Symbolic Model Checking
using SAT procedures instead of BDDs

A. Biere\textsuperscript{1}  A. Cimatti\textsuperscript{2}  E.M. Clarke\textsuperscript{1}  M. Fujita\textsuperscript{3}  Y. Zhu\textsuperscript{1}
February 1, 1999
CMU-CS-99-145

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted for DAC'99

\textsuperscript{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

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

\textsuperscript{3}Fujitsu Laboratories of America, Inc.
595 Lawrence Expressway, Sunnyvale, CA 94086-3922
fujita@fla.fujitsu.com

This research is sponsored 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.
The U. S. Government is authorized to reproduce and distribute reprints for Government purposes
notwithstanding any copyright notation thereon. This manuscript is submitted for public
lication with the understanding that the U. S. Government is authorized to reproduce and dis-
tribute reprints for Governmental purposes.
**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 demonstrate 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 if there is a counterexample of length k. Currently, we use SATO [17], an efficient implementation of the Davis-Putnam 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 Ef where E is the existential path quantifier and f is a temporal formula that contains no path quantifiers. Note that E is the dual of the universal path quantifier A. Finding a witness for Ef is equivalent to finding a counterexample for A¬f.

The implementation of a system is described as a Kripke structure. A Kripke structure is a tuple M = (S, I, T, ℓ) with a finite set of states S, the set of initial states I ⊆ S, a transition relation between states T ⊆ S × S, and the labeling of the states ℓ: S → P(ℓ) with atomic propositions ℓ. 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), ..., s(n)) where s(i) for i = 1, ..., n are propositional variables. We define propositional formulas f₁(s), f₂(s,t) and f₃(s) as follows: f₁(s) iff s ∈ I, f₂(s,t) iff (s,t) ∈ T, and f₃(s) iff p ∈ ℓ(s). For the rest of the paper we simply use T(s,t) instead of f₂(s,t) etc. In
addition, we require that every state has a successor state. That is, for all \( s \in S \) there is an \( t \in S \) with \((s,t) \in T\). For \((s,t) \in T\) we also write \( s \rightarrow t \). For an infinite sequence of states \( \pi = (s_0, s_1, \ldots) \) we define \( \pi(i) = s_i \) and \( \pi' = (s_i, s_{i+1}, \ldots) \) for \( i \in \mathbb{N} \). An infinite sequence of states \( \pi \) is a path if \( \pi(i) \rightarrow \pi(i+1) \) for all \( i \in \mathbb{N} \).

An LTL formula \( \text{Ef} \) is true in a Kripke structure \( M \models \text{Ef} \) iff there exists a path \( \pi \) 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](image)

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 \( \neg s[1] \land \neg s[0] \). The transition relation \( T(s, s') \) describes the increment of the counter at each step. We define \( \text{inc}(s, s') \) as \( (s'[0] \leftrightarrow \neg s[0]) \land (s'[1] \leftrightarrow (s[0] \land s[1])) \), and we define \( T(s, s') \) as \( \text{inc}(s, s') \lor (s[1] \land \neg s[0] \land s'[1] \land \neg s'[0]) \). 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 \( \text{AF} q \), where \( q(s) \) is defined as \( s[1] \land s[0] \). 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 \( \text{EG} p \), where \( p(s) \) is defined as \( \neg s[1] \lor \neg s[0] \). Note that \( \text{EG} p \) is the dual of \( \text{AF} q \).

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 \( \text{EG} p \), or equivalently, a counterexample for \( \text{AF} q \).

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 \( \langle M \rangle \) defined as
$I(s_0) \land T(s_0,s_1) \land T(s_1,s_2)$, 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 $s_0$ or itself. We use $\mathcal{J}_L$ defined as $T(s_2,s_1)$ to denote the transition from $s_2$ to a state $s_1$ where $I \in [0,2]$. To be consistent with the general translation in the next section, we use left subscript in $\mathcal{J}_L$. We define $L$ as $\mathcal{V}_{i=0}^{2} \mathcal{J}_L$. Thus $\neg L$ denotes the case where no loop exists.

