An Optimized Symbolic Bounded Model Checking Engine

Rachel Tzoref, Mark Matusevich, Eli Berger, and Ilan Beer

IBM Haifa Research Lab, Haifa, Israel
rachelt@il.ibm.com

Abstract. It has been shown that bounded model checking using a SAT solver can solve many verification problems that would cause BDD based symbolic model checking engines to explode. However, no single algorithmic solution has proven to be totally superior in resolving all types of model checking problems. We present an optimized bounded model checker based on BDDs and describe the advantages and drawbacks of this model checker as compared to BDD-based symbolic model checking and SAT-based model checking. We show that, in some cases, this engine solves verification problems that could not be solved by other methods.

1 Introduction

As the use of formal verification in industrial settings continues to grow [3,5], contemporary research seeks diverse ways to solve the "state explosion" problem inherent in model checking. In recent years, the traditional methods of BDD-based symbolic model checking [10] have been augmented by methods which are based on Boolean Satifiability (SAT) [13,11] that can solve the Bounded Model Checking (BMC) [7] problem. Unlike the model checking problem that, given a model M and a property ϕ, tries to determine if $M \models \phi$, the BMC problem restricts itself to determining whether $M \models \phi$ on the first k iterations of M. The class of properties that can be checked this way is smaller than the one handled by model checking, as described in Section 2.

The BMC problem is usually solved by reducing the model and the bug detection circuit, unfolded k cycles, to a propositional formula, and then solving this formula using a SAT solver. However, other approaches are also applicable. Bertacco and Olukotun [6] suggest a BDD-based algorithm that unfolds the sequential circuit k times in order to calculate the values of signals on the first k cycles. This algorithm is based on symbolic simulation methods [8], and has some advantages over the SAT approach described in [7]. The main advantage is that the unfolded structure uses BDD variables only for inputs to the model. Therefore, when the number of inputs is small compared to the number of state variables, as in the case of datapath, this approach is advantageous. In this paper, we describe an optimized BDD-based BMC engine, based on this unfolded structure.

2 Basic Concepts

We consider bounded model checking to be the following problem: given a nondeterministic Finite State Machine (FSM) M, n RCTL [4] properties $(\phi_1, \ldots, \phi_n)$ and a

© Springer-Verlag Berlin Heidelberg 2003

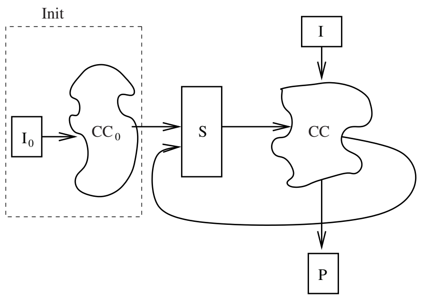

Fig. 1. An FSM

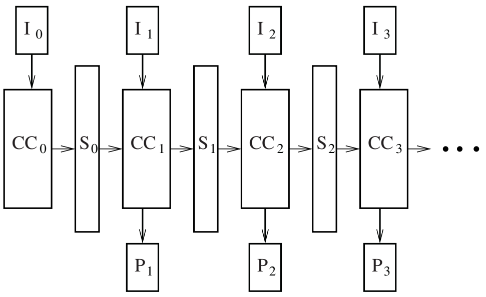

Fig. 2. An unfolded FSM

bound k, we want to check if each of $(\phi_1, \ldots, \phi_n)$ holds in the first k cycles of M. The FSM consists of parts originating from the following sources: a synchronous hardware design to be verified and a nondeterministic environment that defines restrictions on the inputs to the design. In addition, for each property $\phi_m \in (\phi_1, \ldots, \phi_n)$, ϕ_m is translated to an automaton and a formula of the form $AG(p_m)$, where p_m is a Boolean expression, as described in [4], and both the automaton and p_m are included in the FSM (each p_m is an output of a gate). Nondeterministic behavior is translated to free inputs.
An FSM can be defined by the following 6-tuple (CC_0, I_0, CC, I, S, P):

- CC_0 is combinatorial logic that generates the initial states of the flip-flops.
- $I_0 = (i_{(1,0)}, \ldots, i_{(t,0)})$ is an ordered set of Boolean inputs to CC_0.
- CC is combinatorial logic that generates the next state function of the flip-flops.
- $I = (i_1, \ldots, i_q)$ is an ordered set of Boolean inputs to CC.
- $S = (s_1, \ldots, s_r)$ is a set of symbols representing the outputs of the flip-flops.
- $P = (p_1, \ldots, p_n)$ is an ordered set of Boolean outputs representing the properties $(\phi_1, \ldots, \phi_n)$.

