} \quad (1)$$

where $\phi + \delta$ denotes the formula obtained by adding δ to all clock variables occurring in ϕ, computed as $\phi[x_i + \delta/x_i, 1 \leq i \leq n]$, where $x_1, x_2, \ldots, x_n$ are the clock variables in ϕ_i (i.e., not including the zero variable x_0).
Consider the formula on the right hand side of Equation 1. This formula is not in QSL, since it includes expressions that are the sum of two real variables (e.g., $x + \delta$). However, it can be transformed to a QSL formula, by using instead of δ and ϵ, variables $\overline{\delta}$ and $\overline{\epsilon}$ that represent their negations:

$$\exists\overline{\delta}\{\overline{\delta} \leq x_0 \wedge \phi_2 + (-\overline{\delta}) \wedge \forall\overline{\epsilon}[\overline{\delta} \leq \overline{\epsilon} \leq x_0 \implies \phi_1 + (-\overline{\epsilon})]\} \quad (2)$$

Formula 2 is expressible in QSL, since the substitution $\phi[x_i + (-\overline{\delta})/x_i, 1 \leq i \leq n]$ can be computed as $\phi[\overline{\delta}/x_0]$.[3] This yields,

$$\exists\overline{\delta}\{\overline{\delta} \leq x_0 \wedge \phi_2[\overline{\delta}/x_0] \wedge \forall\overline{\epsilon}(\overline{\delta} \leq \overline{\epsilon} \leq x_0 \implies \phi_1[\overline{\epsilon}/x_0])\} \quad (3)$$

Finally, we can rewrite Formula 3 purely in terms of existential quantifiers:

$$\exists\overline{\delta}\{\overline{\delta} \leq x_0 \wedge \phi_2[\overline{\delta}/x_0] \wedge \neg\exists\overline{\epsilon}(\overline{\epsilon} \leq x_0 \wedge \overline{\delta} \leq \overline{\epsilon} \wedge \neg\phi_1[\overline{\epsilon}/x_0])\} \quad (4)$$

A procedure for performing the time elapse operation therefore requires one for eliminating (existential) quantifiers over real variables from a SL formula.
2. **Assignment:** $\phi[A]$, where A is a set of assignments, denotes the formula obtained by simultaneously substituting in ϕ the right hand side of each assignment in A for the left hand side. Formally, if A is the list $b_1 := \phi_1, \ldots, b_k := \phi_k, x_1 := x_{j_1} + c_1, \ldots, x_n := x_{j_n} + c_n$, where each b_i is a Boolean variable, each x_j is a clock variable, and for each x_{j_l}, $j_l = 0$ or $c_l = 0$, then

$$\phi[A] = \phi[\phi_1/b_1, \ldots, \phi_k/b_k, x_{j_1} + c_1/x_1, \ldots, x_{j_n} + c_n/x_n]$$

Assignments are thus performed via substitutions of variables.
3. **Checking Termination:** The termination condition of the fixpoint iteration in the model checking algorithm checks if one set of states, ϕ_{new}, is contained in another, ϕ_{old}. This check is performed by deciding the validity of the SL formula $\phi \doteq \phi_{new} \implies \phi_{old}$ (or equivalently, the satisfiability of $\neg\phi$).

[3] Note that substituting x_0 by $\overline{\delta}$ or $\overline{\epsilon}$ can be viewed as shifting the zero reference point to a more negative value, thus increasing the value of any clock variable relative to zero (as observed, e.g., in [3, 12]).

3 Model Checking Operations using Boolean Encoding

We now show how to implement the fundamental model checking operations using a Boolean encoding of separation predicates. We first describe how our encoding allows us to replace quantification of real variables by quantification of Boolean variables. This builds on previous work on deciding a SL formula by transformation to a Boolean formula [16]. We then show how we represent SL formulas as Boolean formulas, allowing the model checking operations to be implemented as operations in Quantified Boolean Logic (QBL), and enabling the use of QBL packages, e.g., a BDD package.

In the remainder of this section, we will use ϕ to denote a SL formula over real variables $x_1, x_2, \dots, x_n$, and Boolean variables $b_1, b_2, \dots, b_k$. Also, let $\bowtie$, $\bowtie_1$, $\bowtie_2 \in \{>, \geq\}$.

