Symbolic Model Checking using SAT procedures instead of BDDs

A. Biere[1] A. Cimatti[2] E.M. Clarke[1] M. Fujita[3] Y. Zhu[1]

February 1, 1999
CMU-CS-99-145

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted for DAC'99

[1]Computer Science Department, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, U.S.A
{Armin.Biere,Edmund.Clarke,Yunshan.Zhu}@cs.cmu.edu

[2]Istituto per la Ricerca Scientifica e Tecnolgica (IRST)
via Sommarive 18, 38055 Povo (TN), Italy
cimatti@irst.itc.it

[3]Fujitsu Laboratories of America, Inc.
595 Lawrence Expressway, Sunnyvale, CA 94086-3922
fujita@fla.fujitsu.com

This research is sponsered by the National Science Foundation (NSF) under Grant No. CCR-9505472.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or the United States Government.

Keywords: hardware verification, out-of-order execution, temporal logic, symbolic model checking, boolean satisfiability

Abstract

In this paper, we study the application of propositional decision procedures in hardware verification. We introduce the concept of bounded model checking. We show that bounded model checking for linear temporal logic formulas can be reduced to propositional satisfiability. We also present several optimizations that reduce the size of generated propositional formulas. To demonstrate our approach, we have implemented a tool **BMC**. **BMC** accepts a subset of the SMV language and uses state of the art SAT procedures to decide propositional satisfiability. As special cases, equivalence checking and invariant checking can also be handled. In many instances, our SAT-based approach can significantly outperform BDD-based approaches. We observe that SAT-based techniques are particularly efficient in detecting errors in both combinational and sequential designs.

1 Introduction

A complex hardware design can be error-prone and mistakes are costly. Formal verification techniques such as symbolic model checking are gaining wide industrial acceptance. Compared to traditional validation techniques based on simulation, they provide more extensive coverage and can detect subtle errors. Representing and manipulating boolean expressions is critical to many formal verification techniques. BDDs [2] have traditionally been used for this purpose. In this paper, we investigate an alternative approach based on propositional decision procedures.

Model checking [4] is an important technique for verifying sequential designs. In model checking, the specification of a design is expressed in temporal logic and the implementation is described as a finite state machine. Symbolic model checking uses boolean encoding to represent the finite state machine. By replacing explicit state representation with boolean encoding, symbolic model checking [3, 11] can handle much larger designs than explicit state model checking.

By introducing the concept of bounded model checking, we are able to use efficient propositional decision procedures for symbolic model checking. In bounded model checking, only paths of bounded length k are considered. Bounded model checking is thus concerned with finding bugs (or counterexamples) of limited length k. Given a specification in temporal logic and a finite state machine, we construct a propositional formula which is satisfiable iff there is a counterexample of length k. In practice, we look for longer and longer counterexamples by incrementing the bound k, and after a certain number of iterations, we may conclude that no counterexample exists and the specification holds. For example, to verify safety properties, the number of iterations is bounded by the diameter of the finite state machine.

There are known tradeoffs between SAT procedures and BDDs. These tradeoffs are also reflected in SAT-based model checkers and BDD-based model checkers. In particular, BDDs are canonical representations. Once the BDDs are constructed, operations on two boolean expressions can be done very efficiently. On the other hand, by not using a canonical representation, SAT-based model checkers avoid the exponential space blowup of BDDs. They can detect a counterexample without searching through the entire state space. BDD-based approaches often require a good variable ordering. The ordering is either manually generated or by dynamic variable reordering which can be time consuming. In SAT-based model checkers, automatic splitting heuristics are often sufficient. BDDs require a uniform variable ordering. SAT procedures allow different splitting orderings on different branches. This often leads to more efficient search. In bounded model checking, the propositional formula encodes the constraints from the initial state and the specification. Both these constraints can be used to prune the search.

Invariant checking and equivalence checking can both be treated as special cases of bounded model checking. It can be easily shown that invariant checking corresponds to bounded model checking where the bound k equals 1. Equivalence checking is a special case of bounded model checking where the bound k equals 0. The tradeoffs mentioned earlier are also reflected in SAT-based invariant checking and equivalence checking techniques.

We have implemented a tool **BMC** to demonstate our approach. It accepts a subset of the SMV language in which the user can specify a finite state machine and a temporal specification. Given a bound k, **BMC** outputs a propositional formula which is satisfiable iff there is a counterexample of length k. Currently, we use SATO [17], an efficient implementation of the Davis-Putnum technique, and PROVER [1] based on Stålmarck's Method [16] to decide propositional satisfiability. **BMC** can output propositional formulas in either DIMACS format [8] or PROVER format. If a counterexample exists, SATO or PROVER generates a model of the propositional formula produced by **BMC**. We also have developed a script that translates the model back to a sequence of state transitions. We have run a number of examples using **BMC**. We show cases where **BMC** detected a counterexample in seconds where BDD-based approaches failed due to memory limits.

