To be presented at the 18th International Conference on Automated Deduction (CADE'02), Copenhagen, Denmark, July 2002.

Lazy Theorem Proving for Bounded Model Checking over Infinite Domains*

Leonardo de Moura, Harald Rueß, and Maria Sorea**

SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
{demoura, ruess, sorea}@csl.sri.com
http://www.csl.sri.com/

Abstract. We investigate the combination of propositional SAT checkers with domain-specific theorem provers as a foundation for bounded model checking over infinite domains. Given a program M over an infinite state type, a linear temporal logic formula φ with domain-specific constraints over program states, and an upper bound k, our procedure determines if there is a falsifying path of length k to the hypothesis that M satisfies the specification φ. This problem can be reduced to the satisfiability of Boolean constraint formulas. Our verification engine for these kinds of formulas is *lazy* in that propositional abstractions of Boolean constraint formulas are incrementally refined by generating lemmas on demand from an automated analysis of spurious counterexamples using theorem proving. We exemplify bounded model checking for timed automata and for RTL level descriptions, and investigate the lazy integration of SAT solving and theorem proving.

1 Introduction

Model checking decides the problem of whether a system satisfies a temporal logic property by exploring the underlying state space. It applies primarily to finite-state systems but also to certain infinite-state systems, and the state space can be represented in symbolic or explicit form. Symbolic model checking has traditionally employed a boolean representation of state sets using binary decision diagrams (BDD) [4] as a way of checking temporal properties, whereas explicit-state model checkers enumerate the set of reachable states of the system.

Recently, the use of Boolean satisfiability (SAT) solvers for linear-time temporal logic (LTL) properties has been explored through a technique known as

* This research was supported by SRI International internal research and development, the DARPA NEST program through Contract F33615-01-C-1908 with AFRL, and the National Science Foundation under grants CCR-00-86096 and CCR-0082560.
** Also affiliated with University of Ulm, Germany.

bounded model checking (BMC) [7]. As with symbolic model checking, the state is encoded in terms of booleans. The program is unrolled a bounded number of steps for some bound k, and an LTL property is checked for counterexamples over computations of length k. For example, to check whether a program with initial state I and next-state relation T violates the invariant *Inv* in the first k steps, one checks, using a SAT solver:

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge (\neg Inv(s_0) \vee \ldots \vee \neg Inv(s_k)).$$

This formula is satisfiable if and only if there exists a path of length at most k from the initial state s_0 which violates the invariant *Inv*. For finite state systems, BMC can be seen as a complete procedure since the size of counterexamples is essentially bounded by the diameter of the system [3]. It has been demonstrated that BMC can be more effective in falsifying hypotheses than traditional model checking [7, 8].

It is possible to extend the range of BMC to infinite-state systems by encoding the search for a counterexample as a satisfiability problem for the logic of Boolean constraint formulas. For example, the BMC problem for timed automata can be captured in terms of a Boolean formula with linear arithmetic constraints. But the method presented here scales well beyond such simple arithmetic clauses, since the main requirement on any given constraint theory is the decidability of the satisfiability problem on conjunctions of atomic constraints. Possible constraint theories include, for example, linear arithmetic, bitvectors, arrays, regular expressions, equalities over terms with uninterpreted function symbols, and combinations thereof [20, 24].

Whereas BMC over finite-state systems deals with finding satisfying Boolean assignments, its generalization to infinite-state systems is concerned with satisfiability of Boolean constraint formulas. In initial experiments with PVS [21] strategies, based on a combination of BDDs for propositional reasoning and a variant of loop residue [27] for arithmetic, we were usually only able to construct counterexamples of small depths ($\leq$ 5). Clearly, more specialized verification techniques are needed. Since BMC problems are often propositionally intensive, it seems to be more effective to augment SAT solvers with theorem proving capabilities, such as ICS [10], than add propositional search capabilities to theorem provers.

Here, we look at the specific combination of SAT solvers with decision procedures, and we propose a method that we call *lemmas on demand*, which invokes the theorem prover *lazily* in order to efficiently prune out spurious counterexamples, namely, counterexamples that are generated by the SAT solver but discarded by the theorem prover by interpreting the propositional atoms. For example, the SAT solver might yield the satisfying assignment p, $\neg q$, where the propositional variable p represents the atom $x = y$, and q represents $f(x) = f(y)$. A decision procedure can easily detect the inconsistency in this assignment. More importantly, it can be used to generate a set of conflicting assignments that can be used to construct a lemma that further constrains the search. In the above example, the lemma $\neg p \vee q$ can be added as a new clause in the input to the

SAT solver. This process of refining Boolean formulas is similar in spirit to the refinement of abstractions based on the analysis of spurious counterexamples or failed proof attempts [26, 25, 6, 16, 9, 14, 17].

From a set of inconsistent constraints in a spurious counterexample we obtain an *explanation* as an overapproximation of the minimal, inconsistent subset of these constraints. The smaller the explanation that is generated from a spurious counterexample, the greater the pruning in the subsequent search. In this way, the computation of explanations accelerates the convergence of our procedure.

Altogether, we present a method for bounded model checking over infinite-state systems that consists of:

– A reduction to the satisfiability problem for Boolean constraint formulas.
– A lazy combination of SAT solving and theorem proving.
– An efficient method for constructing small explanations.

In general, BMC over infinite-state systems is not complete, but we obtain a completeness result for BMC problems with invariant properties. The main condition on constraints is that the satisfiability of the conjunction of constraints is decidable. Thus, our BMC procedure can be applied to infinite-state systems even when the (more) general model-checking problem is undecidable.

The paper is structured as follows. In Section 2 we provide some background material on Boolean constraints. Section 3 lays the foundation of a refinement-based satisfiability procedure for Boolean constraint logic. Next, Section 4 presents the details of BMC over domain-specific constraints, and Section 5 discusses some simple examples for BMC over clock constraints and the theory of bitvectors. In Section 6 we experimentally investigate various design choices in lazy integrations of SAT solvers with theorem proving. Finally, in Sections 7 and 8 we compare with related work and we draw conclusions.

2 Background

A set of variables $V := \{x_1, \ldots, x_n\}$ is said to be typed if there are nonempty sets D_1 through D_n and a *type assignment* τ such that $\tau(x_i) = D_i$. For a set of typed variables V, a *variable assignment* is a function ν from variables $x \in V$ to an element of $\tau(x)$.

