Symbolic Simulation with Approximate Values *

Chris Wilson[1] and David L. Dill[1] and Randal E. Bryant[2]

[1] Computer Systems Laboratory, Stanford University, Stanford, CA 94035
chriswi@stanford.edu, dill@cs.stanford.edu
[2] Computer Science Dept., Carnegie Mellon University, Pittsburgh, PA 15213
Randy.Bryant@cs.cmu.edu

Abstract. Symbolic methods such as model checking using binary decision diagrams (BDDs) have had limited success in verifying large designs because BDD sizes regularly exceed memory capacity. Symbolic simulation is a method that controls BDD size by allowing the user to specify the number of symbolic variables in a test. However, BDDs still may blow up when using symbolic simulation in large designs with a large number of symbolic variables. This paper describes techniques for limiting the size of the internal representation of values in symbolic simulation no matter how many symbolic variables are present. The basic idea is to use approximate values on internal nodes; an approximate value is one that consists of combinations of the values 0, 1, and X. If an internal node is known not to affect the functionality being tested, then the simulator can output a value of X for this node, reducing the amount of time and memory required to represent the value of this node. Our algorithm uses categorization of the symbolic input variables to determine which node values can be more approximate and which can be more exact.

1 Introduction

Verification methods can be categorized into two basic types: full and partial. Full methods attempt to verify all functionality in one shot while partial methods attempt to verify functionality in pieces. Formal verification attempts to verify functionality fully using symbolic methods. Model checking is a formal verification method that works well on small designs, but does not scale to larger designs because the amount of memory required often exceeds capacity on larger designs. Simulation based methods such as directed and random testing scale easily to large designs but require significant test development effort to cover all functionality.

One approach to improving verification is to use symbolic simulation. In our application, symbolic simulation is a partial verification method that extends directed tests by using symbolic values on inputs. This allows exploring more functionality than is possible with a single directed or random test. We call

* This work is supported by the MARCO/DARPA Gigascale Silicon Research Center (GSRC). We also thank HAL Computer Systems for the use of their designs and resources.

symbolic simulation used in this way *symbolic system simulation* to distinguish it from other forms of symbolic simulation.

This paper addresses the problem of using symbolic system simulation on designs that include both data and control logic. In particular, we are concentrating on the verification of system-level designs. A system-level design is one that integrates a number of units that include both datapath and control logic. The goal of system-level testing is primarily to verify the interaction between units rather than exhaustively verify the functionality of each unit.

One of the characteristics of doing partial verification of large system-level designs is that many system inputs are don't care inputs during a test. A *don't care node* is a node in the circuit whose output value should not affect the outcome of the test; a *don't care input* is a don't care node that is a primary input to the circuit. A *care* node is the opposite of a don't care node.

Conventional symbolic simulation using BDDs does not handle don't care inputs well. For example, suppose a circuit has a multiplier that is not being used for a particular test. The user would like to put symbolic don't care variables on the multiplier inputs. However, BDDs are known to blow up for multipliers and it is likely that the BDDs for the multiplier will cause memory overflow. But, since the the multiplier is unused for this test, memory overflow causes the test to abort unnecessarily.

Our solution to this problem is to use *approximate values* on internal nodes that limit the amount of memory blow-up. An approximate value is one that can have the value 0, 1, and X as a function of the symbolic variables in the test. Using approximate values allows the test to complete no matter what kind of don't care logic is present in the design. If the simulator knows that a particular node in the circuit is a don't care node, it can output an X value for this node. For care nodes, it can compute the exact value represented by a BDD for the node since it knows this value will affect the output.

The basic issue in implementing this is: How does the simulator distinguish between care and don't care nodes? Our method works by categorizing symbolic input values as care or don't care values and then creating more exact or more approximate values on nodes based on this categorization. The result of this is that the amount of approximation needed at each node can be determined by looking at the input values to that node only.

An additional problem in symbolic system simulation is that BDDs may overflow memory for functionality that the user wants to verify. For example, suppose we were trying to verify a multiplier as part of a larger circuit. In system simulation, usually what is being verified is that the multiplier is correctly hooked up and communicates properly with the rest of the chip. In this case, producing some result is more important than exhaustively verifying the entire multiplier. Exhaustive verification of the multiplier can be done in a stand-alone unit environment using methods more suitable for that.

