Optimizing Bounded Model Checking for Linear Hybrid Systems*

Erika Ábrahám[1], Bernd Becker[1], Felix Klaedtke[2], and Martin Steffen[3]

[1] Albert-Ludwigs-Universität Freiburg, Germany
[2] ETH Zurich, Switzerland
[3] Christian-Albrechts-Universität zu Kiel, Germany

Abstract. Bounded model checking (BMC) is an automatic verification method that is based on finitely unfolding the system's transition relation. BMC has been successfully applied, in particular, for discovering bugs in digital system design. Its success is based on the effectiveness of satisfiability solvers that are used to check for a finite unfolding whether a violating state is reachable. In this paper we improve the BMC approach for linear hybrid systems. Our improvements are tailored to lazy satisfiability solving and follow two complementary directions. First, we optimize the formula representation of the finite unfoldings of the transition relations of linear hybrid systems, and second, we accelerate the satisfiability checks by accumulating and generalizing data that is generated during earlier satisfiability checks. Experimental results show that the presented techniques accelerate the satisfiability checks significantly.

1 Introduction

Model checking is widely used for the verification of concurrent state systems, like finite state systems [20, 12] and timed automata [3]. One main reason for the acceptance of model checking is its push-button appeal. A major obstacle to its universal applicability, however, is the inherent size of many real-world systems. This obstacle is often called the state space explosion problem. *Bounded model checking* (BMC) [10] has attracted attention as an alternative to model checking. The bounded model checking problem starts from a more modest question: Does there exist a counterexample of length $k \in \mathbb{N}$ refuting a stipulated property P? In particular, when P is a safety property, a counterexample is simply a finite run leading to the violation. Whether P can be violated in k steps is reduced to checking satisfiability of the formula

$$I(s_0) \wedge T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge \neg P(s_k)\,, \tag{1}$$

where s_i are state variables, I is a unary predicate describing the initial states, and T is a binary predicate describing the transition relation. The bound k is successively increased until either a counterexample is found or some limit is reached (e.g., an upper bound on k or resource limitations).

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

BMC shares with model checking its push-button appeal. However, without further extensions, BMC does not terminate for properties that are fulfilled by the system. While this seems to be a step backwards, BMC has practical relevance. For finite state systems, the formula (1) corresponds to a propositional satisfiability problem that enables the use of state-of-the-art SAT-solvers. Empirical evidence, e.g., in [11] and [13], shows that BMC is often superior to model checking, in particular when the focus is on refuting a property. Extensions for using the BMC approach also for verification are summarized in [9].

The BMC approach for finite state systems cleanly extends to many classes of infinite state systems [16]. For infinite state systems, formula (1) is a Boolean combination of domain-specific constraints depending on the class of the systems. Instead of a SAT-solver we have to use a solver specific to that domain. For instance, BMC has been extended and applied to timed automata in, e.g. [19, 22, 7, 25]. The BMC approach can be further extended to the more general class of *linear hybrid automata* [2, 18]. For linear hybrid automata, the domain-specific constraints are linear (in)equations, where variables range over the reals. Prominent state-of-the-art solvers that can be used in the BMC approach for linear hybrid systems are MathSAT [5], CVC Lite [8], and ICS [15]. All these solvers have in common that the satisfiability checks are done *lazily*. Roughly speaking, this means that these solvers are based on a SAT-solver that calls on demand solvers for conjunctions of the domain-specific constraints.

In this paper we improve the BMC approach for linear hybrid systems by accelerating the satisfiability checks. Our improvements are motivated by a thorough investigation of checking satisfiability of formulas of the form (1), which describe in our context finite runs of a fixed length k of a linear hybrid system. First, we optimize the formula representation of finite runs. The optimized representation is tailored to lazy satisfiability solving. Besides others, one point is to force alternation of the different types of transitions of hybrid systems, namely discrete and time transitions. Second, we accumulate the conflicts returned by the domain-specific solver during the lazy satisfiability check of (1). We use these conflicts as follows. If (1) is unsatisfiable, i.e., there is no counterexample of size k, we generalize the returned conflicts and use these generalized conflicts such that the domain-specific solver is not called again for similar conflicts in forthcoming satisfiability checks. This means, we learn generalized domain-specific conflicts in each satisfiability check. This learning technique also applies to the BMC approach for other classes of infinite state systems.

Both kinds of optimization reduce the demand-driven calls to the domain-specific solver for conjunctions of linear (in)equations. Furthermore, they are complementary in the sense that the optimized encoding leads to fewer conflicts that are generalized and learned. We extensively evaluated our techniques for a number of linear hybrid systems. The outcome of our experiments is that the combination of both techniques increases the bound k on the size of the runs by several orders of magnitudes for which state-of-the-art solvers are able to perform the satisfiability checks in a reasonable amount of time and space.

We proceed as follows. In §2 we review the definition of linear hybrid automata and the BMC approach for linear hybrid automata using lazy satisfiability solvers. In §3 we optimize the encoding of finite runs and in §4 we introduce our learning technique. We

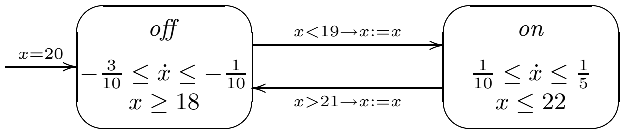


Fig. 1. Thermostat.

present experimental results in §5. In §6 we discuss related work and finally, in §7 we draw conclusions.

2 Bounded Model Checking for Linear Hybrid Systems

Before presenting our work, we first introduce linear hybrid systems and describe a straightforward encoding of finite runs as Boolean combinations of (in)equations. Furthermore, we describe relevant details of state-of-the-art solvers for checking satisfiability of Boolean combinations of linear (in)equations and pinpoint obstacles for using these solvers in the BMC approach for linear hybrid automata.

2.1 Hybrid Systems Background

Hybrid automata [2, 18] have been introduced in control engineering and in computer science as a formal model for systems with both discrete and continuous components.

Hybrid automata are often given graphically, like the one shown in Figure 1. This automaton models a thermostat, which senses the temperature x of a room and turns a heater on and off. In location *off* the heater is off and the temperature falls according to the flow condition $-\frac{3}{10} \leq \dot{x} \leq -\frac{1}{10}$. The location's invariant $x \geq 18$ assures that the heater turns on at latest when the temperature reaches 18 degrees. Analogously for the location on, where the heater is on. Control may move from location *off* to on if the temperature is below 19 degrees, and from on to *off* if the temperature is above 21 degrees. The temperature x does not change by jumping from *off* to on or from on to *off*. Initially, the heater is *off* and the temperature is 20 degrees.

