Bounded Model Checking Using Satisfiability Solving ⋆

Edmund Clarke[1], Armin Biere[2], Richard Raimi[3], and Yunshan Zhu[4]

[1] Computer Science Department, CMU, 5000 Forbes Avenue
Pittsburgh, PA 15213, USA, emc@cs.cmu.edu
[2] Institute of Computer Systems, ETH Zürich
8092 Zürich, Switzerland, biere@inf.ethz.ch
[3] BOPS, Inc., 7719 Woodhollow Drive, Suite 156,
Austin, Texas 78731, USA, richardr@bops.com
[4] Synopsys, Inc., 700 East Middlefield Road
Mountain View, CA 94043, USA, yunshan@synopsys.com

Abstract. The phrase *model checking* refers to algorithms for exploring the state space of a transition system to determine if it obeys a specification of its intended behavior. These algorithms can perform exhaustive verification in a highly automatic manner, and, thus, have attracted much interest in industry. Model checking programs are now being commercially marketed. However, model checking has been held back by the state explosion problem, which is the problem that the number of states in a system grows exponentially in the number of system components. Much research has been devoted to ameliorating this problem.

In this tutorial, we first give a brief overview of the history of model checking to date, and then focus on recent techniques that combine model checking with satisfiability solving. These techniques, known as *bounded model checking*, do a very fast exploration of the state space, and for some types of problems seem to offer large performance improvements over previous approaches. We review experiments with bounded model checking on both public domain and industrial designs, and propose a methodology for applying the technique in industry for invariance checking. We then summarize the pros and cons of this new technology and discuss future research efforts to extend its capabilities.

1 Introduction

Model checking [9, 10, 14, 29] was first proposed as a verification technique some eighteen years ago. The name, *model checking*, encompasses a set of algorithms for verifying properties of state transition systems by a search of their associated state transition graphs. The properties to be checked are expressed in a *temporal logic*, a formalism for reasoning about the ordering of events in time, without introducing time explicitly. In a temporal logic, one could assert, for example, that a property which is not true in the present may eventually become true in a future evolution of a system. Or, that the property would inevitably become true in all future evolutions of the system. Its rich specification language combined with a high degree of automation makes model checking very attractive to industry. As such, the late 1990s have witnessed a growth in the number of CAD companies that are bringing model checkers to market.

The first implementations of model checking in the early 1980s, used explicit representations of state transition graphs and endeavored to explore these with efficient graph traversal techniques. However, the state explosion problem, wherein the number of system states grows exponentially with the number of system components, generally limited such techniques to designs with less than a million states. When dealing with hardware designs, this would limit one to circuits with around twenty latches. Thus, these techniques were unsuitable for most industrial applications. Around 1990, techniques that used *symbolic* state space exploration came into being [8, 15, 27]. In symbolic model checking, a breadth first search of the state space is effected through the use of BDDs (Binary Decision Diagrams)[6]. The BDDs hold the characteristic functions of sets of states, and allow computation of transitions among sets of states rather than individual states.

The first BDD based symbolic model checkers were able to verify examples of significant complexity, such as the Futurebus+ Cache consistency Protocol [11]. However, while these techniques allowed for an order of

⋆ This research is sponsored by the Semiconductor Research Corporation (SRC) under Contract No. 97-DJ-294 and 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 the SRC, NSF or the United States Government.

magnitude increase in the size of designs that could be model checked, this only brought the size up to the level of the smallest component designs of interest in industry, since, below a certain size, it is difficult and often not useful to specify the behavior of a component of a design. The capacity levels of BDD based model checkers have improved somewhat during the 1990s, through enhancements to BDDs and through improvements in abstraction and compositional reasoning. But, while these improvements have paved the way for model checkers to become commercial CAD tools, it remains the case that model checkers lack a certain robustness, in that they cannot consistently handle designs of the size a typical user in industry would like.

Recently, a new type of model checking technique, *bounded model checking* with satisfiability solving [2–4], has given promising results. The method can be applied to both safety and liveness properties, where the verification of a safety property involves checking whether a given set of states is reachable, and the verification of an eventuality property involves detecting loops in a system's state transition graph. Favorable experimental results with bounded model checking have been obtained for safety properties. A simple, yet very important type of safety property is an invariant, a property that must hold in all reachable states. Obviously, if a sequence of states can be found that begins at an initial state and ends in a state where the supposed invariant is false, that property is not an invariant. It turns out that such searches for counterexamples can be done with remarkable efficiency with bounded model checking, on designs that would be difficult for BDD based model checking. Another comparative advantage is that bounded model checking seems to require little by-hand manipulation from the user, while BDD based verifications often require a suitable, manual ordering for BDD variables, or certain by-hand abstractions. While by-hand adjustments could also be necessary in bounded model checking, in our experience the technique seems not to require it very often. The robustness and the capacity increase of bounded model checking make it attractive for industrial use. Behind these advantages is the fact that satisfiability solvers, such as GRASP [33], SATO [39], and Stålmarck's algorithm [35], seldom require exponential space, while BDDs often do. The disadvantages of bounded model checking, to balance the picture, are that the method lacks completeness and the types of properties that can currently be checked are very limited. Additionally, it has not been shown that the method can consistently find long counterexamples or witnesses. However some of these drawbacks have been addressed in more recent work [32, 37, 1] and encouraging results have been obtained.

Essentially, there are two steps in bounded model checking. In the first step, the sequential behavior of a transition system over a finite interval is encoded as a propositional formula. In the second step, that formula is given to a propositional decision procedure, i.e., a satisfiability solver, to either obtain a satisfying assignment or to prove there is none. Each satisfying assignment that is found can be decoded into a state sequence which reaches states of interest. In bounded model checking only finite length sequences are explored. At times, as we will discuss in Section 5.2, a safety property may be entirely verified by looking at only a bounded length sequence. However, if the property cannot be verified as holding, the technique can still be used for finding counterexamples. In this mode, the focus is on finding bugs, rather than proving correctness.

In this paper, we will first give some background information on model checking in general and temporal logics and then briefly review the basics of BDDs and BDD based model checking. We must apologize, in advance, for our inability to cite every important contribution to model checking, as there have been so many. Instead, we have endeavored to provide the reader with a few references, i.e. [14], from which he or she can learn more. After reviewing these past model checking efforts, we introduce bounded model checking with satisfiability solving, and illustrate the method with examples. We then review experimental work and discuss, in this context, an optimization for bounded model checking known as the *bounded cone of influence* [4]. We will also discuss a methodology, first proposed in [4], for using bounded model checking to check invariants on industrial designs. In conclusion, we will summarize what we feel are the advantages and shortcomings of bounded model checking, and discuss future research aimed at minimizing the latter and maximizing the former.

2 Temporal Logic Model Checking

Many designs, especially digital hardware designs, can profitably be modeled as state transition systems for the purposes of design verification. Model checking [9, 10, 14, 29] offers an attractive means of making queries about state transition systems. In model checking, one describes a property of a transition system in a *temporal logic* and then invokes a decision procedure for traversing the state transition graph of the system and determining whether the property holds for that system. The exact decision procedure will vary with the temporal logic and the type of formula; further, for a particular logic and a particular type of formula, there may be several, equally sound model checking procedures.

In this paper, we will focus on CTL [9] (Computation Tree Logic), because it is a temporal logic that has achieved wide acceptance, largely due to its balance of expressive power and moderate decision procedure complexity. CTL is defined over Kripke Structures, which are structures having a set of states, a subset of these being initial states, and a transition relation. In addition, Kripke structures have what is known as a labeling function, which assigns to states certain atomic propositions from an underlying set of atomic propositions. In practice, when using Kripke structures to model digital hardware, states are labeled with unique combinations of Boolean variables representing circuit latches or primary inputs. The transition relation for a Kripke structure relative to a hardware design, then defines which sets of latch valuations can transition to which other.

