Verifying Safety Properties of a PowerPC™* Microprocessor Using Symbolic Model Checking without BDDs**

Armin Biere[1,2,3], Edmund Clarke[2,3], Richard Raimi[4,5], and Yunshan Zhu[2,3]

[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] Verysys Design Automation, Inc.
42707 Lawrence Place, Fremont, CA 94538
[4] Motorola, Inc., Somerset PowerPC Design Center
6200 Bridgepoint Pkwy., Bldg. 4, Austin, TX 78759
Richard_Raimi@email.mot.sps.com
[5] Computer Engineering Research Center, University of Texas at Austin
Austin, TX 78730

Abstract. In [1] *Bounded Model Checking* with the aid of satisfiability solving (SAT) was introduced as an alternative to symbolic model checking 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. We have successfully applied this idea in checking safety properties of a PowerPC microprocessor 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 reason is that model checking, which, to date, has been based on BDDs or on explicit state graph exploration, has not been robust enough for industry.

Model checking [3,12] was first proposed as a verification technique eighteen years ago. However, it was not until the discovery of symbolic model checking

* 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.

N. Halbwachs and D. Peled (Eds.): CAV'99, LNCS 1633, pp. 60–71, 1999.

techniques based on BDDs [2,5,10] around 1990 that it was taken seriously by industry. Unfortunately, BDD based model checkers have suffered from the fact that ordered binary decision diagrams can require exponential space. Recently a new technique called *bounded model checking* [1] has been proposed that uses fast satisfiability solvers instead of BDDs. The advantage of satisfiability solvers like SATO [15], GRASP [13], and Stalmarck's algorithm [14] is that they never require exponential space. In [1], it was shown that this new technique sometimes performed much better than BDD based symbolic model checking. However, the performance was obtained on academic examples, and doubt remained about whether bounded model checking would work well on industrial examples.

In this paper we consider the performance of a bounded model checker, BMC [1], in verifying twenty safety properties on five complex circuits from a PowerPC microprocessor. By any reasonable measure, BMC consistently outperformed the BDD based symbolic model checker, SMV [9]. In part, this performance gain was obtained by utilizing a new *bounded cone of influence reduction* technique which reduces the size of the CNF (conjunctive normal form) formula given to the satisfiability solver.

We believe our new experimental results confirm that bounded model checking can handle industrial 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 feel that this represents a significant milestone for formal verification.

2 Models, Kripke Structures and Safety Properties

For brevity, we focus on the application of bounded model checking to safety properties. The reader is referred to [1] for a more complete treatment of bounded model checking.

We first consider models that can be represented by a set of initial and next state functions.

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 an encoding of the variables in X, i.e., $S = \{0,1\}^m$. A state may also be considered a vector of these m variables, $\bar{x} = (x_1, \ldots, x_n, x_{n+1}, \ldots, x_m)$. 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, denoting the latter with primes, e.g., x'_j. The variables in X serve as atomic propositions, and obviate the need for a labeling function. We define

the transition relation, $T \subseteq S \times S$ and the set of initial states $I \subseteq S$ via 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 not specified for elements $n+1$ through m of $\bar{x}$. These represent primary inputs (PIs) to an underlying sequential circuit.

In practice, we will often consider a set of propositional constraints imposed on a system. Given a model, $M = (X, F, R)$, a constraint function, c, over X, and a Kripke structure, $K = (S, T, I)$, derived from M, a *constrained Kripke structure*, $K_c = (S, T_c, I_c)$, in which c is an invariant, can be obtained as follows:

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

As a specification logic we use Linear Temporal Logic (LTL). In this paper we consider only the unary temporal operators: *eventually*, **F**, and *globally*, **G**.

A path $\pi = (s_0, s_1, \ldots)$ in a model M is an infinite sequence of states in the corresponding Kripke structure K such that $T(s_i, s_{i+1})$ holds for all $i \in \mathbb{N}$. We call π initialized if $I(s_0)$ holds. 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 $s_i(j)$.

We are interested in determining whether $M \models \mathbf{AG}p$ holds, i.e., whether p, a propositional formula, holds in every state along every initialized path is some model, M. We approach this in two ways: (a) by searching for a finite length counterexample showing $M \models \mathbf{EF}\neg p$, or (b) by proving that p is an inductive invariant for M. These have in common that, in both cases, it is not necessary to search unto the *diameter* of the structure, the diameter being that minimal number of transitions sufficient for reaching any state from an initial state.