Our simulator handles BDD overflow using techniques borrowed from satisfiability (SAT) checking. In particular, our algorithm is closely related to the Davis-Putnam algorithm [6]. The algorithm uses recursive case splitting to re-

duce the size of BDDs while maintaining completeness of the verification. Case splitting consists of picking some symbolic input variable and alternately setting it to the constant 0 and re-simulating the circuit and then setting it to the constant 1 and re-simulating.

Case splitting reduces BDD size by converting variables to constants, but increases simulation time due to re-simulation. Recursive case splitting allows the simulator to always produce some result no matter how many care variables are being used since it will turn as many variables as necessary into constants to reduce the BDD size to fit in memory. At the same time, there is no loss of coverage because the simulator enumerates all combinations of values of variables that were split. Practically speaking, completeness is not guaranteed since the number of combinations that needs to be enumerated is exponential in the number of variables split. However, in the event that the user terminates the simulation before all combinations are covered, each combination that has been simulated provides some amount of coverage. This often is sufficient to verify that, for example, the multiplier above is correctly communicating with the rest of the circuit.

2 Related Work

The symbolic simulation methodology most closely related to ours is Symbolic Trajectory Evaluation (STE) [15]. STE encodes sets of ternary vectors as pairs of BDDs which are then propagated through the simulator. The only chance for approximation in this method is in the selection of the ternary vectors, which is done by the user. Our methodology allows the simulator to choose the amount of approximation at each internal node. STE works more efficiently in verifying small datapath units than our method since there is no advantage to approximating values if there are no don't care inputs. However, our method still works well on these cases, whereas symbolic simulation without approximation does not work well on large system-level designs with many don't cares.

A system that is very similar to STE is the commercial symbolic simulator from Innologic. Innologic's simulator is BDD based, but has the ability to handle BDD overflow. Their overflow handling algorithm is not known to us, however.

Some attempts have been made at minimizing BDD overflow in symbolic simulation. Parametric forms [10, 11] are a method of encoding the input space of a symbolic test to reduce the size of BDDs in the simulator. This requires the user to determine how to do this and the simulator has no choice in how to evaluate each node. Another method described by Bertacco et. al. [1] uses dependencies between nodes to reduce the size of BDDs at nodes. This requires non-local knowledge of the circuit to compute values at each node and so it is not clear how scalable this method is.

Our use of SAT methods most closely resembles non-clausal satisfiability checking algorithms. Our method directly incorporates the SAT decision procedure into the simulator. Other verification methods that use SAT methods typically work by generating a clausal formula that is fed to an off-the-shelf SAT

checker. An example of this is Bounded Model Checking (BMC) [2]. The problem with using clausal SAT methods is that they require the circuit to be unrolled for however many cycles are being simulated, requiring memory proportional to the product of design size and the number of cycles being unrolled. Our method does not require unrolling the circuit and so allows larger circuits to be simulated over more cycles.

Approximation is widely used in model checking. There are basically two types used. In one method, an exact model is used, but the state space is approximated to keep BDD sizes down [4, 8]. Another way is to abstract the model itself [9, 13, 7]. The latter method is often used to extract tests for simulation to increase state coverage. These methods all have the problem that they do not scale to large designs easily, they require a lot of work and expertise, and the abstraction often hides many bugs.

Another model checking method uses liberal abstraction to handle BDD overflow. One method [14] tries to find the closest subset to the original function. Liberal abstraction is useful in model checking since it simply reduces the state space searched. However, in symbolic simulation, liberal abstraction results in the wrong answer being produced and it is not clear how to compensate for this. Also, for don't care logic, which comprises most of the values in a simulation run, outputting any other approximation than X is a waste of time, so these methods are probably inefficient compared to our method.

3 Background

3.1 Simulation and Symbolic Tests

The input to the simulation process consists of a circuit and a test which specifies some functionality that needs to be verified and how to test it. A circuit consists of a network of nodes. This paper assumes nodes are either two input AND gates, two input OR gates, NOT gates, or primary inputs and outputs.

A symbolic test performs the following basic actions: creates symbolic variables, sets values from the symbolic domain on inputs, simulates the circuit, and checks outputs against expected values. Although a single test may check many outputs, all of these results are combined to give a final *fail* output. A value of 1 at the *fail* output indicates the existence of a bug and the test fails. A 0 at the *fail* output indicates the test passed and there are no bugs in the functionality being tested.

3.2 Ternary Valued Simulation

Let $\mathcal{T} = \{0, 1, X\}$ be the ternary domain of values that can appear on nodes in the circuit. The value X denotes the fact that the actual value could be 0, 1, or some combination of 0 and 1, but that the simulator does not know or does not care about the actual value.