CTL is built from path quantifiers and temporal operators. There are two path quantifiers, **A** and **E**, with meanings as follows:

- **A**, "for every path"
- **E**, "there exists a path"

A path is an infinite sequence of states such that each state and its successor are related by the transition relation. CTL has four temporal operators. We give English language interpretations of these, below, where, for convenience, we assume p and q are purely propositional formulas:

- $\mathbf{X}p$, "p holds at the next time step".
- $\mathbf{F}p$, "p holds at some time step in the future"
- $\mathbf{G}p$, "p holds at every time step in the future"
- $p\mathbf{U}q$, "p holds until q holds"

A *time step* is usually identified with a computation step, e.g. a clock tick in a synchronous design. The future is considered to be the reflexive future, i.e., it includes the present, and time is considered to unfold in discrete steps.

The syntax of CTL dictates that each usage of a temporal operator must be preceded by a path quantifier. These path quantifier and temporal operator pairs can be nested arbitrarily, but must have at their core a purely propositional formula. CTL formulae may also be connected by Boolean operators. Some typical CTL formulas, and their English language interpretations are:

- $\mathbf{EF}\,(\mathit{Started} \wedge \neg \mathit{Ready})$: it is possible to get to a state where *Started* holds but *Ready* does not hold.
- $\mathbf{AG}\,(\mathit{Req} \Rightarrow \mathbf{AF}\,\mathit{Ack})$: if a *Request* occurs, then it will eventually be *Acknowledged*.
- $\mathbf{AG}\,(\mathbf{AF}\,\mathit{DeviceEnabled})$: *DeviceEnabled* holds infinitely often on every computation path.
- $\mathbf{AG}\,(\mathbf{EF}\,\mathit{Restart})$: from any state it is possible to get to the *Restart* state.

The model checking problem for CTL is, then, to take a formula such as those above, and determine whether the set of states satisfying the formula in a particular system includes or intersects the initial states of the system. If this is the case, then it is said that the system is a *model* of the formula. The choice between requiring inclusion of the set of initial states and intersection, is an implementation choice.

Often, model checking procedures do not check a formula of interest directly, rather, they check its negation. For instance, $\mathbf{AG}\,p$, "p is true in every reachable state" can be shown to be true by showing that $\mathbf{EF}\,\neg p$, "there is a reachable state in which p is false" is false. Properties of the form $\mathbf{AG}\,p$, where p is a purely propositional formula, i.e. p is an invariant, are the only type of safety properties considered in this tutorial. They can be falsified by finding a finite path segment that starts at an initial state and ends at a state in which p is false. If $\mathbf{AG}\,p$ is false, then it must be the case that the dual of the formula, with p negated, i.e., $\mathbf{EF}\,\neg p$, is true. Thus the counterexample for $\mathbf{AG}\,p$, is a witness, or demonstration of validity, for $\mathbf{EF}\,\neg p$. Model checkers can generate instructive counterexamples or witnesses of this type, which is a great advantage of the method.

Properties of the form $\mathbf{AF}\,p$ are liveness properties. $\mathbf{AF}\,p$ has the meaning that reaching a state satisfying p is inevitable, for each and every computation of the system. Likewise, $\mathbf{A}\,[p\ \mathbf{U}\ q]$ is a liveness property. These are also termed eventuality properties, in that they specify behavior that must eventually occur, but need not occur at a specific time. Violations of liveness properties can only be demonstrated by loops in the state transition graph. For instance, a counterexample for $\mathbf{AF}\,p$ is a witness for $\mathbf{EG}\,\neg p$, and this is a loop in the state graph in which no state satisfies p, the loop being reachable from initial states along a path segment in which p is always false. Obviously, since the machine can stay in this loop forever, it need not ever reach a state satisfying p, and thus $\mathbf{AF}\,p$ is false.

2.1 BDDs and Symbolic Model Checking

In the late 1980s, algorithms for storing and manipulating Boolean functions in the form of BDDs (Binary Decision Diagrams) [6] were introduced. BDDs are directed, acyclic graphs that, essentially, hold a compressed form of the truth table of a Boolean function. They obtain the compression by utilizing sharing among different sub-terms of the function. A key advantage of BDDs is that, given a fixed ordering for the variables on which a function depends, the BDD is a canonical form for that function. Further, there are efficient algorithms for obtaining a BDD from Boolean operations on two existing functions in BDD form. Thus, many operations of interest can be performed with BDDs: netlists can be parsed and BDDs formed on the fly, two circuit nodes can be tested for equality (i.e., implementing the same function), satisfiability checking can be carried out in constant time,etc. Of course, the worst case complexity of these operations, such as satisfiability checking, did not improve. Rather, it was transferred into building the BDD. Thus, if one could build the BDD for a function, satisfiability checking could be carried out almost instantly; but, the BDD building could be difficult. For some functions, it is provably the case that building the BDD is exponential in the number of function variables [6].

However, BDDs quite often proved to be a compact and efficient format for Boolean functions. In fact, a whole segment of the current CAD industry, Boolean equivalence checking, has grown up around them[1]. Shortly after the introduction of Bryant's BDD algorithms [6], BDDs began to be used in state transition system traversals, and, therefore, model checking. This type of system was called *symbolic model checking*, because of the use of symbolic, Boolean variables. It is also called *implicit model checking*, because the state transition graph is never explicitly built and is, instead, constructed piecemeal, as necessary, through the BDDs. Around 1990, several researchers independently arrived at methods for doing this type of model checking [8, 15, 27], while Ken McMillan, at Carnegie-Mellon University, was the first to create a publicly available symbolic model checker based on BDDs [26]. This model checker was called SMV.

The main notion on which BDD based model checking is based is that Boolean functions can represent sets. For instance, for a hardware circuit with 3 latches, represented by Boolean variables a, b and c, the function:

$$f = a \wedge b$$

represents the set of circuit states where (a,b,c) are $(1,1,1)$ or $(1,1,0)$. The function $f = a \wedge b$ is called the *characteristic function* of this set. Further, since a relation defines a set, a relation, too, can be represented by a characteristic function. The transition relation of a Kripke structure represents the set of all present and next state pairs that comprise valid transitions. Assuming we are dealing with a typical, digital hardware design, we will now describe how the BDD of the characteristic function of a transition relation may be formed:

1. We allocate sets of BDD variables for the present state of latches, the next state of latches, and the primary inputs of the circuit.
2. We compute the BDD, f_j, for the input function for each latch, j, in the circuit. Each such function is over a vector of present state variables, $\overline{x}$, and a vector of input variables, $\overline{i}$.
3. For each circuit latch, we form the BDD of the transition relation for the j^{th} latch, $x'_j \leftrightarrow f_j(\overline{x},\overline{i})$, where x'_j is the next state variable for the j^{th} latch, and the symbol $\leftrightarrow$ has the meaning of *if and only if* (i.e., XNOR).
4. We form the transition relation, $T(s,s')$, where s and s' denote tuples of present and next state variables, as

$$T(s,s') = \bigwedge_{j=1}^{n} x'_j \leftrightarrow f_j(\overline{x},\overline{i})$$

Over the years, many optimization and approximation techniques have been proposed for this process, since it is often possible to build BDDs for the individual latch transition relations, but difficult to build the BDD for the conjunction of these. Techniques for partitioning the transition relation into clusters are discussed in [7, 31].

Once the transition relation is represented in BDD format, it can be manipulated to *traverse* the underlying transition system. Traversals are done by obtaining *images* or *preimages* of sets of states, these being sets of successor or predecessor states, respectively. The following is the Boolean formula for the *image*, or successors, of the set of states satisfying predicate P:

$$Image_P = \exists s \left[T(s,s') \wedge P(s) \right]$$

[1] Many equivalence checkers use non-BDD based techniques, too; but, most rely on BDDs to a great extent.

and the following for the *preimage*, or predecessors:

$$Preimage_P = \exists s' \left[T(s,s') \wedge P(s') \right]$$

Image and preimage operations on BDDs involve existential quantification of BDD variables. The existential quantification of variable, x_i, from any Boolean function, $f(x_0,\cdots,x_i,\cdots,x_n)$,is defined as:

$$\exists x_i \left[f(x_0,\cdots,x_i,\cdots,x_n) \right] = f(x_0,\cdots,0,\cdots,x_n) \vee f(x_0,\cdots,1,\cdots,x_n)$$

Reachability analysis is usually performed by taking images, starting from the set of initial states, until the set union of reached states does not change. This is an example of what is called reaching a *fixed point*. When computing a fixed point of a function operating on sets, the function is continuously applied to the set resulting from its last application, until, eventually, the function value becomes the same as the function argument. Fixpoint operations are typical of BDD based model checking and are carried out by testing for the equality of characteristic functions of sets. This latter is quite easy with BDDs, which are a canonical form for Boolean functions.