(CC_0, I_0, CC, I, S, P) is illustrated in Figure 1.

3 BDD-Based BMC

This section describes how an FSM is transformed into a combinatorial circuit that represents the first k cycles of the FSM, as well as the computation process applied to the combinatorial circuit in order to evaluate the properties in the first k cycles.

3.1 Circuit Unfolding

The unfolding process transforms an FSM, which is a sequential circuit, into an iterative logic array, as depicted in Figure 2. The combinatorial logic, inputs, and properties of the FSM are duplicated k times, and the flip-flops are replaced by wires connecting the copies of the different iterations. Therefore, the S parts do not actually exist; they are depicted only to indicate where the flip-flops existed previously. Assuming there are no combinatorial loops in CC_0 and CC of the original FSM, there are no combinatorial loops in the combinatorial circuit resulting from the unfolding process.

Definition 1 (Closed machine). *The circuit that results from the unfolding process is called a closed machine.*

We use the netlist representation of the unfolded FSM as our basic data structure. This data structure is referred to as the *circuit*.

3.2 Verification Using the BDD-Based BMC

We use the following terms in the description of the computation process:

- *Cycle* is the pair $(S_{j-1}, (CC_j \bigcup I_j \bigcup P_j))$ (corresponds to cycles in calculations of FSM). This cycle is denoted as cycle number j.
- $p_{m,j}$ is the gate that represents property p_m in cycle j.
- g_j represents the replication of a certain gate g of the FSM in cycle j.
- *The cone* of a gate g_j is the set of all gates on which g_j topologically depends.
- *A fanin* of a gate g_j is a gate $f_{j'}$ whose output is a direct input to g_j.
- *A fanout* of a gate g_j is a gate $h_{j''}$ that has a direct input, which is the output of g_j.

Definition 2 (Gate function). *The function of a gate g_j (denoted $f[g_j]$) is the parametric representation of the gate g_j depending on $(I_0, \ldots, I_k)$. $f[g_j]$ operates on all of the FSM inputs ($I_0 \times \ldots \times I_k$) and goes to $\{0, 1\}$, $f : B^{t+q*k} \rightarrow B$.*

Definition 3 (Frontier). *The frontier F is a set of gates where for each gate $g \in F$, two conditions hold: all of the fanins of g have a calculated BDD and the BDD of g is not yet calculated.*

The initial frontier is built by going backwards from the properties, until we reach primary inputs or gates for which there is a calculated BDD. (These gates were in the cone of influence of properties in previous cycles.) The fanouts of these inputs and gates compose the initial frontier. The frontier may change whenever we calculate a BDD of a gate.

For each gate $p_{m,j}$ of $p_{(1,1)}, \ldots, p_{(n,k)}$, we build the BDD that represents the function of the gate $p_{m,j}$. If the BDD of $p_{m,j}$ equals the function *true*, then p_m holds in cycle j. Otherwise, we extract out of the BDD a non-satisfying assignment as a counter example. In order to calculate the BDD of $p_{m,j}$, we must first calculate the BDDs in the cone of $p_{m,j}$. When building the BDD of g_j, we use the BDDs of all of the fanins of g_j. Therefore, the structure of the closed machine dictates a partial order of calculation on the gates. Note that different copies of the same gate g in different cycles may have different BDDs.

3.3 Advantages and Drawbacks of BDD-Based BMC

The BDD-based BMC approach uses a parametric representation of the state of the flip-flops, depending only on the inputs of the model. That is, the set of reachable states in cycle j is represented by a collection of BDDs representing $f[g_j]$, for all gates g_j that represent outputs of the flip-flops in cycle j. As a result, the BDD-based BMC is only sensitive to the amount of nondeterminism in the model. In contrast, symbolic model checking and SAT solvers represent the states by state variables. Therefore, they are sensitive both to the amount of nondeterminism and to the number of state variables. In addition, the functions computed by the BDD-based BMC describe the *natural* functionality of the original model. Symbolic model checking computes a characteristic representation of the reachable states, which is randomly shaped, and its BDD tends to be bigger than those of the *natural* functions. Another advantage versus SAT is that multiple properties are computed in the same run, without repeating calculations of overlapping cones of influence of these properties. SAT solvers need to backtrack after a counter example is found and thus repeat parts of the calculations. The main drawback of our approach is its sensitivity to the number of calculated cycles. In each cycle, q variables are added and therefore the complexity of calculation increases as the cycles advance. As a result of these advantages and drawbacks, the BDD-based BMC approach performs better than the other methods in wide and shallow circuits (i.e., circuits that have many state variables, but their state space can be covered by a few cycles) and in circuits with many state variables, but with a low amount of nondeterminism.