In the remainder of the paper we only consider the class of linear hybrid automata, which can be described using first-order logic formulas over $(\mathbb{R}, +, <, 0, 1)$. Formally, a *linear hybrid automaton* $\mathcal{H}$ is a tuple

$$\big(L, V, (jump_{\ell,\ell'})_{\ell,\ell' \in L}, (flow_\ell)_{\ell \in L}, (inv_\ell)_{\ell \in L}, (init_\ell)_{\ell \in L}\big)\,,$$

where L and V are finite nonempty sets, and $(jump_{\ell,\ell'})_{\ell,\ell' \in L}$, $(flow_\ell)_{\ell \in L}$, $(inv_\ell)_{\ell \in L}$, $(init_\ell)_{\ell \in L}$ are families of first-order logic formulas over the structure $(\mathbb{R}, +, <, 0, 1)$:

- $L = \{\ell_1, \ldots, \ell_m\}$ is the set of *locations*.
- $V = \{v_1, \ldots, v_n\}$ is the set of *continuous variables*.

– $(jump_{\ell,\ell'})_{\ell,\ell' \in L}$ is an $(L \times L)$-indexed family of formulas with free variables in V and their primed versions. A formula $jump_{\ell,\ell'}(v_1, \ldots, v_n, v'_1, \ldots, v'_n)$ represents the possible *jumps* from location ℓ to location ℓ', where $v_1, \ldots, v_n$ are the values of the continuous variables before the jump and $v'_1, \ldots, v'_n$ are the values of the continuous variables after the jump.
– $(\mathit{flow}_\ell)_{\ell \in L}$ is an L-indexed family of formulas with free variables in V, their primed versions, and t. A formula $\mathit{flow}_\ell(v_1, \ldots, v_n, t, v'_1, \ldots, v'_n)$ represents the *flow* of duration $t \geq 0$ in location ℓ, where the values of the continuous variables change from $v_1, \ldots, v_n$ to $v'_1, \ldots, v'_n$.
– $(inv_\ell)_{\ell \in L}$ is an L-indexed family of formulas with free variables in V. A formula $inv_\ell(v_1, \ldots, v_n)$ represents the *invariant* in location ℓ. We require that all invariants are convex sets.
– $(init_\ell)_{\ell \in L}$ is an L-indexed family of formulas with free variables in V representing the *initial states* of the system.

For instance, the flow in location on of the thermostat in Figure 1 can be described by the formula $\mathit{flow}_{on}(x, t, x') = 10x' - 10x \geq t \wedge 5x' - 5x \leq t$. The other components of the thermostat can be described analogously. Since $(\mathbb{R}, +, <, 0, 1)$ admits quantifier elimination, we assume without loss of generality that the formulas occurring in the description of a linear hybrid automaton are quantifier-free.

Hybrid systems often consist of several hybrid automata that run in parallel and interact with each other. The parallel composition of hybrid automata requires an additional event set for synchronization purposes. The parallel composition is standard but technical and we omit it here. For simplicity and due to space limitations, in the theoretical part of the paper we restrict ourselves to a single linear hybrid automaton.

Encoding Linear Hybrid Automata In the remainder of this subsection, let $\mathcal{H} = \big(L, V, (jump_{\ell,\ell'})_{\ell,\ell' \in L}, (\mathit{flow}_\ell)_{\ell \in L}, (inv_\ell)_{\ell \in L}, (init_\ell)_{\ell \in L}\big)$ be a linear hybrid automaton with $L = \{l_1, \ldots, l_m\}$ and $V = \{v_1, \ldots, v_n\}$, for some $m, n \in \mathbb{N}$. For readability, we write tuples in boldface, i.e., $\boldsymbol{v}$ abbreviates $(v_1, \ldots, v_n)$, and we introduce state variables $s = (at, \boldsymbol{v})$, where at ranges over the locations in L and $\boldsymbol{v} = (v_1, \ldots, v_n)$.

A jump of the automaton $\mathcal{H}$ is described by the formula

$$J(s, s') = \bigvee\nolimits_{\ell,\ell' \in L} \big(at = \ell \wedge at' = \ell' \wedge jump_{\ell,\ell'}(\boldsymbol{v}, \boldsymbol{v}') \wedge inv_{\ell'}(\boldsymbol{v}')\big)$$

and a flow of $\mathcal{H}$ is described by the formula

$$F(s, t, s') = \bigvee\nolimits_{\ell \in L} \big(at = \ell \wedge at' = \ell \wedge t \geq 0 \wedge \mathit{flow}_\ell(\boldsymbol{v}, t, \boldsymbol{v}') \wedge inv_\ell(\boldsymbol{v}')\big)\,,$$

where $s = (at, \boldsymbol{v})$ and $s' = (at', \boldsymbol{v}')$ are state variables, and t is a real-valued variable representing the duration of the flow. Note that we check the invariant of a location after t time units have passed in $F(s, t, s')$ and when we enter the location of s' in a jump $J(s, s')$. Since we assume that invariants are convex sets, we do not have to check at every time point between 0 and t of a flow whether the invariant in the location is satisfied. For $k \in \mathbb{N}$, we recursively define the formula π_k by

$$\pi_0(s_0) = \bigvee\nolimits_{\ell \in L} \big(at_0 = \ell \wedge inv_\ell(\boldsymbol{v}_0)\big)$$

and for $k > 0$,

$$\pi_k(s_0, \ldots, s_k, t_1, \ldots, t_k) =$$
$$\pi_{k-1}(s_0, \ldots, s_{k-1}, t_1, \ldots, t_{k-1}) \wedge \big(J(s_{k-1}, s_k) \vee F(s_{k-1}, t_k, s_k)\big)\,,$$

where $s_0, \ldots, s_k$ are state variables and $t_1, \ldots, t_k$ are real-valued variables. Intuitively, π_k describes the runs of length k of a linear hybrid automaton by glueing together k jumps and flows. Moreover, we have to assure that the first state satisfies the location's invariant.

BMC for Linear Hybrid Automata With the formulas π_k at hand, it is straightforward to obtain a semi-decision procedure for checking whether a linear hybrid automaton violates a state property given by the formula $\mathit{safe}(s)$. For $k \in \mathbb{N}$, we define

$$\varphi_k(s_0, \ldots, s_k, t_1, \ldots, t_k) =$$
$$\big(\bigvee_{\ell \in L}(at_0 = \ell \wedge \mathit{init}_\ell(\boldsymbol{v}_0))\big) \wedge \pi_k(s_0, \ldots, s_k, t_1, \ldots, t_k) \wedge \neg \mathit{safe}(s_k)\,.$$

Starting with $k = 0$ and iteratively increasing $k \in \mathbb{N}$, we check whether φ_k is satisfiable. The algorithm terminates if φ_k is satisfiable, i.e., an unsafe state is reachable from an initial state in k steps.