Specification evaluation, e.g., evaluation of CTL formulae, is usually performed by taking preimages from a set of states satisfying a Boolean formula. For instance, a fixpoint algorithm for checking $\mathbf{EF}P$, i.e., checking whether a state satisfying P is reachable from initial states, is to:

1. Let $Q =$ the BDD of P, and let $Reached =$ the BDD of P.
2. Let $Pre =$ the BDD of the preimage of Q.
3. Let $Union\_Reached =$ the BDD of $Pre \vee Reached$.
4. If $Union\_Reached = Reached$, go to 8.
5. Let $Q = Pre \wedge \neg Reached$. (This is the set difference of the preimage taken in 2 and the set of previously reached states)
6. Let $Reached = Union\_Reached$.
7. Go to 2.
8. If $Reached \wedge I$ is satisfiable (initial state intersection), $\mathbf{EF}P$ holds, if not, $\mathbf{EF}P$ does not hold.

An alternative for step 8 would be to check for inclusion of all initial states in the set of reached states, i.e., $I \rightarrow Reached$. This is an implementation decision, whether to check for inclusion of all initial states or intersection with just some.

While BDD based symbolic model checking enabled an order of magnitude increase in the size of designs that could be verified, in terms of hardware latches (in terms of total states, several orders of magnitude), this still was inadequate for many industrial designs of interest. In addition, manual intervention is often necessary with BDDs. For instance, BDDs are very sensitive to orderings of variables, i.e., the order in which variables are encountered from the root to the leaves of the BDD. The sizes of BDDs can vary dramatically with different orderings, but, unfortunately, all currently known algorithms for finding a good ordering are of exponential complexity, and only heuristics exist. In Section 3, we shall discuss *bounded model checking*, a technique which has enabled orders of magnitude increases in efficiency for some model checking problems, primarily by substituting SAT procedures (propositional satisfiability checking) for BDDs.

2.2 Examples of Model Checking in Practice

In this section, we offer a few examples of where model checkers have found bugs in real designs.

In 1992 Clarke and his students at CMU (Carnegie-Mellon University) used the SMV model checker, developed by Ken McMillan at CMU, to verify the cache coherence protocol in the IEEE Futurebus+ Standard [11]. They constructed a precise model of the protocol and attempted to show that it satisfied a formal specification of cache coherence. In doing so, they found a number of previously undetected errors. This was the first time that formal methods had been used to find errors in an IEEE standard. Although Futurebus+ development started in 1988, all previous attempts to validate the protocol had been based on informal techniques.

In 1996 researchers at Bell Labs offered to check some properties of a High-level Data Link Controller (HDLC) that was being designed at AT&T in Madrid. The design was almost finished, so no errors were expected. Within five hours, six properties were specified and five were verified, using the FormalCheck verifier. The sixth property failed, uncovering a bug that would have reduced throughput or caused lost transmissions. The error was corrected in a few minutes and formally verified.

In 1996, Richard Raimi and Jim Lear, working for Motorola at the PowerPC™ [2] microprocessor design center, Somerset, used Motorola's Verdict model checker to debug a hardware laboratory failure on an early version of the PowerPC 620 microprocessor [30]. The microprocessor had failed to boot an operating system in laboratory testing. With run time in seconds, Verdict produced an example of how the bus interface unit of the microprocessor could deadlock, producing the failure.

3 Bounded Model Checking

In *bounded model checking*, we construct a Boolean formula that is satisfiable if and only if the underlying state transition system can realize a finite sequence of state transitions that reaches certain states of interest. If such a path segment cannot be found at a given length, k, the search is continued for larger k. The procedure is symbolic, i.e., symbolic Boolean variables are utilized; thus, when a check is done for a specific path segment of length k, all path segments of length k are being examined. The Boolean formula that is formed is given to a satisfiability solving program and if a satisfying assignment is found, that assignment is a witness for the path segment of interest.

There are several advantages of bounded model checking. SAT tools, like PROVER [5], SATO [39] and GRASP [33], do not require exponential space and large designs can be checked very fast, since the state space is searched in an arbitrary order. BDD based model checking usually operates in breadth first search consuming much more memory. Further, the procedure is able to find paths of minimal length, which helps the user understand the examples that are generated. Lastly, the SAT tools generally need far less by hand manipulation than BDDs. Usually the default case splitting heuristics are sufficient.

There are several disadvantages to bounded model checking, however. While the method may be extendable, it has thus far only been used for specifications where fixpoint operations are easy to avoid. Additionally the method as applied is generally not complete, meaning one cannot be guaranteed a true or false determination for every specification. This is because the length of the propositional formula subject to satisfiability solving grows with each time step, and this greatly inhibits the ability to find long witnesses or counterexamples, and certainly inhibits the ability to check all possible paths through a machine. However, even with these disadvantages, the advantages of the method make it a valuable complement to existing verification techniques. It is able to find bugs and sometimes to determine correctness, in situations where other techniques fail completely.

Previous to the efforts in bounded model checking, propositional decision procedures (SAT) [16] had been applied in both hardware verification [24, 34, 36] and specification logics [17, 19]. In addition, SAT had been used in the formal verification of railway control systems [5] and in AI (artificial intelligence) planning systems [22].

3.1 Creation of Propositional Formulas

The propositional formula created by a bounded model checker is formed as follows: Given:

- a transition system, M,
- a temporal logic formula, f and
- a user-supplied time bound, k

we construct a propositional formula $[\![M, f]\!]_k$ which will be satisfiable if and only if the formula f is valid along some computation path of M. We form this formula as follows. For state transition system M and time bound k, the *unrolled transition relation* is

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

where $I(s_0)$ is the characteristic function of the set of initial states, and $T(s_i, s_{i+1})$ is the characteristic function of the transition relation. We then form $[\![f]\!]_k$, where $[\![f]\!]_k$ is a formula that will be true if and only if the formula f is valid along a path of length k. Subsequently, we form the conjunction of $[\![M]\!]_k$ and $[\![f]\!]_k$.

[2] PowerPC is a trademark of the International Business Machines Corporation, used under license therefrom.

As an example, we will consider the CTL formula, **EF** p. This formula simply asserts that a state satisfying propositional formula p is reachable from initial states of the system. Let us assume we wish to check whether **EF** p can be verified in two time steps, i.e., $k = 2$. We would then form the following formula:

$$[\![M, f]\!]_2 := I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge (p(s_0) \vee p(s_1) \vee p(s_2))$$

Here, $(p(s_0) \vee p(s_1) \vee p(s_2))$ is $[\![\mathbf{EF}\, p]\!]_2$. We will illustrate this further, and show how the transition relation, $T(s_i, s_{i+1})$, is expanded.

3.2 Safety Property Example

Let us assume we have a 2-bit counter, with the least significant bit represented by a Boolean variable, a, the most significant by a Boolean variable, b. The transition relation of the counter is:

$$(a\prime \leftrightarrow \neg a) \wedge \left(b' \leftrightarrow a \oplus b\right)$$

Here, $\oplus$ has the meaning of XOR, $\leftrightarrow$ of XNOR. Let us assume, initially, both bits are at 0. Suppose we wish to check if (a, b) can transition to $(1, 1)$ within two time steps. Unfolding the transition relation and inserting the formula that performs the reachability check, we get:

$$\begin{aligned}
I(s_0): &\quad (\quad \neg a_0 \wedge \neg b_0 \quad) \wedge \\
T(s_0, s_1): &\quad ((a_1 \leftrightarrow \neg a_0) \wedge (b_1 \leftrightarrow (a_0 \oplus b_0))) \wedge \\
T(s_1, s_2): &\quad ((a_2 \leftrightarrow \neg a_1) \wedge (b_2 \leftrightarrow (a_1 \oplus b_1))) \wedge \\
p(s_0): &\quad (\quad a_0 \wedge b_0 \quad \vee \\
p(s_1): &\quad \quad a_1 \wedge b_1 \quad \vee \\
p(s_2): &\quad \quad a_2 \wedge b_2 \quad)
\end{aligned}$$

This formula is unsatisfiable, as we'd expect, since the count cannot reach $(1, 1)$ from $(0, 0)$ in 2 time steps. If the reader wishes, he or she can expand the formula for a third time step, which would produce a satisfiable formula since the counter *can* transition to $(1, 1)$ in 3 time steps. We now consider liveness properties.