The paper is organized as follows. In the following section, we present the concept of bounded model checking and show the reduction of bounded model checking to propositional satisfiability. In section 3, we present a number of optimization techniques in generating propositional formulas. They help to reduce the complexity of the propositional formula generated by **BMC**. In section 4, we show some experimental results. We have tested **BMC** on a number of examples from symbolic model checking, invariant checking and equivalence checking. Finally, we conclude the paper with some directions for future work.

2 Bounded model checking

We now present our techniques for bounded model checking. First, we give some background and notational conventions that will be used in the rest of the paper. Then we illustrate our approach with a simple example. Finally, we show the reduction of bounded model checking to propositional satisfiability for LTL formulas in general.

2.1 Background

The specification of a system is expressed in linear temporal logic (LTL). We consider the *next time* operator '**X**', the *eventuality* operator '**F**', the *globally* operator '**G**', the *until* operator '**U**', and the *release* operator '**R**'. To simplify our discussion, we consider only existential LTL formulas, i.e. formulas of type $\mathbf{E}f$ where $\mathbf{E}$ is the existential path quantifier and f is a temporal formula that contains no path quantifiers. Note that $\mathbf{E}$ is the dual of the universal path quantifier $\mathbf{A}$. Finding a witness for $\mathbf{E}f$ is equivalent to finding a counterexample for $\mathbf{A}\neg f$.

The implementation of a system is described as a Kripke structure. A *Kripke structure* is a tuple $M = (S, I, T, \ell)$ with a finite set of states S, the set of initial states $I \subseteq S$, a transition relation between states $T \subseteq S \times S$, and the labeling of the states $\ell: S \to \mathcal{P}(\mathcal{A})$ with *atomic propositions* $\mathcal{A}$. In symbolic model checking, we assume that $S = \{0,1\}^n$ and each state can be represented by a vector of state variables $s = (s(1), \ldots, s(n))$ where $s(i)$ for $i = 1, \ldots, n$ are propositional variables. We define propositional formulas $f_I(s)$, $f_T(s,t)$ and $f_p(s)$ as follows: $f_I(s)$ iff $s \in I$, $f_T(s,t)$ iff $(s,t) \in T$, and $f_p(s)$ iff $p \in \ell(s)$. For the rest of the paper we simply use $T(s,t)$ instead of $f_T(s,t)$ etc. In addition, we require that every state has a successor state. That is, for all $s \in S$ there is a $t \in S$ with $(s,t) \in T$. For $(s,t) \in T$ we also write $s \to t$. For an infinite sequence of states $\pi = (s_0, s_1, \ldots)$ we define $\pi(i) = s_i$ and $\pi^i = (s_i, s_{i+1}, \ldots)$ for $i \in \mathbb{N}$. An infinite sequence of states π is a *path* if $\pi(i) \to \pi(i+1)$ for all $i \in \mathbb{N}$.

An LTL formula $\mathbf{E}f$ is true in a Kripke structure M ($M \models \mathbf{E}f$) iff there exists a path π in M with $\pi \models f$ and $\pi(0) \in I$. Model checking is concerned with the problem of determining the truth value of an LTL formula in a given Kripke structure, or equivalently, the problem of determining the existence of a witness for the LTL formula. We now illustrate bounded model checking with a simple example.

2.2 Example

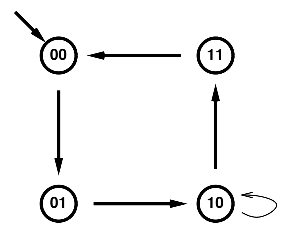

Fig. 1. A two-bit counter with an erroneous transition

Let's consider a two-bit counter. The implementation of the counter is shown as a Kripke structure in Figure 1. There are four states in the Kripke structure. Each state s is represented by two state variables $s[1]$ and $s[0]$, denoting the value of the high bit and the low bit respectively. In the initial state, the value of the counter is 0. Thus the initial state predicate $I(s)$ is defined as $\neg s[1] \wedge \neg s[0]$. The transition relation $T(s,s')$ describes the increment of the counter at each step. We define $inc(s,s')$ as $(s'[0] \leftrightarrow \neg s[0]) \wedge (s'[1] \leftrightarrow (s[0] \wedge s[1]))$, and we define $T(s,s')$ as $inc(s,s') \vee (s[1] \wedge \neg s[0] \wedge s'[1] \wedge \neg s'[0])$. Note that we deliberately add an erroneous transition from state (10) to itself.

Suppose we are interested in the fact that the counter should eventually reach state (11). We can specify the property as $\mathbf{AF}q$, where $q(s)$ is defined as $s[1] \wedge s[0]$. Namely, for all possible execution paths, there exists a state such that $q(s)$ holds. Equivalently, we can check whether there exists a path in which the counter never reaches state (11). The new property is expressed as $\mathbf{EG}p$, where $p(s)$ is defined as $\neg s[1] \vee \neg s[0]$. Note that $\mathbf{EG}p$ is the dual of $\mathbf{AF}q$.

