Verifying Safety Properties of a PowerPC™[1] Microprocessor Using Symbolic Model Checking without BDDs

A. Biere[1,2,4] E. Clarke[2,4] R. Raimi[3,5] Y. Zhu[2,4]

February 1, 1999

CMU-CS-99-146

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted for CAV'99

[1]ILKD, University of Karlsruhe, Postfach 6980, 76128 Karlsruhe, Germany
Armin.Biere@ira.uka.de

[2]Computer Science Department, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213
Edmund.Clarke@cs.cmu.edu, Yunshan.Zhu@cs.cmu.edu

[3]Motorola, Inc., Somerset PowerPC Design Center
6200 Bridgepoint Pkwy., Bldg. 4, Austin, TX 78759
Richard_Raimi@email.mot.sps.com

[4]Verysys Design Automation, Inc.
42707 Lawrence Place, Fremont, CA 94538

[5]Computer Engineering Research Center
University of Texas at Austin
Austin, TX 78730

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

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

Abstract

In [2] *Bounded Model Checking* with the aid of satisfiability solving (SAT) was introduced as an alternative to traditional symbolic model checking based on solving fixpoint equations with BDDs. In this paper we show how bounded model checking can take advantage of specialized optimizations. We present a bounded version of the cone of influence reduction that works very well for verifying safety properties. We have successfully applied this idea to checking safety properties of a PowerPC microprocessor under design at Motorola's Somerset PowerPC design center. Based on that experience, we propose a verification methodology that we feel can bring model checking into the mainstream of industrial chip design.

1 Introduction

Model checking has only been partially accepted by industry as a supplement to traditional verification techniques. The reasons are that model checking, to date, has been based on BDDs or on explicit state graph exploration, and these have not been robust enough for industry. Very often, model checking cannot be carried out without the aid of by hand abstractions, or special partitioning of a design, or intuitive guesses at good BDD variable orderings. Too often, even these interventions are not sufficient for circuits which many designers would consider small, circuits with a few hundred latches and primary inputs. In an industrial environment, it is usually desired that verification be a "background" process, something that can be carried out by a program or script while a busy design team attends to creating the actual design. To date, model checking has needed too much by hand intervention for that to be possible.

Model checking [6, 18] was first proposed as a verification technique eighteen years ago. However, it was not until the discovery of symbolic model checking techniques based on BDDs [5, 9, 16] around 1990 that it was taken seriously by industry. The first BDD based symbolic model checkers were able to verify examples of significant complexity like the Futurebus+ Cache consistency Protocol [7].

Unfortunately, BDD based model checkers have suffered from the fact that ordered binary decision diagrams can require exponential space. In some cases this is due to a bad choice for variable ordering, while in others it is inevitable [4]. However, the search for a good variable ordering can be time consuming and the results unpredictable. For industrial applications, this can make model checking somewhat unreliable; and this has, no doubt, slowed its acceptance by industry.

Recently a new symbolic model checking technique, called *bounded model checking* [2], has been proposed that uses fast satisfiability solvers instead of BDDs. The advantage of satisfiability solvers like SATO [21], GRASP [19], and Stalmarck's algorithm [20] over BDDs is that they never require exponential space. In [2], results were given which showed that this new model checking technique sometimes performed much better than BDD based symbolic model checking. However, these were academic examples, and doubt remained about whether bounded model checking would work well on realistic examples.

In this paper we consider the performance of a bounded model checker, BMC [2], in verifying twenty safety properties on five complex circuits from a current, Power PC microprocessor. By any reasonable measure, BMC consistently outperformed the BDD based symbolic model checker, SMV [15]. SMV failed to terminate on all but one example, and BMC was much faster on that example. In part, this performance gain was obtained by utilizing a new *bounded cone of influence reduction* specifically designed for bounded model checking. The new reduction technique eliminates unnecessary variables and clauses in the CNF (conjunctive normal form) formula used by the satisfiability solver. In the following sections, we explain how this new reduction technique works, and describe in detail the experimental results. We believe that these results should go a long way towards confirming that bounded model checking can efficiently handle realistic examples. Since we, ourselves, are convinced of this, we propose, here, a methodology for using bounded model checking as a supplement to traditional validation techniques in industry. We believe that this methodology can be introduced in a fully automated way today, with the bounded model checking technology that is at hand. Further, we feel that this represents a significant milestone in the progress of formal verification techniques.

Our paper is organized as follows: In Section 2, we describe the model of computation in use throughout the paper, and give a brief explanation of how bounded model checking works. In Section 3 we describe the bounded cone of influence reduction. Section 4 is the heart of the paper, where we discuss the experiments we performed, using bounded model checking to check safety properties of a Power PC microprocessor. Based on the encouraging results we obtained, we propose, in Section 5, a methodology for fully automating this type of validation, in an industrial environment. The paper concludes in Section 6 with a brief summary and some directions for the future.

2 Preliminaries

In this section we give some basic definitions and briefly recall the concepts of bounded model checking presented in [2].