We form the upper semi-lattice $\langle \mathcal{T}, \sqsubseteq \rangle$ defined as $1 \sqsubseteq X$, $0 \sqsubseteq X$, and $a \sqsubseteq a$ for all $a \in \mathcal{T}$. The functions AND, OR, and NOT are defined over this semi-lattice.

In order for the simulator to be sound, these functions must be monotonic, that is the relationship:

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y) \tag{1}$$

must hold. Table 1 shows a monotonic implementation of the AND function over the ternary domain.

$\wedge$	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

Table 1. Table for AND

Ternary simulation is performed by evaluating each node in the circuit using the ternary extension of the Boolean operation defined for each node over the input values at that node.

3.3 Symbolic Simulation

Let $\mathcal{V}$ be the set of all the symbolic variables in a test. A *literal* is a variable or its complement. An assignment to $\mathcal{V}$ is a function $\phi: \mathcal{V} \rightarrow \{0, 1\}$ that maps variables to Boolean values. Let Φ be the set of all possible assignments. The value of a node in the circuit is a function $f: \Phi \rightarrow \{0, 1, X\}$ that maps each assignment in Φ to a ternary value. This function is called the *value function* of the node, which is not to be confused with the operation function (AND, OR, NOT) for that node. Each node in the circuit has its own value and consequently, its own value function.

We pre-define some value functions that will be useful later. For each variable $a \in \mathcal{V}$, let $\hat{a}$ be the value function defined such that $\hat{a}(\phi) = \phi(a)$. Let $\hat{0}$, $\hat{1}$, and $\hat{X}$ be those value functions that return the values 0, 1, and X respectively for all assignments.

Symbolic simulation consists of computing an output value function for each node given value functions for the input nodes. The computation is done pointwise:

$$(f \ \langle op \rangle \ g)(\phi) = f(\phi) \ \langle op \rangle \ g(\phi) \tag{2}$$

where f and g are the input values and $\langle op \rangle$ is the defined Boolean operation for this node. A test case failure is indicated when the value function for the *fail* output is 1 for at least one assignment. A test case passes if the *fail* output is 0 for all assignments.

4 Symbolic Simulation with Approximate Values

4.1 Approximation

A value function f' is an *approximation* of f, written as $f \sqsubseteq f'$, if and only if $\forall \phi . f(\phi) \sqsubseteq f'(\phi)$. Given two approximations, f' and f'' of f, f'' is said to be more approximate than f' if $f' \sqsubseteq f''$. Different approximations of a given value function are not necessarily comparable.

An *exact value* is defined as a value function which ranges over the set $\{0, 1\}$. The exact value of a node is the value function computed for that node using the Boolean operation defined for that node given that both input value functions are exact. An approximate value of a node is any value function which is an approximation of the exact value.

An approximate value for a node can be generated by simply applying the symbolic extension of the Boolean operation to the two approximate input values to produce an approximate output value. The correctness of this method is captured in the following formula:

$$f \sqsubseteq f' \wedge g \sqsubseteq g' \Rightarrow (f \ \langle op \rangle \ g) \sqsubseteq (f' \ \langle op \rangle \ g') \tag{3}$$

where $\langle op \rangle$ is the Boolean operation defined for the node, f' and g' are the input value functions, and f and g are the exact values for the input nodes. The validity of this formula is an immediate consequence of the monotonicity of AND, OR, and NOT.

Normally, input values to the simulator consist of exact values only. Values are approximated at internal nodes by the simulator according to an approximation rule. An *approximation rule* states when a value of X must be returned for some Boolean operation instead of an exact value for some set of assignments. The approximation rule limits the set of value representations allowed in order to limit the size of the internal representation of a value. Time and memory can be traded off simply by varying the approximation rule used by the simulator. This can also be done dynamically to allow the simulator to adjust the level of approximation to get the optimal trade-off between memory and time.

4.2 Improving the Approximation

Approximations are conservative, which means that the final simulation result can be approximate if some internal values are approximate. This is indicated when the value function for the *fail* output ranges over the set $\{0, X\}$. Improving the approximation consists of making values on internal nodes more exact. There are two basic ways of doing this. First, the approximation rule can be relaxed, allowing more exact values to be produced at the expense of using more memory. Second, symbolic input variables can be set to constants while using the same approximation rule internally. Since functions of fewer variables generally have smaller representations, there should be fewer approximations produced.