In bounded model checking, we restrict our attention to paths of length k, that is, paths with $k+1$ states. We start with $k = 0$, and increment k until a witness is found. Let's consider the case where k equals 2. We name the $k+1$ states as s_0, s_1, s_2. We now formulate a set of constraints on s_0, s_1 and s_2 in propositional logic. The constraints guarantee that a path consisting of s_0, s_1, s_2 is indeed a witness of $\mathbf{EG}p$, or equivalently, a counterexample for $\mathbf{AF}q$.

First, we constrain s_0, s_1, s_2 to be a valid path starting from the initial state. Unrolling the transition relation for 2 steps, we derive the propositional formula $[\![\, M \,]\!]$ defined as

$I(s_0) \wedge T(s_0,s_1) \wedge T(s_1,s_2)$, where I and T are predicates for the initial state and the transition relation defined earlier.

Second, we constraint the shape of the path. The sequence of states s_0, s_1, s_2 can be a loop. If so, there is a transition from s_2 to the initial state s_0, s_1 or itself. We use ${}_lL$ defined as $T(s_2, s_l)$ to denote the transition from s_2 to a state s_l where $l \in [0,2]$. To be consistent with the general translation in the next section, we use left subscript in ${}_lL$. We define L as $\bigvee_{l=0}^{2} {}_lL$. Thus $\neg L$ denotes the case where no loop exists.

We further constrain that the specified temporal property $\mathbf{G}p$ holds on the given path s_0, s_1, s_2. In order to be a witness for $\mathbf{G}p$, the path must contain a loop. This constraint has been formulated as L. In addition, property p must hold on every state of the path. We derive a corresponding propositional formula $[\![\, \mathbf{G}p \,]\!]$ defined as $p(s_0) \wedge p(s_1) \wedge p(s_2)$. In the case where no loop exists, $\mathbf{G}p$ does not hold and $[\![\, \mathbf{G}p \,]\!]$ is defined as $false$. Finally, we combine all constraints.

$$[\![\, M \,]\!] \wedge ((\neg L \wedge false) \vee \bigvee_{l=0}^{2} ({}_lL \wedge [\![\, \mathbf{G}p \,]\!])) \tag{1}$$

In general, the constraint imposed by the temporal specification depends on the configuration of the loop. Thus in the formula (1), we put $[\![\, \mathbf{G}p \,]\!]$ within the scope of the disjunction over l. For our particular example the constraint $[\![\, \mathbf{G}p \,]\!]$ is the same for all loop configurations.

In this example, the formula is indeed satisfiable. The satisfying assignment corresponds to a counterexample that is a path from the initial state (00) over (01) to (10) followed by the self-loop at state (10). If the erroneous transition from state (10) to itself is removed then formula (1) becomes unsatisfiable.

2.3 Translation

Given a Kripke structure M, an LTL formula f and a bound k, we will construct a propositional formula $[\![\, M, f \,]\!]_k$. The variables $s_0, \ldots, s_k$ in $[\![\, M, f \,]\!]_k$ denote a finite sequence of states on a path π. Each s_i is a vector of state variables. The formula $[\![\, M, f \,]\!]_k$ represents constraints on $s_0, \ldots, s_k$ such that $[\![\, M, f \,]\!]_k$ is satisfiable iff f is valid along π. To construct $[\![\, M, f \,]\!]_k$, we first define a propositional formula $[\![\, M \,]\!]_k$ that constrains $s_0, \ldots, s_k$ to be on a valid path π in M. Second, we give the translation of an LTL formula f to a propositional formula that constrains π to satisfy f.

Definition 1 (Unfolding the Transition Relation). *For a Kripke structure M, $k \in \mathbb{N}$*

$$[\![\, M \,]\!]_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

Depending on whether a path is a k-loop or not (see Figure 2), we have two different translations of the temporal formula f. In Definition 2 we describe the translation if the path is not a loop. The translation "$[\![\, \cdot \,]\!]_k^i$" maps an LTL formula into a propositional formula. The parameter k is the length of the prefix of the path that we consider and

(a) no loop (b) (k,l)-loop

Fig. 2. The two cases for a *bounded* path.

i is the current position in this prefix (see Figure 2(a)). When we recursively process subformulas, i changes but k stays the same.