2.1 Models, Kripke Structures and LTL

We first consider models that can be represented by a set of initial and next state functions, and the Kripke structures that can be derived from them. The techniques presented in this paper, however, can be lifted to a more general description of state transition systems, and we go on to describe one such system, where a propositional constraint is incorporated into a Kripke structure.

Definition 1 (Model). *Let $X = \{x_1, \ldots, x_n, x_{n+1}, \ldots, x_m\}$ be a set of m Boolean variables, and let $F = \{f_1, \ldots, f_n\}$ be a set of $n \leq m$ Boolean transition functions, each a function over variables in X. Finally, let $R = \{r_1, \ldots, r_n\}$ be a set of initialization functions, each a function over variables in X. Then $M = (X, F, R)$ is called a* model.

From a model M we can construct a Kripke structure $K = (S, T, I)$ in the following way. The set of states, S, is encoded in terms of the set of variables X, from the model M, i.e., $S = \{0,1\}^m$. A state, $s \in S$, then, is an assignment to the variables in X. A state may also be considered a vector of these m variables, and we'll define such a vector as $\bar{x} = (x_1, \ldots, x_n, x_{n+1}, \ldots, x_m)$. Note that we use italic identifiers $s, s_0, \ldots$ for states (elements of $S = \{0,1\}^m$) and overhead bar identifiers $\bar{s}, \bar{s_0}$ for vectors of Boolean variables. We define present and next state versions of the variables in X, where next state variables are denoted by primes, e.g., x_j'. We define the transition relation, $T \subseteq S \times S$ and the set of initial states $I \subseteq S$ by way of their characteristic functions:

$$T(s, s') := \bigwedge_{j=1}^{n} x_j' \leftrightarrow f_j(x) \qquad \text{and} \qquad I(s) := \bigwedge_{j=1}^{n} x_j \leftrightarrow r_j(x)$$

Here, f_j and r_j are the transition and initialization functions, respectively, of the j^{th} element of the variable vector, $\bar{x}$. Note that transition and initialization functions are specified for only the first n elements from $\bar{x}$. Elements $n+1$ through m of $\bar{x}$, are meant to represent primary inputs (PIs), for instance, primary inputs to a circuit which we might represent with a model and a Kripke structure.

In practice, we will often want to consider a set of propositional constraints imposed on a system. For instance, in Section 5, we consider constraints on state variables representing primary inputs to a circuit. Given a model, $M = (X, F, R)$, a Kripke structure, $K = (S, T, I)$, derived from M, and a constraint function, c, over M's set of variables, X, we can derive a *constrained Kripke structure*, $K_c = (S, T_c, I_c)$, by conjoining c with the characteristic function of the transition relation and with the initial states predicate of K:

$$T_c(s, s') := T(s, s') \wedge c(s) \wedge c(s') \qquad \text{and} \qquad I_c(s) := I(s) \wedge c(s)$$

It is clear that c is an invariant for K_c, since all initial states satisfy c, and all successors of all states satisfying c satisfy c as well. It should be noted that, in general, imposing a constraint may produce states with no valid, outgoing transitions, or may even produce an empty set of states. This is not of major concern to us, since such conditions (a) can be easily detected, and handled as considered appropriate and (b) are unlikely to occur using constraint functions over state variables representing inputs to digital circuits, which is our intention. Digital hardware has the property, at the circuit level, of always transitioning to a next state upon all input combinations. To create a "lockup" condition where no next states are possible in digital hardware one would need to remove all input stimuli. It is unlikely that a constraint function would be proposed, in practice, that did this.

As a specification logic we use Linear Temporal Logic (LTL) with state variables as atomic propositions. Therefore we do not need to include a labeling function. In this paper we consider a subset of LTL having only unary temporal operators: the *next* operator $\mathbf{X}$, the *eventually* operator $\mathbf{F}$, and the *globally* operator $\mathbf{G}$. Additionally, formulae are assumed to be in negation normal form (NNF), i.e., negations appear only in front of atomic propositions.

We also adopt the usual semantics with respect to paths (see [2]): A path $\pi = (s_0, s_1, \ldots)$ in a model M is an infinite sequence of states in the corresponding Kripke structure K with the restriction that $T(s_i, s_{i+1})$ holds for all $i \in \mathbb{N}$. In addition we call π initialized if $I(s_0)$ holds. We use the abbreviation $\pi^i := (s_i, s_{i+1}, \ldots)$. It is often convenient to discuss the value of a component variable from the underlying vector, $\bar{x}$, in a certain state along a path. The assignment to element x_j of $\bar{x}$ in state s_i along path π is written as $s_i(j)$.

Definition 2. *For an LTL formula f and a path π in a model M we define the relation $\pi \models f$ by*

$$\begin{array}{llllll} \pi \models (\neg)x_j & \textit{iff} & s_0(j) = \mathsf{true}\ (\mathsf{false}) & \pi \models \mathbf{X}f & \textit{iff} & \pi^1 \models f \\ \pi \models f \vee g & \textit{iff} & \pi \models f \textit{ or } \pi \models g & \pi \models \mathbf{F}f & \textit{iff} & \exists i \in \mathbb{N}.\ \pi^i \models f \\ \pi \models f \wedge g & \textit{iff} & \pi \models f \textit{ and } \pi \models g & \pi \models \mathbf{G}f & \textit{iff} & \forall i \in \mathbb{N}.\ \pi^i \models f \end{array}$$

We write $M \models \mathbf{E}f$ iff there exists an initialized path π in M with $\pi \models f$. Determining whether $M \models \mathbf{E}f$ is called the existential model checking problem. Similarly, we write $M \models \mathbf{A}f$ iff for all initialized paths $\pi \models f$ and this defines the universal model checking problem. Note that the existential model checking problem can be used to solve the universal model checking problem: $M \models \mathbf{E}f$ iff $M \not\models \mathbf{A}\neg f$.