Our goal is to have representations that do not exceed memory capacity. Therefore, our simulator improves approximations by using a combination of

relaxed approximation rules and turning variables into constants. Our method for turning variables into constants is based on the Davis-Putnam (DP) algorithm [6] for proving the satisfiability of propositional formulas. The algorithm uses *case splitting* which selects one of the symbolic input variables and re-runs the simulation twice, once with the selected variable set to 0 and the other with it set to 1. DP recursively case splits until an exact output is generated.

Setting variables to constants is a necessary, but not sufficient condition for improving the approximation at each internal node. The minimum sufficient condition for guaranteeing that an exact value will be produced is that when all variables are set to constant values, a constant value will be output. This is guaranteed if operations on the Boolean domain produce Boolean values, which is normally the case.

The Davis-Putnam method creates a search tree. A leaf node in this tree is one in which either the *fail* output is 1 for some assignment or is the value function $\hat{0}$. If the *fail* output is 1 for some assignment, a bug is found and the test is said to be satisfiable. If the *fail* output value is $\hat{0}$, the algorithm backtracks to a previous decision and tries the other value for the variable that was split. If all branches in the case splitting tree are exhausted, then the circuit is proven to be bug-free for the property being tested and the test is said to be unsatisfiable.

The efficiency of this algorithm is determined by heuristics that select symbolic variables to case split. The goal is to split only variables that affect the outcome of the test. The variable selection heuristic works by propagating a "preferred" split variable through the circuit from the primary inputs to the final output as the circuit is simulated. The data structure for each node consists of the current value of the node and that node's associated split variable. When the node value is updated, the node's associated split variable is also updated. The split variable that is associated with the final *fail* output is the variable that is selected to be split.

The following algorithm guarantees that the value of a node depends on the split variable that is chosen.

1. If the value on the node is a literal, the associated variable is the literal variable.
2. If one of the inputs is a non-controlling value (e.g., 1 is non-controlling for AND) then select the other input's associated variable.
3. Otherwise, select the associated variable of the input that has the lowest index. Normally, the user will assign lower indices to variables that are expected to be split to minimize the amount of case splitting.

4.3 Quasi-Symbolic Simulation

Quasi-symbolic simulation [17] is a particular implementation of symbolic simulation with approximate internal values. Quasi-symbolic simulation restricts value functions to the set $\mathcal{Q} = \{\hat{0}, \hat{1}, \hat{X}, \hat{a}, \neg\hat{a}, \hat{b}, \neg\hat{b}, \ldots\}$ where $a, b, \ldots \in \mathcal{V}$. This domain is called the quasi-symbolic domain because it cannot represent all possible symbolic functions.

Quasi-symbolic value functions can be computed straightforwardly by outputting the value function $\hat{X}$ whenever the exact output value cannot be represented using the quasi-symbolic domain. For example, the computation $\hat{a}$ AND $\hat{b}$ produces the value $\hat{X}$ at the output. If the simulator case splits on a, then the value $\hat{b}$ is produced when $a = 1$ and $\hat{0}$ when $a = 0$. Since both of these values are exact, the approximation has been improved by case splitting.

The approximation rule for quasi-symbolic simulation can be encapsulated as follows:

Approximation rule 1 (Quasi-symbolic Rule) *If the value being produced is a function of two or more variables, output the value $\hat{X}$, else output the correct value from the domain $\mathcal{Q}$.*

Note that this rule allows all values in $\mathcal{Q}$ to be represented by a single word in memory. Consequently, there is no possibility of memory blow up using this approximation rule.

Quasi-symbolic evaluation coupled with recursive case splitting to resolve conservativeness is surprisingly effective at performing symbolic system simulation. The small size of the quasi-symbolic domain causes don't care node values to be turned quickly into $\hat{X}$ values. Since case splitting occurs only over symbolic variables on care inputs, there is no penalty for having many don't care symbolic variables in a test.

4.4 Quasi-Symbolic Simulation with Unit Propagation

Quasi-symbolic simulation can be optimized using a procedure called *unit propagation*. Unit propagation is an implication procedure that is used in the Davis-Putnam method to reduce the size of the search tree. The unit propagation algorithm we use is Propositional Constraint Propagation (PCP) [5]. In this algorithm, two sets of literals are associated with the value at each node. These sets, called C-sets and D-sets, list literals that are conjoined and disjoined respectively with the node value. For example, if the value at a node can be approximated as $\hat{X} \wedge \hat{a} \wedge \hat{b}$ where a and b are literals, then the value of this node is represented as $\hat{X}$ with the C-set $\{a, b\}$. D-sets are constructed similarly with disjoined literals.