3.3 Liveness Property Example

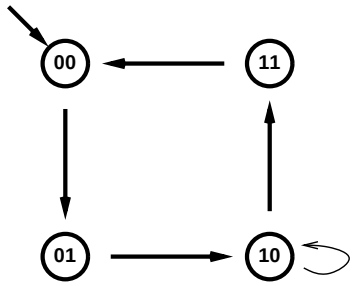

Fig. 1. A two-bit counter with an extra transition

We consider again a two-bit counter, one similar to, but slightly different that the counter in Section 3.2. As in Section 3.2, the counter is encoded by two state variables a and b, denoting the least significant and most significant bits, respectively, and the initial state of the counter is 0, i.e., state $(0, 0)$. However, the transition relation $T(s, s')$ is different from that in Section 3.2, in that we add a transition from state $(1, 0)$ back to itself. This new counter is shown as a Kripke structure in Figure 1. Let us define a function, $inc(s, s')$, as $(a' \leftrightarrow \neg a) \wedge (b' \leftrightarrow (a \oplus b))$.

The reader will recognize this as the transition relation of the counter in Section 3.2. The new transition relation, containing the transition from $(1,0)$ back to itself can then be written as:

$$T(s,s') = inc(s,s') \vee (b \wedge \neg a \wedge b' \wedge \neg a')$$

Suppose we claim that this new counter must eventually reach state $(1,1)$. We can specify this property as $\mathbf{AF}\,(b \wedge a)$. Clearly, from Figure 1, this specification is false. Now, a counterexample that demonstrates this would be a path, starting in the initial state, in which the counter never reaches state $(1,1)$. This can be expressed as $\mathbf{EG}\,p$, where $p = \neg b \vee \neg a$. We check this formula to determine the truth value of $\mathbf{AF}\,(b \wedge a)$.

Let us assume we set the time bound, k, for checking $\mathbf{EG}\,p$ at 2. All candidate paths will then have $k+1$, or 3 states: an initial state, and two others reached upon two successive transitions. We will name these states s_0, s_1, s_2. We now formulate the constraints any s_0, s_1 and s_2 must meet in order to be a witness of $\mathbf{EG}\,p$, or equivalently, a counterexample for $\mathbf{AF}\,(b \wedge a)$.

First, s_0, s_1, s_2 must be part of valid path starting from the initial state. In other words, the unrolled transition relation, formula 1 from Section 3.1, must hold for $k = 2$. This is $[\![M]\!]_2 = I(s_0) \wedge T(s_0,s_1) \wedge T(s_1,s_2)$. Second, the sequence of states s_0, s_1, s_2 must be part of a loop, meaning there must be a transition from the last state, s_2, back to either s_0, s_1, or itself. This is

$$T(s_2,s_3) \wedge (\, s_3 = s_0 \vee s_3 = s_1 \vee s_3 = s_2 \,)$$

We further constrain that the specified temporal property p must hold on every state of the path. Combining all these constraints for the system in Figure 1, we get the following, expanded formula:

$$\begin{array}{rl}
I(s_0): & (\quad \neg a_0 \wedge \neg b_0 \quad) \wedge \\
T(s_0,s_1): & ((a_1 \leftrightarrow \neg a_0) \wedge (b_1 \leftrightarrow (a_0 \oplus b_0)) \vee \\
 & \quad b_1 \wedge \neg a_1 \wedge b_0 \wedge \neg a_0 \quad) \wedge \\
T(s_1,s_2): & ((a_2 \leftrightarrow \neg a_1) \wedge (b_2 \leftrightarrow (a_1 \oplus b_1)) \vee \\
 & \quad b_2 \wedge \neg a_2 \wedge b_1 \wedge \neg a_1 \quad) \wedge \\
T(s_2,s_3): & ((a_3 \leftrightarrow \neg a_2) \wedge (b_3 \leftrightarrow (a_2 \oplus b_2)) \vee \\
 & \quad b_3 \wedge \neg a_3 \wedge b_2 \wedge \neg a_2 \quad) \wedge \\
s_3 = s_0: & (\quad (a_3 \leftrightarrow a_0) \wedge (b_3 \leftrightarrow b_0) \quad \vee \\
s_3 = s_1: & \quad (a_3 \leftrightarrow a_1) \wedge (b_3 \leftrightarrow b_1) \quad \vee \\
s_3 = s_2: & \quad (a_3 \leftrightarrow a_2) \wedge (b_3 \leftrightarrow b_2) \quad) \wedge \\
p(s_0): & (\quad \neg a_0 \vee \neg b_0 \quad) \wedge \\
p(s_1): & (\quad \neg a_1 \vee \neg b_1 \quad) \wedge \\
p(s_2): & (\quad \neg a_2 \vee \neg b_2 \quad)
\end{array}$$

This formula is indeed satisfiable. The satisfying assignment corresponds to a path from initial state $(0,0)$ to $(0,1)$ and then to $(1,0)$, followed by the self-loop at state $(1,0)$, and is a counterexample to $\mathbf{AF}\,(b \wedge a)$. If the self-loop out of $(1,0)$ were removed, giving us the counter in Section 3.2, this would remove the lines of the form

$$b_i \wedge \neg a_1 \wedge b_{i-1} \wedge \neg a_{i-1}$$

from the formula, for i from 1 to 3. Under those circumstances, the reader can verify that the altered version of the formula would then become unsatisfiable, as it should, since $\mathbf{AF}\,(b \wedge a)$ holds for the counter in Section 3.2.

Specifying liveness properties often makes sense only if certain *fairness constraints* are added as well. Fairness constraints are conditions which must occur infinitely often in the continuous operation of a system. For example, circuits on a microprocessor may be verified under fairness constraints that reflect the bus arbitration scheme of the computer system in the processor will eventually operate.

A simple example of a possible fairness constraint would be to assume that continuous operation of the counter in Figure 1 must cause the initial state to be visited infinitely often. While there is no particular reason a counter should work this way, we use the example to illustrate how fairness constraints are imposed in bounded model

checking. Given such a fairness constraint, a counterexample to the liveness property **AF** $(b \wedge a)$ would then need to include a transition to the $(0,0)$ state. This is a path with a loop as before, but with the additional constraint that $\neg a \wedge \neg b$ has to hold somewhere on the loop. This changes the generated Boolean formula as follows. For each backloop, $T(s_2, s_3)$, where the state s_3 is required to be equivalent to either s_0, s_1 or s_2, we add a term that requires $\neg a \wedge \neg b$ to hold on the loop. For example, for the possible loop from s_2 to s_0 (the case where $s_3 = s_0$), we would replace

$$a_3 \leftrightarrow a_0 \wedge b_3 \leftrightarrow b_0$$

by

$$(a_3 \leftrightarrow a_0 \wedge b_3 \leftrightarrow b_0) \wedge (c_0 \vee c_1 \vee c_2)$$

with c_i defined as $\neg a_i \wedge \neg b_i$. As there is no counterexample that would satisfy this fairness constraint, in this case the resulting propositional formula would be unsatisfiable.

3.4 Conversion to CNF

Satisfiability testing for propositional formulae in known to be an NP-complete problem, and all known decision procedures are exponential in the worst case. However, they may use different heuristics in guiding their search and, therefore, exhibit different average complexities in practice. Precise characterization of the "hardness" of a certain propositional problem is difficult and is likely to be dependent on the specific decision procedure used. Many propositional decision procedures assume the input problem to be in CNF (conjunctive normal form). Usually, it is a goal to reduce the size of the CNF version of the formula, although this may not always reduce the complexity of the search. Our experience has been, however, that reducing the size of the CNF does reduce the time for the satisfiability test as well.