The effectiveness of the algorithm depends on the effectiveness of checking whether the φ_ks are satisfiable. Experimental results show that the satisfiability checks of φ_k often become impractical even for small ks and rather small linear hybrid systems, like the railroad crossing example [4], which consists of three linear hybrid automata running in parallel (two of them have 3 locations and one automaton has 4 locations). For instance, the satisfiability check for φ_{10} takes 18 seconds with the state-of-the-art solver ICS [16] and the satisfiability check for φ_{15} takes almost 4 minutes. In order to pinpoint the reasons for the bad running times of the satisfiability checks, we first have to give some ICS details.

2.2 Satisfiability Checking Details and Performance Issues

We first recall details of *lazy theorem proving* [16]. Lazy theorem proving is built on top of a SAT-solver for propositional logic that lazily interacts with a solver for a specific domain. In our context, the domain specific solver checks satisfiability of conjunctions of linear (in)equations over the reals.

Assume that φ is a Boolean combination of the atomic formulas $\alpha_1, \ldots, \alpha_n$. We define the mapping $abs(\alpha_i) = b_i$, where b_i is a fresh Boolean variable. The mapping abs is homomorphically extended to Boolean combinations of (in)equations. We call $abs(\varphi)$ the *Boolean abstraction* of the formula φ. The pseudo-code of the lazy theorem proving algorithm from [16, 14] is shown in Figure 2. We start with the Boolean abstraction $\beta = abs(\varphi)$. In each loop, the SAT-solver suggests a candidate assignment $\nu : \{b_1, \ldots, b_n\} \to \{\mathit{true}, \mathit{false}\}$ satisfying β. If the conjunction $\psi = \bigwedge_{\nu(b_i)=\mathit{true}} \alpha_i \wedge \bigwedge_{\nu(b_i)=\mathit{false}} \neg\alpha_i$ is satisfiable, then φ is satisfiable. Otherwise, we extend β to $\beta \wedge \neg abs(\mathit{explain}(\psi))$, where $\mathit{explain}(\psi)$ is an unsatisfiable subformula of ψ, i.e., a conjunction of some atomic formulas or their negations occurring in ψ that is responsible

procedure *sat*(φ)
  $\beta \leftarrow abs(\varphi)$
  loop
    $\nu \leftarrow$ *SAT-Solver*(β)
    if $\nu = unsatisfiable$ **then return** $unsatisfiable$
    $\psi \leftarrow \bigwedge_{\nu(b_i)=true} \alpha_i \wedge \bigwedge_{\nu(b_i)=false} \neg\alpha_i$
    if *Solver*(ψ) $\neq unsatisfiable$ **then return** $satisfiable$
    $\beta \leftarrow \beta \wedge \neg abs(explain(\psi))$
  end loop

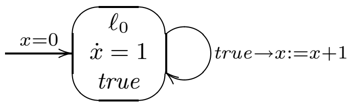
Fig. 2. The lazy theorem proving algorithm for checking satisfiability of a Boolean combination of linear (in)equations.

for the unsatisfiability of ψ. We call the formula $explain(\psi)$ an *explanation*. A simple implementation of *explain* is the identity function, i.e., it returns ψ. Using this simple implementation, there is one loop iteration for each satisfying assignment of $abs(\varphi)$. General techniques for reducing the number of iterations, and in particular more sophisticated implementations of the *explain* function are described in [16, 14].

Less lazy variants of the lazy theorem proving algorithm, like in CVC Lite [8] and ICS [16] consist of a tighter integration of a SAT-solver and the satisfiability checks of a solver for conjunctions of linear (in)equations. In ICS, a truth assignment to a Boolean variable by the SAT-solver adds the corresponding (in)equation to the conjunction of (in)equations for which the corresponding Boolean variables are already assigned to some truth value. A *frequency parameter*, for which the user can provide a threshold, determines after how many truth assignments the SAT-solver checks whether the conjunction of (in)equations is still satisfiable, i.e., the SAT-solver calls the solver for conjunctions of (in)equations. An inconsistency triggers backtracking in the search for Boolean variable assignments and is propagated to the SAT-solver by adding a clause to the formula explaining the inconsistency using the $explain$ function.

Performance Issues The lazy theorem proving algorithm in Figure 2 scales poorly for checking satisfiability of the formulas φ_k. The reason is the large number of loop iterations: for most examples, the number of iterations grows exponentially in k. The following examples illustrate this obstacle more clearly.

Example 1. Consider the following linear hybrid automaton:



Assume that we want to check whether we can reach in k steps a state with $x < 0$. Clearly, a run with x having the initial value 0 and that increases x in each step cannot reach a state with x having a negative value. However, when we only look at a finite unfolding of the transition relation, we must be aware of all changes made on the value of x in order to check that the value of x is not negative after k steps. Independently of the implementation of the $explain$ function, for checking unsatisfiability of φ_k with the lazy theorem proving algorithm, the number of loop iterations is at least 2^k.

The reason for the above exponential behavior can be explained as follows. For each of the 2^k possible sequences of k flows and jumps there is a corresponding satisfying assignment of $abs(\varphi_k)$ assigning $true$ to the Boolean variable for $x_k < 0$ and to the Boolean variables whose (in)equations describe the initial state and the transitions in the sequence. Without loss of generality, the truth values of the other Boolean variables $abs(\varphi_k)$ need not to be considered. For a satisfying assignment of $abs(\varphi_k)$ the $explain$ function has to return a conjunction containing at least the (in)equations in which x_i occurs and for which the Boolean variable is assigned to $true$. Since two such conjunctions of (in)equations are distinct for assignments corresponding to different sequences of k flows and jumps, we have to check at least 2^k conjunctions of (in)equations.

The less lazy variant of the lazy theorem proving algorithm is faced with a similar problem: the number of satisfiability checks for conjunctions of (in)equations corresponding to partial truth assignments of the Boolean variables in the Boolean abstraction is often exponential in the bound k. For the railroad crossing example, we have 95 explanations in the satisfiability check for φ_5, 1047 explanations for φ_{10}, and 6462 explanations for φ_{15}.