Consider the formula $h := p\,\mathbf{U}\,q$ and a path π that is not a k-loop for a given $k \in \mathbb{N}$ (see Figure 2(a)). Starting at π^i for $i \in \mathbb{N}$ with $i \leq k$ the formula h is valid along π^i with respect to the bounded semantics iff there is a position j with $i \leq j \leq k$ and q holds at $\pi(j)$. In addition, for all states $\pi(n)$ with $n \in \mathbb{N}$ starting at $\pi(i)$ up to $\pi(j-1)$ the proposition p has to be fulfilled. Therefore the translation is simply a disjunction over all possible positions j at which q eventually might hold. For each of these positions a conjunction is added that ensures that p holds along the path from $\pi(i)$ to $\pi(j-1)$. Similar reasoning leads to the translation of the other temporal operators.

Definition 2 (Translation of an LTL Formula without a Loop). *For an LTL formula f and $k, i \in \mathbb{N}$, with $i \leq k$*

$$
\begin{array}{rcl@{\qquad}rcl}
[\![\, p \,]\!]_k^i & := & p(s_i) & [\![\, \neg p \,]\!]_k^i & := & \neg p(s_i) \\
[\![\, f \wedge g \,]\!]_k^i & := & [\![\, f \,]\!]_k^i \wedge [\![\, g \,]\!]_k^i & [\![\, f \vee g \,]\!]_k^i & := & [\![\, f \,]\!]_k^i \vee [\![\, g \,]\!]_k^i \\
[\![\, \mathbf{G} f \,]\!]_k^i & := & \mathsf{false} & [\![\, \mathbf{F} f \,]\!]_k^i & := & \bigvee_{j=i}^{k} [\![\, f \,]\!]_k^j \\
[\![\, \mathbf{X} f \,]\!]_k^i & := & \mathit{if}\ i < k\ \mathit{then}\ [\![\, f \,]\!]_k^{i+1}\ \mathit{else}\ \mathsf{false} & & & \\
[\![\, f\,\mathbf{U}\,g \,]\!]_k^i & := & \bigvee_{j=i}^{k} \left([\![\, g \,]\!]_k^j \wedge \bigwedge_{n=i}^{j-1} [\![\, f \,]\!]_k^n \right) & & & \\
[\![\, f\,\mathbf{R}\,g \,]\!]_k^i & := & \bigvee_{j=i}^{k} \left([\![\, f \,]\!]_k^j \wedge \bigwedge_{n=i}^{j} [\![\, g \,]\!]_k^n \right) & & &
\end{array}
$$

Now we consider the case where the path is a k-loop. The translation "${}_l[\![\, \cdot \,]\!]_k^i$" of an LTL formula depends on the current position i and on the length of the prefix k. It also depends on the position where the loop starts (see Figure 2(b)). This position is denoted by l for *l*oop.

Definition 3 (Successor in a Loop). *Let $k, l, i \in \mathbb{N}$, with $l, i \leq k$. Define the successor* $\mathrm{succ}(i)$ *of i in a (k,l)-loop as* $\mathrm{succ}(i) := i+1$ *for $i < k$ and* $\mathrm{succ}(i) := l$ *for $i = k$.*

Definition 4 (Translation of an LTL Formula for a Loop). *Let f be an LTL formula, $k, l, i \in \mathbb{N}$, with $l, i \leq k$.*

$$
\begin{array}{rclrcl}
{}_l[\![\, p \,]\!]_k^i & := & p(s_i) & {}_l[\![\, \neg p \,]\!]_k^i & := & \neg p(s_i) \\
{}_l[\![\, f \wedge g \,]\!]_k^i & := & {}_l[\![\, f \,]\!]_k^i \wedge {}_l[\![\, g \,]\!]_k^i & {}_l[\![\, f \vee g \,]\!]_k^i & := & {}_l[\![\, f \,]\!]_k^i \vee {}_l[\![\, g \,]\!]_k^i \\
{}_l[\![\, \mathbf{G} f \,]\!]_k^i & := & \bigwedge_{j=\min(i,l)}^{k} {}_l[\![\, f \,]\!]_k^j & {}_l[\![\, \mathbf{F} f \,]\!]_k^i & := & \bigvee_{j=\min(i,l)}^{k} {}_l[\![\, f \,]\!]_k^j \\
{}_l[\![\, \mathbf{X} f \,]\!]_k^i & := & {}_l[\![\, f \,]\!]_k^{\mathrm{succ}(i)} & & & \\
{}_l[\![\, f \,\mathbf{U}\, g \,]\!]_k^i & := & \bigvee_{j=i}^{k} \left({}_l[\![\, g \,]\!]_k^j \wedge \bigwedge_{n=i}^{j-1} {}_l[\![\, f \,]\!]_k^n \right) \vee & & & \\
& & \bigvee_{j=l}^{i-1} \left({}_l[\![\, g \,]\!]_k^j \wedge \bigwedge_{n=i}^{k} {}_l[\![\, f \,]\!]_k^n \wedge \bigwedge_{n=l}^{j-1} {}_l[\![\, f \,]\!]_k^n \right) & & & \\
{}_l[\![\, f \,\mathbf{R}\, g \,]\!]_k^i & := & \bigwedge_{j=\min(i,l)}^{k} {}_l[\![\, g \,]\!]_k^j \vee & & & \\
& & \bigvee_{j=i}^{k} \left({}_l[\![\, f \,]\!]_k^j \wedge \bigwedge_{n=i}^{j} {}_l[\![\, g \,]\!]_k^n \right) \vee & & & \\
& & \bigvee_{j=l}^{i-1} \left({}_l[\![\, f \,]\!]_k^j \wedge \bigwedge_{n=i}^{k} {}_l[\![\, g \,]\!]_k^n \wedge \bigwedge_{n=l}^{j} {}_l[\![\, g \,]\!]_k^n \right) & & &
\end{array}
$$