C-sets and D-sets for the result of a Boolean operation are computed using set intersection and union operations over the C-sets and D-sets of the input values to a node based on the Boolean operation defined for the node. For example, for the AND operator, the C-set of the output is equal to the union of the input C-sets. The D-set of the output is equal to the intersection of the input D-sets. Logical NOT is computed by swapping the C-set and D-sets and complementing each literal in the swapped set.

A C-set that contains a variable and its complement is called *basic inconsistent*. In this case the value can be replaced by the constant value $\hat{0}$; similarly a basic inconsistent D-set is replaced by $\hat{1}$.

The case splitting algorithm is modified to allow unit propagation by first examining the C-sets and D-sets of the output, and then based on these, eliminating branches in the tree. If the output value has a non-empty D-set, the test is immediately known to be satisfiable. If the output value has a non-empty C-set, then all literals in the C-set are set to the value 1. This eliminates having to explore the complemented case for each of the C-set literals, reducing the ultimate size of the tree. Literals from C-sets that are set to 1 are said to be unit propagated. After unit propagation is done, the circuit is re-simulated and checked for unit propagation again. Case splitting occurs only if the output value is $\hat{X}$ and both the output C-set and D-sets are empty.

C/D-set based approximations allow value functions of more than one variable. However, approximations with more than one variable are strictly limited to those that can be represented as sets of conjoined or disjoined literals. The next section discusses how to relax these restrictions to allow a richer set of approximations while still controlling the amount of memory used by value functions.

5 Approximation using Variable Categorization

To allow more general approximations, BDDs with extensions to allow approximate values are used in the simulator. The simulator manipulates these BDDs based on a categorization of the symbolic variables. Using BDDs to represent values eliminates the need for case splitting to resolve conservativeness. However, our algorithm still uses case splitting for two reasons. First, case splitting is used as part of the variable categorization algorithm and second, if BDD overflow occurs, the algorithm reverts to case splitting to resolve conservativeness.

5.1 Variable Categories

If quasi-symbolic values only are used to approximate node values, then symbolic variables can be categorized into three types from the simulator's point of view.

- *Care variables* are those that the simulator case splits on and so the result must depend on these variables.
- *Leaf node variables* are variables the output depends on but that are not case split. There can be no more than one of these at each leaf node of the search tree since quasi-symbolic evaluation can only compute functions of a single variable exactly without case splitting.
- *Don't care variables* are symbolic variables that the output *does not* depend on for this test.

These categorizes correspond roughly to the user's view of symbolic variables. In general, the user wants the simulator to split only on control variables and does not want it to split on don't cares. Data variables fall in between; if only simple equivalence checks are required, these can be handled by leaf node variables, but if more complex data manipulations is required, data variables may need to be case split.

The variable categorizations above can be used to guide the selection of the appropriate level of approximation at a given node. Our algorithm works by adaptively changing the categorization of variables as the simulation runs. Multiple simulation runs are required to categorize enough variables such that an exact result is produced.

At any point in time, the simulator has a current variable categorization. A variable is called *marked* if the simulator has categorized it as a care variable and *unmarked* if it has not. Marked variables are always care variables while unmarked variables may be care, don't care, or leaf node variables.

5.2 Approximation using BDDs

Value functions are represented using *ternary BDDs* (TBDDs) which we define as multi-terminal BDDs that have at least three possible terminal nodes: 0, 1, and X. Marked variables can appear without restriction in TBDDs since the goal is to make functions of marked variables as exact as possible. Unmarked variables can only appear in TBDDs in a restricted way.

To enforce these restrictions, TBDDs are computed using a modified version of the *Apply* algorithm [3]. The modified *Apply* algorithm creates the subgraphs for the *if* and *else* branches for a given node and then checks to see if the node exists in a cache called the *unique table*. If the node exists in the unique table, the node is returned. If a new node needs to be created, the approximation rules are checked first. If no approximation is required, the node is created and returned. Otherwise, the node is approximated by returning the value X.

Since we want the simulator to compute C/D-sets for node values, the approximation rule used on BDD nodes with unmarked variables is that the node must be approximated unless it is part of a C-set or D-set. This is implemented using a TBDD in which the TBDD variables are marked care variables only and TBDD terminal nodes are either the ternary constants or are pointers to C/D-sets over unmarked variables.