A formula f in CNF 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 $\{\{a, \neg b, c\}, \{d, \neg e\}\}$. CNF is 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 the distributivity rule and De Morgan's law to convert f to CNF. The size of the converted formula can be exponential with respect to the size of f, the worst case occurring when f is in disjunctive normal form. To avoid the exponential explosion, we use a structure preserving clause form transformation [28].

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

Fig. 2. 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 2 outlines our procedure. Statements which are underlined represent the different cases considered, the symbol "=" denotes assignment while the symbol "==" denotes equality. Given a Boolean formula f, *bool-to-cnf(*f,true*)* returns a set of clauses C which is satisfiable if and only if the original formula, f, is satisfiable. Note that C is not logically equivalent to the original formula, but, rather, preserves its satisfiability. The procedure

traverses the syntactical structure of f, introduces a new variable for each subexpression, and generates clauses that relate the new variables. In Figure 2, we use symbols g and h to denote subexpressions of the Boolean formula f, and we use v_f, v_g and v_h to denote new variables introduced for f, g and h. C_1 and C_2 denote sets of clauses. If a subexpression, q, has been cached, the call to cached(q) returns the variable v_q introduced for q. The procedure, clause(), translates a Boolean formula into clause form. It replaces Boolean connectives such as implication, $\rightarrow$, or equality, $\leftrightarrow$, etc., by combinations of *and*, *or* and *negation* operators and subsequently converts the derived formula into conjunctive normal form. It does this in a brute force manner, by applying the distributivity rule and De Morgan's law. As an example, if u and v are Boolean variables, clause() called on $u \leftrightarrow v$ returns $\{\{\neg u, v\}, \{u, \neg v\}\}$. It should be noted that clause() will never be called by *bool-to-cnf* with more than 3 literals, and so, in practice, the cost of this conversion is quite acceptable. 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, clause($v_f \leftrightarrow v_h \circ v_g$), with no more than 4 clauses, each of which contains no more than 3 literals.

Internally, we represent a Boolean formula f as a directed acyclic graph (DAG), i.e., common subterms of f are shared. In the procedure *bool-to-cnf()*, we preserve this sharing of subterms, in that for each subterm in f, only one set of clauses is generated. 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 the DAG for f.

In Figure 2, we assume that f only involves binary operators, however, the unary operator, negation, can be handled similarly. We have 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 then:

$$\{\{\neg v_f, t_0\}, \{\neg v_f, t_1\}, \ldots, \{\neg v_f, t_n\}, \{v_f, \neg t_0, \ldots, \neg t_n\}\}$$

If we treat $\wedge$ as a binary operator, we need to introduce $n-1$ new variables for the subterms in $\bigwedge_{i=0}^{n} t_i$. 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

At Carnegie-Mellon University, a model checker has been implemented called **BMC**, based on bounded model checking. Its input language is a subset of the SMV language [26]. It takes in a circuit description, a property to be proven, and a user supplied time bound, k. It then generates the type of propositional formula described in Section 3.1. It supports both the DIMACS format [20] for CNF formulae, and the input format for the PROVER Tool [5] which is based on Stålmarck's Method [35]. In our experiments, we have used the PROVER tool, as well as two public domain SAT solvers, SATO [39] and GRASP [33], both of which use the DIMACS format.

We first discuss experiments on circuits available in the public domain, that are known to be difficult for BDD-based approaches. First we investigated a sequential multiplier, the shift and add multiplier of [12]. We specified that when the sequential multiplier is finished, its output is the same as the output of a certain combinational multiplier, the C6288 circuit from the ISCAS'85 benchmark set, when the same input words are applied to both multipliers. The C6288 multiplier is a 16x16 bit multiplier, but we only allowed 16 output bits as in [12], together with an overflow bit. We checked the above property for each output bit individually, and the results are shown in Table 1. For BDD-based model checkers, we used a manually chosen variable ordering where the bits of the registers are interleaved. Dynamic reordering, where the application tries to change reorderings on the fly, failed to find a considerably better ordering in a reasonable amount of time. The proof that the multiplier is finished after a finite number of steps involves the verification of a simple liveness property which can be checked instantly both with BDD based methods and bounded model checking.

In [25] 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. This assumption is modeled by a *fairness constraint* (fairness constraints were explained in Section 3.3). Each cell has exactly 18 gates and therefore the model has $n \cdot 18$ fairness constraints

	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.

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, rather, we showed that there are no counterexamples of particular length k. To illustrate the performance of bounded model checking we chose $k = 5, 10$. The results can be found in Table 2.

We repeated the experiment with a buggy design, by simply removing several fairness constraints. Both PROVER and SATO generate a counterexample (a 2 step loop) nearly instantly (see Table 3).

	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
2	63	7	4	30	0	1	0	2	1	3	10	3
3	275	9	20	50	0	2	1	2	2	5	27	4
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.

	SMV_1		SMV_2		SATO		PROVER	
cells	sec	MB	sec	MB	sec	MB	sec	MB
2	62	7	3	28	0	0	0	1
3	295	9	7	34	0	0	0	1
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.

5 Experiments on Industrial Designs

In this section, we will discuss a series of experiments on industrial designs, checking whether certain predicates were invariants of these designs. First, we explain an optimization for bounded model checking that was used in these experiments.

5.1 Bounded Cone of Influence

The *Cone of Influence Reduction* is a well known technique[3] that reduces the size of a model if the propositional formulae in the specification do not depend on all state variables in the structure. The basic idea of the Cone of Influence (COI) reduction is to construct a dependency graph of the state variables in the specification. In building the dependency graph, a state variable is represented by a node, and that node has edges emanating out to nodes representing those state variables upon which it combinationally depends. The set of state variables in this dependency graph is called the COI of the variables of the specification. In this paper, we call this the "classical" COI, to differentiate it from the bounded version. The variables not in the classical COI can not influence the validity of the specification and can therefore be removed from the model.

This idea can be extended to what we call the Bounded Cone of Influence. The formal definition for the bounded COI is given in [4], and we give, here, an intuitive explanation. The intuition is that, over a bounded time interval, we need not consider every state variable in the classical COI at each time point. For example, if we were to check $\mathbf{EF}\, p$, where p is a propositional formula, for a time bound of $k = 0$, we would need to consider only those state variables upon which p combinationally depends. If the initial values for these were consistent with p holding, then $\mathbf{EF}\, p$ would evaluate to true, without needing to consider any additional state variables in the classical COI. Let us, for convenience, call the set of state variables upon which p combinationally depends its *initial support*. If we could not prove $\mathbf{EF}\, p$ true for $k = 0$, and wanted to check it for $k = 1$, we would need to consider the set of state variables upon which those in the initial support depend. These may include some already in the initial support set, if feedback is present in the underlying circuit. Clearly, the set union of the initial support set plus this second support set are the *only* state variables upon which the truth value of $\mathbf{EF}\, p$ depends for time bound $k = 1$. Again, this will always be a subset of the state variables in the entire classical COI. If we restrict ourselves to expanding formula 1 of Section 3.1 only for those variables in the bounded COI for a particular k, we will get a smaller CNF formula, in general, than if we were to expand it for the entire, classical COI. This is the main idea behind the Bounded Cone of Influence.

[3] The cone of influence reduction seems to have been discovered and utilized by a number of people, independently. We note that it can be seen as a special case of Kurshan's localization reduction [23].

5.2 PowerPC Circuit Experiments

We ran experiments on subcircuits from a PowerPC microprocessor under design at Motorola's Somerset design center, in Austin, Texas. While a processor is under design at Somerset, designers insert assertions into the register transfer level (RTL) simulation model. These Boolean expressions are important safety properties, i.e., properties which should hold at all time points. If an assertion is ever false during simulation, an immediate error is flagged. In our experiments, we checked, using BMC and two public domain SAT checkers, SATO and GRASP, 20 assertions chosen from 5 different processor design blocks. We turned each into an $\mathbf{AG}\,p$ property, where p was the original assertion. For each of these, we:

1. Checked whether p was a tautology.
2. Checked whether p was otherwise an invariant.
3. Checked whether $\mathbf{AG}\,p$ held for various time bounds, k, from 0 to 20.

