SYMBOLIC SIMULATION USING AUTOMATIC ABSTRACTION OF INTERNAL NODE VALUES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

James Christopher Wilson
October 2001

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

David L. Dill
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Randal E. Bryant

Approved for the University Committee on Graduate Studies:

Abstract

This is version 2.1.

Changes from version 2.0 to 2.1.

- Section 1.1 editted to reflect changes in chapter 5,6.
- Section 1.2.1 rewritten for conciseness, clarity.
- Section 1.2-3 stylistic changes.
- Chapter 2-4 stylistic changes only.
- Chapter 5 introduction, stylistic changes.
- Section 5.1.4 Rewritten, algorithm, figure added, for clarity.
- Section 5.1-2 minor editting for clarity/conciseness.
- Section 5.3 added.
- Chapter 6 introduction, rewritten to synthesize results.
- Section 6.1-2, stylistic changes only.

Changes from version 1 to version 2.0.

- Changed the title from "Symbolic Simulation with Approximate Values" to "Symbolic Simulation using Automatic Abstraction of Internal Node Values".
- Section 1.1 editted for conciseness, clarity.
- Section 1.2, 1.2.1 rewritten.
- Section 1.2.2, 1.2.3 stylistic changes only.
- Section 1.3.1 rewritten. Rest of 1.3 stylistic changes only.

- Section 2.1,2.2 editted for conciseness, clarity.
- Section 2.3.1 added.
- Section 2.3.2 editted for conciseness, clarity.
- Chapter 3, editted for conciseness, clarity.
- Section 4.1.1 added.
- Section 4.1-3, editted for conciseness, clarity.
- Section 4.4 rewritten to reflect goals of experiment better.
- Section 4.5 example added for clarification, rewritten.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

It is the thesis of this dissertation that the problem in solving the verification bottleneck in hardware design today is not in finding the few hard bugs, but is in finding the many easy bugs. This thesis proposes using *symbolic simulation* as a replacement for directed and random testing as the primary method for finding all easy bugs in a design. Existing symbolic simulation algorithms have been optimized for finding hard bugs, therefore, this thesis presents methods for optimizing symbolic simulation to find easy bugs efficiently.

1.1 Problem Statement

1.1.1 The Verification Bottleneck

Twenty years ago, very little effort went into pre-silicon verification; most debug was done post-silicon in the lab. Post-silicon debug times of one to two years were not uncommon for large projects. Today's project schedules typically call for no more than three months of post-silicon debug. Consequently, most debug is now done during pre-silicon verification in order to prevent bugs that require time-to-market killing chip spins.

The pressure to continually reduce time-to-market even as chip complexity increases has led to the belief that functional verification has become the bottleneck in hardware design. Reports from industry have shown verification consuming up to 80% of the design effort in a project [37, 82]. The fear is that, if this problem is not addressed, chips will become un-designable within acceptable time frames in the near future.

Many solutions have been proposed for the verification bottleneck, but little effort has been made to try to understand the root cause of the problem. One generally held belief is that simulation based verification methods cannot find all bugs, especially hard bugs [26]. As design sizes grow, the perception is that there are more, harder bugs to be found. Thus, simulation is viewed as becoming less effective in the future as complexity increases. However, there has been no attempt to quantify this belief, and in fact it is not even clear how to validate assumptions about the effectiveness of different methods.

1.1.2 Conventional Solutions

In an attempt to characterize the verification bottleneck, Dill [26] asserted that a common invariant amongst design projects is that the rate at which bugs are found over time starts high and then diminishes [46, 60, 76, 81]. A point is reached at which the bug rate remains very low for a sustained period before tape-out. Dill calls this period *purgatory* to emphasize that it is this period that characterizes the verification bottleneck.

A lower bug rate usually implies that it requires more effort to find each bug. It is commonly assumed that hard bugs require more effort to find. Since the bug rate gets lower over time, the conclusion generally reached is that bugs get harder over time. This line of reasoning has led verification research to focus on finding hard, late stage bugs in order to reduce the time spent in purgatory.

The primary method advocated for doing this is formal verification techniques such as model checking. Model checking, invented in 1981 by Clarke and Emerson [30, 18], is a method for exhaustively exploring the state space of a design looking for bugs. Since every state in the design is explored, even the hardest bugs will be found. Using model checking to check fundamental assumptions early in the design process has been successful [22]. This is usually done using an abstract model of a design that removes everything except the conceptually difficult protocol to be verified. This abstraction is necessary to make the model small enough such that the protocol can be completely verified. However, it has also been found that a verified abstract design does not guarantee that bugs will not be found later on [43].

Model checking is fragile due to the phenomenon of state explosion. It takes a great deal of expertise and time to make model checking work for any given design. However, since the goal is to find late stage bugs, this may be acceptable, as long as the expertise is available and there is sufficient time. This has resulted in mostly large projects, such as microprocessors, designed by large companies, such as Intel, being able to get useful results from model checking.

Many success stories are reported from large companies using model checking [62, 48, 29, 36], however, these designs represent only a small fraction of the design starts throughout the industry. Small companies simply do not have the time or manpower to devote to complex verification methods that may or may not yield useful results. Thus, in the design community at large, model checking has had negligible impact in improving the overall verification process, despite having been around for almost 20 years.

In recognition of this, Dill proposed using *semi-formal* verification. Semi-formal verification gives up the ideal of complete state exploration in order to make these tools easier to use. The primary idea in semi-formal verification that enables this is to have the user guide the tool such that state explosion is reduced, but with the goal of requiring little expertise in order to do so. Most semi-formal verification methods are still based on the assumption that late stage bugs are hard and it is these bugs that cause the verification bottleneck. It is too early to tell if semi-formal verification tools targeted at finding hard late stage bugs will find wide acceptance. So far, these tools have made little impact on the design community at large.

1.2 Thesis

Despite the introduction of formal and semi-formal methods, simulation-based methods such as directed and random testing still find the majority of bugs in today's designs [53, 76, 81, 5]. Rather than view simulation-based methods as being ineffective, this thesis takes the view that directed and random testing are the most effective way of finding all bugs in a design and will continue to be for the foreseeable future. Thus, the direction pursued by this research is to improve upon directed and random testing while maintaining the characteristics that make them the most effective verification methods. The characteristics that must be maintained are: predictability of coverage and run time, good feedback for debugging, and good scalability as design sizes increase. Together, these characteristics are called *reliability*.