3.1 From Real Quantification to Boolean Quantification

Consider the QSL formula $\omega_a \doteq \exists x_a.\phi$, where $a \in [1..n]$.

We transform ω_a to an equivalent QSL formula ω_{bool} with quantifiers over only Boolean variables in the following three steps:

1. *Encode separation predicates:*
   Consider each separation predicate in ϕ of the form $x_i \bowtie x_j + c$ where either $i = a$ or $j = a$. For each such predicate, we generate a corresponding Boolean variable $e_{i,j}^{\bowtie,c}$. Separation predicates that are negations of each other are represented by Boolean literals (true or complemented variables) that are negations of each other; however, for ease of presentation, we will extend the naming convention for Boolean variables to Boolean literals, writing $e_{j,i}^{>,-c}$ for the negation of $e_{i,j}^{\geq,c}$.
   Let the added Boolean variables be $e_{i_1,a}^{\bowtie_{i_1},c_{i_1}}, e_{i_2,a}^{\bowtie_{i_2},c_{i_2}}, \dots, e_{i_m,a}^{\bowtie_{i_m},c_{i_m}}$ for the upper bounds on x_a, and $e_{a,j_1}^{\bowtie_{j_1},c_{j_1}}, e_{a,j_2}^{\bowtie_{j_2},c_{j_2}}, \dots, e_{a,j_{m'}}^{\bowtie_{j_{m'}},c_{j_{m'}}}$ for the lower bounds on it.
   We replace each predicate $x_a \bowtie x_j + c$ (or $x_i \bowtie x_a + c$) in ϕ by the corresponding Boolean variable $e_{a,j}^{\bowtie,c}$ (or $e_{i,a}^{\bowtie,c}$). Let the resulting SL formula be ϕ_{bool}^a.
2. *Add transitivity constraints:*
   Notice that there can be assignments to the $e_{i,a}^{\bowtie,c}$ and $e_{a,j}^{\bowtie,c}$ variables that have no corresponding assignment to the real valued variables. To disallow such assignments, we place constraints on these added Boolean variables. Each constraint is generated from two Boolean literals that encode predicates containing x_a. We will refer to these constraints as *transitivity constraints for x_a*.
   A transitivity constraint for x_a has one of the following types:
   (a) $e_{i,a}^{\bowtie_1,c_1} \wedge e_{a,j}^{\bowtie_2,c_2} \implies (x_i \bowtie x_j + c_1 + c_2)$,
   where if $\bowtie_1$ and $\bowtie_2$ are both $\geq$, then $\bowtie$ is $\geq$, otherwise $\bowtie$ is $>$.
   (b) $e_{i,j}^{\bowtie_1,c_1} \implies e_{i,j}^{\bowtie_2,c_2}$, where $c_1 > c_2$ and either $i = a$ or $j = a$.
   (c) $e_{i,j}^{>,c} \implies e_{i,j}^{\geq,c}$, where either $i = a$ or $j = a$.

Note that a constraint of type (a) involves a separation predicate ($x_i \bowtie x_j + c_1 + c_2$). This predicate might not be present in the original formula ϕ.[4] After generating all transitivity constraints for x_a, we conjoin them to get the SL formula ϕ^a_{cons}.

3. Finally, generate the QSL formula ω_{bool} given below:

$$\exists e^{\bowtie_{i_1},c_{i_1}}_{i_1,a}, e^{\bowtie_{i_2},c_{i_2}}_{i_2,a}, \ldots, e^{\bowtie_{i_m},c_{i_m}}_{i_m,a}. \exists e^{\bowtie_{j_1},c_{j_1}}_{a,j_1}, e^{\bowtie_{j_2},c_{j_2}}_{a,j_2}, \ldots, e^{\bowtie_{j_{m'}},c_{j_{m'}}}_{a,j_{m'}}.[\phi^a_{cons} \wedge \phi^a_{bool}]$$

We formalize the correctness of this transformation in the following theorem.

