Semi-Formal Bounded Model Checking ⋆

Jesse D. Bingham and Alan J. Hu

Department of Computer Science
University of British Columbia
(jbingham,ajh)@cs.ubc.ca

Abstract. This paper presents a novel approach to bounded model checking. We replace the SAT solver by an extended simulator of the circuit being verified. Compared to SAT-solving algorithms, our approach sacrifices some generality in selecting splitting variables and in the kinds of learning possible. In exchange, our approach enables compiled simulation of the circuit being verified, while our simulator extension allow us to retain limited learning and conflict-directed backtracking. The result combines some of the raw speed of compiled simulation with some of the search-space pruning of SAT solvers. On example circuits, our preliminary implementation is competitive with state-of-the-art SAT solvers, and we provide intuition for when one method would be superior to the other. More importantly, our verification approach continuously knows its coverage of the search space, providing useful semi-formal verification results when full verification is infeasible. In some cases, very high coverage can be attained in a tiny fraction of the time required for full coverage by either our approach or SAT solving.

1 Introduction

Model checking [4, 10] has revolutionized formal hardware verification. The underlying engine for model checking has evolved from the original explicit state enumeration to symbolic model checking [3], and then bounded model checking [1]. Although none of these approaches strictly dominates the others, each new approach has enabled applying formal verification to problems that were previously intractable.

In this paper, we present a novel approach to bounded model checking. The basic bounded model checking construction reduces temporal logic model checking into the problem of finding a satisfying input assignment for a combinational circuit. Normally, this combinational circuit is converted to CNF and handed to a SAT-solver. Our approach, in contrast, searches for a satisfying assignment by explicitly simulating input vectors on the constructed circuit. The advantage of a simulation-based engine is that the circuit itself can be compiled into efficient machine code, resulting in very fast simulation. Furthermore, our simulation-based engine can be easily extended to handle non-Boolean devices, such as tri-state drivers, whereas a SAT-solver cannot. The obvious disadvantage of a simulation-based approach is the exponential number of possible input vectors. A key contribution of this work is our extended simulation algorithm that

⋆ This work was supported in part by a research grant and a graduate fellowship from the Natural Science and Engineering Research Council of Canada. Experiments were conducted on a machine donated by Intel Corporation.

prunes the search space analogously to the learning and conflict-directed backtracking of modern SAT-solvers, while still being amenable to compiled simulation.

As with previous model-checking innovations, our approach is inferior to existing methods on some types of problems. On other problems, though, our new approach is competitive with the state-of-the-art in bounded model checking. More importantly, our bounded model-checking engine continuously maintains a conservative bound on the fraction of the search space that has been verified, allowing our method to be used in a **semi-formal** manner when full, formal verification is infeasible. In some cases, very high coverage can be attained in a tiny fraction of the time required for full coverage by either our approach or SAT solving.

2 Background

Bounded model checking [1] forms the front-end for our verification approach, so we start with a brief review. Bounded model checking consists of three key insights. First, many practical verification properties are specified over finite-length sequences of states, so one can define a restricted — but still practically useful — temporal logic with only bounded temporal semantics. Doing so avoids expensive fixpoint computations in the model checking algorithms. Second, since the temporal logic has bounded-time semantics, it is possible to convert the temporal logic model checking problem into a non-temporal logic problem, and a bounded model checking algorithm for some temporal logic must specify how to perform this conversion for any formula in that logic. For example, to verify that pUq holds for the next three clock cycles in a sequential circuit, one could "unroll" the circuit three times, creating a purely combinational circuit with three copies of the inputs and outputs (one for each clock cycle), and then build a small combinational network to check that pUq holds in all three cycles. (See Figure 1.) The third key insight is that modern SAT solvers have become efficient enough to solve the resulting combinational problem in many instances of practical importance. This third insight is simply enabling technology for the practical relevance of bounded model checking and is not integral to the idea. Indeed, a similar approach has been reported using an ATPG tool rather than a SAT solver [2]. In the present work, we rely on the first two insights of bounded model checking, but replace the SAT solver with an engine that offers competitive performance (but with different strengths and weaknesses), and also provides coverage information to allow semi-formal, incomplete verification.