Experimental evaluations [16] have shown that the less lazy variant—as, e.g., implemented in ICS—is superior to the lazy theorem proving algorithm in Figure 2. However, in our experiments we observed that if the Boolean abstraction of a formula has few satisfying assignments then the lazy theorem proving algorithm usually performs better than the less lazy variant, since the solver for conjunctions of (in)equations is called less often. In §4, we will exploit this observation by switching from the less lazy variant to the lazy theorem proving algorithm whenever it is likely that the Boolean abstraction has few satisfying assignments.

Before we present and evaluate the optimizations for the BMC approach for linear hybrid systems we want to comment on the BMC approach for a larger class of hybrid systems. For the BMC approach, it is, in principal, possible to allow first-order logic formulas over $(\mathbb{R}, +, \cdot, <, 0, 1)$ instead of $(\mathbb{R}, +, <, 0, 1)$ in the definition of a hybrid automaton in §2.1. By allowing formulas over $(\mathbb{R}, +, \cdot, <, 0, 1)$ we can describe a much larger class of hybrid systems. Note that the first-order theory over $(\mathbb{R}, +, \cdot, <, 0, 1)$ is decidable since it admits quantifier elimination [24]. The lazy theorem proving algorithm can be easily modified to handle quantifier-free formulas over $(\mathbb{R}, +, \cdot, <, 0, 1)$. However, for a non-linear flow, we have to check a location's invariant in a run for all time points of that flow. This introduces in the formula description of a flow step an additional universally quantified variable, which has to be eliminated before we apply the lazy theorem proving algorithm. The reason why we restrict ourselves to $(\mathbb{R}, +, <, 0, 1)$ is that eliminating such a quantified variable can be expensive. Furthermore, the authors are not aware of a satisfiability checker for quantifier-free formulas over $(\mathbb{R}, +, \cdot, <, 0, 1)$ that performs well in practice for large conjunctions of quantifier-free formulas. However, the following optimizations will also be useful for this larger class of hybrid systems, since they reduce the number of interactions of the SAT-checker and the domain specific solver for the conjunctions.

Formula encoding with optimizations		$k = 5$		$k = 10$		$k = 15$	
		time (secs.)	# expl.	time (secs.)	# expl.	time (secs.)	# expl.
φ_k		0.5	95	18.0	1047	234.5	6462
φ_k	+ §3.1	0.2	21	3.7	349	46.8	1922
φ_k	+ §3.1 + §3.2	0.2	24	2.8	242	35.5	1741
ψ_{2k+1}	+ §3.1 + §3.2	0.2	4	1.8	53	3.6	109
$\psi_1, \ldots, \psi_{2k+1}$	+ §3.1 + §3.2	0.7	14	5.1	144	14.0	396
$\psi^{tau}_{1,2k+1}$	+ §3.1 + §3.2	0.4	14	0.9	21	6.8	169

Table 1. Experimental results for the railroad crossing example.

3 Optimizing the Encoding

For improving the BMC approach for linear hybrid automata, we optimize the formula encoding of finite runs. Our optimized encoding is tailored to the lazy theorem proving algorithms. In order to give an impression of the impact of the different optimizations, we list in Table 1 the improvements for the railroad crossing example. We obtain similar improvements for other examples of hybrid automata (further experiments are in §5).

Let $\mathcal{H} = \big(L, V, (jump_{\ell,\ell'})_{\ell,\ell' \in L}, (\mathit{flow}_\ell)_{\ell \in L}, (inv_\ell)_{\ell \in L}, (init_\ell)_{\ell \in L}\big)$ be a linear hybrid automaton with $V = \{v_1, \ldots, v_n\}$.

3.1 Using Boolean Variables

The lazy theorem proving algorithm in Figure 2 and its variants can be easily extended such that they also handle Boolean combinations of (in)equations *and* Boolean variables. Since the location set L is finite, we can use $\lceil \lg |L| \rceil$ Boolean variables for each $0 \leq i \leq k$ to encode the formulas $at_i = \ell$ with $\ell \in L$ in φ_k. However, the algorithm in Figure 2 replaces (in)equations by fresh Boolean variables; for each $0 \leq i \leq k$, this requires $|L|$ Boolean variables for the atomic formulas $at_i = \ell$ with $\ell \in L$.

Encoding finite sets by Boolean variables is not new. However, we want to point out the benefit of using Boolean variables for the lazy theorem proving algorithm. The Boolean encoding of locations has two advantages over the encoding by equations of the form $at_i = \ell$: The first advantage is that we need exponentially less Boolean variables. The more important advantage is the following. A satisfying assignment of $abs(\varphi_k)$ may assign the corresponding Boolean variables for the equations $at_i = \ell$ and $at_i = \ell'$ with $\ell \neq \ell'$ both to $true$. Such a conflict is not discovered until we call the solver for conjunctions of (in)equations. With Boolean location encoding such conflicts are already discovered by the SAT-solver. This results in less interaction of the SAT-solver and the solver for conjunctions of (in)equations. In particular, note that when using the Boolean encoding of the locations, the assignments returned by the SAT-solver always describe a path in the location graph of the hybrid automaton.

Analogously to the Boolean encoding of locations we can use Boolean variables for all system variables with a finite domain. In order to keep formulas readable, we still write formulas like $at_i = \ell$ as abbreviation for their Boolean encodings.

3.2 Excluding Bad and Initial State Loops

Another optimization is to require that we do not visit an initial state twice and only the last state violates the specification. This means, we add to φ_k the two conjuncts

$$\bigwedge_{0<i\leq k} \bigwedge_{\ell \in L} \neg\big(at_i = \ell \wedge init_\ell(\boldsymbol{v}_i)\big) \qquad \text{and} \qquad \bigwedge_{0\leq i<k} safe(s_i)\,.$$

This optimization has already been proposed in [21] for finite state systems.

It is worth mentioning that the speed-up due to this optimization heavily depends on the underlying linear hybrid automaton and the specification: For specifications containing Boolean variables (or Boolean encodings of locations), the number of assignments for the Boolean abstraction can be reduced this way. On the other hand, if adding the above conjuncts introduces (in)equations that do not occur in φ_k, then it is less likely that this optimization improves the running times of the satisfiability checks. However, it does not significantly slow them down in our examples.

3.3 Alternating Flows and Jumps

Since two successive flows of durations t and t' can always be represented by a single flow of duration $t + t'$, we can require that each flow is followed by a jump. This restriction excludes irrelevant computations, and thus leads to a reduced number of solutions for the Boolean abstractions of the formulas φ_k. Excluding successive flows has already been proposed in [7].