This thesis investigates the use of symbolic simulation to extend directed and random testing. The idea is that symbolic simulation can explore more behavior with a given amount of effort than directed and random testing. The difficulty in doing this is in maintaining the desired reliability characteristics of directed and random simulation Therefore, the primary goal of this thesis is to make symbolic simulation as reliable as random and directed testing.

The next two sections explore the rationale behind this approach in more depth. The first section examines the effectiveness of simulation as a verification method. The second shows how symbolic simulation improves upon directed and random testing.

1.2.1 The Effectiveness of Simulation

The conventional view that simulation cannot keep up with verification needs in the future is based on the view that as designs get larger, bugs become harder to find. What this usually is interpreted to mean is that there is some measure of inherent hardness of a bug and that bugs that are inherently hard require more effort to find. Conventional analysis is predicated on the idea that bug hardness increases with time in a given project and also increases as design size increases. This drives the belief that simulation is an increasingly ineffective way of finding bugs.

Bug hardness, in practice, is a subjective measure. The same bug may be viewed as being either easy or hard depending on factors such as the verification methodology and who is doing the verification. The Intel Pentium FDIV bug [79] is widely quoted as being a hard bug because the probability of failure is on the order of between one in a million and one in nine billion [71]. However, this bug was due to a translation error, not a design error. If it had been a design error, it is quite likely the bug would have been easy to find since it was a simple table error. Exercising all table entries is a standard verification practice, easily done using directed testing. Thus, from a verification standpoint, this bug is not inherently hard to find.

At the same time, hard bugs that slip into silicon are usually easier to work around than simple bugs. The less often a bug occurs during normal operation, the more performance loss that can be tolerated for the failing function. This allows software fixes for hard bugs. Therefore, if bugs are going to slip into silicon, it is better that they be hard bugs that are potentially easier to work around.

Without making any assumptions about bug hardness, the only thing that is certain as design becomes larger is that there will be more bugs to find. Finding all easy bugs will continue to be the primary problem during verification, requiring methods as reliable as directed and random testing, since this is the majority of bugs. Therefore, methods that search only for easy bugs more efficiently may improve the overall verification process. In fact, directed and random testing appear to be extremely efficient methods for finding bugs and it will be difficult to do better than these methods.

1.2.2 Symbolic Simulation

Before describing how symbolic simulation is used to improve upon directed and random testing, it is necessary to give a brief description of what it is. Simulation is a method of representing a device such that the result of applying a stimulus to the simulation is the same as applying the stimulus to the real device. In hardware verification, the device under test (DUT) is a circuit consisting of basic logic gates such as AND, OR, NOT gates, and registers. Signal values are binary, which means they range over the set $\{0, 1\}$.

Binary simulation, which is used in directed and random testing, requires either a one or zero to be applied to each input value for the simulation. For example, if a circuit implements the function $f = A \wedge B$, we could make the input assignments $A = 1$ and $B = 1$ and simulate the function to get the result $f = 1$. To completely verify this circuit would require simulating four different combinations of values for the inputs A and B.

In symbolic simulation, instead of applying a one or a zero to a given input, a symbol is applied. The symbol represents both one and zero at the same time. Values computed at each node in the circuit are represented as expressions over the input symbols. Thus, in the example above, we could make the input assignments $A = x_1$ and $B = x_2$, where x_i is a symbolic variable. Simulating the circuit results in the the symbolic value $f = x_1 \wedge x_2$ being computed. This symbolic value represents all possible combinations of binary values on inputs A and B. Thus, symbolic simulation requires simulating only one pattern to cover all combinations compared to the four required for binary simulation. In several existing systems, BDDs [13, 16] are used to represent symbolic values.

This thesis proposes using symbolic simulation to extend standard directed and random tests. The basic idea is to write tests in essentially the same way as standard directed and random tests, but to replace some of the values injected into the circuit with symbolic values.

To understand how this improves the detection of easy bugs, consider how a test plan is designed. A test plan basically divides the entire space of possible behavior into a set of equivalence classes. For example, consider the creation of the test plan for a simple memory system. First, the set of all tests could be divided into two basic test types: read tests and write tests. Next, each of these classes could be divided into cacheable and uncacheable classes. Next each of these subclasses could be divided into different subclasses based on memory alignment. Each of these subclasses could further be subdivided based on whether the next sequential request was a request to the same address or not. This second request could be further divided by cacheable/uncacheable, memory alignment, and so forth.

Thus, a test plan can be represented as a tree in which each branch of the tree represents a subdivision of the possible set of behaviors. Theoretically, this tree could be infinitely deep, however, the verification engineer decides how deep is necessary in order to cover all possible bug cases. Directed testing then consists of sampling each leaf of this tree. Usually there are far more leaves than can be covered using directed testing. Thus, the verification engineer must chose which subset of the cases to write tests for. Random testing is often used to sample those cases which were not explicitly chosen for directed testing.

There are two ways that bugs may be missed using the above strategy. The first is that there are simply too many leaf nodes and that a bug exists only in a leaf node that was not tested. The second way bugs can be missed is when a bug exists only under certain conditions in a leaf node that was tested and the directed test that sampled this leaf node did not exercise those conditions.

To increase the likelihood of finding the first type of bug requires covering more cases with the same amount of effort by speeding up simulation. It is also often the case that the bug finding rate is limited by how fast tests can be written. Thus, increasing the likelihood of finding the first type of bug also requires more efficient test writing.

Symbolic simulation can be used to provide both of these speedups. First, each leaf node in the test tree can be labelled with a unique identifying number. Symbolic simulation can simulate the entire set of tests in one symbolic simulation run by encoding the test identifier using symbolic values. Also, writing a single symbolic test requires less time than writing many individual tests, although the symbolic test is more complex than any given single test. Thus, symbolic simulation can potentially cover more leaf nodes with a given amount of effort compared to directed and random testing.