Let V be a set of typed variables and L be an associated logical language. A set of constraints in L is called a *constraint theory* $\mathcal{C}$ if it includes constants *true*, *false* and if it is closed under negation; a subset of $\mathcal{C}$ of constraints with free variables in $V' \subseteq V$ is denoted by $\mathcal{C}(V')$. For $c \in \mathcal{C}$ and ν an assignment for the free variables in c, the value of the predicate $[\![c]\!]_\nu$ is called the *interpretation* of c w.r.t. ν. Hereby, $[\![true]\!]_\nu$ ($[\![false]\!]_\nu$) is assumed to hold for all (for no) ν, and $[\![\neg c]\!]_\nu$ holds iff $[\![c]\!]_\nu$ does not hold. A set of constraints $C \subseteq \mathcal{C}$ is said to be *satisfiable* if there exists a variable assignment ν such that $[\![c]\!]_\nu$ holds for every c in C; otherwise, C is said to be *unsatisfiable*. Furthermore, a function $\mathcal{C}\text{-}sat(C)$ is called a $\mathcal{C}$-satisfiability solver if it returns $\bot$ if the set of constraints C is unsatisfiable and a satisfying assignment for C otherwise.

For a given theory $\mathcal{C}$, the set of *boolean constraints* $\mathsf{Bool}(\mathcal{C})$ includes all constraints in $\mathcal{C}$ and it is closed under conjunction $\wedge$, disjunction $\vee$, and negation $\neg$. The notions of satisfiability, inconsistency, satisfying assignment, and satisfiability solver are homomorphically lifted to the set of boolean constraints in the usual way. If $V = \{p_1, \ldots, p_n\}$ and the corresponding type assignment $\tau(p_i)$ is either true or false, then $\mathsf{Bool}(\{\mathit{true}, \mathit{false}\} \cup V)$ reduces to the usual notion of Boolean logic with propositional variables $\{p_1, \ldots, p_n\}$. We call a Boolean satisfiability solver also a SAT solver. N-ary disjunctions of constraints are also referred to as *clauses*, and a formula $\varphi \in \mathsf{Bool}(\mathcal{C}(V))$ is in *conjunctive normal form* (CNF) if it is an n-ary conjunction of clauses. There is a linear-time satisfiability-preserving transformation into CNF [22].

3 Lazy Theorem Proving

Satisfiability solvers for propositional constraint formulas can be obtained from the combination of a propositional SAT solver with decision procedures simply by converting the problem into disjunctive normal form, but the result is prohibitively expensive. Here, we lay out the foundation of a lazy combination of SAT solvers with constraint solvers based on an incremental refinement of Boolean formulas. We restrict our analysis to formulas in CNF, since most modern SAT solvers expect their input to be in this format.

Translation schemes between propositional formulas and Boolean constraint formulas are needed. Given a formula φ such a correspondence is easily obtained by abstracting constraints in φ with (fresh) propositional variables. More formally, for a formula $\varphi \in \mathsf{Bool}(\mathcal{C})$ with atoms $C = \{c_1, \ldots, c_n\} \in \mathcal{C}$ and a set of propositional variables $P = \{p_1, \ldots, p_n\}$ not occurring in φ, the mapping α from Boolean formulas over $\{c_1, \ldots, c_n\}$ to Boolean formulas over P is defined as the homomorphism induced by $\alpha(c_i) = p_i$. The inverse γ of such an abstraction mapping α simply replaces propositional variables p_i with their associated constraints c_i. For example, the formula $\varphi \equiv f(x) \neq x \wedge f(f(x)) = x$ over equalities of terms with uninterpreted function symbols determines the function α with, say, $\alpha(f(x) \neq x) = p_1$ and $\alpha(f(f(x)) = x) = p_2$; thus $\alpha(\varphi) = p_1 \wedge p_2$. Moreover, a Boolean assignment $\nu : P \to \{\mathit{true}, \mathit{false}\}$ induces a set of constraints

$$\gamma(\nu) \;\equiv\; \{c \in \mathcal{C} \,|\, \exists i.\ \text{if } \nu(p_i) = \mathit{true} \text{ then } c = \gamma(p_i) \text{ else } c = \neg\gamma(p_i)\}\ .$$

Now, given a Boolean variable assignment ν such that $\nu(p_1) = \mathit{false}$ and $\nu(p_2) = \mathit{true}$, $\gamma(\nu)$ is the set of constraints $\{f(x) = x, f(f(x)) = x\}$. A consistent set of constraints C determines a set of assignments. For choosing an arbitrary, but fixed assignment from this set, we assume as given a function $\mathit{choose}(C)$.

Theorem 1. Let $\varphi \in \mathsf{Bool}(\mathcal{C})$ be a formula in CNF, $\mathcal{L}$ be the literals in $\alpha(\varphi)$, and $I(\varphi) := \{L \subseteq \mathcal{L} \,|\, \gamma(L) \text{ is } \mathcal{C}\text{-}\mathit{inconsistent}\}$ be the set of $\mathcal{C}$-inconsistencies for φ; then: φ is *$\mathcal{C}$-satisfiable* iff the following Boolean formula is satisfiable:

$$\alpha(\varphi) \wedge (\bigwedge_{\{l_1,\ldots,l_n\} \in I(\varphi)} (\neg l_1 \vee \ldots \vee \neg l_n)).$$

```
**sat**($\varphi$)
    $p := \alpha(\varphi)$;
    **loop**
        $\nu := \mathcal{B}\text{-}sat(p)$;
        **if** $\nu = \bot$ **then return** $\bot$;
        **if** $\mathcal{C}\text{-}sat(\gamma(\nu)) \neq \bot$ **then return** $choose(\gamma(\nu))$;
        $I := \bigvee_{c \in \gamma(\nu)} \neg\alpha(c)$; $p := p \wedge I$
    **endloop**
```

Fig. 1. Lazy theorem proving for $\mathsf{Bool}(\mathcal{C})$.

Thus, every $\mathsf{Bool}(\mathcal{C})$ formula can be transformed into an equisatisfiable Boolean formula as long as the consistency problem for sets of constraints in $\mathcal{C}$ is decidable. This transformation enables one to use off-the-shelf satisfiability checkers to determine the satisfiability of Boolean constraint formulas. On the other hand, the set of literals is exponential in the number of variables and, therefore, an exponential number of *$\mathcal{C}$-inconsistency* checks is required in the worst case. It has been observed, however, that in many cases only small fragments of the set of *$\mathcal{C}$-inconsistencies* are needed.