Due to the static unfolding, the circuit is amenable to static BDD variable ordering, based on its topology. In many cases, this order is sufficient for calculation without a need for dynamic BDD reordering. We can also simplify the evaluation of the properties by performing easy calculations before the difficult ones. Our measure of difficulty is the expected BDD size of the gate, which we estimate according to the sizes of the input BDDs. We traverse first the easier calculations paths, and in many cases, as a result of constant propagation during the computation process, some more difficult calculations that were not yet performed become redundant.

4 Open Machine

We will now introduce a variation of the unfolding algorithm, which enables powerful optimizations to the BDD-based BMC engine, as will be described later. Additionaly, this variation enables us to prove properties in some cases, despite the fact that we are calculating only a bounded number of cycles.

Definition 4 (Open machine). *An open machine is a closed machine whose logic CC_0 is replaced by free inputs, as depicted in Figure 3.*

These free inputs are denoted with ${I_0}'$. Note that the number of inputs in ${I_0}'$ may be different from the number of inputs in I_0.

4.1 The Difference between the Open Machine and the Closed Machine

Let $f^{op}[g_j]$ denote a gate function in the open machine, and $f^{cl}[g_j]$ denote a gate function in the closed machine.

Definition 5 (Equivalence between gate functions). *Two gate functions $f[g_x]$ and $f[g'_y]$ are equal, if and only if the BDD of g_x equals the BDD of g'_y. This equivalence is denoted by $f[g_x] \equiv f[g'_y]$.*

Note that $f^{op}[g_j]$ is not necessarily equal to $f^{cl}[g_j]$.

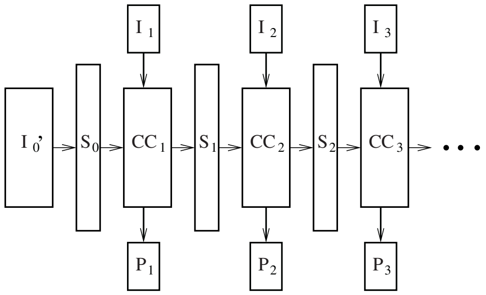

Fig. 3. Open machine

Theorem 6. *If* $f^{op}[g_x] \equiv f^{op}[g'_y]$, *then* $\forall_{j\geq 0}\ f^{op}[g_{x+j}] \equiv f^{op}[g'_{y+j}]$ *and* $f^{cl}[g_{x+j}] \equiv f^{cl}[g'_{y+j}]$.

For proof see [14]. Note that a closed machine version of Theorem 6 does not hold, i.e., if $f^{cl}[g_x] \equiv f^{cl}[g'_y]$, we cannot conclude anything about other gates in the closed machine or in the open machine.

Corollary 7 *It stems from Theorem 6 that if* $f^{op}[g_x] \equiv b, b \in \{0,1\}$, *then* $\forall_{j\geq 0}\ f^{op}[g_{x+j}] \equiv b$ *and* $f^{cl}[g_{x+j}] \equiv b$.

4.2 Uses of the Open Machine

Proving Properties

In some cases, Theorem 6 gives us the ability to prove properties, despite the fact that we are calculating a bounded number of cycles. We prove ϕ_m by calculating the BDD of $p_{m,j}$ for all $j = 1, \ldots, k$ in the open machine. Calculation is performed in the same manner described for the closed machine. If we find that the BDD of $p_{m,j}$ equals true for some $1 \leq j \leq k$, we can conclude that ϕ_m holds both in the open machine and in the closed machine for all cycles $>= j$. As described in [9], we can prove a property in a bounded circuit in this way only if the circuit is k-definite in respect to the property (i.e., the property in each cycle depends only on inputs of at most the last k cycles). While the method in [9] is performed only in order to try and prove properties, we use a more general characteristic of the open machine (introduced in Theorem 6) mainly for optimizations, as described in the next subsection.

An induction-based algorithm, based on a SAT solver, is suggested in [12] for proving safety properties. We chose a different approach in order to accommodate large, real-world, circuits. Our method is suitable only for a subset of the circuits for which the method in [12] is suitable. However, our method can be efficiently implemented using the BDD-based BMC.