Below we define a formula that describes computations with alternating flows and jumps, thereby excluding successive time steps without any overhead. Note that we also exclude runs with successive jumps. However, successive jumps can be expressed using flows of duration 0. Each computation can be rewritten to this form with alternating flows and jumps. The advantage of alternating flows and jumps over excluding successive flows is discussed in Remark 1. For $k \in \mathbb{N}$, we define ψ_k similar to φ_k where π_k is replaced by π'_k:

$$\begin{aligned}\psi_k(s_0, \ldots, s_k, t_1, \ldots, t_k) = \\ \big(\textstyle\bigvee_{\ell \in L}(at_0 = \ell \wedge init_\ell(\boldsymbol{v}_0))\big) \wedge \pi'_k(s_0, \ldots, s_k, t_1, \ldots, t_k) \wedge \neg safe(s_k)\,,\end{aligned}$$

where $\pi'_0(s_0) = \pi_0(s_0)$, and for $k > 0$,

$$\begin{aligned}\pi'_k(s_0, \ldots, s_k, t_1, \ldots, t_k) = \\ \pi'_{k-1}(s_0, \ldots, s_{k-1}, t_1, \ldots, t_{k-1}) \wedge \begin{cases} J(s_{k-1}, s_k) & \text{if } k \text{ is even,} \\ F(s_{k-1}, t_k, s_k) & \text{otherwise.} \end{cases}\end{aligned}$$

Using the above definition for searching iteratively for counterexamples, it suffices to start with $k = 1$ and to increase k in each iteration by 2: We start with a run consisting of a single flow. In each iteration we extend the runs under consideration with a jump that is followed by a flow. Since flows may have the duration 0, there is a counterexample containing k jumps iff ψ_{2k+1} is satisfiable.

Recall that φ_k is satisfiable iff there is a counterexample of length k. Now, if there is a counterexample of length less than or equal to k then there is also a counterexample containing at most k jumps. However, not all runs with at most k jumps can be represented by a run of length less than or equal to k. Consequently, the unsatisfiability of $\psi_1, \psi_3, \ldots, \psi_{2k+1}$ implies the unsatisfiability of $\varphi_0, \varphi_1, \ldots, \varphi_k$. The converse is not true.

The formula ψ_{2k} has twice as many variables as φ_k but the number of distinct (in)equations is approximately the same. Note that for the satisfiability check the number of distinct (in)equations is relevant and not the number of variables. That means, using ψ_{2k+1} instead of φ_k has the advantage that with no overhead the first k iterations check all runs of length less than or equal to $2k+1$ with at most k jumps in addition to the runs of length less than or equal to k, as it is also done by φ_k.

Moreover, the satisfiability check for ψ_{2k+1} is in most cases faster than the satisfiability check for φ_k (see Table 1 and the experiments in §5). The reason is that the number of calls of the solver for conjunctions of (in)equations in the lazy theorem proving algorithms often reduces significantly.

Remark 1. When excluding successive flows we still have the choice of doing a jump or a flow after we have done a jump. This choice is eliminated when we alternate between flows and jumps. In practice, eliminating this choice pays off. For instance, for the hybrid automaton in Example 1, for every $k \geq 0$ there is exactly one satisfying assignment for the Boolean abstraction of ψ_{2k+1} when flows and jumps alternate. Therefore, we have to check only one conjunction of (in)equations. In contrast, by excluding successive flows we would have to cope with exponentially many assignments.

Note that applying the optimization in §3.2 together with the encoding using alternating flows and jumps, we have to allow that the first two states can be initial states, since there are runs that can be described only with a first flow having the duration 0. Similarly, we must allow the last two states to violate the specification.

3.4 Introducing τ-transitions

The BMC approach analyzes in each iteration runs of a certain length. That means, in order to show all runs of a length less than or equal to k to be safe, we must check the satisfiability of $k+1$ formulas. In this section we develop a method to search for counterexamples reachable by runs of length less than or equal to k in a single satisfiability check. To do so, we introduce jumps that do nothing, so-called τ-transitions. Recall that flows may have the duration 0. We require that after a τ-transition only further τ-transitions or flows of duration 0 are possible. Formally, for $k, k' \in \mathbb{N}$ we define $\psi^{tau}_{k',k}$ similar to ψ_k, where π'_k is replaced by $\pi''_{k',k}$:

$$\begin{aligned}\psi^{tau}_{k',k}&(s_0, \ldots, s_k, t_1, \ldots, t_k) = \\ &\big(\textstyle\bigvee_{\ell \in L}(at_0 = \ell \wedge init_\ell(\boldsymbol{v}_0))\big) \wedge \pi''_{k',k}(s_0, \ldots, s_k, t_1, \ldots, t_k) \wedge \neg safe(s_k)\,,\end{aligned}$$

where $\pi''_{k',k}$ describes computations of length k allowing τ-transitions to occur after the first k' steps only. We define $\pi''_{k',k} = \pi'_k$ for $k' \geq k$, and for $k' < k$ we define

$$\pi''_{k',k}(s_0, \ldots, s_k, t_1, \ldots, t_k) = \pi''_{k',k-1}(s_0, \ldots, s_{k-1}, t_1, \ldots, t_{k-1}) \wedge \begin{cases} \big((\neg tau_{k-2} \wedge J(s_{k-1}, s_k)) \vee tau_k\big) & \text{if } k \text{ is even,} \\ F(s_{k-1}, t_k, s_k) \wedge (tau_{k-1} \rightarrow t_k = 0) & \text{otherwise} \end{cases}$$

where tau_k is a shortcut for $false$ if $k \leq 0$ and $s_{k-1} = s_k$, otherwise.

Assume that we already know that there are no counterexamples of length less than or equal to k', and we want to check for some $k > k'$ whether we can reach a bad state in at most k steps. Instead of checking satisfiability of the formulas $\psi_{2k'+3}, \ldots, \psi_{2k+1}$ or $\psi^{tau}_{1,2k+1}$ it suffices to check satisfiability of $\psi^{tau}_{2k'+3,2k+1}$.

The formula $\psi^{tau}_{k',k}$ allows us more flexibility in the BMC approach with hardly any overhead by increasing the length of the runs. The main advantage of using $\psi^{tau}_{k',k}$ is that we only have to call the solver once, and guide the solver not to do unnecessary work, i.e., we force the solver not to look for counterexamples that end in a bad state in less than k' steps.

For the railroad crossing example, the last two rows of Table 1 compare the sums of running times for ψ_{2k+1}, where k ranges from 0–5, 0–10, and 0–15 with the running times of $\psi^{tau}_{1,2k+1}$ for $k \in \{5, 10, 15\}$.