Symbolic simulation can also improve the likelihood of finding the second type of bug. The goal of creating a test plan is to create equivalence classes such that if a test within that equivalence class fails, all tests within that equivalence class fail. Thus, sampling a single member of the class is sufficient to detect any bug within the class of tests. If this is not the case, then there must be some input that distinguishes between those test cases that fail and those that do not. A bug is missed if the test case sampling a particular equivalence class happens to set this input to the wrong value.

Inputs that should not affect the correctness of a particular test case are called *don't care* inputs. If a bug occurs within an equivalence class only if some don't care input is set to a particular value, then that input is a distinguishing input. Putting symbolic values on a don't care input allows the simulator to explore both values of that input. Thus, bugs due to don't care inputs will be detected by the symbolic simulator. Consequently, symbolic simulation increases the likelihood of finding bugs due to don't care inputs by exploring all possible values of don't care inputs.

Because there are a finite number of equivalence classes in the test plan out of a possible infinite set, there may be an infinite number of don't care inputs within any given equivalence class. Conventional symbolic simulation runs into problems in dealing with these types of inputs because the number of BDD variables needed to improve coverage of these cases generally exceeds the capacity of standard BDD algorithms. In addition, if there is, in fact, no bug in the equivalence class, then the work computing symbolic values for don't care nodes in the circuit can actually slow down the simulation compared to directed and random testing. Thus, conventional symbolic simulation does not work well for finding bugs due to don't care inputs.

1.2.3 Symbolic Simulation with Approximate Values

If we make the assumption that bugs are easy, then it is likely that only a few don't care inputs actually cause a bug. Thus, to detect this type of bug, it is not necessary to verify all combinations of values over all don't care inputs. It is usually sufficient to check that some distinguishing don't care input can cause an incorrect value to propagate to an output in order to detect a bug. To do this does not require computing exact values for every don't care node in the circuit during simulation. This thesis proposes using *approximate* values on don't care nodes to minimize computation effort. Approximate values are values which range over the binary values plus a value X, which represents an unknown value; values that are not approximated are called *exact*. Approximate values are chosen such that a sufficient amount of information is maintained to allow easy don't care bugs to be quickly detected.

The algorithms described in this thesis allow a small representation to be used for values on don't care nodes, which minimizes the simulation time for these nodes. However, this is a *conservative approximation*, which means that the simulator may detect a failure in cases that no failure would be detected if exact values were used. This thesis describes methods to automatically resolve this conservativeness such that an exact answer is produced while minimizing memory usage and simulation time.

Approximate values have been used in conventional symbolic simulation [74]. In these methods, the user decides on which input nodes unknown values will be used. The simulator has no choice in how to approximate internal circuit node values and so cannot reduce computation effort on don't care nodes.

This thesis describes methods to allow a simulator to vary dynamically the amount of approximation of a value on a node-by-node and simulation run by simulation run basis. The simulator can determine automatically which internal circuit nodes are don't care nodes and which are not and then increase the level of approximation on don't care nodes while keeping care nodes exact. This allows symbolic simulation to maintain its speedup over conventional directed and random simulation even in the presence of many symbolic don't care input values.

Consequently, symbolic simulation with approximate values can improve upon directed and random testing both in finding bugs due to insufficient leaf coverage and in finding bugs within a leaf that occur only when distinguishing inputs are set to a particular value. Because of these features, automatically abstracing internal node values during simulation gives symbolic simulation the potential to be a primary verification method for finding the majority of bugs in a design.

1.3 Overview of the Thesis

1.3.1 Contributions

A primary contribution of this thesis is approaching the verification problem from a different angle than most researchers using symbolic methods. Specifically, arguing that improving upon directed and random testing as primary verification methods is the best approach to address the verification bottleneck may be the most important contribution of this thesis.

This thesis has chosen symbolic simulation as a way of approaching this problem. In relation to this, the contributions of this thesis are as follows.

- *Symbolic simulation with approximate internal values.* The simulator can approximate values on a node-by-node and cycle-by-cycle basis. This allows the simulator to use smaller symbolic representations for don't care nodes.

- *Variable classification-based approximation.* Symbolic variables are classified as either control, data, or don't care. Variable classification is shown to be a good heuristic for determining the appropriate level of approximation for each value the simulator computes.

- *Automatic approximation improvement.* If the initial variable classification is incorrect, the simulator automatically re-classifies variables and re-runs the test in order to generate the correct approximation for each node value. This process adds little overhead to the simulation process.

- *Quasi-symbolic simulation.* SAT-based case splitting is used to perform approximation improvement with no additional memory required. This provides reliability comparable to random and directed testing. In addition, it is shown to be have good performance on test cases in which bugs are present.

- *Reliable symbolic simulation.* Combining SAT-based and BDD-based approximation improvement methods allows the benefits of BDDs to be used in speeding up simulation while mitigating memory blowup.

- The effectiveness of these techniques is demonstrated on two realistic industrial designs.

1.3.2 Outline of the Thesis

Chapter 1 provides an introduction to the problem and gives arguments for using a different set of assumptions in pursuing a solution for the verification bottleneck.

Chapter 2 is a detailed introduction to symbolic simulation, including background material. After reading this chapter, the reader should be able to write a basic symbolic simulator.

Chapter 3 introduces approximations and gives algorithms for manipulating values that include approximations. Variable classification is introduced as a heuristic for the simulator to use in creating approximate values.

Chapter 4 discusses methods to improve the approximation. The algorithms in Chapter 3 assume a static variable classification that may be wrong. This chapter shows how the classification can be improved automatically with little overhead.

Chapter 5 shows that symbolic simulation can be cast as a satisfiability (SAT) problem. SAT solving using partial assignments is shown to be an instance of symbolic simulation with approximate values and approximation improvement. A method of performing symbolic simulation using SAT-based methods only is presented with the advantage of requiring no additional memory to manipulate symbolic values. SAT-based and BDD-based methods are combined to allow available additional memory to be used for BDDs without sacrificing the reliability of SAT-based symbolic simulation.

Chapter 6 concludes with results and future work.

Chapter 2

Symbolic Simulation

This chapter describes the basic algorithms and data structures related to symbolic simulation. Circuits are defined and event driven simulation is described. Basic Boolean algebra is introduced and BDDs which are used to represent symbolic values are described. A detailed description of the APPLY algorithm, which is one of the primary algorithms used in this thesis, is given. Following that is a formal definition of symbolic simulation and proofs of the correctness of symbolic simulation.