Starting with $p = \alpha(\varphi)$, the procedure **sat**(φ) in Figure 1 realizes a guided enumeration of the set of *$\mathcal{C}$-inconsistencies*. In each loop, the SAT solver *$\mathcal{B}$-sat* suggests a candidate assignment ν for the Boolean formula p, and the satisfiability solver *$\mathcal{C}$-sat* for $\mathcal{C}$ checks whether the corresponding set of constraints $\gamma(\nu)$ is consistent. Whenever this consistency check fails, p is refined by adding a Boolean analogue I of this inconsistency, and *$\mathcal{B}$-sat* is applied to suggest a new candidate assignment for the refined formula $p \wedge I$. This procedure terminates, since, in every loop, I is not subsumed by p, and there are only a finite number of such strengthenings.

Corollary 1. **sat**(φ) in Figure 1 is a satisfiability solver for $\mathsf{Bool}(\mathcal{C})$ formulas in CNF.

We list some essential optimizations. If the variable assignments returned by the SAT solver are partial in that they include *don't care* values, then the number of argument constraints to *$\mathcal{C}$-sat* can usually be reduced considerably. The use of don't care values also speeds up convergence, since more general lemmas are generated. Now, assume a function *explain*(C), which, for an inconsistent set of constraints C, returns a minimal number of inconsistent constraints in C or a "good" overapproximation thereof. The use of *explain*(C) instead of the stronger C obviously accelerates the procedure. We experimentally analyze these efficiency issues in Section 6.

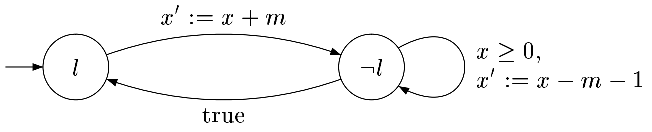

Fig. 2. The *simple* example.

4 Infinite-State BMC

Given a BMC problem for an infinite-state program, an LTL formula with constraints, and a bound on the length of counterexamples to be searched for, we describe a sound reduction to the satisfiability problem of Boolean constraint formulas and we show completeness for invariant properties. The encoding of transition relations follows the now-standard approach already taken in [13]. Whereas in [7] LTL formulas are translated directly into propositional formulas, we use Büchi automata for this encoding. This simplifies substantially the notations and the proofs, but a direct translation can sometimes be more succinct in the number of variables needed. We use the common notions for finite automata over finite and infinite words, and we assume as given a constraint theory $\mathcal{C}$ with a satisfiability solver.

Typed variables in $V := \{x_1, \ldots, x_n\}$ are also called *state variables*, and a *program state* is a variable assignment over V. A pair $\langle I, T \rangle$ is a *$\mathcal{C}$-program* over V if $I \in \mathsf{Bool}(\mathcal{C}(V))$ and $T \in \mathsf{Bool}(\mathcal{C}(V \cup V'))$, where V' is a primed, disjoint copy of V. I is used to restrict the set of initial program states, and T specifies the transition relation between states and their successor states. The set of $\mathcal{C}$-programs over V is denoted by $\mathsf{Prg}(\mathcal{C}(V))$. The semantics of a program P is given in terms of a *transition system* M in the usual way, and, by a slight abuse of notation, we sometimes write M for both the program and its associated transition system. The system depicted in Figure 2, for example, is expressed in terms of the program $\langle I, T \rangle$ over $\{x, l\}$, where the counter x is interpreted over the integers and the variable l for encoding locations is interpreted over the Booleans (the n-ary connective $\otimes$ holds iff exactly one of its arguments holds).

$$\begin{aligned} I(x, l) &:= x \geq 0 \wedge l \\ T(x, l, x', l') &:= (l \wedge x' = x + m \wedge \neg l') \otimes \\ &\quad\; (\neg l \wedge x \geq 0 \wedge x' = x - m - 1 \wedge \neg l') \otimes (\neg l \wedge x' = x \wedge l') \end{aligned}$$

Initially, the program is in location l and x is greater than or equal to 0, and the transitions in Figure 2 are encoded by a conjunction of constraints over the current state variables x, l and the next state variables x', l'.

The formulas of the *constraint linear temporal logic* $\mathsf{LTL}(\mathcal{C})$ (in negation normal form) are linear-time temporal logic formulas with the usual "next", "until", and "release" operators, and constraints $c \in \mathcal{C}$ as atoms.

$$\varphi ::= \mathit{true} \mid \mathit{false} \mid c \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathbf{X}\,\varphi \mid \varphi_1 \,\mathbf{U}\, \varphi_2 \mid \varphi_1 \,\mathbf{R}\, \varphi_2$$

The formula $\mathbf{X}\,\varphi$ holds on some path π iff φ holds in the second state of π. $\varphi_1 \,\mathbf{U}\, \varphi_2$ holds on π if there is a state on the path where φ_2 holds, and at every preceding state on the path φ_1 holds. The release operator $\mathbf{R}$ is the logical dual of $\mathbf{U}$. It requires that φ_2 holds along the path up to and including the first state, where φ_1 holds. However, φ_1 is not required to hold eventually. The derived operators $\mathbf{F}\,\varphi = \mathit{true}\,\mathbf{U}\,\varphi$ and $\mathbf{G}\,\varphi = \mathit{false}\,\mathbf{R}\,\varphi$ denote "eventually φ" and "globally φ". Given a program $M \in \mathsf{Prg}(\mathcal{C})$ and a path π in M, the satisfiability relation $M, \pi \models \varphi$ for an $\mathsf{LTL}(\mathcal{C})$ formula φ is given in the usual way with the notable exception of the case of constraint formulas c. In this case, $M, \pi \models c$ if and only if c holds in the start state of π. Assuming the notation above, the *$\mathcal{C}$-model checking problem* $M \models \varphi$ holds iff for all paths $\pi = s_0, s_1, \ldots$ in M with $s_0 \in I$ it is the case that $M, \pi \models \varphi$. Given a bound k, a program $M \in \mathsf{Prg}(\mathcal{C})$ and a formula $\varphi \in \mathsf{LTL}(\mathcal{C})$ we now consider the problem of constructing a formula $[\![M, \varphi]\!]_k \in \mathsf{Bool}(\mathcal{C})$, which is satisfiable if and only if there is a counterexample of length k for the $\mathcal{C}$-model checking problem $M \models \varphi$. This construction proceeds as follows.