3 Bounded Model Checking for Safety Properties

In bounded model checking [1] the user specifies a number of time steps, k, for searching from initial states. A propositional formula is then generated by introducing $k+1$ vectors of state variables, each representing a state in the prefix of length k, $\bar{s_0}, \ldots, \bar{s_k}$. Then the transition relation is unrolled k times, substituting for states the appropriately labeled state variable vectors:

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

Every initialized path of the model M corresponds to an assignment that satisfies (1). When checking a safety property, $\mathbf{G}p$, where p is a propositional formula, we search for a witness to $f = \mathbf{F}q$, where $q = \neg p$. 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:

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

The final step is to translate the conjunction of (1) and (2) into CNF and check it with SAT tools such as [13,15,14]. Translation into CNF is described in [11].

4 Classical and Bounded Cone of Influence Reduction

The *Cone of Influence Reduction* is a well known technique[1]. For bounded model checking this technique can be specialized to the *Bounded Cone of Influence Reduction*, described below.

The basic idea of COI reduction is to construct a dependency graph of the state variables, rooted at the variables in the specification. The set of state variables in the graph is called the COI of the specification. In this paper, we call this the "classical" COI reduction. Variables not in the classical COI can not influence the validity of the specification and can therefore be removed from the model.

Let $dep(x)$ be the set of successors to variable x in the state variable dependency graph, i.e., the set of variables in the support of the transition function for x. 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 $j < k$. Thus, it is only the copies, in $\bar{s}_{k-1}$, of the variables that are in $dep(x)$ that can determine the value of x in $\bar{s_k}$. Other state variables, and their corresponding transition functions can be removed. If we are looking for violations of a safety property at state $\bar{s_k}$, this argument can be repeated, working backwards, until the initial state is reached.

For instance, consider the following model with five state variables $x_1, \ldots, x_5$ 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. After the 0 has reached x_5 it vanishes, and all state variables stay at 1.

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

[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 [8].

If the property to check is the safety property that x_4 is always true, i.e., $\mathbf{G}x_4$, classical COI reduction would remove just x_5. Now, a counterexample for this property can be found by unrolling the transition relation three times. Let us assume that we only want to check for a counterexample 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 $\bar{s_1}$, which only depends on the initial value of x_1. Therefore we can remove all other variables and their corresponding transitions. This application of bounded COI reduction results in the following 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 for the original formula, $\mathbf{G}x_4$. Without bounded COI, 12 more equalities would have been necessary.

For a formal treatment of the bounded COI reduction we define the *bounded dependency set*, $bdep(\bar{s_i}(j))$, of a component, $\bar{s_i}(j)$, of state variable vector, $\bar{s_i}$, as follows. Here, $\bar{s_i}$ represents a state s_i along a path prefix:

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

The *bounded COI*, $bcoi(\bar{s_i}(j))$, of component $\bar{s_i}(j)$ is defined, recursively, as the least set of variables that includes $\bar{s_i}(j)$, and includes, for each $\bar{s}_{i-1}(l) \in bdep(\bar{s_i}(j))$, if any, the variables in $bcoi(\bar{s}_{i-1}(l))$.

For a fixed k, the length of the considered prefix, we define the bounded COI of an LTL formula, f, as:

$$\text{bcoi}(k, f) := \{x \in bcoi(\bar{s_i}(j)) \mid \bar{s_i}(j) \in \text{var}([\![\ f\ ]\!]_k)\}$$

where $\text{var}([\![\ f\ ]\!]_k)$ is the set of variables of $[\![\ f\ ]\!]_k$.

In (1) we can now remove all factors of the form $\bar{s_i}(j) \leftrightarrow \ldots$ where $\bar{s_i}(j) \notin bcoi(f)$, and derive (for simplicity, we do not remove initial state assignments):

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

where

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

The correctness of the bounded COI reduction is formulated in the following theorem.

Theorem 1. *Let $f = \mathbf{F}q$ be an LTL formula with q a propositional formula. Then $[\![\ f\ ]\!]_k \wedge [\![\ M\ ]\!]_k$ is satisfiable iff $[\![\ f\ ]\!]_k \wedge [\![\ M\ ]\!]_k^{\text{bcoi}(k,f)}$ is satisfiable.*

5 Experiments

We used the bounded model checker, BMC, on subcircuits from a PowerPC microprocessor under design at Motorola's Somerset design center, in Austin,

Texas. BMC accepts a subset of the input format used by the widely known SMV model checker [9].