We call this representation a *CD-MTBDD*. Since the number of terminal nodes can be no larger than $O(2^n)$ for n CD-MTBDD variables and each C/D-set contains only a single instance of a given variable, the worst case size of a CD-MTBDD is exponential in the number of marked care variables only. C/D-sets normally stay small and thus, value functions on don't care nodes represented by CD-MTBDDs generally stay very small using this policy. At the same time, case splitting is reduced to a minimum since functions consisting only of care variables and leaf node variables will appear at the output as exact values.

5.3 Policies for Adaptive Variable Categorization

Initially, variables in the simulator are either all marked or all unmarked. During a simulation run, the simulator reclassifies variables by marking them or unmarking them as appropriate. Our policy is based on the observation that large system-level tests have many more don't care variables than care variables. Thus, our policy is to start with all variables unmarked. The split variable

that is associated with the final *fail* output after a simulation run is known to be a care variable. The simulator can mark this variable as a care variable.

Our algorithm does not immediately mark a variable as a care variable when it is discovered. Instead, variables are case split until a leaf node is discovered. If the run is satisfiable, the test stops, but if backtracking is necessary, the variable that is backtracked is marked as a care variable. The reasoning behind this is that, in our experiments we have seen that if the test is satisfiable, the first leaf node is generally satisfiable. Thus, this heuristic gets to the first leaf node as fast as possible using quasi-symbolic simulation only and if that is not satisfiable, it assumes the test is unsatisfiable and starts marking care variables to allow BDDs to be created to reduce case splitting.

5.4 BDD Overflow Handling

BDD overflow occurs when both input BDDs exist, but there is not enough memory to create the necessary output node. Overflow is handled within the framework of approximation. If the maximum node limit has been exceeded and there is no room to create a new node, our algorithm simply returns the value X instead of creating a new node. Thus, overflow handling can be characterized as simply another approximation rule.

The variable selection heuristic must be modified to allow for overflow. Normally, when the simulator has a choice of variables as the preferred split variable for some node, marked care variables are given lower priority than unmarked variables to allow a new care variable to be discovered after each run. If overflow occurs, a marked care variable must be chosen to be split and this variable must be given priority over unmarked variables when selecting a preferred split variable. Our algorithm selects the variable for the CD-MTBDD node that caused the overflow as the preferred split variable. If multiple nodes overflow, the overflow variable with the lowest index is selected.

6 Experiments

We have implemented a prototype Verilog based symbolic system simulator that supports adaptive variable categorization, case splitting to handle BDD overflow, and uses CD-MTBDDs internally. The CD-MTBDD implementation was built on top of the CMU BDD package [12]. This section reports on the results of running some typical test cases on a large representative system-level circuit.

The test design we use for these experiments is an industrial bus to network bridge for a distributed shared memory multiprocessor [16]. The properties we are verifying use only the bus portion of the design which consists of approximately 140K gates and approximately 2,402 state bits. There are two different sets of tests. In the first test, we are looking for a particular hard to find bug, and in the second test, we are trying to exercise all possible data transfers from bus to network. These tests focus on the bus to network data transfer portion of the design because this is the most complicated part of the chip. This area of the chip had the most bugs during system simulation.

6.1 Experiment 1

For the first experiment, an initial test was written and then debugged using our simulator. For each run, we recorded the results of the test, modified the test and re-ran it until the bug was discovered. Some tests were satisfiable, some were unsatisfiable, and some had timeouts due to test case bugs. Satisfiable tests indicated test case bugs or the hardware bug being discovered, and unsatisfiable tests indicated that we were searching in the wrong area for the bug. A device timeout was typically caused by an error driving a request such that the device never responded.

The initial testing was done using quasi-symbolic simulation with C/D-sets [17]. We have re-run these tests to show that the CD-MTBDD-based method improves performance without sacrificing any of the advantages of quasi-symbolic simulation on all three types of tests. There were 47 variants of the the test run. Table 2 shows the results of running each of these tests in summary form based on whether the test was satisfiable due to a test case error (TESTERR) satisfiable due to a device timeout (TIMEOUT) unsatisfiable because the wrong area was being searched (UNSAT) or satisfiable due to the bug being found (BUG.) The first two columns indicate case type and how many test cases there were for each case. The columns labeled "quasi-symbolic" report the average number of evaluations (times the simulator was run to complete all case splitting) and time for each test using quasi-symbolic values only. Note, that these were run on a version of the simulator optimized for quasi-symbolic values only. The last two columns give these same values using CD-MTBDD-based approximations.