We further constrain that the specified temporal property $Gp$ holds on the given path $s_0, s_1, s_2$. In order to be a witness for $Gp$, 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 $\llbracket Gp \rrbracket$ defined as $p(s_0) \land p(s_1) \land p(s_2)$. In the case where no loop exists, $Gp$ does not hold and $\llbracket Gp \rrbracket$ is defined as false. Finally, we combine all constraints.

\[ \llbracket M \rrbracket \land \left( (\neg L \land \text{false}) \lor \bigvee_{i=0}^{2} (\mathcal{J}_L \land \llbracket Gp \rrbracket) \right) \] (1)

In general, the constraint imposed by the temporal specification depends on the configuration of the loop. Thus in the formula (1), we put $\llbracket Gp \rrbracket$ within the scope of the disjunction over $L$. For our particular example the constraint $\llbracket Gp \rrbracket$ 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 $\llbracket M, f \rrbracket_k$. The variables $s_0, \ldots, s_k$ in $\llbracket M, f \rrbracket_k$ denote a finite sequence of states on a path $\pi$. Each $s_i$ is a vector of state variables. The formula $\llbracket M, f \rrbracket_k$ represents constraints on $s_0, \ldots, s_k$ such that $\llbracket M, f \rrbracket_k$ is satisfiable iff $f$ is valid along $\pi$. To construct $\llbracket M, f \rrbracket_k$, we first define a propositional formula $\llbracket M \rrbracket_k$ that constrains $s_0, \ldots, s_k$ to be on a valid path $\pi$ in $M$. Second, we give the translation of an LTL formula $f$ to a propositional formula that constrains $\pi$ to satisfy $f$.

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

\[ \llbracket M \rrbracket_k := I(s_0) \land \bigwedge_{i=0}^{k-1} T(s_i,s_{i+1}) \]

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 $\llbracket \cdot \rrbracket_k$ maps an LTL formula into a propositional formula. The parameter $k$ is the length of the prefix of the path that we consider and
$i$ 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 $h := p \mathbin{U} q$ and a path $\pi$ that is not a $k$-loop for a given $k \in \mathbb{N}$ (see Figure 2(a)). Starting at $\pi^i$ for $i \in \mathbb{N}$ with $i \leq k$ the formula $h$ is valid along $\pi^i$ 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 $i \leq k$

\[
\begin{align*}
\llbracket p \rrbracket^i_k & := p(s_i) & \llbracket \neg p \rrbracket^i_k & := \neg p(s_i) \\
\llbracket f \land g \rrbracket^i_k & := \llbracket f \rrbracket^i_k \land \llbracket g \rrbracket^i_k & \llbracket f \lor g \rrbracket^i_k & := \llbracket f \rrbracket^i_k \lor \llbracket g \rrbracket^i_k \\
\llbracket GF \rrbracket^i_k & := \text{false} & \llbracket F \rrbracket^i_k & := \lor_{j=i} \llbracket f \rrbracket^j_k \\
\llbracket Xf \rrbracket^i_k & := \text{if } i < k \text{ then } \llbracket f \rrbracket^{i+1}_k \text{ else false} \\
\llbracket f \mathbin{U} g \rrbracket^i_k & := \lor_{j=i} \left( \llbracket g \rrbracket^j_k \land \land_{m=i}^{j-1} \llbracket f \rrbracket^m_k \right) \\
\llbracket f \mathbin{R} g \rrbracket^i_k & := \lor_{j=i} \left( \llbracket f \rrbracket^j_k \land \land_{m=i}^{j} \llbracket g \rrbracket^m_k \right)
\end{align*}
\]

Now we consider the case where the path is a $k$-loop. The translation "$f \llbracket \cdot \rrbracket^i_k$" 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 loop.