1. Definition of $[\![M]\!]_k$ as the unfolding of the program M up to step k from initial states (this requires k disjoint copies of V).
2. Translation of $\neg\varphi$ into a corresponding Büchi automaton $\mathcal{B}_{\neg\varphi}$ whose language of accepting words consists of the satisfying paths of $\neg\varphi$.
3. Encoding of the transition system for $\mathcal{B}_{\neg\varphi}$ and the Büchi acceptance condition as a Boolean formula, say $[\![\mathcal{B}]\!]_k$.
4. Forming the conjunction $[\![M, \varphi]\!]_k := [\![\mathcal{B}]\!]_k \wedge [\![M]\!]_k$.
5. A satisfying assignment for the formula $[\![M, \varphi]\!]_k$ induces a counterexample of length k for the model checking problem $M \models \varphi$.

Definition 1 (Encoding of *$\mathcal{C}$-Programs*). The encoding $[\![M]\!]_k$ of the kth unfolding of a *$\mathcal{C}$-program* $M = \langle I, T \rangle$ in $\mathsf{Prg}(\mathcal{C}(\{x_1, \ldots, x_n\}))$ is given by the $\mathsf{Bool}(\mathcal{C})$ formula $[\![M]\!]_k$.

$$\begin{aligned} I_0(x[0]) &:= I\langle\{x_i \mapsto x_i[0] \mid x_i \in V\}\rangle \\ T_j(x[j], x[j+1]) &:= T\langle\{x_i \mapsto x_i[j] \mid x_i \in V\} \cup \{x'_i \mapsto x_i[j+1] \mid x_i \in V\}\rangle \\ [\![M]\!]_k &:= I_0(x[0]) \wedge \bigwedge_{j=0}^{k-1} T_j(x[j], x[j+1]) \end{aligned}$$

where $\{x_i[j] \mid 0 \leq j \leq k\}$ is a family of typed variables for encoding the state of variable x_i in the jth step, $x[j]$ is used as an abbreviation for $x_1[j], \ldots, x_n[j]$, and $T\langle x_i \mapsto x_i[j]\rangle$ denotes simultaneous substitution of x_i by $x_i[j]$ in formula T.

A two-step unfolding of the *simple* program in Figure 2 is encoded by $[\![\mathit{simple}]\!]_2 := I_0 \wedge T_0 \wedge T_1$ $(*)$.

$$\begin{aligned} I_0 &:= x[0] \geq 0 \wedge l[0] \\ T_0 &:= (\, l[0] \wedge (x[1] = x[0] + m) \wedge \neg l[1]\,) \otimes \\ &\quad\; (\, \neg l[0] \wedge (x[0] \geq 0) \wedge (x[1] = x[0] - m - 1) \wedge \neg l[1]\,) \otimes \\ &\quad\; (\, \neg l[0] \wedge (x[1] = x[0]) \wedge l[1]\,) \end{aligned}$$

$$\begin{aligned} T_1 \ :=\ & (\, l[1] \ \wedge\ (x[2] = x[1] + m) \ \wedge\ \neg l[2]\,) \otimes \\ & (\, \neg l[1] \ \wedge\ (x[1] \geq 0) \ \wedge\ (x[2] = x[1] - m - 1) \ \wedge\ \neg l[2]\,) \otimes \\ & (\, \neg l[1] \ \wedge\ (x[2] = x[1]) \ \wedge\ l[2]\,) \end{aligned}$$

The translation of linear temporal logic formulas into a corresponding Büchi automaton is well-studied in the literature [11] and does not require additional explanation. Notice, however, that the translation of $\mathsf{LTL}(\mathcal{C})$ formulas yields Büchi automata with *$\mathcal{C}$-constraints* as labels. Both the resulting transition system and the bounded acceptance test based on the detection of reachable cycles with at least one final state can easily be encoded as $\mathsf{Bool}(\mathcal{C})$ formulas.

Definition 2 (Encoding of Büchi Automata). Let $V = \{x_1, \ldots, x_n\}$ be a set of typed variables, $\mathcal{B} = \langle \Sigma, Q, \Delta, Q^0, F \rangle$ be a Büchi automaton with labels Σ in $\mathsf{Bool}(\mathcal{C})$, and pc be a variable (not in V), which is interpreted over the finite set of locations Q of the Büchi automaton. For a given integer k, we obtain, as in Definition 1, families of variables $x_i[j]$, $pc[j]$ $(1 \leq i \leq n,\ 0 \leq j \leq k)$ for representing the jth state of $\mathcal{B}$ in a run of length k. Furthermore, the transition relation of $\mathcal{B}$ is encoded in terms of the *$\mathcal{C}$-program* $\mathcal{B}_M$ over the set of variables $\{pc\} \cup V$, and $[\![\mathcal{B}_M]\!]_k$ denotes the encoding of this program as in Definition 1. Now, given an encoding of the acceptance condition

$$acc(\mathcal{B})_k \ :=\ \bigvee_{j=0}^{k-1} \Big(pc[k] = pc[j] \wedge \bigwedge_{v=1}^{n} x_v[k] = x_v[j] \wedge \Big(\bigvee_{l=j+1}^{k} \bigvee_{f \in F} pc[l] = f \Big)\Big)$$

the k-th unfolding of $\mathcal{B}$ is defined by $[\![\mathcal{B}]\!]_k := [\![\mathcal{B}_M]\!]_k \wedge acc(\mathcal{B})_k$.

An $\mathsf{LTL}(\mathcal{C})$ formula is said to be $\mathbf{R}$-free ($\mathbf{U}$-free) iff there is an equivalent formula (in negation normal form) not containing the operator $\mathbf{R}$ ($\mathbf{U}$). Note that $\mathbf{U}$-free formulas correspond to the notion of *syntactic safety formulas* [28, 15]. Now, it can be directly observed from the semantics of $\mathsf{LTL}(\mathcal{C})$ formulas that every $\mathbf{R}$-free formula can be translated into an automaton over finite words that accepts a prefix of all infinite paths satisfying the given formula.