The gate level netlist for each of the 5 design blocks was translated into an SMV file, with each latch represented by a state variable having individual next state and initial state assignments. For the latter, we assigned the 0 or 1 values we knew the latches would have after a designated power-on-reset sequence[4] Primary inputs to design blocks were modeled as unconstrained state variables, i.e., having neither next state nor initial state assignments.

For combinational tautology checking we eliminated all initialization statements and ran BMC with a bound of $k = 0$, checking the inner, propositional formula, p, from each of the $\mathbf{AG}\,p$ specifications. Under these conditions, the specification could hold only if p was true for all assignments to the state variables in its support.

Invariance checking entails checking whether a propositional formula holds in all initial states and is preserved by the transition relation, the latter meaning that all successors of states satisfying the formula also satisfy it. If these conditions are met, we call the predicate an *inductive invariant*. We ran BMC on input files with all initialization assignments intact, for each design block and each p in each $\mathbf{AG}\,p$ specification, with a time bound of $k = 0$. This determined whether each formula, p, held in the single, valid initial state of each design. We then ran BMC in a mode in which, for each design block and each $\mathbf{AG}\,p$ specification, all initialization assignments were removed from the input file, and, instead, an initial states predicate was added that indicated the initial states should be all those states satisfying p. Note that we did not really believe the initial states actually were those satisfying p. This technique was simply a way of getting the BMC tool to check all successors of all states satisfying p, in one time step. The time bound, k, was set to 1, and the $\mathbf{AG}\,p$ specification was checked. If the specification held, this showed p was preserved by the transition relation, since $\mathbf{AG}\,p$ could only hold, under these circumstances, if the successors of every state satisfying p also satisfied p. Note that $\mathbf{AG}\,p$ not holding under these conditions could possibly be due exclusively to behaviors in unreachable states. For instance, if an unreachable state, s, existed which satisfied p but had a successor, s', which did not, then the check would fail. Therefore, because of possible "bad" behaviors in unreachable states, this technique can only show that p *is* an invariant, but cannot show that it is not. However, we found this type of inductive invariance checking to be very inexpensive with bounded model checking, and, therefore, very valuable. In fact, we made it a cornerstone of the methodology we recommend in Section 6.

In these experiments, we used both the GRASP [33] and SATO [38] satisfiability solvers. When giving results, however, we do not indicate from which solver they came, rather, we just show the best results from the two. There is actually an interesting justification for this. In our experience, the time needed for satisfiability solving is often just a few seconds, and usually no more than a few minutes. However, there are problem instances for which a particular SAT tool will labor far longer, until a timeout limit is reached. We have quite often found that when one SAT solving tool needs to be aborted on a problem instance, another such tool will handle it quickly; and, additionally, the same solvers often switch roles on a later problem instance, the former slow solver suddenly becoming fast, the former fast one, slow. Since the memory cost of satisfiability solving is usually slight, it makes sense to give a particular SAT problem, in parallel, to several solvers, or to versions of the same solvers with different command line arguments, and simply take the first results that come in. So, this method of running multiple solvers, as we did, on each job, is something which we recommend.

The SMV input files were given to a recent version of the SMV model checker (the SMV_1version referred to earlier) to compare to BDD based model checking. We did 20 SMV runs, checking each of the $\mathbf{AG}\,p$ specifications, separately. When running SMV, we used command line options that enabled the early detection, during

[4] Microprocessors are generally designed with specified reset sequences. In PowerPC designs, the resulting values on each latch are known to the designers, and this is the appropriate initial state for model checking.

reachability analysis, of false **AG** *p* properties. In this mode, the verifier did not need to compute a fixpoint if a counterexample existed, which made the comparison to BMC more appropriate. We also enabled dynamic variable ordering when running SMV.

All experiments were run with wall clock time limits. The satisfiability solvers were given 15 minutes wall clock time, maximum, to complete each run, while SMV was given an hour for each of its runs. BMC, itself, was never timed, as its task of translating the design description and the specification is usually done quite quickly. The satisfiability solving and SMV runs were done on RS6000 model 390 workstations, having 256 Megabytes of local memory.

5.3 Environment Modeling

We did not model the interfaces between the subcircuits on which we ran our experiments and the rest of the microprocessor or the external computer system in which the processor would eventually be placed. This is commonly referred to as "environment modeling". One would ideally like to do environment modeling on subcircuits such as we experimented on, since these are not closed systems. Rather, they depend for their correct functioning upon input constraints, i.e., certain input combinations or sequences *not* occurring. The rest of the system must guarantee this [21]. However, in the type of invariant checking we did, one would always be assured of true positives, since if a safety property holds with a totally unconstrained environment, then it holds in the real environment (this is proven in [13, 18]).

It is likely that an industrial design team would first check safety properties with unconstrained environments, since careful environment modeling can be time consuming. They would then decide, on an individual basis, what to do about properties that failed: invest in the environment modeling for more accurate model checking, in order to separate false failures from real ones, or hope that digital simulation will find any real violations that exist. Importantly, the model checker's counterexamples could provide hints as to which simulations, on the complete design not just the subcircuit, may need to be run. For instance, the counterexample may indicate that certain instructions need to be in execution, certain exceptions occurring, etc. The properties that pass the invariance test need no more digital simulation, and thus conserve CPU resources.

In the examples we did run, all the negatives proved, upon inspection with designers, to be false negatives. The experiments still yield, however, useful information on the capacity and speed of bounded model checking. Further, in Section 6, we describe a methodology that can reduce or eliminate false negatives.

5.4 Experimental Results

As mentioned, we checked 20 safety properties, distributed across 5 design blocks from a single PowerPC microprocessor. These were all *control* circuits, having little or no datapath elements. Their sizes were as follows:

Circuit	Latches	PIs	Gates
bbc	209	479	4852
ccc	371	336	4529
cdc	278	319	5474
dlc	282	297	2205
sdc	265	199	2544

Circuit	Spec	Latches	PIs
bbc	1 - 4	150	242
ccc	1 - 2	77	207
cdc	1 - 4	119	190
dlc	1 - 6	119	170
dlc	7	119	153
sdc	1 - 2	113	121
sdc	3	23	15

Table 4. Before and After Classical COI (PI = Primary Inputs)

In table 4, we report the sizes of the circuits before and after classical COI reduction has been applied. Each **AG** *p* specification is given an arbitrary numeric label, on each circuit. These do not relate specifications on different design blocks, e.g., specification 2 of *dlc* is in no way related to specification 2 of *sdc*. Many properties involved much the same cone of circuitry on a design block, as can be seen by the large number of specifications

having cones of influence with the same number of latches and PIs. However, these reduced circuits were not identical, from one specification to another, though they shared much circuitry.

Table 5 gives the results of tautology and inductive invariance checking for each p from each **AG** p specification. These runs were done with bounded COI enabled. There are columns for tautology checking, for preservation by the transition relation and for preservation in initial states. The last two conditions must both hold for a Boolean formula to be an inductive invariant. A"Y" in the leftmost part of a column indicates the condition holding, an "N" that it does not, When a "Y" is recorded, time and memory usage may appear after it, separated by slashes. These are recorded only for times ≥ 1 second, and memory usage ≥ 5 megabytes, otherwise a "-" appears for insignificant time and memory. As can be seen, tautology and invariance checking can be remarkably inexpensive. This is an extremely important finding, as these can be quite costly with BDD based methods, and are at the heart of the verification methodology we propose in Section 6.

We were surprised by the small number of assertions that were combinational tautologies. We had expected that designers would try to insure safety properties held by relying on combinational, as opposed to sequential circuitry. However, the real environment may, in fact, constrain inputs to design blocks combinationally such that these are combinational tautologies. See Section 6 for a discussion of this.