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

5
Definition 4 (Translation of an LTL Formula for a Loop). Let $f$ be an LTL formula, $k, l, i \in \mathbb{N}$, with $l, i \leq k$.

\[
\begin{align*}
\llbracket p \rrbracket_k^l & := p(s_i) & \llbracket \neg p \rrbracket_k^l & := \neg p(s_i) \\
\llbracket f \land g \rrbracket_k^l & := \llbracket f \rrbracket_k^l \land \llbracket g \rrbracket_k^l & \llbracket f \lor g \rrbracket_k^l & := \llbracket f \rrbracket_k^l \lor \llbracket g \rrbracket_k^l \\
\llbracket G f \rrbracket_k^l & := \bigwedge_{j=\min(l,i)}^{j} \llbracket f \rrbracket_k^j & \llbracket F f \rrbracket_k^l & := \bigvee_{j=\min(l,i)}^{j} \llbracket f \rrbracket_k^j \\
\llbracket X f \rrbracket_k^l & := \llbracket f \rrbracket_{\text{succ}(i)} & \\
\llbracket f U g \rrbracket_k^l & := \bigvee_{j=i}^{\min(l,j)} \left( \llbracket g \rrbracket_k^j \land \bigwedge_{n=i}^{j-1} \llbracket f \rrbracket_k^n \right) \lor \\
& & \bigvee_{j=i}^{\min(l,j)} \left( \llbracket g \rrbracket_k^j \land \bigwedge_{n=i}^{j-1} \llbracket f \rrbracket_k^n \land \bigwedge_{m=i}^{j-1} \llbracket f \rrbracket_k^m \right) \\
\llbracket f R g \rrbracket_k^l & := \bigwedge_{j=\min(l,i)}^{j} \llbracket g \rrbracket_k^j \lor \\
& & \bigvee_{j=i}^{\min(l,j)} \left( \llbracket g \rrbracket_k^j \land \bigwedge_{n=i}^{j-1} \llbracket f \rrbracket_k^n \right) \lor \\
& & \bigvee_{j=i}^{\min(l,j)} \left( \llbracket g \rrbracket_k^j \land \bigwedge_{n=i}^{j-1} \llbracket f \rrbracket_k^n \land \bigwedge_{m=i}^{j-1} \llbracket f \rrbracket_k^m \land \bigwedge_{n=m}^{j-1} \llbracket g \rrbracket_k^n \right)
\end{align*}
\]

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 $L_k := T(s_k, s_l)$, $L_k := \llbracket f \rrbracket_{i=0}^{l} i L_k$

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

\[
\llbracket M, f \rrbracket_k := \llbracket M \rrbracket_k \land \left( \left( \neg L_k \land \llbracket f \rrbracket_0^k \right) \lor \bigvee_{i=0}^{k} \left( L_k \land \llbracket f \rrbracket_0^k \right) \right)
\]

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 $i$ of a loop are tried and the translation for a $(k, l)$-loop is conjuncted with the corresponding $L_k$ loop condition. The following theorem shows the correctness of our translation.

Theorem 1. $M \models E f$ if and only if $\llbracket M, f \rrbracket_k$ 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, $((a \lor \neg b \lor c) \land (d \lor \neg e))$ 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 $\neg, \lor$ and $\land$ and apply distributivity rule and De Morgan’s law to convert $f$ into its conjunctive normal form $f_{\text{CNF}}$. The size of $f_{\text{CNF}}$ 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].

```plaintext
procedure bool-to-cn(f,v)
{
    if (cached(f,v)) return(clause(vf \leftrightarrow v));
    case
        atomic(f) : return(clause(f \leftrightarrow vf));
        f == h \land g:
            C1 = bool-to-cn(h,vh);
            C2 = bool-to-cn(g,vg);
            assert(cached(f,vf));
            return(clause(vf \leftrightarrow vh \land vg) \cup C1 \cup C2);
    esac;
}
```

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. ‘$\lor$’ represents a boolean operator.

Figure 3 outlines our procedure. Given a boolean formula $f$, bool-to-cn($f$,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 ‘$\lor$’ is a boolean operator, $v \leftrightarrow (v_h \lor v_g)$ 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-cn($f$), 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-cn. For any boolean formula $f$, bool-to-cn($f$,true) generates a clause set $C$ with $O(|f|)$ variables and $O(|f|)$ clauses, where $|f|$ is the size of DAG for $f$. 