2.1 Simulation

The input to the simulation process consists of a circuit and a set of input/output patterns which specify how to test some functionality that needs to be verified. The simulation process itself consists of computing values for each node in the circuit for each step of simulated time.

2.1.1 Circuits

There are many possible circuit abstractions that can be used depending on the type of verification being performed. Protocols and high-level algorithms often are given in terms of operations on states, thus a Mealy or Moore FSM model is appropriate.

Equivalence checking is often done between the gate and transistor level representations of a design.

Since we are interested in functional verification of the design as entered by the logic designer, the most relevant circuit description is what the designer enters. In most cases, this is a register transfer level (RTL) description, written using a hardware description language (HDL) such as Verilog or VHDL.

RTL represents a design as a set of unabstracted registers and a behavioral-level description of the combinational logic. The syntax and semantics of RTL are complex to define. Therefore this thesis will assume a gate level description of the design is used. Since synthesis from RTL to the gate level is straightforward, any conclusions that are derived for the algorithms presented in this thesis on gate level representations are also claimed to hold for RTL representations.

A *gate level circuit* consists of a network of nodes connected using wires. Each node in the circuit implements a Boolean function. All Boolean combinational functions can be implemented using only two-input AND gates and NOT gates. Therefore, without loss of generality, nodes are assumed to be either two-input AND, NOT, primary input (PI,) or primary output (PO) nodes. Each circuit node has an associated Boolean value which is a function of the operation the node performs and the input values to the node. Primary input values change as a function of time as controlled by the test case; consequently, node values change as a function of time. Figure 2.1 shows an example of a simple circuit consisting of three two-input AND gates, three NOT gates, three primary inputs and one primary output.

This definition of a circuit mentions nothing about state holding elements such as latches or registers. However, it does allow cyclic connections in the circuit network. Consequently, state holding elements can be implemented by creating gate level representations of latches and registers. Thus, this circuit definition allows both sequential and combinational circuits to be represented. [1]

[1]It also allows asynchronous circuits to be represented. Although simulation of asynchronous circuits is not precluded by the algorithms presented in this thesis, this is not an explicit goal of this thesis.

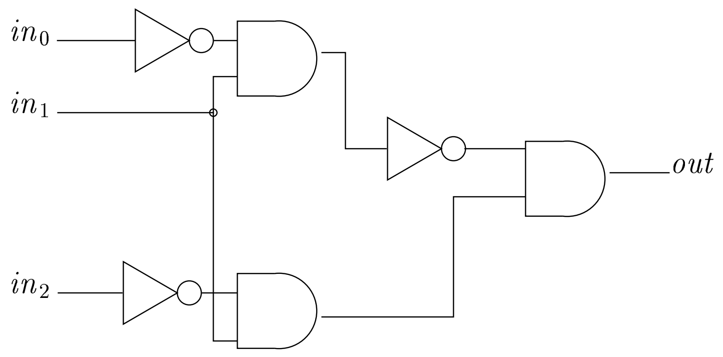


Figure 2.1: An example of a simple circuit

2.1.2 Test Environment

The test environment is the interface between the user and the circuit. The goal of the test environment is to allow the user to control the value of each input or output at each time during the simulation. The test environment specifies a value for each primary input at each time step of the simulation. It also specifies which outputs are checked for correctness at each time step and the correct values for each checked output.

Most input and output values can be generated algorithmically based on the protocols that the circuit understands. Normally, these protocols are written using behavioral code, generally at a higher level of abstraction than RTL. This behavioral code relieves the user from having to specify every value of every input on every cycle, which can be quite a tedious task. Symbolic simulation creates some problems when simulating test environments written using behavioral code. These issues will generally be ignored in this thesis unless they affect a relevant algorithm.

We can assume that there is some mechanism for applying values to inputs and for checking outputs. A simple way to think about this is to assume that there is a table of values that for every time step lists the value of every input and the value of every output and whether it is checked. The user creates the table and the simulator steps through the table applying values, simulating the circuit and then checking the output values.

Algorithm 1 BASIC_SIMULATION_LOOP

```
1: $t \leftarrow \mathbf{0}$; {t is the current time step}
2: $stop \leftarrow \mathbf{0}$;
3: $Initialize\_all\_node\_values()$;
4: while $\neg stop \wedge \neg fail$ do
5:     $\{stop, fail\} \leftarrow Simulate(t)$;
6:     $t \leftarrow Next\_time\_step(t)$;
7: end while
```

There are two special outputs that are generated by the test bench that the simulator understands to have a special meaning. These are the *stop* and *fail* outputs. The *stop* output indicates that the test is finished and the simulator should stop simulation. The *fail* output indicates that an output miscompare was detected by the test bench. It is assumed that the simulator stops if the *fail* output is ever asserted. A test passes if *stop* is asserted without *fail* being asserted.

There are two basic types of tests: *time-bounded* and *reactive*. Time-bounded tests run for a fixed amount of simulation time. If *MAX_TIME* represents the amount of time we want the simulation to run, then *stop* is the function $t \geq MAX\_TIME$. Reactive tests run until the circuit generates some condition. For example, the test bench may inject a response and then wait for a response from the circuit. The *stop* condition would then be a function of the circuit's response. The specification may not specify a maximum time for the circuit to respond. Therefore, the test bench must allow an unbounded amount of time. Thus, reactive tests are also called *unbounded* tests. To achieve the goals of this thesis, supporting reactive tests is a requirement. Therefore, issues in dealing with reactive tests will be discussed when relevant in this thesis.

2.1.3 Simulation Algorithm

Simulation consists of applying inputs to a circuit, computing the value of each node in the circuit for each time step of the simulation, and checking outputs. The basic algorithm for simulating a circuit is given in Algorithm 1.

This algorithm starts the simulation at time 0 and simulates each time step until completion. The $Initialize\_all\_node\_values()$ function generates an initial value for all nodes in the circuit.

The most time consuming part of simulation usually is in computing the values of each node in the circuit. As circuit sizes have grown, much work has been done trying to improve the performance of binary simulators. Consequently, there are many different algorithms that can be used to compute values during simulation. Most of these algorithms can be classified into two basic methods: *event driven simulation* and *compiled code simulation*.