2.2 Bounded Model Checking

In bounded model checking [2] the universal model checking problem is handled by checking the dual of the formula, i.e., by solving the existential LTL model checking problem. This, in turn, is translated into a propositional satisfiability test. Efficient satisfiability solvers (SAT) such as [10, 20] are used to perform the satisfiability test. In traditional symbolic model checking [5, 15] ordered binary decision diagrams [3] are the underlying data structure.

The bounded model checking procedure of [2] works as follows. Given an LTL formula f, a model M and a bound $k \in \mathbb{N}$ we generate a propositional formula such that every satisfying assignment of this formula can be interpreted as a prefix of length k of a path π that is a witness for f ($\pi \models \mathbf{E}f$). Let d be the *diameter* of M (see below). If all such generated formulae are unsatisfiable for all $k \leq d$, then we have proven that f is not existentially valid in M ($M \not\models \mathbf{E}f$).

In generating the propositional formula we first introduce $k+1$ vectors of state variables, each representing a state in the prefix of length k, $\bar{s_0}, \ldots, \bar{s_k}$. We use the notation that $\bar{s_i}(j)$ denotes the copy of the j^{th} state variable, x_j, in any such vector, $\bar{s_i}$. Then the transition relation is unrolled k times, substituting for states the appropriately labeled state variable vectors:

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

An assignment to the propositional variables in (1) corresponds, then, to a prefix of $k+1$ states along an initialized path π in K. This initialized path π can be extended to an infinite path since our type of models are lockup free, meaning each state has at least one successor. Every initialized path can also be interpreted as an assignment that satisfies (1), in which case we write $\pi([\![\ M\ ]\!]_k) = \mathsf{true}$.

If the specification is a simple safety property $\mathbf{G}p$ where p is a propositional formula then the negated formula for which we search for a witness is $f = \mathbf{F}q$, where q is a propositional formula in NNF that is equivalent to $\neg p$. A satisfying assignment to (1) can be extended to a path that is a witness for f (and a counterexample for $\mathbf{G}p$) iff q holds at one of the $k+1$ states or equivalently the assignment also satisfies:

$$[\![\ f\ ]\!]_k := q(\bar{s_0}) \vee q(\bar{s_1}) \vee \cdots q(\bar{s_k}) \tag{2}$$

With this notation we can formulate the following theorem. It shows that bounded model checking is correct and complete for universal model checking of simple safety properties or equivalently for existential model checking of simple liveness properties.

Theorem 3. *Let $f = \mathbf{F}q$ be an LTL formula with q a propositional formula then $M \models \mathbf{E}f$ iff there exists $k \in \mathbb{N}$ for which $[\![\ f\ ]\!]_k \wedge [\![\ M\ ]\!]_k$ has a satisfying assignment.*

For general LTL formulae the translation is more involved. Particularly, back loops from the last state to a previous state have to be considered for liveness properties [2]. We omit this discussion, since our focus here is on safety properties.

The final step is to translate the generated propositional formula into CNF (conjunctive normal form) since several SAT tools, such as [19, 21], expect their input in this format. The basic mechanism for this translation is to introduce a new variable for each subformula and add constraints in clause form that relate these variables. This is done such that the resulting CNF is satisfiable iff the original formula is satisfiable [1, 17]. The translation is linear in the size of the original formula. As an example, if the generated propositional formula contains a subformula $g \leftrightarrow g_1 \wedge g_2$ and u, u_1 and u_2, in that order, are the propositional variables introduced for g, g_1 and g_2 then we add the constraint $(u \to u_1) \ \wedge \ (u \to u_2) \ \wedge \ (u_1 \wedge u_2 \to u)$ which is equivalent to $(\neg u \vee u_1) \ \wedge \ (\neg u \vee u_2) \ \wedge \ (\neg u_1 \vee \neg u_2 \vee u)$ in CNF. For other boolean operators similar constraints are used.

To prove that a safety property $\mathbf{AG}p$, with p a propositional formula, is valid in a model M, we have to show that no witness for the dual formula $\mathbf{EF}\neg p$ exists for a large enough k. This k can be chosen as the minimal number of steps in which every state can be reached. Alternatively, we can compute the *diameter* of M, a hopefully small upper bound on this. The diameter is defined as follows. Let M be a model such that for all reachable states s and t for which t is reachable from s there exists a path from s to t with at most $d-1$ intermediate states. Then d is called the *diameter* of M.