The translation of the formula depends on the shape of the path (whether it is a loop or not). We now define a loop condition to distinguish these cases.

Definition 5 (Loop Condition). *For $k, l \in \mathbb{N}$, let ${}_l\mathrm{L}_k := T(s_k, s_l)$, $\mathrm{L}_k := \bigvee_{l=0}^{k} {}_l\mathrm{L}_k$*

Definition 6 (General Translation). *Let f be an LTL formula, M a Kripke structure and $k \in \mathbb{N}$*

$$
[\![\, M, f \,]\!]_k := [\![\, M \,]\!]_k \wedge \left(\left(\neg \mathrm{L}_k \wedge [\![\, f \,]\!]_k^0 \right) \vee \bigvee_{l=0}^{k} \left({}_l\mathrm{L}_k \wedge {}_l[\![\, f \,]\!]_k^0 \right) \right)
$$

The left side of the disjunction is the case where there is no back loop and the translation without a loop is used. On the right side all possible start positions l of a loop are tried and the translation for a (k,l)-loop is conjuncted with the corresponding ${}_lL_k$ loop condition. The following theorem shows the correctness of our translation.

Theorem 1. *$M \models \mathbf{E} f$ iff $[\![\, M, f \,]\!]_k$ is satisfiable for some $k \in \mathbb{N}$.*

3 Conversion to CNF

Many propositional decision procedures assume the input problem to be in conjunctive normal form. In this section, we focus on techniques for converting arbitrary boolean formulas to conjunctive normal form. In particular, we investigate optimization techniques that reduce the number of variables and clauses in the CNF generated. Satisfiability test for propositional problems is NP-complete. All known propositional decision procedures are exponential in the worst case. However, they may use different heuristics in guiding their search and exhibit different complexity in subsets of the propositional problems. Precise characterizations of the "hardness" of propositional problems is difficult and is likely to be dependent on specific propositional decision procedures used.

Reducing the size of CNF may not always reduce the complexity of the problem. Our optimization techniques are heuristics in nature as well. Experimental results show that these optimization techniques reduce the size of the CNF as well as the time for satisfiability test.

A formula f in conjunctive normal form is represented as a set of clauses, each clause is a set of literals, and each literal is either a positive or negative propositional variable. In other words, a formula is a conjunction of clauses, and a clause is a disjunction of literals. For example, $((a \vee \neg b \vee c) \wedge (d \vee \neg e))$ is represented as $\{\{a, \neg b, c\}, \{d, \neg e\}\}$. Conjunctive normal form is also referred to as clause form.

Given a boolean formula f, one may replace boolean operators in f with $\neg, \wedge$ and $\vee$ and apply distributivity rule and De Morgan's law to convert f into its conjunctive normal form f_{CNF}. The size of f_{CNF} can be exponential with respect to the size of f. For example, the worse case occurs when f is in disjunctive normal form. To avoid the exponential explosion, we use a structure preserving clause form transformation [14].

```
procedure bool-to-cnf(f,v_f)
{
  if (cached(f,v)) return(clause(v_f ↔ v));
  case
     atomic(f) : return(clause(f ↔ v_f));
     f == h ∘ g:
        C_1 = bool-to-cnf(h,v_h);
        C_2 = bool-to-cnf(g,v_g);
        assert(cached(f,v_f));
        return(clause(v_f ↔ v_h ∘ v_g) ∪ C_1 ∪ C_2);
  easc;
}
```

Fig. 3. An algorithm for generating conjunctive normal form. f, g and h are boolean formulas. v, v_h and v_g are boolean variables. '$\circ$' represents a boolean operator.

Figure 3 outlines our procedure. Given a boolean formula f, *bool-to-cnf(*f*,*true*)* returns a set of clauses C which is satisfiable iff f is satisfiable. The procedure traverses the syntactical structure of f, introduces a new variable (e.g. v_h, v_g) for each subexpression, and generates clauses that relate the new variables. If u and v are boolean variables, $u \leftrightarrow v$ is equivalent to $\{\{\neg u, v\}, \{u, \neg v\}\}$. If v, v_h, v_g are boolean variables and '$\circ$' is a boolean operator, $v \leftrightarrow (v_h \circ v_g)$ has a logically equivalent clause form with no more than 4 clauses, each of which contains no more than 3 literals. Note that C is not logically equivalent to the original formula f, but it preserves the satisfiability of f.