Optimizations Based on the Open Machine

Before applying the computation process to the closed machine, we perform two powerful optimizations that simplify further calculations, based on the open machine:

1. **Constant propagation**. There are constant signals in the FSM that originate in restrictions of the environment on the design's inputs. When we find that g_i is the constant b in the open machine, we automatically propagate b to all g_j, for $j \geq i$, both in the open machine and in the closed machine, according to Corollary 7. Due to the special data structure, described later, the time complexity of the propagation is independent of k.
2. **Logical equivalence**. If a gate g is k-definite, the copy of g in cycle j has the same BDD as the copy of g in cycle $j + k$, for all $j \geq 1$. Another case in which different gates have equal BDDs occurs as a result of logic duplication in the original model. We find in the open machine sets of gates with equal BDDs and gather them in equivalence sets. Each equivalence set actually represents up to an infinite number of equivalence sets, since the next cycle replications of the gates in each equivalence set are also an equivalence set. When the computation process runs in the closed machine, we only calculate one BDD for each equivalence set.

Data Structure for the BDD-Based BMC

Our data structure represents both the closed machine and the open machine. While our implementation of the data structure conceptually allows us to perform operations on each of the $2 \times k$ replications of each gate g at any time, initially there is only one object (whose size is independent of k) in the data structure for every gate g of the original FSM. This representation may change as various operations are performed on the circuit. As a result, the common size of the objects representing the replications of g may grow and, in the worst case, depend on k. In practice, most of the data structure remains folded during the entire run. When an operation is performed on a gate g_j in the open machine, it also applies to all of the relevant gates of the subsequent cycles, according to Theorem 6. In most cases, the time complexity is independent of k, since all of the relevant gates are a single object in the data structure.

5 Under-Approximation

Despite the simplification methods and despite applying reordering algorithms, the BDDs can still grow as the cycles advance and may eventually outgrow the memory resources. One solution is to perform under-approximations, although this compromises on coverage. Each under-approximation is performed by choosing an input $i_l \in I_j : 0 \leq j \leq k$ (denoted $i_{l,j}$) and setting it to a constant value $b \in \{0, 1\}$ for the rest of the run. Next, we simplify the already calculated BDDs accordingly. The heuristics we use to choose $i_{l,j}$ and b, try to find the best variable assignment that will balance between causing a significant reduction in the BDDs sizes and leaving many behaviors in the scope of the calculation. The heuristics also take into account that if $i_{l,j}$ was set to b and we are performing a new under-approximation, then we prefer not to choose any of the inputs $i_{l,(j+t)}$ for $t \neq 0$, or if we choose one of them, then set it to $\neg b$. In this way, we degenerate the behavior of an input only in a specific cycle, rather than for the entire run. Examples of heuristics for choosing $i_{l,j}$ and b appear in [14]. Running the computation process with under-approximations is especially useful for finding bugs that, on one hand occur after many cycles, and therefore an exhaustive search would

circuit	in	FF	props	optimized BDD-based BMC					SAT solver			
				time	mem	cycles	res	# app	time	mem	cycles	res
design1	4	279	1	63	353	49	F	0	957	169	49	F
design2(*)	32	363	15	1948	606	100	-	0	out	571	70	-
design3(**)	58	202	2	33 (535)	136 (285)	6,7	F,F	1	out	9	0	-
design4	175	1124	1	21529	816	100	-	0	out	937	35	-
design5	39	377	1	77 (80)	176 (200)	23	F	1	415	207	23	F
design6	112	375	1	17 (950)	96 (317)	10	F	5	10	24	10	F
design7	14	109	1	43 (5189)	74 (memlimit)	16	F	15	23	19	16	F

Fig. 4. Optimized BDD-based BMC versus SAT

be difficult, and on the other hand are quite common (occur for many possible sets of inputs) and therefore can be found even when the search is partial.

We also implemented a mode that combines under-approximations with backtracking, to perform exact evaluation of the properties. In this mode, whenever reaching the cycle bound, we backtrack and compute parts of the search space which were neglected as a result of previous under-approximations.

6 Experimental Results

We implemented the optimized BDD-based BMC in the framework of IBM's model checker RuleBase [2], and used the CUDD package [1] for BDD calculations. The table in Figure 4 presents the results of our engine versus an IBM zChaff-based SAT solver. The engines ran on real-life examples taken from various projects. Both engines operated using default configurations. We set a timeout of 36000 seconds, memory limit of 1G, and a bound of 100 cycles.