Theorem 1. *ω_a and ω_{bool} are equivalent.*

Example 1. Let $\omega_a = \exists x_a.\phi$ where $\phi = x_a \leq x_0 \wedge x_1 \geq x_a \wedge x_2 \leq x_a$. Then, $\phi^a_{bool} = e^{\geq,0}_{0,a} \wedge e^{\geq,0}_{1,a} \wedge e^{\geq,0}_{a,2}$. ϕ^a_{cons} is the conjunction of the following constraints:

1. $e^{\geq,0}_{0,a} \wedge e^{\geq,0}_{a,2} \implies x_0 \geq x_2$
2. $e^{\geq,0}_{1,a} \wedge e^{\geq,0}_{a,2} \implies x_1 \geq x_2$

Then, $\omega_{bool} = \exists e^{\geq,0}_{0,a}, e^{\geq,0}_{1,a}, e^{\geq,0}_{a,2}.[\phi^a_{cons} \wedge \phi^a_{bool}]$ evaluates to $x_0 \geq x_2 \wedge x_1 \geq x_2$. □

The quantifier transformation procedure described here works even when ϕ is replaced by a QSL formula with quantifiers only over Boolean variables.

3.2 Deciding SL formulas

Suppose we want to decide the satisfiability of ϕ. Note that ϕ is satisfiable iff the QSL formula $\omega_{1..n} = \exists x_1, x_2, \ldots, x_n.\phi$ is.

Using Theorem 1, we can transform $\omega_{1..n}$ to an equivalent QSL formula ω_{bool} with existential quantifiers only over Boolean variables encoding all separation predicates. As ω_{bool} is a QBL formula with only existential quantifiers, we can simply discard the quantifiers and use a Boolean satisfiability checker to decide the resulting Boolean formula.

Note that the transformation described above can be viewed as one way to implement the procedure of Strichman et al. [16].

3.3 Representing SL Formulas as Boolean Formulas

In our presentation up to this point, we have not used any specific representation of SL formulas. In practice, we encode a SL formula ϕ as a Boolean formula β. The encoding is performed as follows. Consider each separation predicate $x_i \bowtie x_j + c$ in ϕ. As in Section 3.1 earlier, we introduce a Boolean variable $e^{\bowtie,c}_{i,j}$ for $x_i \bowtie x_j + c$, only this time we do it for every single separation predicate. Also as before, separation predicates that are negations of each other are represented by Boolean literals that are negations of each other. We then replace each separation predicate in ϕ by its corresponding Boolean literal. The resulting Boolean formula is β.

[4] This addition is analogous to the "tightening" step performed in difference-bound matrix based algorithms

Clearly, β, by itself, stores insufficient information for generating transitivity constraints. Therefore, we also store the 1-1 mapping of separation predicates to the Boolean literals that encode them. However, this mapping is used only lazily, i.e., when generating transitivity constraints during quantification and in deciding SL formulas.

Substitution. Given the representation described above, we can implement substitution of a clock variable as follows. For a clock variable x_i, we perform the substitution $[x_i \leftarrow x_k + d]$ (where $k = 0$ or $d = 0$), by replacing all Boolean variables of the form $e_{i,j}^{\bowtie,c}$ and $e_{j,i}^{\bowtie',c'}$, for all j, by variables $e_{k,j}^{\bowtie,c-d}$ and $e_{j,k}^{\bowtie',c'+d}$ respectively, creating fresh replacement variables if necessary. Substitution of a Boolean state variable by the Boolean encoding of a separation formula is done by Boolean function composition.

4 Optimizations

The methods presented in Section 3 can be optimized in a few ways. First, we can be more selective in deciding when to add transitivity constraints. Second, we can compute the time elapse operator more efficiently by a method that does not explicitly introduce the bound real variable $\overline{\epsilon}$, and hence does not introduce new quantifiers over Boolean variables. The final optimization concerns eliminating paths in a BDD representation that violate transitivity constraints. As is well-known, the size of a BDD is very sensitive to the number and ordering of BDD variables. In the case of model checking timed automata, new Boolean variables are created as the model checking proceeds, while generating transitivity constraints, and while performing substitutions of clock variables. This has the potential to add several BDD variables on each iteration. While all three techniques presented in this section help in reducing the number of BDD variables, only the last technique is specialized for BDDs.