Event driven simulation and compiled code simulation compare differently for symbolic simulation than for binary simulation. The difference between these methods lies in the number of node evaluations that must be performed and the overhead of handling a node event. The performance difference is a function of the *activity factor*, which is the percentage of nodes that have input value changes during a given time step. Symbolic simulation has much higher activity factors than binary simulation which potentially makes compiled code simulation more attractive for symbolic simulation. However, node value computation time dominates over event processing time in symbolic simulation. This means that neither method has a particular advantage for symbolic simulation. Therefore, the experiments in this thesis use an event driven simulation algorithm since this is simpler to implement.

Algorithm 2 lists the event driven simulation algorithm. In this algorithm, the value of a node is computed only if one of its input values changes. At the beginning of each time step, the test bench applies values to each primary input (line 1). The *fanout* of a node is the set of nodes which are driven by this node. If a primary input value is different from its previous value, all of the fanouts of this primary input will be added to the pending event list.

The outer loop (lines 2–13) simulates for one time step. At the beginning of this loop, the set of pending events becomes the set of active events. The inner loop (lines 5–12) computes the value of each node on the active event list, checks to see if the node value changes and, if so, puts the set of fanouts for this node on the pending list.

Algorithm 2 SIMULATE_EVENT_DRIVEN(t)

```
 1: $pending\_event\_list \leftarrow get\_testbench\_inputs(t)$;
 2: while $\neg is\_empty(pending\_event\_list)$ do
 3:    $active\_event\_list \leftarrow pending\_event\_list$;
 4:    $pending\_event\_list \leftarrow \emptyset$;
 5:    while $\neg is\_empty(active\_event\_list)$ do
 6:       $node \leftarrow Pop(active\_event\_list)$;
 7:       $val \leftarrow Compute\_value\_of(node)$;
 8:       if $val \neq node.val$ then
 9:          $Push(fanout\_of(node), pending\_event\_list)$;
10:          $node.val \leftarrow val$;
11:       end if
12:    end while
13: end while
14: $fail \leftarrow testbench\_check\_outputs(t)$;
15: $stop \leftarrow testbench\_stop(t)$;
16: return $\{stop, fail\}$;
```

This loop continues until both the active and pending lists are empty. This indicates that all nodes have correct values for the current time step. Next, the test bench checks any primary outputs that need to be checked and computes the *stop* and *fail* indications which are returned to the main simulator loop (lines 14–16).

2.2 Values and Their Representation

The simulation algorithms given in the previous section are independent of the values a simulator supports. Circuits use a Boolean domain. Binary simulators use binary values that represent a Boolean domain. This section describes basic Boolean algebra.

Symbolic simulators use symbolic values. The major data structure used in this thesis to represent symbolic values is the *Binary Decision Diagram (BDD)*. This section also describes BDDs and the algorithms used to manipulate them.

2.2.1 Boolean Algebra

The lowest level of value abstraction that is used for functional verification is the binary domain $\{\mathbf{0}, \mathbf{1}\}$. Values are computed based on Boolean algebra, thus, $\mathbf{0}$ is interpreted to mean *FALSE* and $\mathbf{1}$ is interpreted to mean *TRUE*.

A Boolean *formula* is an expression over an alphabet consisting of the constants $\mathbf{0}$ and $\mathbf{1}$, variables $x_0, x_1, x_2, \ldots$, the connectives $\wedge, \neg$, and parentheses $(,)$. All Boolean functions over a set of variables can be represented by a Boolean formula requiring only the connectives $\wedge$ and $\neg$. However, it useful to abbreviate particular functions for convenience. Some examples are shown in Table 2.1.

Function	Symbol	Expression
Inclusive-or (OR)	$f \vee g$	$\neg(\neg f \wedge \neg g)$
Exclusive-or (XOR)	$f \oplus g$	$(f \wedge \neg g) \vee (\neg f \wedge g)$
Exclusive-nor (XNOR)	$f \equiv g$	$(\neg f \wedge \neg g) \vee (f \wedge g)$

Table 2.1: Boolean Function Definitions

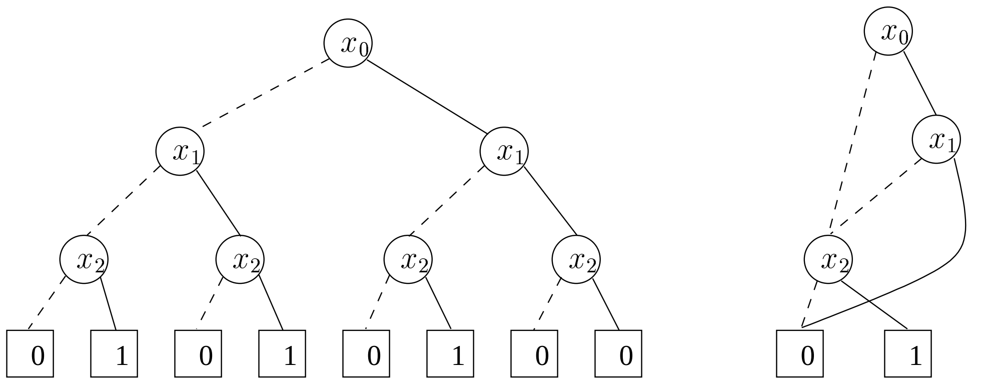
An example of a Boolean formula is

$$(x_0 \wedge \neg x_1) \vee (\neg x_0 \wedge x_2)$$

2.2.2 Binary Decision Diagrams

BDDs [14] are a data structure used to represent Boolean functions. BDDs are widely used in formal verification, synthesis, and other CAD applications to manipulate symbolic values represented as Boolean functions. There are many variants of BDDs. The most widely used variant is the *Reduced Ordered BDD (ROBDD)*. In fact, the term BDD generally is interpreted to mean a ROBDD.

ROBDDs have the feature that Boolean functions can have a significantly more compact representations for many functions than representations such as truth tables or sum-of-products. Given a fixed variable order, ROBDDs are *canonical*, meaning that if two functions are the same they will have the same ROBDD representation and if they are different, they will have different representations.