Although our method replaces the SAT solver, the motivation, algorithms, and weaknesses in our approach can be better understood against the backdrop of the techniques and inefficiencies in typical, modern Boolean SAT solvers. The field of Boolean satisfiability checking has a long and extensive research literature, but all of the leading, freely available, non-commercial SAT solvers used for bounded model checking (e.g., [8, 12, 9]) are based on the approach of Davis, Putnam, Logemann, and Loveland [6, 5]. The basic idea is to **choose** heuristically a good variable on which to case split, assign a value to that variable and **propagate any constraints** that can be logically deduced from the assignment, **backtrack** if our choices and deductions lead to an obviously unsatisfiable formula, and possibly **learn** relationships among the variables by memorizing variable choices that guarantee a non-satisfying truth assignment. This process is repeated until either a satisfying assignment is found, or the entire search

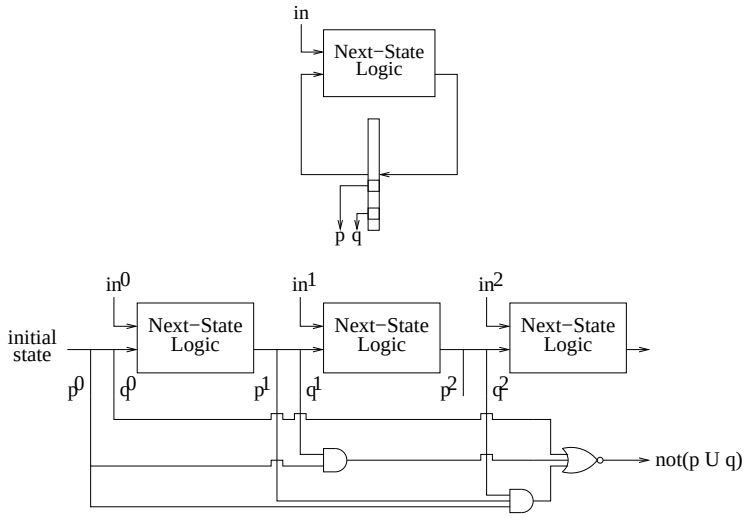
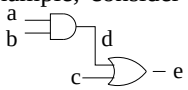


Fig. 1. Converting a Temporal Property to a Combinational One. To verify pUq over three clock cycles on a sequential circuit, we can unroll the circuit three times and create a combinational circuit whose output is true for any counterexample sequence.

space has been exhausted. For example, consider the simple combinational circuit:



Standard SAT solvers work on formulas in conjunctive normal form (CNF), so if we wish to find an input assignment that makes the output true, the typical translation creates the CNF formula:[1]

$$(a+\overline{d})(b+\overline{d})(\overline{a}+\overline{b}+d)(\overline{c}+e)(\overline{d}+e)(c+d+\overline{e})e.$$

The first three clauses ensure the AND gate behaves as an AND gate, the next three clauses handle the OR gate, and the last clause specifies that the output must be true. The last clause has only the single literal e. Such clauses are called "unit clauses", and all SAT solvers immediately assign unit clauses to their forced values, simplify the resulting formula, and look for newly generated unit clauses to continue this process. For example, after the unit clause e has been propagated, we get the simpler CNF formula:

$$(a+\overline{d})(b+\overline{d})(\overline{a}+\overline{b}+d)(c+d).$$

At this point, the choice heuristic might choose to try making d true, and unit clause propagation will result in the satisfying assignment in which a and b are true as well.

[1] For such a small example, it is tempting to build the CNF for the output as a function of the circuit inputs. However, the CNF for a function given as a circuit is in general exponentially larger than the circuit. The typical translation we show here is linear in the size of the circuit, but introduces variables for each internal wire.

The basic SAT algorithm appears to be little more than an explicit search through the possible truth assignments. Progress on SAT solving, however, has produced intelligent heuristics for choosing the variables for case-splitting, faster implementations for propagating constraints, clever ways to backtrack more efficiently, and heuristics for adding new clauses in order to learn not to repeat previous mistakes [8, 12, 9]. The resulting tools can be amazingly efficient on many SAT instances.

Let us now compare SAT-solving to a brute-force attack for the problem of finding an input assignment that satisfies a combinational circuit. The brute-force approach would be to systematically try all possible input assignments to the circuit, evaluating the circuit on each input assignment and looking for a satisfying assignment. Such an approach actually has several advantages over the SAT solver. First, the search space is much smaller, corresponding to only the inputs of the circuit, rather than to all the variables the SAT solver uses to model the internal wires of the circuit. Next, given an input assignment, propagating the results of that assignment from inputs to outputs can be implemented extremely efficiently — for example, the circuit could be compiled into straight-line code that needs at most a few machine instructions to evaluate each gate. In contrast, constraint propagation for a SAT solver dominates the run time (over 90% [9]), and is slow, typically requiring several non-sequential (i.e., cache-miss-prone) memory accesses to walk through the data structures storing the formula, and several data-dependent (i.e., hard-to-predict) branches. On modern processors, the penalty for an L2 cache miss is around 50–100 cycles, and on a Pentium 4, the branch mispredict penalty is at least 19 cycles, so the compiled circuit simulation enjoys an enormous speed advantage. On the other hand, the SAT solver has several advantages over the brute-force attack. First, the SAT solver has the freedom to choose any variable in the system for case-splitting, and the choice of the right splitting variable can sometimes simplify a problem enormously. Empirical results, however, suggest that for bounded model checking, an excellent strategy is usually to choose the variables in a breadth-first manner moving exclusively forward from the inputs to the outputs, or exclusively backwards from the outputs to the inputs [11]. In the forward case, the strategy is essentially a very slow implementation of circuit simulation. The backward case, on the other hand, does give the SAT solver an option unavailable to the brute-force solver. The important advantages in favor of the SAT solver are the backtracking and learning strategies. In particular, modern SAT solvers use some form of non-chronologic or conflict-directed backtracking, in which the tool backtracks all the way back to a relevant decision that could avoid the unsatisfiable sub-problem, rather than simply to the most recent decision. Learning allows the SAT solver to remember combinations of decisions that led to unsatisfiable sub-problems, so that they can be avoided in the future. Our work essentially adds non-chronologic backtracking and learning to the brute-force solver, in a manner that still permits compiled simulation.