As stated above, many of our examples exhibited false negatives, and they did so at low time bounds. Other of our examples were found to be inductive invariants. Satisfiability solving went quickly at high values of k if counterexamples existed at low values of k or if the property was an invariant. The more difficult SAT runs are those for which neither counterexamples nor proofs of correctness were found. Table 6 shows the four examples which were of this type, *bbc* specs 1, 3 and 4, and *sdc* spec 1. All results, again, were obtained using bounded COI. We also ran these examples using just classical COI, and we observed that the improvement that bounded COI brings relative to classical COI wears off at higher k values, specifically, at values near to 10. Intuitively, this is due to the fact that, as we extend further in time, we eventually compute valuations for all the state variables in the classical cone of influence. However, since we expect bounded model checking to be most effective at finding short counterexamples, bounded COI is helping augment the system's strengths.

In table 6, "long k" is the highest k value at which satisfiability solving was accomplished, and "vars" and "clauses" list the number of literals and clauses in the CNF file at that highest k level. The "time" column gives CPU time, in seconds, for the run at that highest k value. Regarding memory usage, this usually does not exceed a few tens of megabytes, and is roughly the storage needed for the CNF formula, itself.

Table 7 lists the circuits and specifications which were either shown to be inductive invariants or for which counterexamples were found. Under the column "holds", a "Y" indicates a finding of being an inductive invariant, a "N" the existence of a counterexample. For the counterexamples, the next column, "fail k", gives the value of k for which the counterexample was found. Since all counterexamples were found with k values ≤ 2, we did not list time and memory usage, as this was extremely slight. In each case, the satisfiability solving took less than a second, and memory usage never exceeded more than 5 megabytes.

Lastly, the results of BDD-based model checking are that SMV was given each of the 20 properties separately, but completed only one of these verifications. The 19 others all timed out at one hour wall clock time. SMV was run when the Somerset computer network allowed it unimpeded access to the CPU it was running on; and still, under these circumstances, SMV was only able to complete the verification of *sdc*, specification 3. Classical COI for this specification gave a very small circuit, having only 23 latches and 15 PIs. SMV found the specification false in the initial state, in approximately 2 minutes. Even this, however, can be contrasted to BMC needing 2 seconds to translate the specification to CNF, and the satisfiability solver needing less than 1 second to check it!

5.5 Comparison to BDD Based Model Checking

It is useful to reflect on what the experiments on PowerPC microprocessor circuits show and what they do not. First, the experiments should not be interpreted as evidence that BDD based model checkers cannot handle circuits of the size given. There are approximation techniques, for instance where certain portions of a circuit are deleted or approximated with simpler Boolean functions that still yield true positives for invariance checking, and these could have been employed. Some of the verifications may have gone through under these circumstances. However, the experiments, as run, do give a measure of the size limits of BDD based and SAT based model checking.

Without input constraints it proved easy to reach states that violated the purported invariants. It has been noted, empirically, by many users of BDD based tools, that it is much harder to build BDDs for incorrect designs than it is for correct designs. There is no theoretical explanation of why this is so, but it may very well be that

Circuit	Spec	Tautology	Tran Rel'n	Init State
bbc	1	N - -	N - -	Y - -
bbc	2	N - -	Y - -	N - -
bbc	3	N - -	N - -	Y - -
bbc	4	N - -	N - -	Y - -
ccc	1	N - -	N - -	Y - -
ccc	2	N - -	N - -	Y - -
cdc	1	N - -	N - -	Y - -
cdc	2	Y - -	Y - -	Y - -
cdc	3	Y - -	Y - -	Y - -
cdc	4	Y - -	Y - -	Y - -
dlc	1	N - -	N - -	Y - -
dlc	2	N - -	N - -	Y - -
dlc	3	N - -	N - -	Y - -
dlc	4	N - -	N - -	Y - -
dlc	5	N - -	N - -	Y - -
dlc	6	N - -	N - -	Y - -
dlc	7	N - -	N - -	Y - -
sdc	1	N - -	Y / 15 / 5	Y - -
sdc	2	N - -	N / 60 / 6.5	Y - -
sdc	3	N - -	N 15 -	N - -

Table 5. Tautology and Invariance Checking Results

circuit	spec	long k	vars	clauses	time
bbc	1	4	7873	30174	35.4
bbc	3	10	16814	63300	58
bbc	4	5	9487	35658	18
sdc	1	4	5554	20893	72

Table 6. Size Measures for Difficult Examples

circuit	spec	holds	fail k
bbc	2	N	0
ccc	1	N	1
ccc	2	N	1
cdc	1	N	2
cdc	2	Y	-
cdc	3	Y	-
cdc	4	Y	-
dlc	1	N	2
dlc	2	N	2
dlc	3	N	2
dlc	4	N	1
dlc	5	N	2
dlc	6	N	2
dlc	7	N	0
sdc	2	Y	-
sdc	3	N	0

Table 7. Invariants and Counterexamples

SMV, or another BDD based model checker, could have successfully completed many of the property checks on versions of these designs having accurate input constraints. However, in a way this is to the credit of bounded model checking, in that it seems able to handle problem instances which are difficult for BDDs.

Now, another observation is that when a design has a large number of errors, random, digital simulation can find counterexamples quickly. Many commercial formal verification tools first run random digital simulations on a design, to see if property violations can be detected easily. While we did not do this in our experiments, we feel it is likely this, too, would have found quick counterexamples. However, this only shows that bounded model checking is at least as powerful as this method, on buggy designs—yet, bounded model checking has the additional capability of conducting exhaustive searches, within certain limits.

As to those limits, a big question with bounded model checking is whether it can, or will, find long counterexamples. Clearly, it is to the advantage of BDD based model checking that if the BDDs can be built and manipulated, all infinite computation paths, i.e., all loops through the state graph, can be examined. But, all too often, as mentioned, the BDDs cannot be built or manipulated. In those cases, even if bounded model checking cannot be run over many time steps, it does give exhaustive verification at each time step, and certainly is worth running. Most of our experiments did not produce information that would answer the question as to the expected length of counterexamples, but a few did. Out of the verifications attempted, 4 yielded neither counterexamples nor proofs of correctness, and simply timed out. This means for the property being checked, these designs were *not* buggy, up to the depth checked. Of these four, *bbc* specs 1, 3 and 4, *sdc* spec 1, BMC was able to go out to 4, 10, 5 and 4 time steps, respectively (see table 6). Thus, we expect that with current technology, we might be limited to between 5 and 10 time steps on large designs. Of course, we could have let the SAT tools run longer, and undoubtedly we would have extended some of these numbers. But, that was not the goal of our experiment. We tried to see what one could expect running large numbers of designs through a verifier, where not much time could be spent on any individual verification, as we felt this would replicate conditions that would occur in industry. Still even if we end up limited, in the end, to explorations within 5 to 10 time steps of initial states, if such explorations can be done quickly and are exhaustive, it is certain they will aid in finding design errors in industry. And, of course, we hope to extend these limits by further research.

Lastly, the results for invariance checking speak for themselves. We believe the performance would only improve given accurate input constraints. There is no logical reason to believe otherwise. Yet, it is hard to improve on the existing performance, since nearly every invariance check completed in under 1 second!

6 A Verification Methodology

Our experimental results lead us to propose an automated methodology for checking safety properties on industrial designs. In what follows, we assume a design divided up into separate blocks, as is the norm with hierarchical VLSI designs. Our methodology is as follows:

1. Annotate each design block with Boolean formulae required to hold at all time points. Call these the block's *inner assertions*.
2. Annotate each design block with Boolean formulae describing constraints on that block's inputs. Call these the block's *input constraints*.
3. Use the procedure outlined in Section 6.2 to check each block's inner assertions under its input constraints, using bounded model checking with satisfiability solving.

This methodology could be extended to include *monitors* for satisfaction of sequential constraints, in the manner described in [21], where input constraints were considered in the context of BDD based model checking.

6.1 Incorporating Constraints

Let us consider propositional input constraints with which the valuations of circuit inputs must always be consistent.