(a) OBDD (b) ROBDD

Figure 2.2: An example of an OBDD and ROBDD

The compact representation ROBDDs achieve is the source of the speedup of symbolic simulation over binary simulation. Also, since verification of a circuit usually involves comparing the value computed by the circuit to some reference value, canonicity allows these comparisons to be carried out efficiently. Thus, ROBDDs are an attractive representation for verification, in general, and symbolic simulation in particular.

The data structure of a BDD is a rooted, directed acyclic graph. There are two types of nodes in the graph: *terminal* nodes and *non-terminal* nodes. A terminal node is labelled with either the constant **0** or the constant **1** and has no outgoing edges. Each non-terminal node v is labelled with a variable $var(v)$ and has two outgoing edges, each connected to a different child node. There is a single *root* node of the graph that has only outgoing edges.

Let $f(x_0, x_1, \ldots, x_n)$ be a Boolean function. This function is equivalent to its Shannon expansion: $(\bar{x} \wedge f(\mathbf{0}, x_1, \ldots, x_n)) \vee (x \wedge f(\mathbf{1}, x_1, \ldots, x_n))$. The root node of BDD f represents $f(x_0, x_1, \ldots, x_n)$ and has $var(f) = x_0$. One of its outgoing edges, called $if(f)$ points to the BDD node representing $f(\mathbf{1}, x_1, \ldots, x_n)$ and the other edge, labelled $else(f)$ points to the BDD node representing $f(\mathbf{0}, x_1, \ldots, x_n)$. The *if* and *else* nodes are recursively decomposed this way until only the terminal functions **0** and **1** are left.

An *Ordered Binary Decision Diagram (OBDD)* is a BDD in which all paths starting from the root node and traversing the graph to a terminal node have the variables appear in the same order. Since it is possible for a path from root to terminal node to traverse all variables, there is a total ordering of variables that must be maintained in order to enforce this property. The *level* of a variable is its order of appearance in this total order with the root node being at level one and the order increasing as the graph is traversed toward the terminal nodes. Figure 2.2a illustrates the OBDD for a simple function.

A ROBDD is an OBDD in which there are no duplicate nodes. A duplicate terminal node is one that is labelled with the same constant as another. Two non-terminal nodes u and v are duplicates if $var(u) = var(v)$ and $if(u) = if(v)$ and $else(u) = else(v)$.

It is possible to create an ROBDD from an OBDD by recursively applying the following two rules starting from the terminal nodes and working up the graph.

- **Remove duplicate nodes.** For each set of equivalent nodes, choose one of them to remain and remove all others. All incoming edges to this set of nodes should be changed to point to the remaining node.
- **Remove redundant nodes.** The removal of equivalent nodes can result in nodes having $if(v) = else(v)$. These nodes are redundant and can be removed and all incoming edges can be re-directed to point to $if(v)$.

Figure 2.2b shows the ROBDD generated from Figure 2.2a as a result of applying these rules. Normally it is not practical to create a OBDD and then reduce it. Instead a ROBDD is built directly from two existing ROBDDs. Apply is a general algorithm that takes two BDDs and an algebraic operation as arguments and produces the result of applying the operation to the two BDDs.

2.2.3 The Apply Algorithm

This thesis will present modifications to the basic Apply algorithm in the next chapters in order to support the creation of approximate values. Therefore, it is useful to

Algorithm 3 SIMPLE_APPLY(f,g,⟨*op*⟩)

```
1:  if $terminal\_case(f, g, \langle op \rangle)$ then
2:      return $handle\_terminal\_case f, g, \langle op \rangle)$;
3:  end if
4:  $top\_var \leftarrow min(var(f), var(g))$;
5:  $f_{if}, f_{else} \leftarrow cofactor(f, top\_var)$;
6:  $g_{if}, g_{else} \leftarrow cofactor(g, top\_var)$;
7:  $t_{if} \leftarrow$ SIMPLE_APPLY$(f_{if}, g_{if}, \langle op \rangle)$;
8:  $t_{else} \leftarrow$ SIMPLE_APPLY$(f_{else}, g_{else}, \langle op \rangle)$;
9:  if $t_{if} = t_{else}$ then
10:     $result \leftarrow t_{if}$;
11: else
12:     $result \leftarrow create\_node(top\_var, t_{if}, t_{else})$;
13: end if
14: return $result$;
```

understand the basic APPLY algorithm in some detail. Algorithm 3 presents an unoptimized version of APPLY that is easier to understand than the complete, optimized version. This simple algorithm will be presented first followed by the full algorithm with optimizations.

Lines 1–3 handle terminal cases, in which the result of the operation applied to the two input BDDs can be directly determined by examining the two input BDDs. For example, if the operation is AND and one of the input BDDs is the terminal node **0**, the result is the terminal node **0**. The function *terminal_case*() detects this and *handle_terminal_case*() computes the correct result.[2] The complete set of cases handled as terminal functions is given in the following table (note that symmetric cases are not shown).

f	g	terminal value for $\wedge$
$\mathbf{0}$	g	$\mathbf{0}$
$\mathbf{1}$	g	g
f	f	f
f	$\neg f$	$\mathbf{0}$

[2]These functions are usually performed by the same subroutine in a BDD package.

If the result is not going to be a terminal node, then the computation is decomposed by splitting it into two cases. This is done by selecting the root variable in f or g that has the lowest order and then performing a Shannon expansion over this variable on both BDDs.

The *cofactor*() function (lines 5–6) returns the *if* and *else* branches of BDD f if $var(f)$ is equal to *top_var*. If $var(f)$ is greater than *top_var*, then f is not a function of *top_var* and so *cofactor*() simply returns the BDD that was passed to it for both cofactors.

The SIMPLE_APPLY algorithm is called recursively (lines 7–8) on the *if* and *else* cofactors separately. These two calls will return BDD nodes representing the *if* and *else* functions for the computed function. Elimination of redundant nodes is done by detecting that the *if* and *else* nodes are equal (lines 9–10). If the node is not redundant, the BDD node representing the result of applying $\langle op \rangle$ on f and g can be computed by creating a node, labelling it with *top_var* and pointing the *if* and *else* branches to the appropriate BDD nodes returned by the recursive calls to SIMPLE_APPLY (line 12).