We represent a boolean formula f as a directed acyclic graph (DAG), i.e., common subterms of f are shared. The DAG representation is important in practice. For example, the size of formula $inc(a)$ is linear with a DAG representation, and is quadratic otherwise. In the procedure *bool-to-cnf()*, we preserve the sharing of subterms. Namely, for each subterm in f, only one set of clauses is generated. The sharing is reflected in line 1 of *bool-to-cnf*. For any boolean formula f, *bool-to-cnf(*f*,*true*)* generates a clause set C with $O(|f|)$ variables and $O(|f|)$ clauses, where $|f|$ is the size of DAG for f.

In Figure 3, we assume that f only involves binary operators. Unary operator, i.e. negation, can be handled similarly. We also extended the procedure to handle operators with multiple operands. In particular, we treat conjunction and disjunction as N-ary operators. For example, let us assume that v_f represents the formula $\bigwedge_{i=0}^{n} t_i$. The clause form for $v_f \leftrightarrow \bigwedge_{i=0}^{n} t_i$ is $\{\{\neg v_f, t_0\}, \{\neg v_f, t_1\}, \ldots, \{\neg v_f, t_n\}, \{v_f, \neg t_0, \ldots, \neg t_n\}\}$. If we treat $\wedge$ as a binary operator, we need to introduce $n-1$ new variables for the subterms in $\bigwedge_{i=0}^{n} t_i$. For instance, with this optimization, the comparison between two 16 bit registers r and s occurring as a subformula, $\bigwedge_{i=0}^{15}(r[i] \leftrightarrow s[i])$, can be converted into clause form without introducing new variables.

4 Experimental Results

We have implemented a model checker **BMC** based on bounded model checking. Its input language is a subset of the SMV language [11]. It outputs a propositional formula. Two different formats for the propositional formula are supported. The first format is the DIMACS format [8] for satisfiability problems. The SATO tool [17] is an efficient implementation of the Davis & Putnam Procedure [6] and it uses the DIMACS format. We also support the input format of the PROVER Tool [1] which is based on Stålmarck's Method [16]. As comparisons, we use the official version of the CMU model checker SMV and a version by Bwolen Yang from CMU with improved support for conjunctive partitioning. We refer to them as $\text{SMV}_1$and SMV_2respectively.

4.1 Model Checking

As benchmarks we chose examples that are difficult for BDD-based approaches. First we investigated a sequential multiplier, the shift and add multiplier of [5]. We formulated as *model checking* problem the following property: when the sequential multiplier is finished its output is the same as the output of a combinational multiplier (the C6288 circuit from the ISCAS'85 benchmarks) applied to the same input words. These multipliers are 16x16 bit multipliers but we only allowed 16 output bits as in [5] together with an overflow bit. We proved the property for each output bit individually and the results are shown in Table 1. Note that the overflow bit depends on all the bits of the sequential multiplier and occurs in the specification. Thus, the cone of influence reduction could not remove anything. For BDD-based model checkers, we used a manually chosen variable ordering where the bits of registers are interleaved. Dynamic reordering failed to find a considerably better ordering in a reasonable amount of time.

In [10] an asynchronous circuit for distributed mutual exclusion is described. It consists of n cells for n users that want to have exclusive access to a shared resource. We proved the liveness property that a request for using the resource will eventually be acknowledged. This liveness property is only true if each asynchronous gate does not delay execution indefinitely. We model this assumption by a fairness constraint for each individual gate. Each cell has exactly 18 gates and therefore the model has $n \cdot 18$ fairness constraints where n is the number of cells. Since we do not have a bound for the maximal length of a counterexample for the verification of this circuit we could not verify the liveness property completely. We only showed that there are no counterexamples of

	SMV_1		SMV_2		SATO		PROVER	
bit	sec	MB	sec	MB	sec	MB	sec	MB
0	919	13	25	79	0	0	0	1
1	1978	13	25	79	0	0	0	1
2	2916	13	26	80	0	0	0	1
3	4744	13	27	82	0	0	1	2
4	6580	15	33	92	2	0	1	2
5	10803	25	67	102	12	0	1	2
6	43983	73	258	172	55	0	2	2
7	>17h		1741	492	209	0	7	3
8				>1GB	473	0	29	3
9					856	1	58	3
10					1837	1	91	3
11					2367	1	125	3
12					3830	1	156	4
13					5128	1	186	4
14					4752	1	226	4
15					4449	1	183	5
sum	71923		2202		23970		1066	

Table 1. 16x16 bit sequential shift and add multiplier with overflow flag and 16 output bits (sec = seconds, MB = Mega Byte).

particular length k. To illustrate the performance of bounded model checking we have chosen $k = 5, 10$. The results can be found in Table 2.