7
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 $v_f \iff \bigwedge_{i=0}^{n} t_i$ is $\{\neg v_f, t_0\}, \{\neg v_f, t_1\}, \ldots, t_n, \{v_f, \neg t_0, \ldots, \neg t_n\}$. If we treat $\land$ 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, $\bigwedge_{i=0}^{15} (r[i] \iff s[i])$, 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 Bwoln Yang from CMU with improved support for conjunctive partitioning. We refer to them as SMV1 and SMV2 respectively.

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
<table>
<thead>
<tr>
<th>bit</th>
<th>SMV₁</th>
<th>SMV₂</th>
<th>SATO</th>
<th>PROVER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sec</td>
<td>MB</td>
<td>sec</td>
<td>MB</td>
</tr>
<tr>
<td>0</td>
<td>919</td>
<td>13</td>
<td>25</td>
<td>79</td>
</tr>
<tr>
<td>1</td>
<td>1978</td>
<td>13</td>
<td>25</td>
<td>79</td>
</tr>
<tr>
<td>2</td>
<td>2916</td>
<td>13</td>
<td>26</td>
<td>80</td>
</tr>
<tr>
<td>3</td>
<td>4744</td>
<td>13</td>
<td>27</td>
<td>82</td>
</tr>
<tr>
<td>4</td>
<td>6580</td>
<td>15</td>
<td>33</td>
<td>92</td>
</tr>
<tr>
<td>5</td>
<td>10803</td>
<td>25</td>
<td>67</td>
<td>102</td>
</tr>
<tr>
<td>6</td>
<td>43983</td>
<td>73</td>
<td>258</td>
<td>172</td>
</tr>
<tr>
<td>7</td>
<td>&gt;17h</td>
<td>1741</td>
<td>&gt;1GB</td>
<td>209</td>
</tr>
<tr>
<td>8</td>
<td>&gt;1GB</td>
<td>473</td>
<td>0</td>
<td>29</td>
</tr>
<tr>
<td>9</td>
<td>856</td>
<td>1</td>
<td>58</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>1837</td>
<td>1</td>
<td>91</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>2367</td>
<td>1</td>
<td>125</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>3830</td>
<td>1</td>
<td>156</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>5128</td>
<td>1</td>
<td>186</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>4752</td>
<td>1</td>
<td>226</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>4449</td>
<td>1</td>
<td>183</td>
<td>5</td>
</tr>
<tr>
<td>sum</td>
<td>71923</td>
<td>2202</td>
<td>23970</td>
<td>1066</td>
</tr>
</tbody>
</table>