3 Verification Algorithm

We first present the brute-force compiled simulation algorithm, and then show how it can be modified to incorporate intelligent backtracking and learning.

3.1 Brute-Force Compiled Simulation

We assume we are given a gate-level sequential circuit, an initial state, a verification wire, and a time bound k. The verification problem is to find a sequence of inputs that causes the verification wire to be true at time k. Different bounded model checking constructions can be handled by pre-unrolling the circuit into a combinational circuit, and then using our algorithm with $k = 0$.

More formally, let C be a sequential circuit with n input variables $\{x_0, \ldots, x_{n-1}\}$ and m state variables $\{s_0, \ldots, s_{m-1}\}$. We use superscripts to denote time indices, so the initial state I is an assignment of Boolean values to s_i^0, for $i = 0, \ldots, m-1$. Label the verification wire f, so we seek an input sequence that causes $f^k = 1$.

The brute-force approach would take an instance of the verification problem and generate a program with the following structure:

```
while (vector=choose_an_untried_vector()) {
  set_inputs(vector);  // Assign vector to circuit inputs
  simulate_circuit(k); // Simulate time 0 through k
  if (circuit.f.value==TRUE) return SATISFIABLE;
  record_unsuccessful_trial(vector,...);
}
return UNSATISFIABLE;
```

The heart of the program is the `simulate_circuit` function. For a combinational circuit, the code generator declares a variable for each wire in the circuit, then does a topological sort on the gates, and generates code that evaluates the output of each gate as a function of its inputs. For example, if our circuit contains the gate `a = AND(b,c)`, the emitted code could be as simple as:

```
circuit.a.value = circuit.b.value & circuit.c.value;
```

which would compile to as little as one instruction and at most a few instructions in machine code. The generated simulator has no expensive data structure for storing and manipulating the circuit and performs no traversals over the circuit; instead, the only representation of the circuit is embedded in the evaluation code itself. To simulate several cycles of a sequential circuit, we simply simulate the next-state logic combinationally, update the state variables, and repeat. The simplest way to choose an untried vector and record unsuccessful trials is to count sequentially through all possible vectors. Obviously, we will need to introduce more effective ways to do this, but any method that makes progress on each iteration will produce the correct answer.

3.2 Skip Cubes

The key idea behind the advanced backtracking techniques used in modern SAT solvers is that many of the decisions (assignments to variables) made before reaching a conflict have no effect at all on that conflict. Therefore, the backtrack should not bother revising irrelevant decisions. Analogously, we will now introduce a mechanism, which we call "skip cubes", by which the circuit simulator can tell which input variables did not affect the value of f^k. Note that this computation is done for the specific input vector being simulated, so this reduction is more specific than the cone-of-influence reduction, which

can only eliminate portions of the circuit which do not affect f^k for any possible input vector. For each vector simulated, therefore, we also compute a potentially large set of other vectors that are guaranteed to produce the same result and can therefore be pruned from further consideration.

Define the universal set U to consist of all binary vectors of length $N = n(k+1)$. An element $v = v_0 \dots v_{N-1} \in U$ corresponds to an input sequence over $k+1$ time steps with $x_i^t = v_{tn+i}$, so we will use the terms "vector" and "input sequence" interchangeably. Starting in initial state I and given some $v \in U$, let w_v^t represent the value on a wire w of the circuit C at time step t.

Definition 1 (Skip Set) *The skip set of a wire w at time t with respect to input vector v is defined $\mathcal{S}_v(w^t) = \{u \mid u \in U \wedge w_v^t = w_u^t\}$.*

Intuitively, $\mathcal{S}_v(w^t)$ is the set of all vectors that cause w^t to have the same value as when C is simulated with the input sequence v. Specifically, simulating the circuit with a vector v will drive f^k to the same value as any other vector in $\mathcal{S}_v(f^k)$, so if f_v^k is false, we may skip any subset of these vectors when searching for a satisfying assignment.