		quasi-symbolic		*CD-MTBDD-based*	
test	tests	*evals*	*time*	*evals*	*time*
TESTERR	17	3.8	30.8	3.0	46.6
TIME	20	1.7	50.1	1.9	94.8
UNSAT	9	52.3	445.9	7.8	131.9
BUG	1	78	863.0	17	363.6

Table 2. Results of Directed Test Experiment

The results show that the amount of case splitting is virtually identical for satisfiable tests between the two methods. This is because all the satisfiable tests stopped at the first leaf node and, thus, no BDDs were created due to our variable marking policy that does not mark variables until a leaf node is hit. The minor difference in the average amount of case splitting between the two methods is due to changing the order of control variables to always come before data variables in the BDD-based tests. This ordering of control variables before data variables was not a requirement in the original algorithm that used quasi-symbolic values only. The difference in execution times between the two methods for the satisfiable cases is due to inefficiencies in the off-the-shelf BDD package we were using.

The unsatisfiable cases show that CD-MTBDDs reduce the amount of case splitting by a factor of 6.7 on average with a maximum of 13.9 and a minimum of one for one case that only required a single evaluation even with quasi-symbolic values. The maximum number of marked care variables over all the unsatisfiable tests was five. There was no BDD overflow for any of these cases and the largest BDD created was 17 nodes despite the fact that the number of don't care variables ranged from 590 to 1697 amongst all the unsatisfiable tests. This low size is due to the variable categorization policy and the unmarked variable BDD restrictions that convert values quickly to quasi-symbolic values in don't care nodes.

6.2 Experiment 2

In this experiment, a general data transfer test was written in which symbolic control variables select all possible combinations of events that affect data transfer. This test was expected to be unsatisfiable since there were no known bugs in the area being tested. This experiment started with all symbolic control variables set to a constant value in the test. We then performed a number of runs in which each run increased by one the number of control variables that were made symbolic. The graph in figure 1 plots execution time versus the number of control variables that were made symbolic in each test for both the quasi-symbolic only case and BDD-based approximation case with the quasi-symbolic cases being run on the optimized quasi-symbolic simulator.

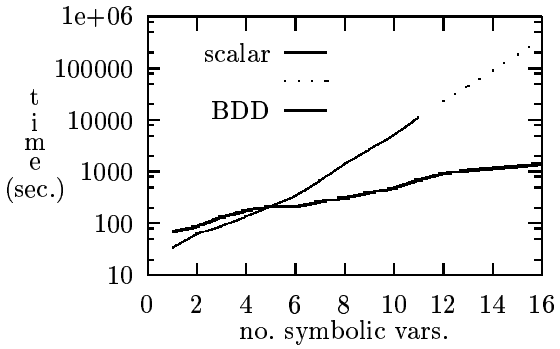

Fig. 1. Execution Time of Quasi-Symbolic and CD-MTBDD Based Approximations

This figure shows that execution time increases exponentially using quasi-symbolic values only. Using CD-MTBDDs, the growth is still exponential, but at a much lower rate. Quasi-symbolic tests were aborted due to excessive case splitting when more than 11 symbolic care variables were present. At this point, the execution time was roughly doubling for each additional control variable that was made symbolic as is indicated by the dashed line in the plot. To under-

stand the difference between quasi-symbolic and CD-MTBDD-based simulation further, we need to look at how various other parameters scale.

Table 3 lists the values of various parameters for each run. The first column lists the number of variables that were case split using quasi-symbolic only simulation and this is equal to the number of control variables made symbolic in a given run. This also turns out to be the number of variables that were marked as care variables during simulation with CD-MTBDDs. The second column gives the number of simulator runs required to complete the test using quasi-symbolic values only; the number of evaluations roughly doubles for each added symbolic variable. The next column gives the number of case splits required when using CD-MTBDDs. The relationship here is that the number of case splits is roughly double the number of marked care variables. This is due to our policy of not marking a variable as a care variable until a leaf node is hit and then marking variables one by one during backtracking. Thus, for each variable marked, there are two evaluations, one going down the tree and one going back up. The cases for which the number of CD-MTBDD splits is less than double the number of marked variables are due to unit propagation. Consequently, the amount of case splitting has been reduced by an exponential factor using CD-MTBDDs without a substantial increase in the time per evaluation.

The next two columns give BDD package statistics. The first of these columns indicates the largest BDD that was created in each test and the last is the total number of BDD nodes created. The largest BDD in each case is remarkably small considering that these tests created over 300 BDD variables including control, data, and don't care variables. These small sizes are due to the unmarked variable restrictions. It is hard to gauge this effect exactly since we cannot easily determine which nodes are don't cares and which are not. The total number of BDD nodes created grows exponentially, but these are mostly BDDs of size ten nodes or less.