We repeated the experiment with a buggy design. For the liveness property we simply removed several fairness constraints. Both PROVER and SATO generate a counterexample (a 2-loop) instantly (see Table 3).

4.2 Invariant Checking

Safety properties can be verified by providing an *inductive* invariant that has to hold at the initial state, is preserved by the transition relation and implies the safety property [7]. These three conditions can all be formulated as propositional satisfiability problems and verified by a propositional decision procedure. We implemented this approach in the tool **BMC** as follows. The user formulates the model as usual and specifies the invariant as a safety property (with **AG**). Then **BMC** generates two instances of a satisfiability problem. One formula for checking that the invariant is preserved by the transition relation and another formula for checking that the invariant holds initially. The third condition has to be formulated by the user.

As an example for this technique we verified that two different implementations of a queue of a particular length behave the same. This example is taken from [12] and it is known that no variable ordering exists such that the (RO)BDDs for the set of reachable states remain small. In columns SMV_1 and SMV_2, we used two versions of SMV to verify the safety property that the outputs of the two queues are always the same. In the other experiments of Table 4 an invariant was used that relates the contents of the

	SMV_1		SMV_2		SATO $k=5$		PROVER $k=5$		SATO $k=10$		PROVER $k=10$	
cells	sec	MB	sec	MB	sec	MB	sec	MB	sec	MB	sec	MB
4	846	11	159	217	0	3	1	3	3	6	54	5
5	2166	15	530	703	0	4	2	3	9	8	95	5
6	4857	18	1762	703	0	4	3	3	7	9	149	6
7	9985	24	6563	833	0	5	4	4	15	10	224	8
8	19595	31		$>$1GB	1	6	6	5	16	12	323	8
9	$>$10h				1	6	9	5	24	13	444	9
10					1	7	10	5	36	15	614	10
11					1	8	13	6	38	16	820	11
12					1	9	16	6	40	18	1044	11
13					1	9	19	8	107	19	1317	12
14					1	10	22	8	70	21	1634	14
15					1	11	27	8	168	22	1992	15

Table 2. Liveness for one user in the DME (sec = seconds, MB = Mega Bytes).

	SMV_1		SMV_2		SATO		PROVER	
cells	sec	MB	sec	MB	sec	MB	sec	MB
4	799	11	14	44	0	1	0	2
5	1661	14	24	57	0	1	0	2
6	3155	21	40	76	0	1	0	2
7	5622	38	74	137	0	1	0	2
8	9449	73	118	217	0	1	0	2
9	segmentation		172	220	0	1	1	2
10	fault		244	702	0	1	0	3
11			413	702	0	1	0	3
12			719	702	0	2	1	3
13			843	702	0	2	1	3
14			1060	702	0	2	1	3
15			1429	702	0	2	1	3

Table 3. Counterexample for liveness in a buggy DME (sec = seconds, MB = Mega Bytes).

two queues. As discussed above, three conditions have to be verified for each particular length of the queues. Beside propositional decisions procedures (see columns SATO and PROVER in Table 4), we also used model checking, similar to [7], to prove their correctness (see columns SMV_3 and SMV_4).

These experiments indicate that invariant checking can handle larger designs than traditional fixpoint computations. This result also applies to BDD-based approaches but the real potential of invariant checking becomes apparent when used in combination with propositional decision procedures.

	SMV_1		SMV_2		SMV_3		SMV_4		SATO		PROVER	
L	sec	MB	sec	MB	sec	MB	sec	MB	sec	MB	sec	MB
12	18	10	4	55	11	17	7	51	60	7	9	2
13	44	13	6	60	29	20	11	56	68	8	11	2
14	109	19	11	70	37	27	20	65	287	12	15	2
15	291	31	18	86	82	40	36	81	102	10	19	2
16	711	55	43	196	207	66	80	197	411	6	6	2
17	2126	102	159	393	573	119	191	393	1701	16	45	3
18	6103	195	459	753	1857	223	422	754	302	14	58	3
19	23405	383	1491	920	5765	430	1101	817	1551	20	70	3
20	$>$17h			$>$1GB	30809	845	9136	977	1377	20	86	3
21						$>$1GB		$>$1GB	$>$40h		99	3
22											120	3
23											149	4
24											167	4

Table 4. Comparison between queues (L = length of queues, SMV_3 = SMV_1 with invariant checking, SMV_4 = SMV_2 with invariant checking, MB = Mega Byte, sec = seconds). In the case of invariant checking the accumulated time and the maximal memory requirements are shown.