4 Learning Explanations

The bottleneck of the lazy theorem proving algorithm and its less lazy variants for the satisfiability check of a Boolean combination φ of (in)equations is the large number of calls to the solver for conjunctions of (in)equations. In the BMC approach, the number of calls usually grows exponentially with respect to the bound k. In this section we present a simple but effective method for reducing the calls of the solver for conjunctions of (in)equations.

The idea is that we make use of the knowledge of the unsatisfiability of the explanations that were generated during the previous satisfiability checks of the BMC algorithm. Assume that there is no counterexample of length less than k, i.e., the formulas $\psi_1, \ldots, \psi_{2k-1}$ are unsatisfiable. Moreover, assume that $\gamma_1, \ldots, \gamma_n$ are the explanations that are generated during the satisfiability checks for $\psi_1, \ldots, \psi_{2k-1}$. Since the γ_is are unsatisfiable conjunctions of (in)equations, we can check satisfiability of $\psi_{2k+1} \wedge \left(\bigwedge_{1 \leq i \leq n} \neg\gamma_i \right)$ instead of ψ_{2k+1} in the next iteration of the BMC algorithm. Intuitively, this means that we "learn" for the next iteration the unsatisfiability of the explanations $\gamma_1, \ldots, \gamma_n$.

In practice it turned out that just adding explanations from the previous satisfiability checks does not result in much speed-up. However, we can do better. In order to describe our method of exploiting the knowledge of the unsatisfiability of the explanations, we need the following definitions.

Definitions Let $\gamma = \bigwedge_{1 \leq i \leq m} \alpha_i$ and $\gamma' = \bigwedge_{1 \leq i \leq m'} \alpha'_i$ be explanations. The explanation γ *(syntactically) subsumes* γ' if for every $1 \leq i \leq m$ there is a $1 \leq j \leq m'$ such that α_i and α'_j are syntactically equal. The explanation γ is *minimal* if for every $1 \leq j \leq m$, the conjunction $\bigwedge_{1 \leq i \leq m \text{ and } i \neq j} \alpha_i$ is satisfiable. For an integer s, *shift*(γ, s) denotes the formula γ where each variable index i occurring in γ is replaced by $i + s$. The motivation of shifting indices in explanations is that the lazy theorem proving algorithm often checks similar conjunctions of (in)equations that only differ by the indices of the variables. Note that shifting the indices does not change the satisfiability of a formula. Let *min*(γ) be the smallest index occurring in γ and let *max*(γ)

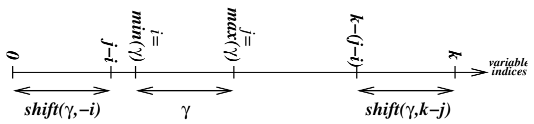


Fig. 3. Shifting an explanation γ.

be the largest index occurring in γ. Figure 3 illustrates the possible range of values for shifting the indices in the explanation γ up to some bound k. The set of all variations of γ due to index shifting such that all indices are between 0 and k is defined as

$$\mathit{SHIFT}(\gamma, k) = \big\{\mathit{shift}(\gamma, s) \,\big|\, -\mathit{min}(\gamma) \leq s \leq k - \mathit{max}(\gamma) \text{ and } s \text{ is even}\big\}\,.$$

Observe that we always shift indices by an even integer. An (in)equation in an explanation γ describing a flow rarely describes also a jump. Since flows and jumps alternate in the formula ψ_{2k-1}, it is unlikely that for an odd s, the additional conjunct $\mathit{shift}(\gamma, s)$ prunes the search space in the satisfiability check of $\psi_{2k+1} \wedge \neg\mathit{shift}(\gamma, s)$.

Learning Method The learned explanations should not contain irrelevant (in)equations. Therefore we first minimize every explanation that is generated during a satisfiability check. We do minimization greedily: We eliminate the first (in)equation α in an explanation γ if γ without α is still unsatisfiable; otherwise we do not remove α. We proceed successively with the other (in)equations in γ in the same way. After minimizing an explanation γ we delete all other explanations that are subsumed by γ. Finally, using shifting, we generalize all the remaining explanations for the next BMC iteration. In the kth BMC iteration we check satisfiability of the formula

$$\psi_{2k+1}^{learning} = \psi_{2k+1} \wedge \big(\textstyle\bigwedge_{\gamma \in E} \bigwedge_{\gamma' \in SHIFT(\gamma, 2k+1)} \neg\gamma'\big)\,,$$

where E is the set of all minimized explanations that occurred in the first $k-1$ iterations and that are not subsumed by other explanations.

We point out that with the additional conjunct $\big(\bigwedge_{\gamma \in E} \bigwedge_{\gamma' \in SHIFT(\gamma, 2k+1)} \neg\gamma'\big)$ we not only learn explanations that have been generated during earlier satisfiability checks, but due to index shifting we also apply them to the whole length of computations. Our case studies have shown that the same conflicts occur in different iterations with shifted indices, i.e., at another part of the computation sequence.

Due to our learning method, the Boolean abstractions of the formulas $\psi_{2k+1}^{learning}$ often have very few satisfying assignments. For such formulas, it is often more efficient to use the lazy theorem proving algorithm than the less lazy variant of it, since the solver for conjunctions of (in)equations has to be called less often. We pursue the policy that if in the last two iterations there are less than a threshold number (we use 50) of explanations then we assign a large value to the frequency parameter (see §2.2) of ICS, i.e., ICS switches to a "very" lazy variant of the lazy theorem proving algorithm. The running times heavily depend on this threshold.

Example	Last iteration below 200 secs. of CPU time		
	naive	**optimized**	**optimized+learning**
Thermostat	70	> 1500	> 1500
Water-level monitor	39	> 1500	> 1500
Railroad crossing	14	52	872
Extended railroad crossing	10	12	80
Fischer's protocol (2 processes)	10	15	1254
Fischer's protocol (3 processes)	9	14	31
Bakery protocol (2 processes)	10	45	742
Nuclear reactor	20	82	> 1500
Audio-control protocol	20	62	357

Table 2. Maximal number of BMC iterations k.

5 Experimental Results

We carried out tests for evaluating the BMC approach for linear hybrid systems with the different encodings and techniques described in §3 and §4. Our test suite[1] consists of standard examples, e.g., examples that come with the HyTech tool and the Bakery protocol[2]. All experiments were performed on a SUN Blade 1000 with 8 Gbytes of main memory and two 900 Mhz UltraSparc III+ processors; each one with an 8 Mbyte cache. We used ICS (version 2.0b) [16] for checking satisfiability of the formulas in the BMC approach. The reason for us to use ICS was that in most cases ICS behaves at least as good as other state-of-the-art solvers [15]. We expect similar running times with other state-of-the-art solvers, like e.g., CVC Lite [8], since they use similar techniques as described in §2.2 for checking satisfiability of Boolean combinations of (in)equations.