Computing $\mathcal{S}_v(w^t)$ for each gate output could be done in a straightforward manner at the same time that the output value is computed. For example, if w is the output of an AND or NAND gate with inputs a and b, then

$$\mathcal{S}_v(w^t) = \begin{cases} \mathcal{S}_v(a^t) \cup \mathcal{S}_v(b^t) & \text{if } a_v^t = 0 \wedge b_v^t = 0 \\ \mathcal{S}_v(a^t) \cup \overline{\mathcal{S}_v(b^t)} & \text{if } a_v^t = 0 \wedge b_v^t = 1 \\ \overline{\mathcal{S}_v(a^t)} \cup \mathcal{S}_v(b^t) & \text{if } a_v^t = 1 \wedge b_v^t = 0 \\ \mathcal{S}_v(a^t) \cap \mathcal{S}_v(b^t) & \text{if } a_v^t = 1 \wedge b_v^t = 1 \end{cases}$$

Other gate rules are similar. Note, however, that $\mathcal{S}_v(w^t)$ is simply either the on-set or the off-set of w^t, so any exact computation of skip sets amounts to computing the exact functionality of each wire, which will blow-up for many practical examples.

Instead, in our approach we propagate conservative approximations $\mathcal{A}_v(w^t)$ such that $\{v\} \subseteq \mathcal{A}_v(w^t) \subseteq \mathcal{S}_v(w^t)$ and where $\mathcal{A}_v(w^t)$ has a succinct representation. The conservative approximations to the skip sets are special sets called *cubes*:

Definition 2 (Cube) *A cube is simply the Boolean subspace generated by assigning constants to some variables. Specifically, a set $B \subseteq U$ is a cube if $B = \{v \mid v_{i_1} = b_1, v_{i_2} = b_2, \dots v_{i_\ell} = b_\ell\}$, where $0 \leq i_1 < i_2 < \cdots i_\ell \leq N-1$ and the b_i are arbitrary Boolean constants. The indices $i_1, i_2, \dots, i_\ell$ are called the* specified bits*; all others are called* unspecified*. If bit i_j is specified, then b_j is called the* specified value*.*

We may express a cube B as a length N vector over the alphabet $\{0, 1, -\}$, where B_i is the specified value of bit i if specified, or "$-$" if unspecified. We now defined the *skip cube* of a wire:

Definition 3 (Skip Cube) *The* skip cube *$\mathcal{A}_v(w^t)$ of a wire w at time t with respect to input vector v is defined depending on the type of gate driving w:*

1. *If w is a two-input gate with controlling value μ (e.g., an AND gate with $\mu = 0$ or an OR gate with $\mu = 1$) and input wires a and b, then*

$$\mathcal{A}_v(w^t) = \begin{cases} \mathcal{A}_v(a^t) \cap \mathcal{A}_v(b^t) & \text{if } a_v^t = \overline{\mu} \wedge b_v^t = \overline{\mu} \\ \mathcal{A}_v(b^t) & \text{if } a_v^t = \overline{\mu} \wedge b_v^t = \mu \\ \mathcal{A}_v(a^t) & \text{if } a_v^t = \mu \wedge b_v^t = \overline{\mu} \\ \max(\mathcal{A}_v(a^t), \mathcal{A}_v(b^t)) & \text{if } a_v^t = \mu \wedge b_v^t = \mu \end{cases}$$

   where $\max(\ldots)$ *returns the set of greater cardinality.*
2. *If w is the output of an inverter with input a, then*

$$\mathcal{A}_v(w^t) = \mathcal{A}_v(a^t)$$

3. *If w is a state holding element with next state signal a, then*

$$\mathcal{A}_v(w^t) = \begin{cases} \mathcal{A}_v(a^{t-1}) & \text{if } t > 0 \\ U & \text{if } t = 0 \end{cases}$$

4. *If w is an input x_i, then*

$$\mathcal{A}_v(w^t) = (B_0, \ldots, B_{N-1}), \text{where } B_j = \begin{cases} w_v^t & \text{if } j = tn + i \\ - & \text{otherwise} \end{cases}$$

5. *If w is a two-input gate without a controlling value (i.e., exclusive-OR or exclusive-NOR), with input signals a and b, then*

$$\mathcal{A}_v(w^t) = \mathcal{A}_v(a^t) \cap \mathcal{A}_v(b^t)$$

Theorem 1. *$\mathcal{A}_v(w^t)$ as defined in Definition 3 is always a cube.*

Proof: The intersection of two cubes is always either another cube or the empty set. The latter case occurs only if the two cubes disagree on at least one specified bit. In Definition 3, all specified bits always agree with the input vector v. ∎