When a processor is under design at Somerset, designers insert assertions into the RTL simulation model. These Boolean expressions are important safety properties. The simulator flags an error if these are ever false. We checked, with BMC, 20 assertions chosen from 5 different processor design blocks. For each assertion, p, we:

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

Each circuit latch was represented by a state variable having individual next state and initial state assignments. For the latter, we assigned the 0 or 1 value the latch would have after a designated power-on-reset sequence known to the designer. Primary inputs were modeled as unconstrained state variables, having neither next state nor initial state assignments.

For combinational tautology checking we deleted all initialization statements and ran BMC with $k = 0$, giving the propositional formula, p, as the specification. Under these conditions, the specification could hold only if p held for all assignments to the variables in its support.

We then checked whether p was an inductive invariant. A formula is an inductive invariant if it holds in all initial states and is preserved by the transition relation. Leaving all initialization assignments intact, for each design block and each formula p, we gave p as the specification and set $k = 0$. This determined whether each p held in the single, valid initial state of each design. Then, for each design block and for each formula, p, we removed all initialization assignments and specified p as an initial state predicate. We set $k = 1$ and checked the specification $\mathbf{AG}p$. If the specification held, this meant the successors of every state satisfying p, also satisfied p. Note that $\mathbf{AG}p$ could fail to hold exclusively due to transitions out of unreachable states. Therefore, this technique can only show that p *is* an invariant, it cannot show that it is not.

The output of BMC is a Boolean formula in CNF that is given to a satisfiability solver. In these experiments, we used both the GRASP [13] and SATO [15] satisfiability solvers. When giving results, we give the best result from the two.

We also ran a recent version of the SMV model checker on each of the 20 $\mathbf{AG}p$ specifications. We used command line options that enabled the early detection, during reachability analysis, of false $\mathbf{AG}p$ properties, so that SMV did not need to compute a fixpoint. This made the comparison to BMC more appropriate. We also enabled dynamic variable ordering when running SMV, and used a partitioned transition relation.

All experiments were run with wall clock time limits. The satisfiability solvers had 15 minutes for each run, while SMV had an hour. BMC was not timed, as the task of translating to CNF is usually done quite quickly. The satisfiability solving and SMV runs were done on RS6000 model 390 workstations, having 256 megabytes of local memory.

We did not model the interfaces between the 5 design blocks and the rest of the microprocessor or the external computer system in which the processor would be placed. This is commonly referred to as "environment modeling". One would ideally like to do environment modeling, since subcircuits usually work correctly only under certain input constraints. However, one will get true positives for safety properties with a totally unconstrained 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 [4,6] that if f is an $ACTL$ formula, as are all the properties in these experiments, then $M' \models f$ implies $M \models f$.

Our experiments did result in false negatives. Upon inspection, and after checking with circuit designers, it seems all the counterexamples generated were due to impossible input behaviors. However, our purpose in these experiments was to show the capacity and speed of bounded model checking, and the false negatives did not obscure these results. We discuss, in Section 6, a methodology wherein false negatives could be lessened or eliminated, by incorporating input constraints into the bounded model checking. We certainly feel this would be the way to use bounded model checking in a non-experimental, industrial application. The reader may also want to refer to [7], where input constraints are considered in a BDD based verification environment.

5.1 Experimental Results

The 5 design blocks we chose all came from a single PowerPC microprocessor, and 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

On the left, we report the original size of each circuit, and on the right, the sizes after classical COI reduction. Each specification is given an arbitrary numeric label. These do not relate across design blocks, e.g., specification 2 of *dlc* is in no way related to specification 2 of *sdc*. Many properties involved much the same circuitry on a design block, as can be seen by the large number of cones of influence having identical numbers of latches and PIs. However, these reduced circuits were not identical, though they may have differed only in how

the variables in the specification depended, combinationally, upon latches and PIs.

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	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	Y
sdc	2	N	N	Y
sdc	3	N	N	N

Tautology and Invariance Checking

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, only classical COI, and both classical and bounded COI. In the table labeled "Average Bounded COI Reduction", we give average sizes of all these CNF files. We averaged the number of literals and clauses (a clause is a disjunct of literals) in all the CNF files for each k, i.e., for all specifications, for all design blocks,

for that k. We checked, by hand, that this averaging did not obscure the median case. In the table, we give to the left of a slash, the average number of literals for a k value, and to the right, the average number of clauses. It can be seen that the advantage of bounded COI decreases with increasing k. Intuitively, this is because, going out in time, eventually values are computed for all state variables in the classical cone of influence. However, at k up to 10, bounded COI gives distinct benefit. Since bounded model checking seems to be most effective at finding short counterexamples, and, since tautology and invariance checking are run at low k, we feel bounded COI augments the system's strengths.