We report on experimental results for the following three different encodings of finite runs: (A) the *naive* encoding as described in §2.1; (B) the *optimized* encoding as described in §3.1–§3.3; (C) the optimized encoding as in (B) with additional *learning* of explanations as described in §4. Table 2 lists for each example the maximal number of BMC iterations for which every satisfiability check could be performed within a time limit of 200 seconds.

Additionally, we recorded the running times for each iteration and the numbers of explanations that are generated during the satisfiability checks. In the following, we describe the outcome of our experiments separately.

Running Times Figure 4 shows the running times for the encodings (A), (B), and (C) for some of our examples with k ranging from 0 to 200.

Checking satisfiability of the formulas φ_k becomes impractical even for small ks. For example, the satisfiability check for the railroad crossing example with $k = 15$ needs more than 230 seconds of CPU time. Although the optimization of the representation with alternating flows and jumps leads to a reduction of the running times, checking satisfiability of ψ_{2k+1} is also limited to rather small ks. For the railroad crossing

[1] A detailed description of our test suite and all the experimental results is in [1].

[2] The Bakery protocol is not a hybrid system but a discrete infinite state system. Our techniques can also be used for the BMC approach of such systems.

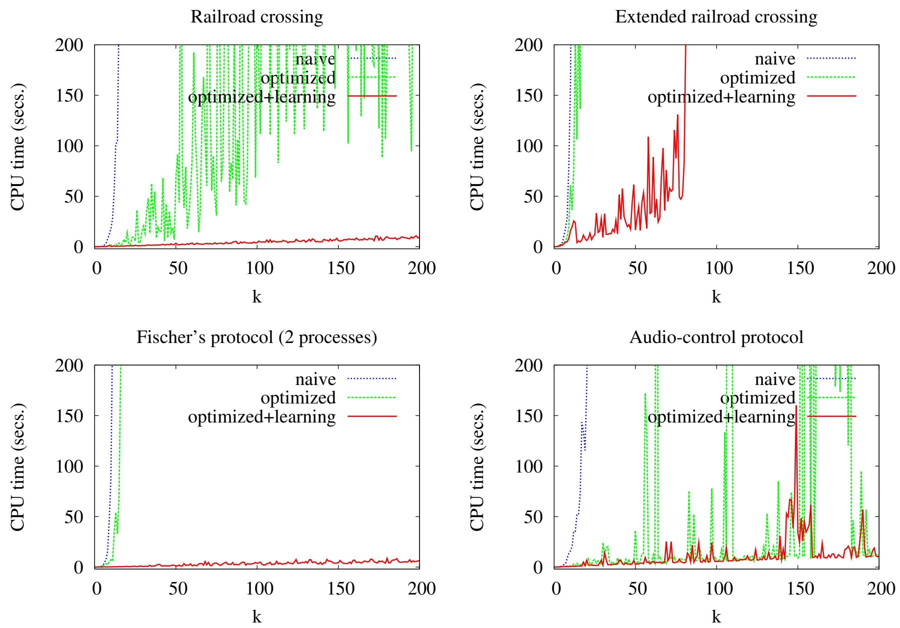


Fig. 4. Running times for the satisfiability checks for the naive encoding, the optimized encoding, and the optimized encoding with learning explanations.

example each satisfiability check for $k < 53$ needs less than 200 seconds; for $k = 53$ the satisfiability check exceeds our time limit of 200 seconds. The technique of learning explanations reduces the running times significantly. More importantly, the running times of satisfiability checks often scale much better for our examples. For instance, for the railroad crossing example each satisfiability check for $k \leq 200$ is under 11 seconds. The running times for computing the set of explanations that are added to the formula are not included. For the railroad crossing example, the sum of CPU times that ICS needs for the explanation minimization amounts to 15 seconds in the first 12 iterations; there are no explanations generated in later iterations. The reason for not including the times for minimizing explanations and the subsumption checks is twofold: First, we are interested in the speed-up of the satisfiability check that is due to the learning of explanations. Second, the implementation of the minimization and subsumption check is currently rather naive. For instance, we call ICS for each minimization step.

Number of Explanations Additionally to the running times, we also recorded the numbers of explanations that are generated during the satisfiability checks. The running times strongly correlate with the numbers of explanations. A detailed statistics on the number of explanations for the railroad crossing example is listed in Table 3. We obtained similar numbers for the other examples.

The second and third column in Table 3 list the numbers of explanations generated during the satisfiability checks of φ_k and of ψ_{2k+1} with the optimizations of §3.1–§3.2, respectively, for some different ks. The optimizations significantly reduce the number

k	naive # expl.	optimized # expl.	optimized+learning # expl.	 # expl. after subsumption check	 mean expl. size	 mean expl. size after minimization
0	1	1	1	1	3	2
3	31	3	1	1	25	18
6	179	12	0	0	0	0
9	651	40	27	6	19	8
12	2500	20	9	2	21	13
15	6462	109	0	0	0	0

Table 3. Number of explanations that are generated during the satisfiability checks for the railroad crossing example.

of generated explanations. Further reduction can be reached by learning explanations, as illustrated in the fourth column. Only a few explanations (column 5) are left over after minimization and removing subsumed explanations. The sizes of the explanations, i.e., the numbers of (in)equations in the explanations (column 6) are reduced by minimization. Column 7 shows the mean sizes of the minimized explanations that remain after subsumption. These sizes are often moderate in comparison to the bound k. For the railroad crossing example with optimization and learning explanations, ICS only generates explanations for $k \in \{0, \ldots, 12\}$.

6 Related Work

BMC has been extended to verify properties for finite state systems [21] by introducing termination conditions that are again checked by a SAT-solver. A generalization and extension of these methods to infinite state systems is presented in [17]. We have also applied our presented optimizations for checking termination conditions. We obtained similar improvements as for the satisfiability checks of the counterexample search.[3]

A complementary method of learning conflicts discovered in previous satisfiability checks is described in [23]. The conflicts that are learned by the two methods originate from different kinds of inconsistencies. The method in [23] learns conflicts that are discovered by the SAT-solver and our method learns conflicts that are discovered by the domain-specific solver.