The check that a given model M has diameter d can be formulated as a validity test of a *Quantified Boolean Formula* (QBF). However with SAT we can only check the validity of propositional formulae. An alternative is to prove an upper bound on the diameter which is called the *recurrence diameter* [2]. The recurrence diameter is defined as the least number r such that at most r consecutive states in a path are different. It is the least number r for which the following propositional formula is valid:

$$T(\bar{s_0}, \bar{s_1}) \wedge \cdots \wedge T(\bar{s}_{r-1}, \bar{s_r}) \ \to \ \bigvee_{i<j} \bar{s_i} = \bar{s_j}$$

Note that in this case initial state constraints on $\bar{s_0}$ can be included as well.

3 Cone of Influence

The *Cone of Influence Reduction* is a well known technique[1] that reduces the size of a model if the propositional formulae in the specification do not depend on all state variables in the structure. For bounded model checking this technique can be specialized to the *Bounded Cone of Influence Reduction*, described below.

3.1 Classical Cone of Influence Reduction

The basic idea of the Cone of Influence (COI) reduction is to construct a dependency graph of the state variables, and then traverse it starting from the variables in the specification. The set of state variables reached during this traversal is called the COI of the variables of the specification. In this paper, we call this the "classical" COI reduction, to differentiate it from the bounded version, which we introduce later. The variables not in the classical COI can not influence the validity of the specification and can therefore be removed from the model.

Let the model M be given as in Definition 1. Then, define the immediate dependency set, $dep(x_j)$, of a state variable x_j as

$$dep(x_j) := \{x_l \mid x_l \text{ occurs in } f_j\}$$

where f_j is the transition function for x_j. The *Cone of Influence* (COI) $coi(x_j)$ of a state variable x_j is the least set of variables that contains x_j and includes $dep(x_l)$ for all $x_l \in coi(x_j)$. The COI of an LTL formula f is defined as $coi(f) := \bigcup\{coi(x_j) \mid x_j \in var(f)\}$ where $var(f)$ is the set of variables that occur in f. Obviously, $coi(x_j)$ is the solution of a least fixpoint equation. With respect to a particular LTL formula f we define a reduced model $coi(M, f)$ as $coi(M, f) := (coi(x), coi(t), coi(r))$ where all the state variables not in the COI and their corresponding transition and initialization functions are removed:

$$coi(\mathbf{x}) = (x_{\tau_1}, \ldots, x_{\tau_\rho}), \quad \mathrm{coi}(\mathbf{t}) = (f_{\tau_1}, \ldots, f_{\tau_\rho}), \quad coi(\mathbf{r}) = (r_{\tau_1}, \ldots, r_{\tau_\rho})$$

with $\{x_{\tau_1}, \ldots, x_{\tau_\rho}\} = coi(f)$. The following theorem, given without proof, allows us to reduce the size of the model in model checking if the formula does not depend on all state variables:

Theorem 4. $M \models f \quad$ *iff* $\quad \mathrm{coi}(M, f) \models f$

[1] 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 [13].

3.2 Bounded COI Reduction

The *Bounded Cone of Influence Reduction* is based on the observation that, for any state s_k along a path, the value of an arbitrary state variable, x, in the associated state variable vector, $\bar{s_k}$, can depend only on state variables in state variable vector $\bar{s_j}$, with $j < k$. In addition only the copies, in state variable vector $\bar{s}_{k-1}$, of the variables that are in $dep(x)$, can directly influence the value of x in $\bar{s_k}$. For instance if x is the only state variable appearing in the specification for which COI is being performed, then all variables other than the copy of x and those in $dep(x)$ can be removed from $\bar{s}_{k-1}$. Likewise, their corresponding transition functions in $T(\bar{s}_{k-1}, \bar{s_k})$ can be removed. Classical COI reduction would miss such reduction possibilities. This argument, that the only variables in a preceding state that need to be preserved are those in the immediate dependency set of variables in a current state, can be repeated, working backwards, until the initial state is reached. This is the case, at least, if we are only looking for violations of a safety property at state $\bar{s_k}$ (in which case, we'd be replacing (2) by $q(\mathbf{s}_k)$).

For instance consider the following model with five state variables $x_1, \dots, x_5$ and transition functions

$$f_1 = 1, \quad f_2 = x_1, \quad f_3 = x_2, \quad f_4 = x_3, \quad f_5 = x_4$$

Assume the state variables are initialized to constants:

$$r_1 = 0, \quad r_2 = 1, \quad r_3 = 1, \quad r_4 = 1, \quad r_5 = 1$$

This model has only one execution sequence in which the 0 value is moved from x_1 to x_5 over x_2, x_3 and x_4. After the 0 has reached x_5 it vanishes in the next step and all state variables stay at 1.

$$01111 \;\rightarrow\; 10111 \;\rightarrow\; 11011 \;\rightarrow\; 11101 \;\rightarrow\; 11110 \;\rightarrow\; 11111 \;\rightarrow\; \cdots$$

If the property, f, is the safety property that x_4 is always true ($f = \mathbf{G}x_4$), classical COI reduction would remove just x_5. Now, a counterexample for this property can be found by unrolling the transition relation three times ($k = 3$). Let us assume that we only want to check for $\neg x_4$ in the last state s_3. To apply bounded COI we observe that x_4 in $\bar{s_3}$ only depends on x_3 in $\bar{s_2}$ which in turn depends on x_2 in $\bar{s_1}$. The value of x_2 in $\bar{s_1}$ only depends on the initial value of x_0. Therefore we can remove all other variables and their corresponding transitions. For example, in the transition from $\bar{s_1}$ to $\bar{s_2}$, the variable x_1 is not pertinent, and can be eliminated.