4.1 Determining if Bounds are Conjoined

Suppose ϕ is a SL formula with Boolean encoding β, and we wish to eliminate the quantifier in $\exists x_a.\phi$. As described in Section 3.1, a transitivity constraint for x_a involves two Boolean literals that encode separation predicates involving x_a. For a syntactic representation of β, as the number of constraints grows, so does the size of $[\beta^a_{cons} \wedge \beta^a_{bool}]$, the Boolean encoding of $[\phi^a_{cons} \wedge \phi^a_{bool}]$. Further, new separation predicates can be added when a transitivity constraint is generated from an upper bound and a lower bound on x_a. For a BDD-based implementation, this corresponds to the addition of a new BDD variable. We would therefore like to avoid adding transitivity constraints wherever possible.

In fact, we only need to add a constraint involving an upper bound literal and a lower bound literal if they are conjoined in a minimized DNF representation of β.[5] From a geometric viewpoint, this means that we check that the predicates

[5] A conservative, syntactic variant of this idea was earlier proposed by Strichman [15].

corresponding to the two literals are bounds for the same convex clock region. This check can be posed as a Boolean satisfiability problem, which is easily solved using a BDD representation of β. Let the literals be e_1 and e_2. Then, we use cofactoring and Boolean operations to compute the following Boolean formula:

$$e_1 \wedge e_2 \wedge [\beta|_{e_1=\mathbf{true}} \wedge \neg(\beta|_{e_1=\mathbf{false}})] \wedge [\beta|_{e_2=\mathbf{true}} \wedge \neg(\beta|_{e_2=\mathbf{false}})] \tag{5}$$

Formula 5 expresses the Boolean function corresponding to the disjunction of all terms in the minimized DNF representation of β that contain both e_1 and e_2 in true form. Therefore, if Formula 5 is satisfiable, it means that e_1 and e_2 are conjoined, and we must add a transitivity constraint involving them both.

Note however, that since β does not, by itself, represent the original SL formula ϕ, finding that e_1 and e_2 are conjoined in β does not imply that they are bounds in the same convex region of ϕ. However, the converse is true, so our method is sound.

4.2 Quantifier Elimination by Eliminating Upper Bounds on x_0

The definition of the time elapse operation introduces two quantified non-clock real variables: $\overline{\delta}$ and $\overline{\epsilon}$. We can exploit the special structure of the QSL formula for the time elapse operation so as to avoid introducing $\overline{\epsilon}$ altogether. Thus, we can avoid adding new quantified Boolean variables encoding predicates involving $\overline{\epsilon}$.

Consider the inner existentially quantified SL formula in Formula 4 in Section 2.3, reproduced here:

$$\exists\overline{\epsilon}(\overline{\epsilon} \leq x_0 \wedge \overline{\delta} \leq \overline{\epsilon} \wedge \neg\phi_1[\overline{\epsilon}/x_0])$$

Grouping the inequality $\overline{\delta} \leq \overline{\epsilon}$ with the formula $\neg\phi_1[\overline{\epsilon}/x_0]$, we get:

$$\exists\overline{\epsilon}\{\overline{\epsilon} \leq x_0 \wedge (\overline{\delta} \leq x_0 \wedge \neg\phi_1)[\overline{\epsilon}/x_0]\} \tag{6}$$

Finally, treating $\overline{\delta}$ as a clock variable, we can revert back to ϵ from $\overline{\epsilon}$, transforming Formula 6 to the following form:

$$\exists\epsilon[\epsilon \geq x_0 \wedge (\overline{\delta} \leq x_0 \wedge \neg\phi_1) + \epsilon] \tag{7}$$

Formula 7 is a special case of the formula ω_ϵ given by

$$\omega_\epsilon = \exists\epsilon.\epsilon \geq x_0 \wedge \phi + \epsilon$$

for some SL formula ϕ. From a geometric viewpoint, ϕ is a region in $\mathbb{R}^n$ and ω_ϵ is the shadow of ϕ for a light source at ∞^n. Examples of ϕ and the corresponding ω_ϵ are shown in Figures 1(a) and 1(c) respectively.