Definition 3. Given an automaton $\mathcal{B}$ over finite words and the notation as in Definition 2, the encoding of the k-ary unfolding of $\mathcal{B}$ is given by $[\![\mathcal{B}_M]\!]_k \wedge acc(\mathcal{B})_k$ with the acceptance condition

$$acc(\mathcal{B})_k \ :=\ \bigvee_{j=0}^{k} \bigvee_{f \in F} pc[j] = f \ .$$

Consider the problem of finding a counterexample of length $k = 2$ to the hypothesis that our running example in Figure 2 satisfies $\mathbf{G}\,(x \geq 0)$. The negated property $\mathbf{F}\,(x < 0)$ is an $\mathbf{R}$-free formula, and the corresponding automaton $\mathcal{B}$ over finite words is displayed in Figure 3 (l_1 is an accepting state.). This automaton is translated, according to Definition 3, into the formula

$$[\![\mathcal{B}]\!]_2 := I(\mathcal{B}) \wedge T_0(\mathcal{B}) \wedge T_1(\mathcal{B}) \wedge acc(\mathcal{B})_2 \ . \qquad (**)$$

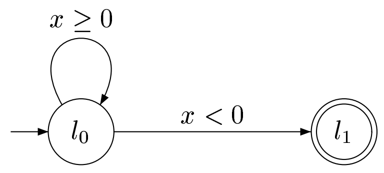

Fig. 3. Automaton for $\mathbf{F}\,(x < 0)$.

The variables $pc[j]$ and $x[j]$ ($j = 0, 1, 2$) are used to represent the first three states in a run.

$$\begin{aligned}
I(\mathcal{B}) &:= pc[0] = l_0 \\
T_0(\mathcal{B}) &:= (pc[0] = l_0 \wedge x[0] \geq 0 \wedge pc[1] = l_0) \otimes (pc[0] = l_0 \wedge x[0] < 0 \wedge pc[1] = l_1) \\
T_1(\mathcal{B}) &:= (pc[1] = l_0 \wedge x[1] \geq 0 \wedge pc[2] = l_0) \otimes (pc[1] = l_0 \wedge x[1] < 0 \wedge pc[2] = l_1) \\
acc(\mathcal{B})_2 &:= pc[0] = l_1 \vee pc[1] = l_1 \vee pc[2] = l_1
\end{aligned}$$

The bounded model checking problem $[\![simple]\!]_2 \wedge [\![\mathcal{B}]\!]_2$ for the *simple* program is obtained by conjoining the formulas $(*)$ and $(**)$. Altogether, we obtain the counterexample $(0, l) \rightarrow (m, \neg l) \rightarrow (-1, l)$ of length 2 for the property $\mathbf{G}\,(x \geq 0)$.

Theorem 2 (Soundness). Let $M \in \mathsf{Prg}(\mathcal{C})$ and $\varphi \in \mathsf{LTL}(\mathcal{C})$. If there exists a natural number k such that $[\![M, \varphi]\!]_k$ is satisfiable, then $M \not\models \varphi$.
Proof sketch. If $[\![M, \varphi]\!]_k$ is satisfiable, then so are $[\![\mathcal{B}]\!]_k$ and $[\![M]\!]_k$. From the satisfiability of $[\![\mathcal{B}]\!]_k$ it follows that there exists a path in the Büchi automaton $\mathcal{B}$ that accepts the negation of the formula φ.

In general, BMC over infinite-state systems is not complete. Consider, for example, the model checking problem $M \models \varphi$ for the program $M = \langle I, T \rangle$ over the variable $V = \{x\}$ with $I = (x = 0)$ and $T = (x' = x + 1)$ and the formula $\varphi = \mathbf{F}\,(x < 0)$. M can be seen as a one-counter automaton, where initially the value of the counter x is 0, and in every transition the value of x is incremented by 1. Obviously, it is the case that $M \not\models \varphi$, but there exists no $k \in \mathbb{N}$ such that the formula $[\![M, \varphi]\!]_k$ is satisfiable. Since $\neg\varphi$ is not an $\mathbf{R}$-free formula, the encoding of the Büchi automaton $\mathcal{B}_k$ must contain, by Definition 2, a finite accepting cycle, described by $pc[k] = pc[0] \wedge x[k] = x[0]$ or $pc[k] = pc[1] \wedge x[k] = x[1]$ etc. Such a cycle, however, does not exist, since the program M contains only one noncycling, infinite path, where the value of x increases in every step, that is $x[i + 1] = x[i] + 1$, forall $i \geq 0$.

Theorem 3 (Completeness for Finite States). Let M be a $\mathcal{C}$*-program* with a finite set of reachable states, φ be an $\mathsf{LTL}(\mathcal{C})$ formula φ, and k be a given bound; then: $M \not\models \varphi$ implies $\exists k \in \mathbb{N}.\, [\![M, \varphi]\!]_k$ is satisfiable.
Proof sketch. If $M \not\models \varphi$, then there is a path in M that falsifies the formula. Since the set of reachable states is finite, there is a finite k such that $[\![M, \varphi]\!]_k$ is satisfiable by construction.

For a $\mathbf{U}$-free formula φ, the negation $\neg\varphi$ is $\mathbf{R}$-free and can be encoded in terms of an automaton over finite words. Therefore, by considering only $\mathbf{U}$-free properties one gets completeness also for programs with an infinite set of

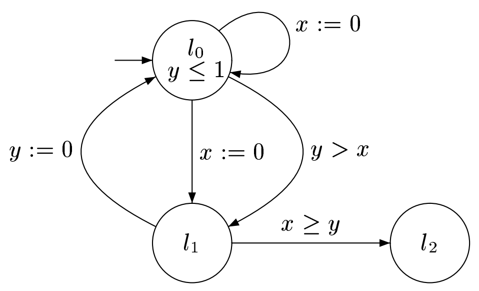

Fig. 4. Timed automata example.

reachable states. A particularly interesting class of **U**-free formulas are invariant properties.

Theorem 4 (Completeness for Syntactic Safety Formulas). Let M be a *$\mathcal{C}$-program*, $\varphi \in \mathsf{LTL}(\mathcal{C})$ be a **U**-free property, and k be some given integer bound. Then $M \not\models \varphi$ implies $\exists k \in \mathbb{N}.\, [\![M, \varphi]\!]_k$ is satisfiable.
Proof sketch. If $M \not\models \varphi$ and φ is **U**-free then there is a finite prefix of a path of M that falsifies φ. Thus, by construction of $[\![M, \varphi]\!]_k$, there is a finite k such that $[\![M, \varphi]\!]_k$ is satisfiable.

This completeness result can easily be generalized to all safety properties [15] by observing that the prefixes violated by these properties can also be accepted by an automaton on finite words.

5 Examples

We demonstrate BMC over clock constraints and the theory of bitvectors by means of some simple but, we think, illustrative examples.