In this example, the application of bounded COI reduction would result in the following propositional formula:

$$\bar{s_0}(1) \leftrightarrow 0 \wedge \bar{s_1}(2) \leftrightarrow \bar{s_0}(1) \wedge \bar{s_2}(3) \leftrightarrow \bar{s_0}(2) \wedge \bar{s_3}(4) \leftrightarrow \bar{s_0}(3) \wedge \neg \bar{s_3}(4)$$

This formula is satisfiable, and its only satisfying assignment can be extended to a counterexample which is the only execution sequence falsifying the original formula. Without bounded COI, 12 more equalities would have been necessary.

For a formal treatment of the bounded COI reduction we define a special type of immediate dependency set, $bdep(\bar{s_i(j)})$, for the components $\bar{s_i}(j)$ of the state variable vectors representing a prefix of a path,

$$bdep(\bar{s_i}(j)) \quad := \quad \text{if } i = 0 \text{ then } \emptyset \text{ else } \{\bar{s}_{i-1}(l) \mid x_l \in dep(x_j)\}$$

The bounded COI $bcoi(s_i(j))$ of a component of a state variables vector, $\bar{s_i}(j)$ is defined, recursively, as the least set of variables that includes $\bar{s_i}(j)$ and includes all elements from the immediate dependency sets of all variables in $bcoi(s_i(j))$. Finally we define the bounded COI of an LTL formula f as $\text{bcoi}(f) := \bigcup\{\text{bcoi}(\bar{s_i}(j)) \mid \bar{s_i}(j) \in \text{var}(\llbracket f \rrbracket_k)\}$. In (1) we can now remove all factors of the form $\bar{s_i}(j) \leftrightarrow \dots$ where $\bar{s_i}(j) \notin bcoi(f)$ and derive (for simplicity, we do not remove initial state assignments):

$$\llbracket M \rrbracket_k^{\text{bcoi}(f)} := I(\bar{s_0}) \wedge T_0(\bar{s_0}, \bar{s_1}) \wedge \cdots \wedge T_{k-1}(\bar{s}_{k-1}, \bar{s_k})$$

where

$$T_{i-1}(\bar{s}_{i-1}, \bar{s_i}) := \bigwedge_{\bar{s_i}(j) \in \text{bcoi}(f)} \bar{s_i}(j) \leftrightarrow f_j(\bar{s}_{i-1}) \qquad \text{for } i = 1 \dots k$$

The correctness of the bounded COI reduction is formulated in the following theorem (compare with theorem 3).

Theorem 5. *Let $f = \mathbf{F}q$ be an LTL formula with q a propositional formula. Then $[\![\ f\ ]\!]_k \wedge [\![\ M\ ]\!]_k$ is satisfiable iff $[\![\ f\ ]\!]_k \wedge [\![\ M\ ]\!]_k^{\mathrm{bcoi}(f)}$ is satisfiable.*

In the bounded model checker, BMC, we have implemented bounded COI as follows. After the propositional formula has been generated, equalities are removed that do not contain any variable of the bounded COI or represent an assignment of a variable not included in the bounded COI. By the latter, we refer to the fact that all equalities in the propositional formula generated by BMC are terms of the form $\bar{s}_i(j) \leftrightarrow f_j(\bar{s}_{i-1})$, where f_j is the transition function of $\bar{s}_i(j)$. This term would be dropped if $\bar{s}_i(j)$ was not in the bounded COI. In BMC, we added an integer array of size $k+1$ for each variable $x \in \bar{x}$. In this array, entry i is set to 1 during the dependency analysis iff the copy of the state variable in the i^{th} state is in the bounded COI of the specification. The dependency analysis uses the same graph structure as classical COI to represent dependencies between variables. The information stored in the array is generated in a depth first traversal through this graph while maintaining a counter that represents the present level in the traversal.

For liveness properties a back loop from the last state, represented by $\bar{s_k}$, to a previous state, $\bar{s_l}$, has to exist [2]. Thus, $\bar{s_k}(j)$ has to be included into the bounded COI iff $\bar{s_l}(j)$ is contained in the bounded COI. Since our focus is on safety properties, we omit further discussion of bounded COI for liveness properties.

4 Experiments

We ran experiments using the bounded model checker, BMC, to test out the ideas set forth in this paper. BMC accepts files in a subset of the input format used by the widely known SMV model checker [15]. The experiments were run on subcircuits from a PowerPC microprocessor currently under design at Motorola's Somerset design center, in Austin, Texas. We believe that the results demonstrate the utility not only of bounded COI, but also of bounded model checking in an industrial setting.