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 SMV₁ and SMV₂, 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
<table>
<thead>
<tr>
<th>cells</th>
<th>SMV₁ sec</th>
<th>SMV₂ MB</th>
<th>SATO k = 5 sec</th>
<th>SATO MB</th>
<th>PROVER k = 5 sec</th>
<th>PROVER MB</th>
<th>SATO k = 10 sec</th>
<th>SATO MB</th>
<th>PROVER k = 10 sec</th>
<th>PROVER MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>846</td>
<td>11</td>
<td>159</td>
<td>217</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>2166</td>
<td>15</td>
<td>530</td>
<td>703</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>4857</td>
<td>18</td>
<td>1762</td>
<td>703</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>9985</td>
<td>24</td>
<td>6563</td>
<td>833</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>19595</td>
<td>31</td>
<td>&gt;1GB</td>
<td></td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>&gt;10h</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>5</td>
<td>24</td>
<td>13</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>7</td>
<td>10</td>
<td>5</td>
<td>36</td>
<td>15</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>8</td>
<td>13</td>
<td>6</td>
<td>38</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>9</td>
<td>16</td>
<td>6</td>
<td>40</td>
<td>18</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>9</td>
<td>19</td>
<td>8</td>
<td>107</td>
<td>19</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>10</td>
<td>22</td>
<td>8</td>
<td>70</td>
<td>21</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>11</td>
<td>27</td>
<td>8</td>
<td>168</td>
<td>22</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>cells</th>
<th>SMV₁ sec</th>
<th>SMV₂ sec</th>
<th>SATO sec</th>
<th>PROVER sec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>799</td>
<td>11</td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1661</td>
<td>14</td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>3155</td>
<td>21</td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>5622</td>
<td>38</td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>9449</td>
<td>73</td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>segmentation fault</td>
<td>244 702 0 1 0 3</td>
<td>122 220 0 1 2</td>
<td>13</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>244 702</td>
<td>0 1 0 3</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>413 702</td>
<td>0 1 0 3</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>719 702</td>
<td>0 2 1 3</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td>843 702</td>
<td>0 2 1 3</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td>1060 702</td>
<td>0 2 1 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1429 702</td>
<td>0 2 1 3</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>L</th>
<th>SMV_1</th>
<th>SMV_2</th>
<th>SMV_3</th>
<th>SMV_4</th>
<th>SATO</th>
<th>PROVER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sec MB</td>
<td>sec MB</td>
<td>sec MB</td>
<td>sec MB</td>
<td>sec MB</td>
<td>sec MB</td>
</tr>
<tr>
<td>10</td>
<td>18 10</td>
<td>4 55</td>
<td>11 17</td>
<td>7 51</td>
<td>60 7</td>
<td>9 2</td>
</tr>
<tr>
<td>11</td>
<td>34 13</td>
<td>6 60</td>
<td>29 20</td>
<td>11 56</td>
<td>68 8</td>
<td>11 2</td>
</tr>
<tr>
<td>14</td>
<td>109 19</td>
<td>11 70</td>
<td>37 27</td>
<td>20 65</td>
<td>287 12</td>
<td>15 2</td>
</tr>
<tr>
<td>15</td>
<td>291 31</td>
<td>18 86</td>
<td>82 40</td>
<td>36 81</td>
<td>102 10</td>
<td>19 2</td>
</tr>
<tr>
<td>16</td>
<td>711 55</td>
<td>43 196</td>
<td>207 66</td>
<td>80 197</td>
<td>411 6</td>
<td>6 2</td>
</tr>
<tr>
<td>17</td>
<td>2126 102</td>
<td>159 393</td>
<td>573 119</td>
<td>191 393</td>
<td>1701 16</td>
<td>45 3</td>
</tr>
<tr>
<td>18</td>
<td>6103 195</td>
<td>459 753</td>
<td>1857 223</td>
<td>422 754</td>
<td>302 14</td>
<td>58 3</td>
</tr>
<tr>
<td>19</td>
<td>23405 383</td>
<td>1491 920</td>
<td>5765 430</td>
<td>1101 817</td>
<td>1551 20</td>
<td>70 3</td>
</tr>
<tr>
<td>&gt;17h</td>
<td>&gt;1GB</td>
<td>&gt;1GB</td>
<td>&gt;1GB</td>
<td>&gt;1GB</td>
<td>&gt;40h</td>
<td>&gt;99 3</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>#ins</th>
<th>#outs</th>
<th>#gates</th>
<th>sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Industry1</td>
<td>203</td>
<td>8</td>
<td>738</td>
<td>233</td>
</tr>
<tr>
<td>Industry2</td>
<td>317</td>
<td>232</td>
<td>15242</td>
<td>8790</td>
</tr>
<tr>
<td>Industry3</td>
<td>96</td>
<td>32</td>
<td>1032</td>
<td>210</td>
</tr>
</tbody>
</table>

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

In our first example, Industry1, the logic of one circuit has been considerably opti-
mized 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 opti-
mization 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 in-
put patterns can differentiate the two circuits (2^{20} out of 2^{26}). There is little hope that
random simulation can identify the non-equality. Also, due to their complex logic func-
tionality, 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 investi-
gated. Previous work from other fields such as artificial intelligence may be relevant 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