We can transform ω_ϵ to an equivalent SL formula ϕ_{ub} by eliminating upper bounds on x_0, i.e., Boolean variables of the form $e_{i,0}^{\bowtie,c}$. The transformation is performed iteratively in the following steps:

1. Let $\phi_0 = \phi$. Let $e_{i_1,0}^{\bowtie_1,c_1}, e_{i_2,0}^{\bowtie_2,c_2}, \ldots, e_{i_m,0}^{\bowtie_m,c_m}$ be Boolean literals encoding all upper bounds on x_0 that occur in ϕ.
2. For $j = 1, 2, \ldots, m$, we construct ϕ_j as follows:
   (a) Replace all occurrences of $x_{i_j} \bowtie_j x_0 + c_j$ in ϕ_{j-1} with $e_{i_j,0}^{\bowtie_j,c_j}$ to get $\phi_{bool}^{0,j-1}$.
   (b) Construct $\phi_{cons}^{0,j-1}$, the conjunction of all transitivity constraints[6] for x_0 involving $e_{i_j,0}^{\bowtie_j,c_j}$ and clock variables in $\phi_{bool}^{0,j-1}$.
   (c) Construct the formula ϕ_j, a disjunction of two terms:

   $$\phi_j = \{(\phi_{bool}^{0,j-1} \wedge \phi_{cons}^{0,j-1})|_{e_{i_j,0}^{\bowtie_j,c_j}=\mathbf{true}}\} \vee \{[\neg(x_{i_j} \bowtie_j x_0 + c_j)] \wedge [\phi_{bool}^{0,j-1}|_{e_{i_j,0}^{\bowtie_j,c_j}=\mathbf{false}}]\}$$

   The first disjunct is the region obtained by dropping the bound $x_{i_j} \bowtie_j x_0 + c_j$ from convex sub-regions of ϕ_{j-1} where it is a lower bound on x_{i_j}, while letting time elapse backward. The second disjunct corresponds to sub-regions where $\neg(x_{i_j} \bowtie_j x_0 + c_j)$ is an upper bound; these regions are left unchanged.

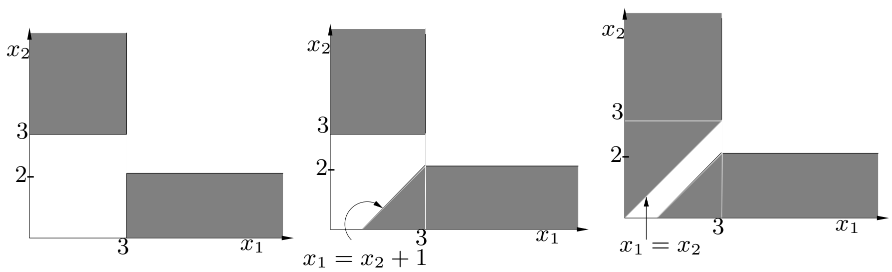
The output of the above transformation, ϕ_{ub}, is given by $\phi_{ub} = \phi_m$. The correctness of this procedure is formalized in the following theorem.

Theorem 2. *ω_ϵ and ϕ_{ub} are equivalent.*

Example 2. Let the subformula ϕ of ω_ϵ be

$$\phi = (x_1 \geq x_0 + 3 \wedge x_2 \leq x_0 + 2) \vee (x_1 < x_0 + 3 \wedge x_2 \geq x_0 + 3)$$

ϕ is depicted geometrically as the shaded region in Figure 1(a). It comprises two



(a) $\phi_0 = \phi$ (b) ϕ_1 (c) $\phi_2 = \omega_\epsilon$

Fig. 1. Eliminating upper bounds on x_0. Only the positive quadrant is shown.

sub-regions, one for each disjunct. The lower bounds on these regions, $x_1 \geq x_0 + 3$ and $x_2 \geq x_0 + 3$, are upper bounds on x_0. We encode these by $e_{1,0}^{\geq,3}$ and $e_{2,0}^{\geq,3}$.

Figure 1(b) shows ϕ_1, the result of eliminating $e_{1,0}^{\geq,3}$. Formally, we calculate

$$\phi_{bool}^{0,0} = (e_{1,0}^{\geq,3} \wedge x_2 \leq x_0 + 2) \vee (\neg e_{1,0}^{\geq,3} \wedge x_2 \geq x_0 + 3)$$
$$\phi_{cons}^{0,0} = (e_{1,0}^{\geq,3} \wedge x_2 \leq x_0 + 2) \implies (x_1 \geq x_2 + 1)$$

[6] We can use the optimization technique of Section 4.1 in this step.