4.3 Equivalence Checking

Recently, there has been a lot of progress in boolean equivalence checking[9, 13]. State-of-the-art equivalence checkers can handle designs with more than 1 million gates. These tools utilize the correspondence between internal signals and partition large circuits into much smaller ones. If the two circuits to be compared have significantly different structures, equivalence checkers can perform poorly, even on much smaller designs (less than 10K gates). Most equivalence checkers are BDD-based. We have investigated how propositional decision procedures (SAT procedures) can be used instead of BDDs for checking equivalence.

To determine if two given circuits are equivalent, we use **BMC** to convert the equivalence checking problem to a propositional satisfiability problem. The output of **BMC** is a formula in CNF which is fed into SATO. We observed that SATO can verify almost all designs with less than 10K gates, even if the two circuits are significantly different. In Table 5, we list some industrial circuits that cannot be processed by state-of-the-art equivalence checkers (based on BDDs and similarity of the two circuits) but that can be verified by SATO. In all cases, state-of-the-art equivalence checkers cannot finish within one day.

Circuit	#ins	#outs	#gates	sec
Industry1	203	8	738	233
Industry2	317	232	15242	8790
Industry3	96	32	1032	210

Table 5. Equivalence checking using SAT procedures (sec = seconds).

In our first example, Industry1, the logic of one circuit has been considerably optimized and the other is unoptimized. The structure of the two circuits is quite different and BDDs cannot be built for them because of their complex logic functionality. SATO finishes the verification in four minutes.

In the second case, Industry2, because of the size and the dissimilarity of the two circuits, we never expected the verification to finish. The result suggests that efficient SAT procedures have real potential in handling hard equivalence checking problems. For both Industry1 and Industry2, we applied logic optimization using SIS [15] on the circuits before submitting them for equivalence checking. This extra step of logic optimization greatly speeds up our verification. Without it, Industry1 takes 8246 seconds and Industry2 takes more than 1 day. The use of logic transformation to speed up SAT procedures seems promising for future research.

Industry3 is another particularly interesting example. In the two circuits that are compared, some outputs are not equivalent. However, only a small fraction of the input patterns can differentiate the two circuits (2^{20} out of 2^{96}). There is little hope that random simulation can identify the non-equality. Also, due to their complex logic functionality, BDDs cannot be built for the circuits. SATO could identify counterexamples in a few seconds for every non-equivalent output! SATO's heuristics to generate case-splitting variables work very well in this case. This example supports our belief that SAT-based approaches can detect errors efficiently.

5 Conclusion

Our results demonstrate the potential of SAT-based techniques in various domains of hardware verification. We believe that SAT-based approaches complement the existing BDD-based approaches well. There are some promising directions of future research. Optimization techniques in generating propositional formulas need to be further investigated. Previous work from other fields such as artificial intelligence may be relavant as well. Also, heuristics of SAT procedure need to be studied for the domain of hardware verification. For instance, in BDDs, interleaving the bits often provides a good variable ordering. Similar techniques may work well as splitting heuristics for SAT procedures.

References

1. Arne Borälv. The industrial success of verification tools based on Stålmarck's Method. In Orna Grumberg, editor, *International Conference on Computer-Aided Verification (CAV'97)*, number 1254 in LNCS. Springer-Verlag, 1997.
2. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
3. J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.
4. E. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the IBM Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.
5. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.
6. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.
7. D. Deharbe. Using induction and BDDs to model check invariants. In D. Probst, editor, *CHARME'97*. Chapman & Hall, 1997.
8. D. S. Johnson and M. A. Trick, editors. *The second DIMACS implementation challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1993. (see `http://dimacs.rutgers.edu/Challenges/`).
9. W. Kunz. HANNIBAL: An efficient tool for logic verification based on recursive learning. In *ICCAD'93*, pages 538–543, 1993.
10. A. J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In H. Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on Very Large Scale Integration*, 1985.
11. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
12. K. L. McMillan. A conjunctively decomposed boolean representation for symbolic model checking. In Rajeev Alur and Thomas Henzinger, editors, *Computer Aided Verification, 8th International Conference, CAV'96*, volume 1102, pages 13–25. Springer-Verlag, 1996.
13. R. Mukherjee, J. Jain, K. Takayama, M. Fujita, J. A. Abraham, and D. S. Fussell. FLOVER: Filtering oriented combinational verification approach. In *Proc. of International Workshop on Logic Synthesis*, 1995.
14. D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.
15. E. M. Sentovich, K. J. Singh, L. Lavagno, C. M., R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. *SIS: A System for Sequential Circuit Synthesis*. Memorandum No. UCB/ERL M92/41, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, 1992.
16. G. Stålmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula,1989. Swedish patent no. 467 076(1992), U.S. patent no. 5 276 897(1994), European patent no. 0404 454(1995).
17. H. Zhang. SATO: An efficient propositional prover. In *International Conference on Automated Deduction (CADE'97)*, number 1249 in LNAI, pages 272–275. Springer-Verlag, 1997.