While a processor is under design at Somerset, designers insert assertions into the RTL simulation model. These Boolean expressions are important safety properties, i.e., they should hold at all time points. If an assertion is ever false during simulation, an immediate error is flagged. In our experiments, we checked, with BMC, 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[2] 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. 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 really believe the initial states actually were

[2] 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.

those satisfying p. Rather, we knew each design block had only a single, valid initial state, which may or may not satisfy 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 either did not satisfy p or had a successor, s', which did not, then the check would fail. Therefore, this technique can only show that p *is* an invariant, but cannot show that it is not. However, we found this type of invariant 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 5.

The output of BMC is a Boolean formula in CNF (conjunctive normal form) that is given to a satisfiability solver. In these experiments, we used both the GRASP [19] and SATO [21] satisfiability solvers. When giving results, we do not indicate from which solver they came, rather, we just show the best results from the two.

The SMV input files were given to a recent version of the SMV model checker, in order 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 reachability analysis, of false $\mathbf{AG}p$ properties. Note that the verifier did not need, under these conditions, to compute a fixpoint if a counterexample existed. This 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.

4.1 Environment Modeling

A typical PowerPC microprocessor simulation model can have hundreds or even thousands of assertions. We wanted to demonstrate that bounded model checking could quickly prove that some of these held, eliminating the need to check them during simulation. Additionally, for assertions that did not hold, we wanted to demonstrate that useful information on possible failure modes could be generated.

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 [12]. However, if a safety property holds with a totally unconstrained environment, then it holds in the real environment. Given Kripke structures M' and M, M' representing a design block with an unconstrained environment and M the same block with its real, constrained environment, it is obvious that M' simulates M, i.e. $M \leq M'$ in the simulation preorder. It has been shown in [8, 11] that if f is an $ACTL$ formula, as are all the properties in these experiments, then $M' \models f$ implies $M \models f$.

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, or hope that simulation will find any real violations that are possible. 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, e.g., a page fault, outstanding, and so on.

4.2 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

Before COI

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

After (classical) COI

In the right table we report the sizes of the circuits 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.

The effectiveness of bounded COI can best be measured by looking at the CNF output of BMC. We ran BMC for values of k of $0, 1, 2, 3, 4, 5, 10, 15$ and 20, on each specification. For each of these, we had BMC create CNF files having no COI reduction, having only classical COI, and having both classical and bounded COI.

In the table labeled "Average Bounded COI Reduction", we give average sizes of all these CNF files, for each value of k. We summed the number of literals and clauses in all the CNF files for each k, for all specification for all design blocks for that k, and divided by the total number of such files. While averaging can sometimes obscure the common occurrence of a phenomenon, we performed a by hand inspection to verify this would not be the case. In the table, we give the average number of literals to the left of a slash, and the average number of clauses to the right. It can be seen that the advantage of bounded COI decreases with increasing k. 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, at values of k up to around 10, bounded COI gives distinct benefit. Since we expect bounded model checking to be most effective at finding short counterexamples, and, since tautology and invariance checking are run at low k, we feel bounded COI is helping augment the system's strengths.

The table labeled "Tautology and Invariance Checking" gives the results of these types of checks 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 must both hold for a Boolean formula to be an invariant. A"Y" in the leftmost part of a column indicates the condition holding, an "N" that it does not, The center and rightmost parts of a column give time and memory usage, respectively. 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 5.

The result on *bbc* specification 2 is interesting. This property is preserved by the transition relation—but does not hold in the initial state! Separating the check on initial states from the check on the transition relation enabled us to quickly see this. We were somewhat surprised by the small number of assertions that were 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 tautologies. See Section 5 for a discussion of this.

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 - -

Tautology and Invariance Checking

k	Bounded COI	Classic COI	No COI
0	137 / 449	234 / 546	376 / 688
1	1023 / 3762	1801 / 6790	3402 / 12749
2	2330 / 8946	3367 / 13025	6426 / 24801
3	3755 / 14631	4931 / 19259	9450 / 36851
4	5259 / 20608	6496 / 25492	12473 / 48901
5	6820 / 26821	8060 / 31725	15496 / 60951
10	14643 / 57987	15883 / 62891	30613 / 121202
15	22466 / 89153	23706 / 94057	45730 / 181452
20	30288 / 120319	31529 / 125223	60846 / 241702

Average Bounded COI Reduction

circuit	spec	long k	vars	clauses	time	mem	holds	fail k
bbc	1	4	7873	30174	35.4	NR	Y	
bbc	2	15	34585	93922	5.5	84	N	0
bbc	3	10	16814	63300	58	NR	Y	
bbc	4	5	9487	35658	18	NR	Y	
ccc	1	5	9396	40450	1.3	36	N	1
ccc	2	5	9148	38841	1.4	39	N	1
cdc	1	20	49167	207764	128	77	N	2
cdc	2	20	50825	213137	4.7	NR	Y	
cdc	3	20	50571	213614	4.7	NR	Y	
cdc	4	20	50491	212406	4.8	NR	Y	
dlc	1	20	18378	71291	2.9	64	N	2
dlc	2	20	18024	69830	2,8	63	N	2
dlc	3	20	17603	68333	2.6	60	N	2
dlc	4	20	18085	69942	2.73	61	N	1
dlc	5	20	18378	71291	2.9	60	N	2
dlc	6	20	17712	68714	2.7	NR	N	2
dlc	7	20	16217	63781	2.4	64	N	0
sdc	1	4	5554	20893	72	14	Y	
sdc	2	4	5545	20841	548	21	Y	
sdc	3	20	4119	15168	-	3	N	0