The number of inputs, flip-flops, and properties is shown for each circuit. The total run-time is in seconds and the memory is in MB. The *cycles* column is the number of cycles the engine calculated until reaching either the cycle bound, timeout, or memory limit, or until all properties failed. The *res* column displays whether the engine managed to disprove the properties. The *# app* column displays the number of under-approximations performed during the computation process. We also ran several symbolic model checkers on these examples, all of these outgrew memory resources on design1 to design5, while computing the set of initial states. When under-approximations were used, we report, in parentheses, the time and memory consumption of the run without under-approximations. These results demonstrate the significant decrease in time and memory demands our under-approximations achieve.
(*) The SAT solver reached timeout after 70 cycles in each of the 15 runs.
(**) The SAT solver reached timeout while constructing the CNF formula. Using a SAT expert advice, we ran the SAT solver without the bounded cone of influence reduction. With this configuration, it found a counter example for the first property after 189 seconds and for the second property after 139 seconds — about 10 times slower than the unfolding engine (combining the run-time of the two properties).

The table in Figure 5 reports the run-time in seconds of the constant propagation performed on the FSM unfolded 100 cycles. The *open and closed machine* column

circuit	open and closed machine	only closed machine
design1	21.5	140.2
design2	28.5	131.3
design3	6.4	191
design4	12.9	out
design5	5	30.5
design6	2.9	106.7
design7	2.1	53.3

Fig. 5. Constant propagation time

circuit	100 cycles	300 cycles
design1	3.8	4.3
design2	3.7	4.9
design3	6.9	8.2
design4	10.0	16.5
design5	4.2	5.2
design6	3.9	4.4
design7	2.8	3.1

Fig. 6. Construction time

presents the run-time of constant propagation, as it is performed in our optimized engine — first on the open machine (according to Corollary 7) and then on the closed machine. Note that constant propagation on the open machine changes both the topology of the open machine and of the closed machine. The *only closed machine* column presents constant propagation as it would have been performed in a standard implementation (i.e., only on the closed machine). We conclude that there is a significant decrease in run-time when performing constant propagation on the open machine. Note, that in many cases, constant propagation on the closed machine alone dominates the running time and may even cause timeout.

The table in Figure 6 reports the run-time for each circuit in seconds of unfolding the FSM k cycles, out of the netlist representation of the original FSM, for $k = 100$ and for $k = 300$. This table demonstrates the fact that, due to our data structure, the circuit unfolding time does not have a linear dependency on the cycle-bound k.

Acknowledgments. The authors would like to thank Danny Geist and Cindy Eisner for their valuable contribution to this paper.

References

1. *CUDD-2.3.1. http://vlsi.Colorado.edu/ fabio*.
2. I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: An industry-oriented formal verification tool. In *Design Automation Conference*, pages 655–660, June 1996.
3. Ilan Beer, Shoham Ben-David, Cindy Eisner, Daniel Geist, L. Gluhovsky, Tamir Heyman, Avner Landver, P. Paanah, Yoav Rodeh, G. Ronin, and Yaron Wolfsthal. Rulebase: Model checking at IBM. In *Computer Aided Verification*, pages 480–483, 1997.
4. Ilan Beer, Shoham Ben-David, and Avner Landver. On-the-fly model checking of RCTL formulas. In *Computer Aided Verification*, pages 184–194, 1998.
5. Bob Bentley. Validating the intel pentium 4 microprocessor. In *Design Automation Conference*, pages 244–248, 2001.
6. V. Bertacco and K. Olukotun. Efficient state representation for symbolic simulation. In *39th ACM/IEEE Design Automation Conference*, 2002.
7. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.
8. R. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. Cosmos: A compiled simulator for mos circuits. In *Proceedings of the Design Automation Conference*, pages 9–16, 1987.

9. Randal E. Bryant. A methodology for hardware verification based on logic simulation. *Journal of the ACM (JACM)*, 38(2):299–328, 1991.
10. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, Norwell, MA, 1993.
11. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.
12. M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a sat-solver. In *Formal Methods in Computer Aided Design*, 2000.
13. G. P. M. Silva and K. A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Trans. on Computers*, pages 506–516, 1999.
14. R. Tzoref, M. Matusevich, E. Berger, and I. Beer. An optimized symbolic bounded model checking engine. Technical Report H-0185, IBM Haifa Research Laboratory, 2003.