SIMPLE_APPLY computes an ordered BDD, but not an ROBDD. The *if*, *else* nodes and *top_var* value computed for some call to SIMPLE_APPLY may be equivalent to those of some other call. Since a new node will be created in both cases, one of the nodes is a duplicate of the other.

The optimized APPLY algorithm fixes this problem by keeping a list of all created nodes in a hash table called the *unique table*. This table is looked up after the *if* and *else* branches are computed to see if this node was previously created. A hash tag is created consisting of the *if* and *else* nodes and *top_var*. If the node exists in the unique table, it is returned as the result of this APPLY. If not, a new node is created as in the SIMPLE_APPLY algorithm and inserted in the unique table.

APPLY can be called with the same set of arguments multiple times. The first call with this set of arguments will create the node and all subsequent calls with the same set of arguments will return a node from the unique table. However, this will only be detected after recursively computing the entire *if* and *else* sub-trees, each node of which will also be returned from the unique table.

Algorithm 4 APPLY(f,g,$\langle op \rangle$)

```
1: if $terminal\_case(f, g, \langle op \rangle)$ then
2:     return $handle\_terminal\_case f, g, \langle op \rangle)$;
3: end if
4: if $cache\_hit(\{f, g, \langle op \rangle)\}$ then
5:     return $cache\_lookup(\{f, g, \langle op \rangle\})$;
6: end if
7: $top\_var \leftarrow min(var(f), var(g))$;
8: $f_{if}, f_{else} \leftarrow cofactor(f, top\_var)$;
9: $g_{if}, g_{else} \leftarrow cofactor(g, top\_var)$;
10: $t_{if} \leftarrow$ APPLY$(f_{if}, g_{if}, \langle op \rangle)$;
11: $t_{else} \leftarrow$ APPLY$(f_{else}, g_{else}, \langle op \rangle)$;
12: if $t_{if} = t_{else}$ then
13:     $result \leftarrow t_{if}$;
14: else if $unique\_hit(top\_var, t_{if}, t_{else})$ then
15:     $result \leftarrow unique\_lookup(\{top\_var, t_{if}, t_{else}\})$;
16: else
17:     $result \leftarrow create\_node(top\_var, t_{if}, t_{else})$;
18:     $unique\_insert(\{top\_var, t_{if}, t_{else}\}, result)$;
19: end if
20: $cache\_insert(\{f, g, \langle op \rangle)\}, result)$;
21: return $result$;
```

This computation can be eliminated using another hash table, this time tagged by the incoming arguments f, g, and $\langle op \rangle$. This hash table is checked before any recursive computation is done and the result immediately returned if the value exists in this hash table.

Note that this hash table is not necessary for correctness, thus, entries can be deleted from this table with only a (potentially exponential) performance penalty. This hash table is often called the *node cache* to emphasize this aspect. BDD node caching almost always significantly reduces BDD computation time.

Algorithm 4 is the optimized APPLY algorithm including both types of hash table. Lines 4–6 check the cache to see if this function has been computed before and return the computed value if so. If this function was not already computed, then after the node has been created, the new node is inserted into the cache at line 20.

Lines 14–15 perform the unique table lookup on the computed node and return the node from the table if it exists. Line 17 creates a new node and line 18 inserts the created node value into the unique table.

Now that the basic algorithms for ROBDD creation have been presented, the term BDD will refer to a ROBDD from here on in this thesis unless otherwise stated.

2.2.4 BDD Variable Ordering for Symbolic Simulation

An equally important aspect in creating BDDs is determining a good BDD variable order. The chosen order can have a dramatic effect on the size of the BDD. For most functions, there exists variable orders such that the BDD size is exponential in the number of BDD variables. For many functions, most variable orders result in exponential sized BDDs and there are few orders that result in polynomial sized BDDs. Finding a good BDD variable order is an NP-complete problem, thus research has been done on finding heuristics that work well in most cases [59].

It is not an objective of this thesis to study BDD variable ordering for symbolic simulation. However, finding good variable orders is an important part in making symbolic simulation practical. It is a goal of this thesis to make symbolic simulation efficient in verifying control dominated circuits. In effect, this is done by exploring as many combinations of control variables as possible mixed with some data values that are used to check the correctness of the results. The following two simple heuristics have been found to be effective in producing good variable orders in this case.

- **Order control variables before data**. Different settings of control variables often have the effect of causing different data to be checked at the output of the circuit. If timing is abstracted away and the effects of other inputs are abstracted, then the circuit is reduced to a multiplexor with the control variables selecting different data inputs. Multiplexors have well known BDD variable ordering characteristics. Basically, ordering the select variables ahead of the data variables results in a minimum sized BDD. Thus, ordering control variables before data variables will, in general, minimize BDD sizes.

- **Interleave variables representing values that are summed.** This heuristic generally is applied to data variables. When verifying an adder, interleaving the variables for the two operand inputs minimizes BDD size. This heuristic also works for comparators and equality detectors, elements which occur quite frequently in circuits.

Another subtle use of this heuristic is to apply it to control variables. For example, verification of data transfer protocols such as bus controllers can done by injecting a number of requests and then checking that each request transferred data properly. Since it is covering all timing cases that is difficult in these protocols, it is desirable to test as many timing combinations between requests as possible.

One way to do this is to have a delay value associated with each request. Let d be the delay before injecting the first request and e be the delay after the first request before injecting the second request. The cycle at which the second request is injected is equal to the sum of the two delays, $d + e$. At any given cycle, the *request_valid* signal that indicates a request is to be injected is a function of d and e. For the second request, if T is a constant representing the current cycle, then the symbolic value on the *request_valid* signal represents the relation $d + e = T$. Thus, the best ordering for the variables representing d and e is to interleave the Boolean variables encoding the integers d and e.

Dynamic variable ordering [73] is a technique that automatically varies the variable order while BDDs are being built in order to find a variable order that minimizes BDD size. Dynamic variable ordering is a very time consuming technique, but is effective in certain circumstances. It is usually the case that once a good order is found, this order works well on future runs even if the BDDs being generated change due to design changes. Thus, a standard practice is to perform dynamic variable ordering once to find a good order and then use this order for subsequent runs.