The timed automaton [1] in Figure 4 has two real-valued clocks x, y, the transitions are decorated with clock constraints and clock resets, and the invariant $y \leq 1$ in location l_0 specifies that the system may stay in l_0 only as long as the value of y does not exceed 1. The transitions can easily be described in terms of a program with linear arithmetic constraints over states (pc, x, y), where pc is interpreted over the set of locations $\{l_0, l_1, l_2\}$ and the clock variables x, y are interpreted over $\mathbb{R}_0^+$. Here we show only the encoding of the time *delay* steps.

$$\begin{aligned}
&delay(pc, x, y, pc', x', y') := \\
&\quad \exists\, \delta \geq 0.\, ((pc = l_0 \Rightarrow y' \leq 1) \ \wedge\ (x' = x + \delta) \ \wedge\ (y' = y + \delta) \ \wedge\ (pc' = pc)).
\end{aligned}$$

This relation can easily be transformed into an equivalent quantifier-free formula. Now, assume the goal of falsifying the hypothesis that the timed automaton in Figure 4 satisfies the $\mathsf{LTL}(\mathcal{C})$ property $\varphi = (\mathbf{G}\, \neg l_2)$, that is, the automaton never reaches location l_2. Using the BMC procedure over linear arithmetic constraints one finds the counterexample

$$(l_0, x = 0, y = 0) \rightarrow (l_1, x = 0, y = 0) \rightarrow (l_2, x = 0, y = 0)$$

of length 2. By using Skolemization of the delay step δ instead of quantifier elimination, explicit constraints are synthesized for the corresponding delay steps in countertraces.

Now, we examine BMC over a theory $\mathcal{B}$ of bitvectors by encoding the shift register example in [3] as follows.

$$I_{BS}(x_n) := \mathit{true} \qquad T_{BS}(x_n, y_n) := (y_n \ = \ x_n[1 : n-1] \star 1_1)$$

The variables x_n and y_n are interpreted over bitvectors of length n, $x_n[1 : n-1]$ denotes extraction of bits 1 through $n-1$, $\star$ denotes concatenation, and 0_n (1_n) is the constant bitvector of length n with all bits set to zero (one). In the initial state the content of the register x_n is arbitrary. Given the $\mathsf{LTL}(\mathcal{B})$ property $\varphi \ = \ \mathbf{F}\,(x_n = 0_n)$ and $k = 2$ the corresponding BMC problem reduces to showing satisfiability of the $\mathsf{Bool}(\mathcal{B})$ formula

$$\begin{array}{l}(x_1 \ = \ x_0[1 : n-1] \star 1_1) \ \wedge \ (x_2 \ = \ x_1[1 : n-1] \star 1_1) \ \wedge \\ (x_0 \neq 0_n \vee x_1 \neq 0_n \vee x_2 \neq 0_n) \ \wedge \ (x_0 = x_2 \vee x_1 = x_2).\end{array}$$

The variables x_0, x_1, x_2 are interpreted over bitvectors of size n, since they are used to represent the first three states in a run of the shift register. The satisfiability of this formula is established by choosing all unit literals to be true. Using theory-specific canonization (rewrite) steps for the bitvector theory $\mathcal{B}$ [18], we obtain an equation between variables x_2 and x_0.

$$x_2 \ = \ x_1[1 : n-1] \star 1_1 \ = \ (x_0[1 : n-1] \star 1_1)[1 : n-1] \star 1_1 \ = \ x_0[2 : n-1] \star 1_2$$

This canonization step corresponds to a symbolic simulation of depth 2 of the synchronous circuit. Now, in case the SAT solver decides the equation $x_0 = x_2$ to be true, the bitvector decision procedures are confronted with solving the equality $x_0 \ = \ x_0[2 : n-1] \star 1_2$. The most general solution for x_0 is obtained using the solver in [18] and, by simple backsubstitution, one gets a satisfying assignment for x_0, x_1, x_2, which serves as a counterexample for the assertion that the shift register eventually is zero. The number of case splits is linear in the bound k, and, by leaving the word size uninterpreted, our procedure invalidates a family of shift registers without runtime penalties.

6 Efficiency Issues

The purpose of the experiments in this section is to identify useful concepts and techniques for obtaining efficient implementations of the lazy theorem proving approach. For these experiments we implemented several refinements of the basic lazy theorem proving algorithm from Section 3, using SAT solvers such as Chaff [19] and ICS [10] for deciding linear arithmetic constraints. These programs either returns $\perp$ in case the input Boolean constraint problem is unsatisfiable or an assignment for the variables. We describe some of our experiments using the Bakery mutual exclusion protocol (see Figure 5). Usually, the y_i counters

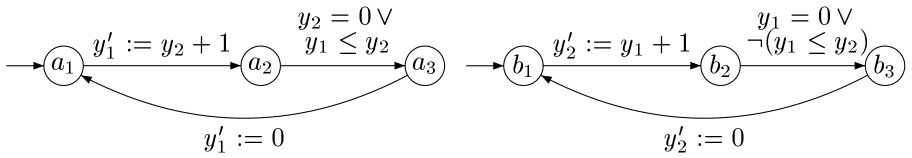

Fig. 5. Bakery Mutual Exclusion Protocol.

are initialized with 0, but here we simultaneously consider a family of Bakery algorithms by relaxing the condition on initial values of the counters to $y_1 \geq 0 \wedge y_2 \geq 0$. Our experiments represent worst-case scenarios in that the corresponding BMC problems are all unsatisfiable. Thus, unsatisfiability of the BMC formula for a given k corresponds to a verification of the mutual exclusion property for paths of length $\leq k$.

Initial experiments with a direct implementation of the refinement algorithm in Figure 1 clearly show that this approach quickly becomes impractical. We identified two main reasons for this inefficiency.

First, for the interleaving semantics of the Bakery processes, usually only a small subset of assignments is needed for establishing satisfiability. This can already be demonstrated using the *simple* example in Figure 2. Suppose a satisfying assignment ν (counterexample) corresponding to executing the transition $l \longrightarrow \neg l$ with $x' = x + m$ in the first step; that is, $[\![l[0]]\!]_\nu$, $[\![x[1] = x[0] + m]\!]_\nu$ and $[\![\neg l[1]]\!]_\nu$ hold. Clearly, the value of the literals $x[0] \geq 0$, $x[1] = x[0] - m - 1$, and $x[1] = x[0]$ are *don't cares*, since they are associated with some other transition. Overly eager assignment of truth values to these constraints results in useless search. For example, if $[\![x[1] = x[0]]\!]_\nu$ holds, then an inconsistency is detected, since $m > 0$, and $x[1] = x[0] + m = x[0]$. Consequently, the assignment ν is discarded and the search continues. To remedy the situation we analyze the structure of the formula before converting it to CNF, and use this information to assign *don't care* values to literals corresponding to unfired transitions in each step.

