Symbolic Model Checking using SAT procedures instead of BDDs *

A. Biere[1,2], A. Cimatti[3], E.M. Clarke[1,2], M. Fujita[4], Y. Zhu[1,2]

[1] Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, U.S.A.
[2] Verysys Design Automation, Inc., 42707 Lawrence Place, Fremont, CA 94538, U.S.A.
[3] Istituto per la Ricerca Scientifica e Tecnolgica (IRST), via Sommarive 18, 38055 Povo (TN), Italy
[4] Fujitsu Laboratories of America, Inc. 595 Lawrence Expressway, Sunnyvale, CA 94086-3922, U.S.A.
{armin+,emc+,zhu+}@cs.cmu.edu, cimatti@irst.itc.it, fujita@fla.fujitsu.com

Abstract

In this paper, we study the application of propositional decision procedures in hardware verification. In particular, we apply bounded model checking, as introduced in [1], to equivalence and invariant checking. We present several optimizations that reduce the size of generated propositional formulas. 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 [3] have traditionally been used for this purpose. In this paper, we investigate an alternative approach based on propositional decision procedures.

Model checking [5] 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 representation for the finite state machine. By replacing explicit state representation with boolean encoding, symbolic model checking [4, 10] can handle much larger designs than explicit state model checking.

With the introduction of bounded model checking [1], 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. However, to determine the diameter automatically involves a decision procedure for quantified boolean expressions.

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.

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 iff there is a counterexample of length k. Currently, we use SATO [16], an efficient implementation of the Davis-Putnum technique, and PROVER [2] based on Stålmarck's Method [15] to decide propositional satisfiability. If a counterexample exists, SATO or PROVER generates a model of the propositional formula produced by **BMC**. 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 with an example. 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.

*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.

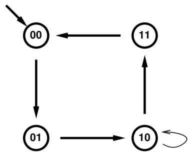

Figure 1: A two-bit counter with an erroneous transition

2 Bounded model checking

We now briefly describe our techniques for bounded model checking. First, we give some background and notational conventions. Then we illustrate our approach with a simple example. More details can be found in [1].

2.1 Background

The specification of a system is expressed in linear temporal logic (LTL). In this paper we only consider the *eventuality* operator '**F**', and the *globally* operator '**G**'. The techniques can also be applied to a more general class of operators [1]. 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. The labeling function can be omitted if every atomic propositions corresponds to a state variable.

An infinite sequence of states $\pi = (s_0, s_1, \ldots)$ is a *path* iff $(s_i, s_{i+1}) \in T$ for all $i \in \mathcal{N}$. 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

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. Initially, the value of the counter is (00). 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 $inc(s, s') \vee (s(1) \wedge \neg s(0) \wedge s'(1) \wedge \neg s'(0))$. 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 $[\![\, M \,]\!]_2$ 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, in order to be a witness for $\mathbf{G}p$, the path must contain a loop. Therefore there is a transition from s_2 to the initial state s_0, s_1 or itself. We define $L_2 := T(s_2, s_0) \vee T(s_2, s_1) \vee T(s_2, s_2)$. Additionally, property p must hold on every state of the path. We derive a corresponding propositional formula $[\![\, \mathbf{G}p \,]\!]_2$ defined as $L_2 \wedge p(s_0) \wedge p(s_1) \wedge p(s_2)$. The combination of both constraints, $[\![\, M \,]\!]_2 \wedge [\![\, \mathbf{G}p \,]\!]_2$, is the result of the translation.

In this example, the resulting propositional 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 the propositional formula becomes unsatisfiable.

This idea can be generalized to arbitrary LTL formulae. The details are described in [1] together with the following theorem.

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

3 Conversion to Clause Form

Many propositional decision procedures assume the input problem to be in conjunctive normal form, or equivalently *clause form* (CF). In this section, we focus on techniques for converting arbitrary boolean formulas to CF. In particular, we investigate optimization techniques that reduce the number of variables and clauses in the CF 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 characterization of the "hardness" of propositional problems is difficult and is likely to be dependent on specific propositional decision procedures used. Reducing the size of CF 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 CF as well as the time for satisfiability test.