Theorem 2. *Let $\mathcal{S}_v(w^t)$ and $\mathcal{A}_v(w^t)$ be defined as in Definitions 1 and 3. Then,*

$$\{v\} \subseteq \mathcal{A}_v(w^t) \subseteq \mathcal{S}_v(w^t).$$

Proof: The base cases are that the skip cube for a primary input wire has that input bit specified and all other bits unspecified, which is clearly in the skip set for that input wire, and that the skip cube for a latch at time 0 is completely unspecified, which is clearly in the skip set for that wire (because the inputs don't affect the reset state of the latch). For the inductive step, assume that the skip cubes for the inputs of any gate are contained in their skip sets, and therefore that any vector in that cube would not change the value of that input. Then, the skip cube for the output of the gate computed according to Definition 3 contains only vectors that would not change the value of that output, and are therefore contained in the skip set. This can be easily verified by a case analysis of all the rules. ∎

Corollary 1. *Let $B = \mathcal{A}_v(w^t)$. For all $i = 0, \ldots, N-1$, if B_i is a specified bit, then $B_i = v_i$*

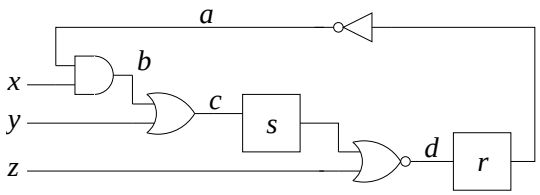
The above results establish the correctness of the optimization that, upon completion of simulating vector v and finding $f_v^k = 0$, skips simulation of all vectors in $\mathcal{A}_v(f^k)$.

We now consider how the computation of the skip cubes can be efficiently integrated into compiled simulation. During simulation of C against input v, we store v as a string of N bits in memory, padded to the nearest machine word boundary. A skip cube B can also be stored as a same-sized bit string, with bit $i = 1$ in memory if and only if bit i is specified in B. If B_i is specified, the specified value is v_i by Corollary 1 and is thus readily available. Note from Definition 3 that the skip cube computation propagates from the inputs to the outputs of each gate, exactly as the value computation does. Accordingly, the code generator can allocate a (value, skip cube) pair for each wire in the original circuit, and the `simulate_circuit` function will contain code to compute both the value and the skip cube for each wire. In most cases, computing the skip cube of w^t is a straightforward copy of the array storing the skip cube of one of the gate inputs. The cases pertaining to a primary input or an initial state variable are also trivial to compute. The cube intersection operations required in cases 1 and 5 of Definition 3 can be achieved by computing a bitwise OR of the bit strings for the respective gate input skip cubes. The max operation of case 1 is the only slower operation, consisting of selecting the skip cube with the fewer specified bits. We implement this step by performing a population count on the skip cube bit strings.

For example, consider the circuit of Figure 2. This circuit has two latches s and r with initial states $s^0 = r^0 = 0$. The table gives the skip cubes for all relevant wires with respect to the input vector $v = 011110100$, where the bits of v from left to right respectively give the input values for $x^0, y^0, z^0, x^1, y^1, z^1, x^2, y^2$, and z^2. The leftmost two columns give the wire name/time index and the bit value, respectively. The column labeled "skip cube" gives the skip cube for the wire. The rightmost two columns state the source of the skip cube and the rule from Definition 3 applied to obtain the skip cube. This example demonstrates the power of the skip cube technique. Suppose we wish to verify that latch r must be 0 at time 2, i.e., $r^2 = 0$. Observe that the skip cube for r^2 is $\mathcal{A}_v(r^2) = \{v \mid y^0 = 1\}$ and $r_v^2 = 0$. Thus, we know that any vector with $y^0 = 1$ implies $r_v^2 = 0$, so we can skip all other vectors with $y^0 = 1$. Hence, our search space has been reduced by a factor of 2.

3.3 Learning and Coverage

Upon simulating any vector v and finding that the value at time k of the verification wire f_v^k is false, the skip cube $\mathcal{A}_v(f^k)$ that we have simultaneously computed gives us a set of vectors that also would have made f_v^k false. The search procedure should remember this skip cube to ensure that it will never again try any vectors in this cube, thereby pruning the search space analogously to the learning and non-chronologic backtracking of conventional SAT procedures. For example, if some input x_i^0 always causes the verification wire to be false regardless of the other inputs, the first vector that we simulate with x_i^0 true will generate a skip cube for f_v^k that shows this fact, and our search procedure will never try any other vectors with x_i^0 true. Thus, the search procedure has effectively backtracked non-chronologically to the decision for x_i^0, and learned the relationship that x_i^0 implies $\overline{f^k}$.