Highest k Values

The table labeled “Highest k Values” shows the results of increasing the time bound, k. These runs, again, were with bounded COI. We ran to large k even after finding counterexamples, or finding that the properties were invariants, at lower k. We did so simply to get statistics on runs with large k values. We found that the satisfiability solving went quickly at high values of k if counterexamples existed at low values of k or if the property was an invariant. While these are quite different outcomes, we surmised that, in both cases, checking satisfiability might be much easier.

In the table for the different k runs, NR means not recorded (data lost). It was sometimes difficult to obtain memory usage statistics during satisfiability solving; but, it should be kept in mind this often does not exceed that needed to store the CNF formula. Time is given in seconds, memory usage in megabytes, with dashes appearing where these were insignificant. The “vars” and “clauses” columns give the number of literals and clauses in the CNF file for the highest value of k on which satisfiability solving completed, the k in the “long k” column. The time and memory usage lisings are for satisfiability solving on the CNF file for this, particular k value. A “Y” in the “holds” column indicates the property held through all values of k tested, and an “N” indicates a counterexample was found. Counterexamples were found for 12 of the 20 properties. When these were found, the “fail k” column gives the the first k at which a counterexample appeared. Time and memory consumption are not listed for the runs giving counterexamples, because the satisfiability solving took less than a second, and no more than 5 megabytes of memory, in each case!

Lastly, the results of BDD-based model checking are that SMV was given each of the 20 properties separately, but completed only one of the these verifications. The 19 others all timed out at one hour of wall clock time, and, in each of these cases, SMV could not build the BDDs for the transition relation in the alloted 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 A Verification Methodology

The experimental results of Section 4.2 lead us to believe that the checking of safety properties, an extremely important class of properties, can be efficiently automated for industrial chip designs. In what follows, we assume a design divided up into separate blocks, as is the norm with hierarchical VLSI designs. A methodology we would recommend, and which can be implemented with existing technology, is as follows.

- Annotate each design block with Boolean formulae that should hold at all time points. Call these the block’s *inner assertions*.
- Annotate each design block with Boolean formulae describing constraints which inputs to that block must obey. Call these the block’s *input constraints*.
- Use the procedure outlined in Section 5.2 to check each block’s inner assertions under its input constraints, using bounded model checking with satisfiability solving. Implement this as a program that runs as a background job.

The goal is to determine whether, for each block, each inner assertion, p, is an invariant.

The input constraints would be written in terms of conditions that should always hold. For instance, if circuit inputs a and b should never be true at the same time, the constraint would be written as $\neg(a \wedge b)$. Ideally, we would like to have design teams notate sequential input constraints as well, which could be handled as LTL formulae. But, there are limitations as to which properties bounded model checking can currently check, and we focus here on what can be implemented, today.

The methodology we outline here should be compared to that proposed in [12], where input constraints were considered in the context of BDD based model checking.

5.1 Modeling Constrained Systems

Let us assume we have translated the description of a design block, into a Kripke structure, K. In the presence of propositional input constraints, c, we need to check whether an inner block assertion, p, is an invariant of the *constrained Kripke structure*, K_c, derived from K as described in Section 2.1. However, for bounded model checking, we need not form K_c directly, and can work with the unconstrained Kripke structure, K. Note that unrolling the transition relation of the constrained structure, K_c, as per formula (1) of Section 2.2, is entirely equivalent to unrolling the transition relation of the unconstrained structure, K, and conjoining each term with the constraint function, c, at each time step:

$$[\![\ M \ ]\!]_k := I(\bar{s_0}) \wedge c(\bar{s_0}) \wedge T(\bar{s_0}, \bar{s_1}) \wedge c(\bar{s_1}) \wedge \cdots \wedge T(\bar{s}_{k-1}, \bar{s_k}) \wedge c(\bar{s_k}) \qquad (3)$$

Being able to work with the unconstrained system makes the implementation simple.

There are potential limitations to COI reductions on constrained systems. The constraint function, c, may depend upon variables not in the classical COI of the specification, p. Likewise, when applying bounded COI, the constraint function, at a given time step, may depend upon variables not in the bounded COI, at that time step. So, it may be that variables get reintroduced, via the constraint function, that COI reductions, classical or bounded, would have removed. However, we do not expect this to be a major problem. We expect most constraints to be given as individual Boolean formulae to be conjoined together, and thus, unneeded variables may often be eliminated by dropping one or more conjuncts which contain only those.

5.2 Safety Property Checking Procedure

Let c be the block input constraints, for some design block, D, let p be an inner block assertion for D, let K be D's unconstrained Kripke structure, and let K_c be its constrained Kripke structure. When checking a specification over K_c, assume the transition relation of K will be unrolled as per formula (3), directly above; and, when we checking a specification over K, assume the transition relation of K will be unrolled as per formula (1) of Section 2.2. The steps for checking whether p is an invariant under the block input constraints, c, are outlined, below.