7 Conclusion

Symbolic system simulation has the potential to be better than directed and random testing in verifying large system-level designs with mixed control and data logic. Straightforward BDD-based symbolic simulation is not optimized for system-level testing in which there are many don't care inputs. This paper presented an algorithm that uses approximations on internal nodes and heuristics based on variable categorization that allow the simulator to automatically select the appropriate level of abstraction at each node. The key to making this work is having restrictions on the BDDs that represent values on nodes. If one of these restrictions is violated, it indicates that the node is either a don't care node or if it is a care node, that further case splitting must occur. In either case, the amount of case splitting is not affected by making this node more approximate.

We also presented a method for handling BDD overflow that uses the built-in case splitting mechanism to control BDD size at the expense of increased simulation run time. Our experiments show that CD-MTBDD-based approximations

control vars	*quasi-symbolic case splits*	*BDD case splits*	*max BDD size*	*tot. BDD nodes*
1	3	3	1	2352
2	6	4	3	4435
3	8	6	3	4478
4	12	8	3	4739
5	20	10	3	5422
6	32	10	3	6309
7	62	12	7	13001
8	124	14	14	38775
9	246	16	22	110729
10	450	16	45	259236
11	993	17	82	570216
12	-	18	123	867923
13	-	20	123	897016
14	-	22	123	936028
15	-	24	123	1000766
16	-	26	151	1121177

Table 3. Results of Experiment 2

improve performance compared to using quasi-symbolic approximations without increasing the probability of memory overflow significantly. We have not demonstrated that the set of abstraction policies we chose is optimal and in the future, we hope to explore different policy tradeoffs.

References

1. V. Bertacco, M. Damiani, and S. Quer. Cycle-based symbolic simulation of gate-level synchronous circuits. In *Proc. of 36th Design Automation Conf.*, pages 391–396, 1999.
2. A. Biere, A. Cimatti, E. M. Clarke, M.Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proc. of 36th Design Automation Conf.*, pages 317–320, 1999.
3. R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
4. H. Cho, G. Hachtel, E. Macii, B. Pleisser, and F. Somenzi. Algorithms for approximate fsm traversal based on state space decomposition. *IEEE Trans. on Comp.-Aided Design of Integrated Circuits and Systems*, 15(12):1465–1478, December 1996.
5. M. Dalal. Efficient propositional constraint propagation. In *Proc. of the Tenth National Conf. on Artificial Intelligence (AAAI-92)*, pages 409–414, 1992.
6. M. Davis, G. Logemann, and D. Loveland. Machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
7. M. Ganai, A. Aziz, and A. Kuehlman. Augmenting simulation with symbolic algorithms. In *Proc. of 36th Design Automation Conf.*, pages 385–390, 1999.

8. S. Govindaraju, D. L. Dill, A. J. Hu, and M. A. Horowitz. Approximate reachability with bdds using overlapping projections. In *Proceedings of the 35th Design Automation Conference*, June 1998. San Francisco, CA.
9. R. Ho and M. Horowitz. Validation coverage analysis for complex digital designs. In *1996 IEEE International Conference on Computer-Aided Design*, pages 146–151, 1996.
10. P. Jain and G. Gopalakrishnan. Efficient symbolic simulation based verification using the paramteric form of boolean expressions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):1005–1015, 1994.
11. R. Jones, M. Aagard, and C.-J. Seger. Formal verification using parametric representations of boolean constraints. In *Proc. of 36th Design Automation Conf.*, pages 402–407, 1999.
12. D. E. Long. Cmu bdd package, 1993.
13. D. Moundanos, J. A. Abraham, and Y. V. Hoskote. Abstraction techniques for validation coverage analysis and test generation. *IEEE Transactions on Computers*, 47(1):2–14, January 1998.
14. K. Ravi, K. McMillan, T. Shiple, and F. Somenzi. Approximation and decomposition of binary decision diagrams. In *Proc. of 35th Design Automation Conf.*, pages 445–450, 1998.
15. C.-J. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.
16. W.-D. Weber et al. The mercury interconnect architecture: A cost-effective infrastructure for high-performance servers. In *Proc. of the 24th Annual Intl. Symp. on Computer Architecture (ISCA97)*, 1997.
17. C. Wilson and D. L. Dill. Reliable verification using symbolic simulation with scalar values. In *Proceedings of the 37th Design Automation Conference*, June 2000. Los Angeles, CA.