Then, applying step 2(c) of the transformation, we get

$$\phi_1 = (x_2 \leq x_0 + 2 \wedge x_1 \geq x_2 + 1) \vee (x_1 < x_0 + 3 \wedge x_2 \geq x_0 + 3)$$

Similarly, in the next iteration, we introduce and eliminate $e_{2,0}^{\geq,3}$ to get ϕ_2, shown in Figure 1(c), which is equivalent to ω_ϵ. □

Note that the QSL formula obtained after eliminating the inner quantifier in Formula 4 is not of the form ω_ϵ, and so we cannot avoid introducing the $\overline{\delta}$ variable.

4.3 Eliminating Infeasible Paths in BDDs

Suppose β is the Boolean encoding of SL formula ϕ. Let ϕ_{cons} denote the conjunction of transitivity constraints for all real-valued variables in ϕ, and let β_{cons} denote its Boolean encoding. Finally, denote the BDD representations of β and β_{cons} by $\mathsf{Bdd}(\beta)$ and $\mathsf{Bdd}(\beta_{cons})$ respectively.

We would like to eliminate paths in $\mathsf{Bdd}(\beta)$ that violate transitivity constraints, i.e., those corresponding to assignments to variables in β for which $\beta_{cons} =$ **false**. We can do this by using the BDD `Restrict` operator, replacing $\mathsf{Bdd}(\beta)$ by `Restrict(`$\mathsf{Bdd}(\beta)$`, `$\mathsf{Bdd}(\beta_{cons})$`)`. Informally, `Restrict(`$\mathsf{Bdd}(\beta)$`, `$\mathsf{Bdd}(\beta_{cons})$`)` traverses $\mathsf{Bdd}(\beta)$, eliminating a path on which β_{cons} is **false** as long as it doesn't involve adding new nodes to the resulting BDD. Details about the `Restrict` operator may be found in the paper by Coudert and Madre [7].

Since eliminating infeasible paths in a large BDD can be quite time consuming, we only apply this optimization to the BDD for the set of reachable states, once on each fixpoint iteration.

5 Experimental Results

We implemented a BDD-based model checker called TMV, that is written in OCaml and uses the CUDD package[7]. We have performed preliminary experiments comparing the performance of our model checker for both reachability and non-reachability TCTL properties. For reachability properties, we compare against the other unbounded, fully symbolic model checkers, viz., a DDD-based checker (DDD) [12] and Red version 4.1 [17], which have been shown to outperform Uppaal2k and Kronos for reachability analysis. For non-reachability properties, such as checking that a system is non-zeno, we compare against Kronos and Red, the only other unbounded model checkers that check such properties.

We ran two experiments using Fischer's time-based mutual exclusion protocol. The first experiment compared our model checker against DDD and Red, checking that the system preserves mutual exclusion (a reachability property). In the second, we compared against Kronos and Red for checking that the

[7] `http://vlsi.colorado.edu/~fabio/CUDD`

product automaton is non-zeno (a non-reachability property). All experiments were run on a notebook computer with a 1 GHz Pentium-III processor and 128 MB RAM, running Linux. We ran DDD, Kronos, and Red with their default options. TMV used a static BDD variable ordering that is the same as the one used for the corresponding Boolean variables and separation predicates in DDD and is described in detail in our technical report [14].

Table 1(a) shows the results of the comparison against DDD and Red for checking mutual exclusion for increasing numbers of processes. For DDD and TMV, the table lists both the run-times and the peak number of nodes in the decision diagram for the reachable state set. We find that DDD outperforms TMV due to the blow-up of BDDs. In spite of the optimizations of Section 4, the peak node count in the case of DDD is less than that for TMV, since the local reduction operations performed by DDD during node creation can eliminate unnecessary DDD nodes without adding any time overhead. For example, DDD can reduce a function of the form $e_1 \wedge e_2 \wedge e_3$ under the transitivity constraint $[e_1 \wedge e_2] \implies e_3$ to simply the conjunction $e_1 \wedge e_2$. The BDD `Restrict` operator cannot always achieve this as it is sensitive to the BDD variable ordering. Furthermore, TMV contains many other BDDs, such as those for the transitivity constraints, to which we do not apply the `Restrict` optimization due to its runtime overhead. Finally, in comparison to Red, we see that while TMV is faster on the smaller benchmarks, Red's superior memory performance enables it to complete for 7 processes while TMV runs out of memory.