w^t	w_v^t	skip cube t=0 t=1 t=2 xyz xyz xyz	source	case from Def. 3
x^0	0	0--------	$\{u \mid u_0 = 0\}$	4
y^0	1	-1-------	$\{u \mid u_1 = 1\}$	4
z^0	1	--1------	$\{u \mid u_2 = 1\}$	4
s^0	0	---------	U	3
r^0	0	---------	U	3
a^0	1	---------	$\mathcal{A}(r^0)$	2
b^0	0	0--------	$\mathcal{A}(x^0)$	1
c^0	1	-1-------	$\mathcal{A}(y^0)$	1
d^0	0	--1------	$\mathcal{A}(z^0)$	1
x^1	1	---1-----	$\{u \mid u_3 = 1\}$	4
y^1	1	----1----	$\{u \mid u_4 = 1\}$	4
z^1	0	-----0---	$\{u \mid u_5 = 0\}$	4
s^1	1	-1-------	$\mathcal{A}(c^0)$	3
r^1	0	--1------	$\mathcal{A}(d^0)$	3
a^1	1	--1------	$\mathcal{A}(r^1)$	2
b^1	1	--11-----	$\mathcal{A}(a^1) \cap \mathcal{A}(x^1)$	1
c^1	1	----1----	$max(\mathcal{A}(b^1), \mathcal{A}(y^1))$	1
d^1	0	-1-------	$\mathcal{A}(s^1)$	1
x^2	1	------1--	$\{u \mid u_6 = 1\}$	4
y^2	0	-------0-	$\{u \mid u_7 = 0\}$	4
z^2	0	--------0	$\{u \mid u_8 = 0\}$	4
s^2	1	----1----	$\mathcal{A}(c^1)$	3
r^2	0	-1-------	$\mathcal{A}(d^1)$	3
a^2	1	-1-------	$\mathcal{A}(r^2)$	2
b^2	1	-1------0	$\mathcal{A}(a^2) \cap \mathcal{A}(x^2)$	1
c^2	1	-1------0	$\mathcal{A}(b^2)$	1
d^2	0	----1----	$\mathcal{A}(s^2)$	1

Fig. 2. Skip Cube Computation Example. We assume an input vector $v = 011110100$, which provides the values for the three inputs x, y, and z, over three time steps 0, 1, and 2. The table shows the skip cubes that are computed by our algorithm.

Our current implementation maintains a BDD which represents the covered set V of input vectors that have been either explicitly simulated already or else covered via skip cubes. After each simulation iteration, the resulting skip cube $\mathcal{A}_v(f^k)$ is disjoined into V, and the next simulation vector is chosen randomly from the complement set $\overline{V}$. Note that V is not directly related to the functionality of the circuit under verification, and thus the BDD for V does not necessarily blow up even when the BDD for the circuit would have exponential size. The algorithm completes when the entire space U is covered, or when an input sequence is discovered to make f^k true.

The use of a BDD to store the covered set has several advantages. First, all the operations needed by the algorithm can be done efficiently: converting a cube to a BDD, disjoining a BDD for the cube into the BDD for the covered set, and retrieving a random vector not in the covered set. BDDs also provide an easy way to incorporate input don't-cares into the verification method, by initializing V to include all don't-care vectors. Most important is the fact that the ratio $c = |S| / |U|$ can be computed in time linear in the number of BDD nodes. The coverage ratio c tells us the fraction of the search space that has been explored and eliminated by our algorithm. Periodically computing c allows the algorithm to communicate a progress metric to the user. Furthermore, in the event of a time-out or space-out, the coverage ratio provides an informative verification result and increases confidence that the property being verified holds. Having an accurate measure of progress and coverage greatly enhances the usability of verification tools, especially on challenging problems that can't be verified quickly or completely.

4 Experimental Results

We have implemented our algorithm to test its performance. The tool takes a sequential circuit in a slightly modified version of ISCAS89 format and outputs the simulator for that circuit as a C++ program. This translation step is virtually instantaneous. The simulator is then compiled and run to perform the verification. We report compile and run times for the simulator. All experiments were conducted on a PC with a 1.5Ghz Intel Pentium 4 processor and 1GB of RDRAM. Memory usage is not reported, as it was never significant. The operating system was Linux 2.4.9, and the compiler was g++ version 2.95.2 using the `-O3` optimization level. The compiler missed an obvious peephole optimization (two adjacent `addl` instructions modifying the stack pointer), so we used a simple Perl script to perform this optimization, resulting in a performance improvement of a couple percent.

For comparison, we ran against a leading, free, non-commercial SAT solver for bounded model checking, Z-Chaff. Our experiments were conducted with version Z2001.2.17. We used our own translator from ISCAS89 format to CNF, but ignore the negligible translation time. We believe our translator produces CNF comparable to other bounded model checking tools. For example, the multiplier in Section 4.1 is closely modeled on the example presented by Biere et al. [1], and Chaff is able to solve our generated CNF formulas slightly faster than the ones supplied by Biere.