Our work is in the line of the works by Audemard et. al. [7, 6] and by Sorea et. al. [22, 16] on the BMC approach for timed systems using lazy satisfiability solvers for Boolean combinations of (in)equations. The papers [7] and [22] extend the BMC approach to timed automata for properties written as LTL formulas. For simplicity, we only considered state properties. The paper [7] proposes several optimizations for encoding finite runs of timed systems. For instance, Audemard et. al. avoid successive flows and encode some form of symmetry reduction. The symmetry reduction only applies to certain timed systems, e.g., for systems consisting of identical components. As explained in Remark 1 in §3.3, alternating between flows and jumps is superior to excluding successive flows. Alternating between flows and jumps also appears in [22] with a different motivation. Sorea argues that alternation guarantees nonzenoness and often leads to smaller

[3] Due to space limitations we omit a description on checking termination conditions. Details and the experimental results are in [1].

completeness thresholds for timed automata. In contrast, our motivation is that alternating between flows and jumps accelerates lazy satisfiability solving. We show that alternation significantly speeds up the satisfiability checks. The papers [6, 16] extend and generalize the work in [7, 22].

In [19] bounded-length verification problems for timed automata are translated into formulas in difference logic. Another approach of BMC for timed automata is presented in [25]. In contrast to the work by Audemard et. al., Sorea et. al., and ours, the core of their work is a reduction from the BMC problem for timed automata to a SAT problem exploiting the region graph construction for timed automata.

7 Conclusion

In this paper we presented complementary optimizations for improving the BMC approach for linear hybrid automata and explained why these optimizations speed-up lazy satisfiability solving. Experimental results substantiate the benefit of the optimizations. The speed-up stems from reducing the interactions of the SAT-solver used as well as the domain-specific solver. Our first optimization tunes the encodings of finite runs of linear hybrid automata and the second optimization speeds up the satisfiability checks by learning generalized conflicts. The learning technique can also be used in the BMC approach for other classes of infinite state systems.

Other verification tools for linear hybrid systems, like the model checker HyTech, are faster on some of our test examples. One reason is that, on small examples, the reachable set computation terminates already after a few iterations. However, many larger systems cannot be handled by model checkers due to state explosion. The BMC approach for hybrid system verification is still in its infancy, but this paper shows, that there is a large potential for further improvements to be successful also for larger examples.

Our future work includes developing a tighter integration of generalized conflict learning and satisfiability solving. One task here is to develop methods that determine the usefulness of conflicts in later satisfiability checks and data structures that efficiently store generalized conflicts with fast look-ups. We also want to develop a more dynamic adjustment of the "laziness" in the satisfiability checks. Moreover, minimizing explanations is a crucial subtask in lazy satisfiability solving. At the moment, minimization is done greedily. Other methods for explanation minimization have to be developed.

Acknowledgements We thank Christian Herde and Martin Fränzle for the fruitful discussions, and Leonardo de Moura and Harald Rueß for answering our numerous ICS-related questions.

References

1. E. Ábrahám, B. Becker, F. Klaedtke, and M. Steffen. Optimizing bounded model checking for linear hybrid systems. Technical Report TR214, Albert-Ludwigs-Universität Freiburg, Fakultät für Angewandte Wissenschaften, Institut für Informatik, 2004. Online available at `http://www.informatik.uni-freiburg.de/tr/`.
2. R. Alur, C. Courcoubetis, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138:3–34, 1995.

3. R. Alur and D. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126:183–235, 1994.
4. R. Alur, T. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22:181–201, 1996.
5. 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. of CADE'02*, volume 2392 of *LNAI*, pages 195–210, 2002.
6. G. Audemard, M. Bozzano, A. Cimatti, and R. Sebastiani. Verifying industrial hybrid systems with MathSAT. In *Proc. of BMC'04*, 2004.
7. G. Audemard, A. Cimatti, A. Korniłowicz, and R. Sebastiani. Bounded model checking for timed systems. In *Proc. of FORTE'02*, volume 2529 of *LNCS*, pages 243–259, 2002.
8. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. of CAV'04*, volume 3114 of *LNCS*, pages 515–518, 2004.
9. A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.
10. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of TACAS'99*, volume 1579 of *LNCS*, pages 193–207, 1999.
11. A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC$^{\mathrm{TM}}$ microprocessor using symbolic model checking without BDDs. In *Proc. of CAV'99*, volume 1633 of *LNCS*, pages 60–71, 1999.
12. E. Clarke and E. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic specifications. In *Proc. of the Workshop on Logic of Programs 1981*, volume 131 of *LNCS*, pages 244–263, 1982.
13. F. Copty, L. Fix, R. Fraer, E. Guinchiglia, G. Kamhi, and M. Vardi. Benefits of bounded model checking in an industrial setting. In *Proc. of CAV'01*, volume 2102 of *LNCS*, pages 436–453, 2001.
14. L. de Moura and H. Rueß. Lemmas on demand for satisfiability solvers. In *Proc. of SAT'02*, pages 244–251, 2002.
15. L. de Moura and H. Rueß. An experimental evaluation of ground decision procedures. In *Proc. of CAV'04*, volume 3114 of *LNCS*, pages 162–174, 2004.
16. L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proc. of CADE'02*, volume 2392 of *LNAI*, pages 438–455, 2002.
17. L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In *Proc. of CAV'03*, volume 2725 of *LNCS*, pages 14–26, 2003.
18. T. Henzinger. The theory of hybrid automata. In *Proc. of LICS'96*, pages 278–292, 1996.
19. P. Niebert, M. Mahfoudh, E. Asarin, M. Bozga, N. Jain, and O. Maler. Verification of timed automata via satisfiability checking. In *Proc. of FTRTFT'02*, volume 2469 of *LNCS*, pages 225–244, 2002.
20. J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the 5th International Symposium on Programming 1981*, volume 137 of *LNCS*, pages 337–351, 1982.
21. M. Sheeran, S. Singh, and G. Stalmårck. Checking safety properties using induction and a SAT-solver. In *Proc. of FMCAD'00*, volume 1954 of *LNCS*, pages 108–125, 2000.
22. M. Sorea. Bounded model checking for timed automata. *Electronic Notes in Theoretical Computer Science*, 68, 2002.
23. O. Strichman. Accelerating bounded model checking of safety properties. *Formal Methods in System Design*, 24(1):5–24, 2004.
24. A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, Berkeley, 2nd edition, 1951.
25. B. Woźna, A. Zbrzezny, and W. Penczek. Checking reachability properties for timed automata via SAT. *Fundamenta Informaticae*, 55(2):223–241, 2003.