A formula f in CF 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 $\{\{a, \neg b, c\}, \{d, \neg e\}\}$.

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 CF. The size of the resulting CF 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 CF transformation [13].

procedure bool-to-cf(f, v_f)
{
case
cached(f, v):
return clause($v_f \leftrightarrow v$);
atomic(f):
return clause($f \leftrightarrow v_f$);
$f == h \circ g$:
C_1 = bool-to-cf(h, v_h);
C_2 = bool-to-cf(g, v_g);
assert cached(f, v_f);
return clause($v_f \leftrightarrow (v_h \circ v_g)) \cup C_1 \cup C_2$;
esac;
}

Figure 2: An algorithm for generating CF. 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-cf(f,*`true`*)* returns a set of clauses C which is satisfiable iff f is satisfiable. The procedure traverses the syntactic 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 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 CF, clause($v_f \leftrightarrow (v_h \circ v_g)$), 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.

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 with a tree representation. In the procedure *bool-to-cf()*, 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-cf*. For any boolean formula f, *bool-to-cf(f,*`true`*)* generates a clause set with $O(|f|)$ variables and $O(|f|)$ clauses, where $|f|$ is the size of the DAG for f.

In Figure 2, we assume that f only involves binary operators. Unary operators, 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 $\bigwedge_{i=0}^{n} t_i$. We can generate CF for $v_f \leftrightarrow \bigwedge_{i=0}^{n} t_i$ as $\{\{\neg v_f, t_0\}, \{\neg v_f, t_1\}, \ldots, \{\neg v_f, t_n\}, \{v_f, \neg t_0, \ldots, \neg t_n\}\}$. 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, this optimization is useful in generating CF for the comparison between two 16 bit registers r and s $\bigwedge_{i=0}^{15}(r[i] \leftrightarrow s[i])$.

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 [10]. It outputs a propositional formula. Two different formats for the propositional formula are supported. The first format is the DIMACS format for satisfiability problems. The SATO tool [16] 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 [2] which is based on Stålmarck's Method [15]. As comparisons, we use the official version of the CMU model checker SMV and a version by Bwolen Yang from CMU with improvements including support for conjunctive partitioning. We refer to them as SMV_1 and SMV_2 respectively.

	SMV_1		SMV_2		SATO		PROVER	
cells	sec	MB	sec	MB	sec	MB	sec	MB
5	1661	14	24	57	0	1	0	2
7	5622	38	74	137	0	1	0	2
9	seg. fault		172	220	0	1	1	2
11			413	702	0	1	0	3
13			843	702	0	2	1	3
15			1429	702	0	2	1	3

Table 1: Counterexample for liveness in a buggy DME (sec = seconds, MB = Mega Bytes).

4.1 Model Checking

In [1] we reported the results of applying bounded model checking to some academic examples. We showed that model checking safety properties for a sequential shift and add multiplier and finding short counterexamples for liveness properties of an asynchronous circuit could be done much faster with SAT than with BDD based methods. In particular, we studied the design of an asynchronous circuit for distributed mutual exclusion [9]. We introduced a bug by removing some fairness constraints. The buggy design violated the liveness property that a request for a resource will be eventually acknowledged, and there is a counterexample of length 2. We applied model checkers to find the counterexample. Table 1 shows the results of this experiment.

4.2 Invariant Checking

Safety properties can be verified by proving an *inductive* invariant that holds 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. Of course, being an inductive invariant is a sufficient but not necessary condition for a safety property. 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 the equivalence between two different implementations of a queue of a particular length. This example is taken from [11] and it is known that no variable ordering exists such that the (RO)BDDs for the set of reachable states remain small. In the experiments of Table 2 an invariant that relates the contents of the two queues was manually constructed. As discussed above, three conditions have to be verified for each particular length of the queues. We compared the results of SAT-based and BDD-based techniques.