		Compiled Simulation with Skip Cubes				
b	Chaff	compile	full	0.9999	0.999	0.99
0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	8.5	0.0	0.0	0.0	0.0
2	0.0	8.9	0.0	0.0	0.0	0.0
3	0.0	9.7	0.1	0.0	0.0	0.0
4	0.1	11.0	0.2	0.1	0.1	0.0
5	0.4	12.5	0.8	0.1	0.1	0.0
6	2.4	14.6	4.0	0.2	0.2	0.1
7	16.1	17.1	19.6	0.4	0.3	0.1
8	89.7	20.1	71.3	0.8	0.6	0.1
9	234.6	24.0	126.4	2.2	0.8	0.2
10	221.4	28.9	222.3	5.8	1.4	0.2
11	165.4	34.5	332.7	11.4	1.2	0.3
12	134.8	41.5	502.2	28.7	2.8	0.4
13	99.2	48.6	662.6	48.3	2.7	0.5
14	34.9	58.8	840.4	74.9	4.4	0.6
15	26.3	69.5	946.7	115.3	3.9	0.7

Table 1. 16×16 Multiplier Results with Original Specification (Eq. 1). Compile and run times are in seconds. The column labeled "compile" gives the compilation time for the simulation program, while the rightmost four columns give the running time required of our tool to reach the indicated coverages, e.g., 126.4 seconds to fully verify bit 9, and only 2.2 seconds to attain 99.99% coverage.

4.1 Original 16×16 Multiplier Example

Our first example is a 16×16-bit multiplier with 16-bit output. We designed this problem instance closely following the one reported in the original bounded model checking paper [1]. The specification verified was

$$(done \wedge \neg overflow) \rightarrow (out_b = out'_b) \tag{1}$$

where *done* is asserted when the output register has converged to the final value, $overflow$ is asserted if the product exceeds 16 bits, and out_b and out'_b are the bth output bits of a reference combinational multiplier and the sequential multiplier under verification, respectively. Separate runs were performed for $b = 0, \ldots, 15$, and the time bound used in each case was $k = b + 1$.

Table 1 gives the results for our tool and for Chaff. For semi-formal verification (rightmost three columns), our tool gives very high coverage extremely quickly. This illustrates the effectiveness of the skip cube propagation at quickly eliminating large parts of the search space. For complete, formal verification, our tool is competitive with Chaff up to bit 9, but then, surprisingly, the Chaff run times drop sharply. One normally expects output bit $n-1$ of an $n \times n$ multiplier to be the most difficult bit, but this curious behavior can be explained by the presence of $\neg overflow$ in the antecedent of the specification (Eq. 1) in conjunction with the time bound $b+1$. Although the circuit correctly computes the values of all output bits for all input values, almost all input word pairs actually raise $overflow$, making the specification vacuously true. A SAT solver

		Compiled Simulation with Skip Cubes				
b	Chaff	compile	full	0.9999	0.999	0.99
0	0.0	8.1	0.0	0.0	0.0	0.0
1	0.0	8.4	0.0	0.0	0.0	0.0
2	0.0	8.9	0.0	0.0	0.0	0.0
3	0.0	9.7	0.0	0.0	0.0	0.0
4	0.1	10.7	0.1	0.1	0.0	0.0
5	0.4	12.3	0.7	0.5	0.0	0.0
6	2.2	14.4	3.6	3.1	0.0	0.0
7	17.1	16.9	18.2	17.5	7.4	0.0
8	111.3	20.2	102.1	100.5	69.9	0.0
9	1081.2	23.9	533.7	519.4	447.6	0.0
10	time	28.4	2797.9	2928.4	2667.0	855.2
11	time	34.1	time(0.9776)	time(0.9776)	time(0.9776)	time(0.9776)
12	time	40.8	time(0.9418)	time(0.9418)	time(0.9418)	time(0.9418)
13	time	48.8	time(0.8769)	time(0.8769)	time(0.8769)	time(0.8769)
14	time	58.2	time(0.7509)	time(0.7509)	time(0.7509)	time(0.7509)
15	time	68.9	time(0.5004)	time(0.5004)	time(0.5004)	time(0.5004)

Table 2. Full-Sized 16×16 Multiplier Results with Specification (Eq. 2). "time" indicates timeout after 1 hour. When our tool times out, the attained coverage is indicated in parentheses. For bit 10, we actually ran Chaff to completion, which took over 17 hours. Our result for bit 10 with 0.9999 coverage is anomalous, taking slightly longer than full coverage. This might be explained by extra floating-point comparisons performed by our tool when a target coverage is specified, and we have also observed slightly different page fault behavior, but we are still investigating.

can propagate constraints backwards from the overflow flag, quickly pruning the circuit down to essentially an 8×8 multiplier, making the high-order bits easy to verify.

4.2 Full-Size 16×16 Multiplier Example

In the preceding multiplier example, the combinational reference multiplier is actually a full-size 16×16 multiplier with 32-bit output. Similarly, the sequential multiplier would correctly compute all 32 output bits if the output register were wider. Accordingly, we removed the overflow logic from the preceding example, creating a true, full-sized 16×16 multiplier, and verified the specification:

$$done \rightarrow (out_b = out'_b) \tag{2}$$

Table 2 presents the results for this experiment. This problem is much more difficult than that of Section 4.1. Here, we observe our approach running about as fast as Chaff for the low-order bits, and beating Chaff for bits 8, 9, and 10. For the higher-order bits, both tools timeout, but the compiled simulator provides a high coverage while the SAT solver reveals no information.