The table labeled "Tautology and Invariance Checking" has columns for tautology checking, for preservation by the transition relation and for preservation in initial states. The last two must both hold for a formula to be an inductive invariant. These runs were done with bounded COI enabled. A "Y" in a column indicates a condition holding, an "N" that it does not. Time and memory usage are not listed, since these were ≤ 1 second ≤ 5 megabytes in all but three cases. In the worst case, *sdc* specification 2, 60 seconds of CPU time and 6.5 megabytes of memory were required, for checking preservation by the transition relation. Clearly, tautology and invariance checking can be remarkably inexpensive. In contrast, these can be quite costly with BDD based methods.

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 k. These runs, again, were with bounded COI. We ran to large k regardless of whether

we found counterexamples, or determined a property was an invariant, at lower k. It was sometimes difficult to obtain memory usage statistics during satisfiability solving; but, this usually does not exceed that needed to store the CNF formula. In the table, NR means not recorded (data unavailable). 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 listings are for satisfiability solving at this highest 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. 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 BDD-based model checker, SMV, completed only one of the 20 verifications it was given. The 19 others all timed out at one hour of wall clock time, with SMV unable to build the BDDs for the partitioned transition relation. 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!

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 6.1 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 [7], where input constraints were considered in the context of BDD based model checking.

6.1 Safety Property Checking Procedure

Let us consider a Kripke structure, K, for a design block having input constraints, c. A *constrained Kripke structure*, K_c, can be derived from K as in

Section 2. To check whether an inner block assertion, p, is an invariant in K_c, we need not work with K_c directly. Unrolling the transition relation of K_c, as per formula (1) of Section 3, is entirely equivalent to unrolling the transition relation of K, and conjoining each term with the constraint function, c:

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

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 K. If it is, exit.
2. Check whether p is an inductive invariant for K. If it is, exit.
3. Check whether p is a combinational tautology in K_c. If it is, go to step 6.
4. Check whether p is an inductive invariant for K_c. It it is, go to step 6.
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[2]. 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, however, against circular reasoning. 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 Conclusion

In this paper, we have outlined a specialized version of cone of influence reduction for bounded model checking. The present set of experiments, on a large and complex PowerPC microprocessor, are compelling. They tell us that, for some applications, the efficiency of model checking has increased by orders of magnitude. The fact that the BDD-based SMV model checker failed to complete on all but one of 20 examples, underscores this point. We still believe, however, that BDD-based model checking fills important needs. Certainly, it seems to be the

[2] This is implied by the theorems for ACTL formulae in [4,6], which we referred to in Section 5

only technique that can presently find long counterexamples, though, of course, this can be done only for designs that fall within its capacity limitations.

We feel that new verification methodologies can now be introduced in industry, to take advantage of bounded model checking. We have outlined one such procedure here, for checking safety properties. Our hope is that the widened use of model checking will illuminate further possibilities for optimization.

References

1. A. Biere, A. Cimatti, Edmund M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS'99*, 1999. to appear. 60, 61, 61, 61, 61, 62
2. 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). 61
3. 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. 60
4. 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. 66, 70
5. 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 Converence*, pages 23–32, 1990. 61
6. O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16:843–872, May, 1994. 66, 70
7. M. Kaufmann, A. Martin, and C. Pixley. Design constraints in symbolic model checking. In *Proc. 10th Int'l Computer Aided Verification Converence*, June, 1998. 66, 69
8. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*, pages 170–172. Princeton University Press, Princeton, New Jersey, 1994. 63
9. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993. 61, 65
10. C. Pixley. Verifying temporal properties of sequential machines without building their state diagrams. In *Proc. 10th Int'l Computer Aided Verification Converence*, pages 54–64, 1990. 61
11. D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986. 63
12. J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int. Symp. in Programming*, 1981. 60
13. J. P. M. Silva. Search algorithms for satisfiability problems in combinational switching circuits. *Ph.D. Dissertation, EECS Department, University of Michigan*, May 1995. 61, 63, 65
14. 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. 61, 63
15. H. Zhang. A decision procedure for propositional logic. *Assoc. for Automated Reasoning Newsletter*, 22:1–3, 1993. 61, 63, 65