We discussed Kripke structures in Section 2, and how these can be used to model digital hardware systems. We defined the unrolled transition relation of a Kripke structure in formula 1, of Section 3.1. We can incorporate

input constraints into the unrolled transition relation as shown below, where we assume the input constraints are given by a propositional formula, c, over state variables representing inputs.

$$[\![M]\!]_k := I(\bar{s_0}) \wedge c(\bar{s_0}) \wedge T(\bar{s_0}, \bar{s_1}) \wedge c(\bar{s_1}) \wedge \cdots \wedge T(\bar{s}_{k-1}, \bar{s_k}) \wedge c(\bar{s_k}) \quad (2)$$

Below, when we speak of checking invariants under input constraints, we mean using formula 2 in place of formula 1 for the unrolled transition relation, $[\![M]\!]_k$,

6.2 Safety Property Checking Procedure

The steps for checking whether a block's inner assertion, p, is an invariant under input constraints, c, are:

1. Check whether p is a combinational tautology in the unconstrained K, using formula 1. If it is, exit.
2. Check whether p is an inductive invariant for the unconstrained K, using formula 1. If it is, exit.
3. Check whether p is a combinational tautology in the presence of input constraints, using formula 2. If it is, go to step 6.
4. Check whether p is an inductive invariant in the presence of input constraints, using formula 2. It it is, go to step 6.
5. Check if a bounded length counterexample exists to **AG** p in the presence of input constraints, using formula 2. If one is found, there is no need to examine c, since the counterexample would exist without input constraints[5]. If a counterexample is not found, go to step 6. The input constraints may need to be reformulated and this procedure repeated from step 3.
6. Check the input constraints, c, on pertinent design blocks, as explained below.

Inputs that are constrained in one design block, A, will, in general, be outputs of another design block, B. To check A's input constraints, we turn them into inner assertions for B, and check them with the above procedure. One must take precautions against circular reasoning while doing this. Circular reasoning can be detected automatically, however, and should not, therefore, be a barrier to this methodology.

The ease with which we carried out tautology and invariance checking indicates the above is entirely feasible. Searching for a counterexample, step 5, may become costly at high k values; however, this can be arbitrarily limited. It is expected that design teams would set limits for formal verification and would complement its use with simulation, for the remainder of available resources.

7 Conclusions

We can summarize the advantages of bounded model checking as follows. Bounded model checking entails only slight memory and CPU usage, especially if the user is willing to not push the time bound, k, to its limit. But there are some encouraging results for larger values of k as well [32]. The technique is extremely fast for invariance checking. Counterexamples and witnesses are of minimal length, which make them easy to understand. The technique lends itself well to automation, since it needs little by-hand intervention. The disadvantages of bounded model checking are that, at present, the implementations are limited as to the types of properties that can be checked, and there is no clear evidence the technique will consistently find long counterexample or witnesses.

From this discussion it follows that at the current stage of development bounded model checking alone can not replace traditional symbolic model checking techniques based on BDDs entirely. However in combination with traditional techniques bounded model checking is able to handle more verifications tasks consistently. Particularly for larger designs where BDDs explode, bounded model checking is often still able to find design errors or as in our experiments violations of certain environment assumptions.

Since bounded model checking is a rather recent technique there are a lot of directions for future research:

1. The use of domain knowledge to guide search in SAT procedures.
2. New techniques for approaching completeness, especially in safety property checking, where it may be the most possible.
3. Combining bounded model checking with other reduction techniques.

[5] This is implied by the theorems in [13, 18], mentioned in Section 5.2

4. Lastly, combining bounded model checking with a partial BDD approach.

The reader may also refer to [32], which presents successful heuristics for choosing decision variables for SAT procedures in the context of bounded model checking of industrial designs. In [37] early results on combining BDDs with bounded model checking are reported. See also [1] for a related approach.

While our efforts will continue in these directions, we expect the technique to be successful in the industrial arena as presently constituted, and this, we feel, will prompt increased interest in it as a research area. This is all to the good, as it will impel us faster, towards valuable solutions.

References

1. P. A. Abdulla, P. Bjesse, and N. Een. Symbolic reachability analysis based on sat-solvers. In *TACAS'00, 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 2000.
2. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking using SAT procedures instead of BDDs. In *Design Automation Conference, (DAC'99)*, June, 1999.
3. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS'99*, 1999.
4. A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu. Verifying Safety Properties of a PowerPC Microprocessor Using Symbolic Model Checking without BDDs. In *International Conference on Computer-Aided Verification (CAV'99)*, July, 1999.
5. A. Borälv. The industrial success of verification tools based on Stålmarck's Method. In O. Grumberg, editor, *International Conference on Computer-Aided Verification (CAV'97)*, number 1254 in LNCS. Springer-Verlag, 1997.
6. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
7. J. R. Burch, E. M. Clarke, and D. Long. Representing Circuits more Efficiently in Symbolic Model Checking. *Proc. Design Automation Conference*, 1991.
8. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, June 1992. Originally presented at the 1990 Symposium on Logic in Computer Science (LICS90).
9. E. M. 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.
10. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finie-state concurrent systems using temporal logic specifcations. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
11. E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebux+ Cache Coherence Protocol. In *Proc. 11th Intl. Symp. on Computer Hdwe. Description Lang. and their Applications*, April, 1993.
12. E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.
13. E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. In *Proc. 19th Ann. ACM Symp. on Principles of Prog. Lang.*, Jan., 1992.
14. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.
15. O. Coudert, J. C. Madre, and C. Berthet. Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams. In *Proc. 10th Int'l Computer Aided Verification Conference*, pages 23–32, 1990.
16. M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.
17. F. Giunchiglia and R. Sebastiani. Building Decision Procedures for Modal Logics from Propositional Decision Procedures - the case study of modal K. In *Proc. of the 13th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.
18. O. Grumberg and D. E. Long. Model Checking and Modular Verification. *ACM Transactions on Programming Languages and Systems*, 16:843–872, May, 1994.
19. D. Jackson. An Intermediate Design Language and its Analysis. In *Proceedings of Foundations of Software Engineering*, November, 1998.
20. 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/`).
21. M. Kaufmann, A. Martin, and C. Pixley. Design constraints in symbolic model checking. In *Proc. 10th Int'l Computer Aided Verification Conference*, June, 1998.
22. H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. AAAI'96*, Portland, OR, 1996.
23. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*, pages 170–172. Princeton University Press, Princeton, New Jersey, 1994.

24. T. Larrabee. Test Pattern Generation using Boolean Satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 11:4–15, 1992.
25. 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.
26. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
27. C. Pixley. A Computational Theory and Implementation of Sequential Hardware Equivalence. In *CAV'90 DIMACS series, vol.3, also DIMACS Tech. Report 90-31*, pages 293–320, 1990.
28. D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.
29. J. P. Quielle and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proc. 5th Int. Symp. in Programming*, 1981.
30. R. Raimi and J. Lear. Analyzing a PowerPC 620 Microprocessor Silicon Failure using Model Checking. In *Proc. Int'l Test Conference*, 1997.
31. R. Ranjan, A. Aziz, R. Brayton, B. Plessier, and C. Pixley. Efficient BDD Algorithms for FSM Synthesis and Verification. In *Int'l Workshop on Logic Synthesis*, 1995.
32. O. Shtrichman. Tuning sat checkers for bounded model-checking. In *Computer Aided Verification, 12th International Conference (CAV'00)*. Springer-Verlag, 2000.
33. J. P. M. Silva. Search Algorithms for Satisfiability Problems in Combinational Switching Circuits. *Ph.D. Dissertation, EECS Department, University of Michigan*, May 1995.
34. J. P. M. Silva, L. M. Siveira, and J. Marques-Silva. Algorithms for Solving Boolean Satisfiability in Combinational Circuits. In *Design, Automation and Test in Europe (DATE)*, 1999.
35. G. Stålmarck and M. Säflund. Modeling and verifying systems and software in propositional logic. In B. K. Daniels, editor, *Safety of Computer Control Systems (SAFECOMP'90)*, pages 31–36. Pergamon Press, 1990.
36. P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinational Test Generation Using Satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 15:1167–1176, 1996.
37. P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and sat procedures for efficient symbolic model checking. In *Computer Aided Verification, 12th International Conference (CAV'00)*. Springer-Verlag, 2000.
38. H. Zhang. A Decision Procedure for Propositional Logic. *Assoc. for Automated Reasoning Newsletter*, 22:1–3, 1993.
39. 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.

This research was sponsored in part by National Science Foundation (NSF) grant no. CCR-0122581.