Num.	Red	DDD		TMV	
Proc.	Time (sec.)	Time (sec.)	Reach Set (peak nodes)	Time (sec.)	Reach Set (peak nodes)
3	0.21	0.06	130	0.11	101
4	1.13	0.14	352	0.38	316
5	4.53	0.33	854	1.85	1127
6	15.11	0.90	2375	17.41	4685
7	46.31	2.65	6346	*	*

(a) Mutual Exclusion

Num.	Kronos	Red	TMV	
Proc.	Time (sec.)	Time (sec.)	Time (sec.)	Reach Set (peak nodes)
3	0.03	0.28	0.24	28
4	0.23	1.30	0.44	39
5	1.98	5.05	0.80	54
6	*	17.80	2.15	69
7	*	57.95	6.61	88

(b) Non-Zenoness

Table 1. Checking properties of Fischer's protocol for increasing numbers of processes. A "*" indicates that the model checker ran out of memory.

Table 1(b) shows the comparison with Kronos and Red for checking non-zenoness. The time for Kronos is the sum of the times for product construction and backward model checking. We notice that while Kronos does better for smaller numbers of processes, the product automaton it constructs grows very quickly, becoming too large to construct at 6 processes. The run times for TMV, on the other hand, grow much more gradually, demonstrating the advantages of a fully symbolic approach. For this property, the BDDs remain small even for larger numbers of processes. Thus, TMV outperforms Red, especially as the number of processes increases. These results indicate that when the representation (BDDs) remains small, Boolean methods for quantifier elimination and deciding SL can outperform non-Boolean methods by a significant factor.

Although they are preliminary, our results indicate that our model checker based on a general purpose BDD package can outperform methods based on specialized representations of SL formulas. We are working on improving our current methods of eliminating unnecessary BDD nodes, and are also starting to work on a SAT-based implementation.

Acknowledgments. We thank Joël Ouaknine, Ofer Strichman, and the reviewers for useful comments. We also thank the authors of DDD and Red for providing their tools and answering our queries. This research was supported in part by a NDSEG Fellowship and by ARO grant DAAD19-01-1-0485.

References

1. L. Aceto, P. Bouyer, A. Burgueno, and K. G. Larsen. The power of reachability testing for timed automata. *Theoretical Computer Science*, 300(1-3):411-475, 2003.
2. R. Alur. Timed automata. In *Proc. CAV'99*, LNCS 1633, pages 8–22.
3. G. Audemard, A. Cimatti, A. Korniowicz, and R. Sebastiani. Bounded model checking for timed systems. In *Proc. FORTE'02*, LNCS 2529, pages 243–259.
4. G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In *Proc. CAV'99*, LNCS 1633, 341–353.
5. D. Beyer. Improvements in BDD-based reachability analysis of timed automata. In *Proc. FME'01*, LNCS 2021, pages 318–343.
6. M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In *Proc. CAV'97*, LNCS 1254, pages 179–190.
7. O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proc. ICCAD'90*, pages 126–129.
8. G. B. Dantzig and B. C. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory A*, 14:288–297, 1973.
9. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.
10. M. Koubarakis. Complexity results for first-order theories of temporal constraints. In *Proc. KR'94*, pages 379–390.
11. K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proc. CAV'02*, LNCS 2404, pages 250–264.
12. J. B. Møller. Simplifying fixpoint computations in verification of real-time systems. In *Proc. RTTOOLS'02* workshop.
13. P. Niebert, M. Mahfoudh, E. Asarin, M. Bozga, N. Jain, and O. Maler. Verification of timed automata via satisfiability checking. In *Proc. FTRTFT'02*, LNCS 2469.
14. S. A. Seshia and R. E. Bryant. A Boolean approach to unbounded, fully symbolic model checking of timed automata. Technical Report CMU-CS-03-117, Carnegie Mellon University, 2003.
15. O. Strichman. Optimizations in decision procedures for propositional linear inequalities. Technical Report CMU-CS-02-133, Carnegie Mellon University, 2002.
16. O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with SAT. In *Proc. CAV'02*, LNCS 2404, pages 209–222.
17. F. Wang. Efficient verification of timed automata with BDD-like data-structures. In *Proc. VMCAI'03*, LNCS 2575, pages 189–205.
18. S. Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1/2):123–133, Oct. 1997.