4.3 SRT Divider Example

Our last experiment is the most difficult. We verify a $2n$-bit by n-bit radix-2 SRT divider with redundant quotient representation [7] against a combinational divider. The

specification asserts that if the SRT divisor is normalized, and if the the combinational divider does not overflow, then the two dividers produce the same result. In particular, we verify all bits of the quotient and remainder in a single run.

The results for $n = 4, \ldots, 8$ are given in Table 3, which clearly demonstrate that our approach is more effective than Chaff on this problem. For $n = 7$, we find our approach to be almost 4 times faster than Chaff even when compilation time is included. For $n = 8$, both tools time-out (again set at 1 hour), but our tool reports the coverage attained.

		Compiled Simulation with Skip Cubes	
n	Chaff	compile	full
4	1.2	7.2	0.4
5	7.5	10.8	4.7
6	98.1	15.7	56.8
7	2848.4	22.3	735.2
8	time	30.4	time(0.7737)

Table 3. 2n-bit by n-bit Radix-2 SRT Divider Results

5 Conclusion and Future Work

We have presented a novel approach to bounded model checking. Our search procedure has competitive performance with state-of-the-art SAT solvers on many problems. Intuition and experimental results suggest that SAT solvers have the advantage on smaller circuits and on circuits in which clever case-splitting heuristics can quickly establish unsatisfiability, whereas our new method has the advantage for larger circuits that aren't amenable to such attacks. Furthermore, our method continually provides coverage information, which is useful as a progress indicator for lengthy verification runs, and as a semi-formal verification result for runs that time out. Our work provides a valuable additional tool for model checking when other methods (e.g., BDDs, SAT) fail.

We believe our implementation could be substantially optimized. For example, our implementation generates C++, which introduced many inefficiencies. A production tool should generate the simulator machine code directly, bypassing the compiler, which is not tuned for the very large, simply structured `simulate_circuit` function that we generate. There is little need for global optimizations, so the code generation would be straightforward.

More algorithmic directions for further research are to explore various design tradeoffs. For example, we could compute more conservative approximations of the skip cubes using branchless code, which might run faster, but need more vectors. Alternatively, we could compute more accurate approximations of the skip sets, reducing the number of vectors needed, but slowing down the simulation as well. In some cases, it would be useful to shift between strategies, starting with skip cubes, for example, and then switching to an alternative if the skip cubes become too small.

From a theoretical perspective, we would like to understand what factors influence the rate of convergence of the coverage ratio. Intuitively, if there exists a vector that produces a large skip cube, it is plausible that many other vectors (such as the other vectors in the skip cube) would also generate a large skip cube, so large skip cubes

would be covered early. If this intuition is true, one could estimate statistically the total run time based on the first few coverage ratios computed, which would further enhance the usability of the model checker.

Acknowledgments

We would like to thank Armin Biere for supplying examples and details on the experimental procedures reported in [1], and Sally McKee for discovering that the g++ optimizer was missing an obvious optimization and for suggesting a work-around.

References

1. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207. Springer, 1999. Lecture Notes in Computer Science Vol. 1579.
2. Vamsi Boppana, Sreeranga P. Rajan, Koichiro Takayama, and Masahiro Fujita. Model checking based on sequential ATPG. In *Computer-Aided Verification: Eleventh International Conference*, pages 418–430. Springer, 1999. Lecture Notes in Computer Science Number 1633.
3. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Conference on Logic in Computer Science*, pages 428–439, 1990. An extended version of this paper appeared in *Information and Computation*, Vol. 98, No. 2, June 1992.
4. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Workshop on Logics of Programs*, pages 52–71, May 1981. Published 1982 as Lecture Notes in Computer Science Number 131.
5. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.
6. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.
7. David Goldberg. *Computer Arithmetic*. Appendix A in David A. Patterson and John L. Hennessy, *Computer Architecture: A Quantitative Approach*, 2nd Ed., Morgan Kaufmann, 1996.
8. João P. Marques Silva and Karem A. Sakallah. GRASP — a new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227. IEEE/ACM, 1996.
9. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference*, pages 530–535. ACM/IEEE, 2001.
10. Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in Cesar. In *5th International Symposium on Programming*, pages 337–351. Springer, 1981. Lecture Notes in Computer Science Number 137.
11. Ofer Shtrichman. Tuning SAT checkers for bounded model checking. In *Computer-Aided Verification: 12th International Conference*, pages 480–494. Springer, 2000. Lecture Notes in Computer Science Vol. 1855.
12. Hantao Zhang. SATO: An efficient propositional prover. In *14th Conference on Automated Deduction*, pages 272–275. Springer, 1997. Lecture Notes in Artificial Intelligence Vol. 1249.