Second, the convergence of the refinement process must be accelerated by finding concise overapproximations *explain*(C) of the minimal set of inconsistent constraints C corresponding to a given Boolean assignment. There is an obvious trade-off between the conciseness of this approximation and the cost for computing it. We are proposing an algorithm for finding such an overapproximation based on rerunning the decision procedures $O(m \times n)$ times, where m is some given upper bound on the number of iterations (see below) and n is the number of given constraints.

The run in Figure 6 illustrates this procedure. The constraints in Figure 6.(a) are asserted to ICS from left-to-right. Since ICS detects a conflict when asserting $y_6 \leq 0$, this constraint is in the minimal inconsistent set. Now, an overapproximation of the minimal inconsistent sets is produced by connecting constraints with common variables (Figure 6.(a)). This overapproximation is iteratively refined by collecting the constraints in an array as illustrated in Figure 6.(b). Configu-

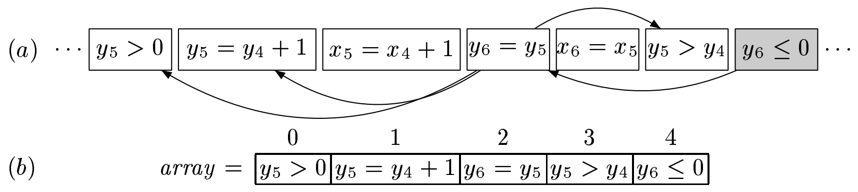

Fig. 6. Trace for linear time *explain* function.

rations consist of triples (C, l, h), where C is a set of constraints guaranteed to be in the minimal inconsistent set, and the integers l, h are the lower and upper bounds of constraint indices still under consideration. The initial configuration in our example is $(\{y_6 \leq 0\},\ 0,\ 3)$. In each refinement step, we maintain the invariant that $C \cup\ \{array[i] \mid l \leq i \leq h\}$ is inconsistent. Given a configuration (C, l, h), individual constraints of index between l and h are added to C until an inconsistency is detected. In the first iteration of our running example, we process constraints from right-to-left, and an inconsistency is only detected when processing $y_5 > 0$. The new configuration $(\{y_6 \leq 0, y_5 > 0\}, 1, 3)$ is obtained by adding this constraint to the set of constraints already known to be in a minimal inconsistent set, by leaving h unchanged, and by setting l to the increment of the index of the new constraint. The order in which constraints are asserted is inverted after each iteration. Thus, in the next step in our example, we successively add constraints between 1 and 3 from left-to-right to the set $\{y_6 \leq 0, y_5 > 0\}$. An inconsistency is first detected when asserting $y_6 = y_5$ to this set, and the new configuration is obtained as $(\{y_6 \leq 0, y_5 > 0, y_6 = y_5\}, 1, 1)$, since the lower bound l is now left unchanged and the upper bound is set to the decrement of the index of the constraint for which the inconsistency has been detected. The procedure terminates if C in the current configuration is inconsistent or after m refinements. In our example, two refinement steps yield the minimal inconsistent set $\{y_5 > 0, y_6 = y_5, y_6 \leq 0\}$. In general, the number of assertions is linear in the number of constraints, and the algorithm returns the exact minimal set if its cardinality is less than or equal to the upper bound m of iterations.

Given these refinements to the satisfiability algorithm in Figure 1, we implemented an *offline* integration of Chaff with ICS, in which the SAT solver and the decision procedures are treated as black boxes, and both procedures are restarted in each lazy refinement step. Table 1 includes some statistics for three different configurations depending on whether *don't care* processing or the linear *explain* are enabled. For each configuration, we list the total time (in seconds) and the number of conflicts detected by the decision procedure. This table indicates that the effort of assigning don't care values depending on the asynchronous nature of the program and the use of explain functions significantly improves performance.

Recall that the experiments so far represent worst-case scenarios in that the given formulas are unsatisfiable. For BMC problems with counterexamples, however, our procedure usually converges much faster. Consider, for example the mutual exclusion problem of the Bakery protocol with a guard $y_1 \geq y_2 - 1$ instead of $\neg(y_1 \leq y_2)$. The corresponding counterexample for $k = 5$ is produced

	don't cares, no explain		no don't cares, explain		don't cares, explain	
depth	time	conflicts	time	conflicts	time	conflicts
5	0.71	66	45.23	577	0.31	16
6	2.36	132	83.32	855	0.32	18
7	12.03	340	286.81	1405	1.75	58
8	56.65	710	627.90	1942	2.90	73
9	230.88	1297	1321.57	2566	8.00	105
10	985.12	2296	-	-	15.28	185
15	-	-	-	-	511.12	646

Table 1. Offline lazy theorem proving ('-' is time $\geq$ 1800 secs).

	no explain			explain		
depth	time	conflicts	calls to ICS	time	conflicts	calls to ICS
5	0.03	24	162	0.01	7	71
6	0.08	48	348	0.01	7	83
7	0.19	96	744	0.02	7	94
8	0.98	420	3426	0.05	29	461
9	2.78	936	7936	0.19	70	1205
10	8.60	2008	17567	0.26	85	1543
15	-	-	-	4.07	530	13468

Table 2. Online lazy theorem proving.

in a fraction of a second after eight refinements.

$$(a_1, k_1, b_1, k_2) \rightarrow (a_2, 1 + k_2, b_1, k_2) \rightarrow (a_3, 1 + k_2, b_1, k_2) \rightarrow (a_3, 1 + k_2, b_2, 2 + k_2) \rightarrow (a_3, 1 + k_2, b_3, 2 + k_2)$$

This counterexample actually represents a family of traces, since it is parameterized by the constants k_1 and k_2, with $k_1, k_2 \geq 0$, which have been introduced by the ICS decision procedures.

In the case of lazy theorem proving, the *offline* integration is particular expensive, since restarts implies the reconstruction of ICS logical contexts repetitively. Memoization of the decision procedure calls does not improve the situation significantly, since the assignments produced by Chaff in subsequent calls usually do not have long enough common prefixes. This observation, however, might not be generalizable, since it depends on the specific, randomized heuristics of Chaff for choosing variable assignments.