1. Check whether p is a combinational tautology in K. If it is, then p holds regardless of c, and we do not need to check further.
2. Check whether p is otherwise an invariant for K. If it is, p is an invariant regardless of c, and we need not check further.
3. Check whether p is a combinational tautology in the constrained Kripke structure, K_c. If it is, go to step 6 to check c.
4. Check whether p is otherwise an invariant for K_c. It it is, go to step 6 to check c.
5. Check if a bounded length counterexample exists to $\mathbf{AG}p$ in K_c. If one is found, there is no need to examine c, since the counterexample would exist without input constraints[3]. If a counterexample is not found, we do need to check c (i.e., go to step 6). The input constraints may need to be reformulated and the check on p in K_c repeated, i.e., this procedure repreated, starting at step 3.
6. Check the input constraints, c, for being an invariant of design blocks pertinent to it (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 design block B, and use the procedure outlined above to check them. One must take precautions, here, against circular reasoning. Eventually, a chain of assumptions must be guaranteed by discharging the last unconditionally. The detection of circular reasoning is possible to automate, however, and so should not be a barrier to using this methodology.

There are two reasons to check both the unconstrained and constrained systems, as we do above:

1. It may not be necessary to check c, if all of a block's inner assertions, p, pass on step 1 or 2.

[3] This is implied by the theorems for ACTL formulae in [8, 11], which we refered to in Section 4.1

2. It may be useful to know which inner assertions are invariants regardless of c.

Regarding the last point, it is comforting to know that a design block's correctness is independent of behaviors at its inputs. In fact, such independence may even be required for some design blocks.

In the experiments of Section 4.2, we could not follow the above procedure, as we did not have a list of input constraints. But, the ease with which we carried out tautology and invariance checking indicates that the above is entirely feasible. It should be noted that bounded COI is most effective at low k values, and so steps 1 through 4, above, benefit a great deal from this optimization. This would be important when hundreds of safety properties need to be checked at frequent intervals. Searching for a counterexample, step 5, may become CPU and memory intensive at high k values; however, this can be arbitrarily limited, in order to check a large number of properties. For instance, we set wall clock time limits in our experiments. It is expected that design teams would operate in this manner, i.e., give a percentage of limited resources to formal verification, and then hope that simulation would complement this effort with the remainder of available resources.

6 Conclusion

In this paper, we have outlined a specialized version of cone of influence reduction for bounded model checking. The concept of bounded model checking is just beginning to be explored, and we expect other reduction techniques will be found. In our future research, we will seek these out.

We were fortunate to have had access to a large and complex PowerPC microprocessor design for our experiments. Previous experiments with bounded model checking using satisfiability solving had been confined to academic examples, and could possibly have been dismissed as unrealistic. The present set of experimental results, however, are compelling. They tell us that, for some types of properties, these new techniques have increased the efficiency of model checking by orders of magnitude, with respect to time and memory usage. Our results using BDD based model checking, in which the SMV model checker failed to complete on all but one of 20 examples, accentuate this difference. We still expect, however, that BDD-based model checking methods will have a place in the overall verification "arsenal". Certainly, they seem to be the only techniques that can presently find long counterexamples, though, of course, they can only do so on designs that fall within their capacity limitations.

Lastly, our experiments lead us to believe that new and newly appropriate verification methodologies can be introduced in industry, to take advantage of these new efficiencies. In this paper, we have outlined one such procedure for checking safety properties. Our hope is that once such methodologies are accepted, the widespread use of model checking will illuminate further possibilities for optimization.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *DAC'99*, 1999. submitted.

[2] A. Biere, A. Cimatti, Edmund M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS'99*, 1999. to appear.

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

[4] R. E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, 1991.

[5] 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).

[6] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1981.

[7] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In L. Claesen, editor, *Proc. 11th CHDL*. North-Holland, April 1993.

[8] 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.

[9] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In Kurshan and Clarke [14], pages 23–32.

[10] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[11] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16:843–872, May, 1994.

[12] M. Kaufmann, A. Martin, and C. Pixley. Design constraints in symbolic model checking. In *Proc. 10th Int'l Computer Aided Verification Converence*, June, 1998.

[13] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*, pages 170–172. Princeton University Press, Princeton, New Jersey, 1994.

[14] R. P. Kurshan and E. M. Clarke, editors. *Computer-Aided Verification (CAV'90)*, volume 531 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

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

[16] C. Pixley. Verifying temporal properties of sequential machines without building their state diagrams. In Kurshan and Clarke [14], pages 54–64.

[17] D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[18] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int. Symp. in Programming*, 1981.

[19] J. P. M. Silva. Search algorithms for satisfiability problems in combinational switching circuits. *Ph.D. Dissertation, EECS Department, University of Michigan*, May 1995.

[20] G. Stalmarck 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.

[21] H. Zhang. A decision procedure for propositional logic. *Assoc. for Automated Reasoning Newsletter*, 22:1–3, 1993.