4.3 Equivalence Checking

Recently, there has been a lot of progress in boolean equivalence checking[8, 12]. 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

	SMV$_1$		SMV$_2$		SATO		PROVER	
l	sec	MB	sec	MB	sec	MB	sec	MB
15	82	40	36	81	102	10	19	2
16	207	66	80	197	411	6	6	2
17	573	119	191	393	1701	16	45	3
18	1857	223	422	754	302	14	58	3
19	5765	430	1101	817	1551	20	70	3
20	30809	845	9136	977	1377	20	86	3
21	>1GB		>1GB		>40h		99	3
22							120	3

Table 2: Comparison between queues (l = length of queues, MB = Mega Byte, sec = seconds).

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

Table 3: Equivalence checking using SAT procedures (sec = seconds).

poorly even on much smaller designs. Most equivalence checkers are BDD-based. We have investigated how propositional decision procedures (SAT procedures) can be used instead of BDDs for equivalence checking.

We use **BMC** to convert the equivalence checking problem to a propositional satisfiability problem. The output of **BMC** is a formula in CF which is fed into SATO. In Table 3, we list some industrial circuits that cannot be processed by some state-of-the-art equivalence checkers within 24 hours.

In Industry1 and Industry2, the logic of one circuit has been considerably optimized and the other is unoptimized. For both examples, we applied logic optimization using SIS [14] 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. In Industry3, some outputs of the two circuits 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. SATO was able to identify counterexamples in a few seconds for every non-equivalent output! 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 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

[1] BIERE, A., CIMATTI, A., CLARKE, E. M., AND ZHU, Y. Symbolic model checking without BDDs. In *TACAS'99* (1999). to appear.

[2] BORÄLV, A. The industrial success of verification tools based on Stålmarck's Method. In *International Conference on Computer-Aided Verification (CAV'97)* (1997), O. Grumberg, Ed., no. 1254 in LNCS, Springer-Verlag.

[3] BRYANT, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers 35*, 8 (1986), 677–691.

[4] BURCH, J. R., CLARKE, E. M., AND MCMILLAN, K. L. Symbolic model checking: 10^{20} states and beyond. *Information and Computation 98* (1992), 142–170.

[5] CLARKE, E., AND EMERSON, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the IBM Workshop on Logics of Programs* (1981), vol. 131 of *LNCS*, Springer-Verlag, pp. 52–71.

[6] DAVIS, M., AND PUTNAM, H. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery 7* (1960), 201–215.

[7] DEHARBE, D. Using induction and BDDs to model check invariants. In *CHARME'97* (1997), D. Probst, Ed., Chapman & Hall.

[8] KUNZ, W. HANNIBAL: An efficient tool for logic verification based on recursive learning. In *ICCAD'93* (1993), pp. 538–543.

[9] MARTIN, A. J. The design of a self-timed circuit for distributed mutual exclusion. In *Proceedings of the 1985 Chapel Hill Conference on Very Large Scale Integration* (1985), H. Fuchs, Ed.

[10] MCMILLAN, K. L. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.

[11] MCMILLAN, K. L. A conjunctively decomposed boolean representation for symbolic model checking. In *CAV'96* (1996), vol. 1102 of *LNCS*, Springer-Verlag, pp. 13–25.

[12] MUKHERJEE, R., JAIN, J., TAKAYAMA, K., FUJITA, M., ABRAHAM, J. A., AND FUSSELL, D. S. FLOVER: Filtering oriented combinational verification approach. In *Proc. of International Workshop on Logic Synthesis* (1995).

[13] PLAISTED, D., AND GREENBAUM, S. A structure-preserving clause form translation. *Journal of Symbolic Computation 2* (1986), 293–304.

[14] SENTOVICH, E. M., SINGH, K. J., LAVAGNO, L., M., C., MURGAI, R., SALDANHA, A., SAVOJ, H., STEPHAN, P. R., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. *SIS: A System for Sequential Circuit Synthesis*. Memorandum No. UCB/ERL M92/41, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, 1992.

[15] STÅLMARCK, G. 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).

[16] ZHANG, H. SATO: An efficient propositional prover. In *International Conference on Automated Deduction (CADE'97)* (1997), no. 1249 in LNAI, Springer-Verlag, pp. 272–275.