In an *online* integration, choices for propositional variable assignments are synchronized with extending the logical context of the decision procedures with the corresponding atoms. Detection of inconsistencies in the logical context of the decision procedures triggers backtracking in the search for variable assignments. Furthermore, detected inconsistencies are propagated to the propositional search engine by adding the corresponding inconsistency clause (or, using an explanation function, a good overapproximation of the minimally inconsistent set of atoms in the logical context). Since state-of-the-art SAT solvers such as Chaff are missing the necessary API for realizing such an online integration, we developed a homegrown SAT solver which has most of the features of modern SAT solvers and integrated it with ICS. The results of using this online integration for the Bakery example can be found in Table 2 for two different configurations. [1] For each configuration, we list the total time (in seconds), the number of conflicts detected by ICS, and the total number of calls to ICS. Altogether, using an explanation facility clearly pays off in that the number of refinement iterations (conflicts) is reduced considerable.

7 Related Work

There has been much recent work in reducing the satisfiability problem of Boolean formulas over the theory of equality with uninterpreted function symbols to a SAT problem [5, 12, 23] using *eager* encodings of possible instances of equality axioms. In contrast, lazy theorem proving introduces the semantics of the formula constraints *on demand* by analyzing spurious counterexamples. Also, our procedure works uniformly for much richer sets of constraint theories. It would be interesting experimentally to compare the eager and the lazy approach, but benchmark suites (e.g. www.ece.cmu.edu/~mvelev) are currently only available as encodings of Boolean satisfiability problems.

In research that is most closely related to ours, Barrett, Dill, and Stump [2] describe an integration of Chaff with CVC by abstracting the Boolean constraint formula to a propositional approximation, then incrementally refining the approximation based on diagnosing conflicts using theorem proving, and finally adding the appropriate conflict clause to the propositional approximation. This integration corresponds directly to an online integration in the lazy theorem paradigm. Their approach to generate good explanations is different from ours in that they extend CVC with a capability of abstract proofs for overapproximating minimal sets of inconsistencies. Also, optimizations based on *don't cares* are not considered in [2]. The experimental results in [2] coincide with ours in that they suggest that lazy theorem proving without explanations (there called the *naive* approach) and offline integration quickly become impractical. Using equivalence checking for pipelined microprocessors, speedups of several orders of magnitude over their earlier SVC system are obtained.

8 Conclusion

We developed a bounded model checking (BMC) procedure for infinite-state systems and linear temporal logic formulas with constraints based on a reduction to the satisfiability problem of Boolean constraint logic. This procedure is shown to be sound, and although incomplete in general, we establish completeness

[1] The differences in the number of conflicts compared to Table 1 are due to the different heuristics of the SAT solvers used.

for invariant formulas. Since BMC problems are propositionally intensive, we propose a verification technique based on a *lazy* combination of a SAT solver with a constraint solver, which introduces only the portion of the semantics of constraints that is relevant for constructing a BMC counterexample.

We identified a number of concepts necessary for obtaining efficient implementations of lazy theorem proving. The first one is specialized to BMC for asynchronous systems in that we generate partial Boolean assignments based on the structure of program for restricting the search space of the SAT solver. Second, good approximations of minimal inconsistent sets of constraints at reasonable cost are essential. The proposed any-time algorithm uses a mixture of structural dependencies between constraints and a linear number of reruns of the decision procedure for refining overapproximations. Third, offline integration and restarting the SAT solver results in repetitive work for the decision procedures. Based on these observations we realized a lazy, online integration in which the construction of partial assignments in the Boolean domain is synchronized with the construction of a corresponding logical context for the constraint solver, and inconsistencies detected by the constraint solver are immediately propagated to the Boolean domain. First experimental results are very promising, and many standard engineering can be applied to significantly improve running times.

We barely scratched the surface of possible applications. Given the rich set of possible constraints, including constraints over uninterpreted function symbols, for example, our extended BMC methods seems to be suitable for model checking open systems, where environments are only partially specified. Also, it remains to be seen if BMC based on lazy theorem proving is a viable alternative to specialized model checking algorithms such as the ones for timed automata and extensions thereof for finding bugs, or even to AI planners dealing with resource constraints and domain-specific modeling.

Acknowledgements. We would like to thank the referees for their invaluable comments for improving this paper. S. Owre, J. Rushby, and N. Shankar provided many useful inputs.

References

1. R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. *5th Symp. on Logic in Computer Science (LICS 90)*, pages 414–425, 1990.
2. C. W. Barrett, D. L. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT, 2002. To be presented at CAV 2002.
3. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zh. Symbolic model checking without BDDs. *LNCS*, 1579, 1999.
4. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
5. R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. *LNCS*, 1633:470–482, 1999.
6. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. *LNCS*, 1855:154–169, 2000.

7. E.M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
8. F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of bounded model checking in an industrial setting. *LNCS*, 2101:436–453, 2001.
9. Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Symposium on Logic in Computer Science*, pages 51–60. IEEE, 2001.
10. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonizer and Solver. *LNCS*, 2102:246–249, 2001.
11. Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
12. A. Goel, K. Sajid, H. Zhou, and A. Aziz. BDD based procedures for a theory of equality with uninterpreted functions. *LNCS*, 1427:244–255, 1998.
13. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, June 1994.
14. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. *ACM SIGPLAN Notices*, 31(1):58–70, 2002.
15. Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
16. Yassine Lachnech, Saddek Bensalem, Sergey Berezin, and Sam Owre. Incremental verification by abstraction. *LNCS*, 2031:98–112, 2001.
17. M.O. Möller, H. Rueß, and M. Sorea. Predicate abstraction for dense real-time systems. *Electronic Notes in Theoretical Computer Science*, 65(6), 2002.
18. O. Möller and H. Rueß. Solving bit-vector equations. *LNCS*, 1522:36–48, 1998.
19. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.
20. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.
21. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.
22. David A. Plaisted and Steven Greenbaum. A structure preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.
23. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. *LNCS*, 1633:455–469, 1999.
24. H. Rueß and N. Shankar. Deconstructing Shostak. In *16th Symposium on Logic in Computer Science (LICS 2001)*. IEEE Press, June 2001.
25. Vlad Rusu and Eli Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. *LNCS*, 1579:178–192, 1999.
26. H. Saïdi. Modular and incremental analysis of concurrent software systems. In *14th IEEE International Conference on Automated Software Engineering*, pages 92–101. IEEE Computer Society Press, 1999.
27. Robert Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1981.
28. A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–512, 1994.