The characteristics of BDDs created during symbolic simulation are different than those created by other applications such as model checking. In particular, model checking tends to create a relatively small number of very large BDDs while symbolic simulation creates a very large number of small BDDs. Because of the large number of BDDs created by symbolic simulation, it is highly likely that most of them will be exponential in size, albeit over a small number of variables. It is unlikely that dynamic variable ordering will be able to significantly reduce the BDD size in this case. Consequently, this thesis does not explore the use of dynamic variable ordering during symbolic simulation.

2.3 Symbolic Simulation

2.3.1 History and Related Work

In the broadest sense, symbolic simulation encompasses any simulation method that uses values other than from the Boolean domain. The first such instance was the incorporation of the X value to represent physically indeterminate values during simulation by IBM in the 1960s [28]. The X value was found to be useful for detecting hazards in logic. A few years later the use of X for increasing test vector coverage was explored [49]. This work pointed out the effectiveness of using X values to verify all possible reset conditions of memories.

An early paper by Breuer [9] explored the use of multiple X values, each with a unique identity. It was concluded that the use of such values would result in expression blow-up and therefore did not look promising.

The term "symbolic simulation" was coined by researchers at IBM in the late 1970s who were exploring the application of symbolic software verification techniques to hardware verification [24, 19]. This work also suffered from a lack of good symbolic representation and did not advance beyond this initial exploration[14].

BDD-based symbolic simulation appeared soon after the introduction of OBDDs. MOSSYM [12] was the first BDD-based symbolic simulator and was designed to verify switch level representations of circuits. For this class of problem, modeling indeterminate values is very important. Therefore, from the very beginning, BDD-based symbolic simulation has always supported the ternary base domain. This line of research evolved into the COSMOS switch-level symbolic simulation [10] and STE [11].

The work in this thesis is a direct descendent of the the MOSSYM/COSMOS/STE line of symbolic simulation. The primary difference between this thesis and these other methods is the introduction of automatic approximation methods in the simulator.

Another symbolic simulation methodology uses a two-step property verification approach. In this methodology, symbolic simulation is used to extract a formula which is fed to a validity checker. This allows the symbolic expressions in the simulator to be non-canonical and is typically used with high-level data types such as integers, reals, and arrays. A primary application of this is the verification of pipelined microprocessor implementations [17]. A successful application of this approach uses the Stanford Validity Checker (SVC) which supports quantifier-free first order logic [50]. Another application used ACL2 [64] as the validity checker for verifying processor microcode [39].

As the basic symbolic simulation algorithms have increased in power, more attention has been paid to applications and the methodology of using symbolic simulation. Parametric representations [45, 1] are a methodology that allows symbolic simulation to be more robust for verifying data path elements, such as floating point adders, that do not allow a polynomial-sized BDD representation. Self-consistency checking [52] is another technique that reduces the effort required in writing reference models for verification. Specifying a symbolic address for a RAM effectively requires accessing all locations in the RAM simultaneously during symbolic simulation. To alleviate the work in accessing many physical locations, an Efficient Memory Model (EMM) was developed for use in symbolic simulation [77]. The basic idea of this model is to store the state of the entire RAM using a single BDD which is updated as writes are performed.[3]

One of the initial applications of ternary simulation was the switch-level verification of RAM circuits [15]. This led to one of the first successful commercial applications of BDD-based symbolic simulation on PowerPC memory arrays [67]. A company, Innologic, whose main product is a BDD-based symbolic simulator was founded. The most common application of their symbolic simulator has been in verifying memory arrays.

[3]An EMM is implemented in the program *qsym* which embodies the algorithms in this thesis and was used for some of the experiments in this thesis.

Symbolic simulation has also been applied to the verification of blocks within commercial microprocessors. Several blocks within recent Intel microprocessors have been successfully tested using symbolic simulation [51].

2.3.2 Theory

Symbolic simulation was informally introduced in Chapter 1 as a method in which symbolic variables replaced some of the binary values in a directed or random test. It was claimed that each symbolic variable applied to an input covered the case of that input being set to both a $\mathbf{0}$ and $\mathbf{1}$. However, this informal description does not have clearly defined semantics. This section will formalize the concepts of symbolic simulation and tie together the pieces of this chapter that we have seen so far.

Our informal definition of symbolic simulation is imprecise because we have not been given a formal definition of what the symbols mean. It is tempting to think of symbols as values unto themselves, however, a more useful interpretation is to think of symbolic values as functions that map the value of sets of Boolean variables onto a finite domain.

The finite domain is called the *base domain.* The base domain is the set of scalar values that the simulator supports. For binary simulation, the base domain is the Boolean, or binary, domain $\mathcal{B} = \{\mathbf{0}, \mathbf{1}\}$. In Chapter 3, the ternary domain $\mathcal{T} = \{\mathbf{0}, \mathbf{1}, X\}$ will be used. Thus, we can refer to symbolic simulation using each of these base domains as *binary symbolic simulation* and *ternary symbolic simulation* respectively.

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 $\phi\colon \mathcal{V} \to \{\mathbf{0}, \mathbf{1}\}$ 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 $f\colon \Phi \to \{\mathbf{0}, \mathbf{1}, X\}$ 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 $\hat{\mathbf{0}}$, $\hat{\mathbf{1}}$, and $\hat{X}$ be those value functions that return the values $\mathbf{0}$, $\mathbf{1}$, 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 point wise:

$$(f \ \langle op \rangle \ g)(\phi) = f(\phi) \ \langle op \rangle \ g(\phi) \tag{2.1}$$

where f and g are the input values and $\langle op \rangle$ is the defined Boolean operation for this node.

We will not consider the ternary domain until the next chapter, therefore, the remainder of this chapter will assume that symbolic values are over a Boolean base domain. It is straightforward to represent symbolic values as defined above using BDDs. The computation of values at nodes is done using the Apply algorithm.

As an example, we can formalize the simple example of an AND gate introduced in Chapter 1. Let $\mathcal{V} = \{x_1, x_2\}$ and the test case $\mathcal{P}$ be the input assignments $\{X = x_1, Y = x_2\}$. Let $\Phi = \{\phi_0, \phi_1, \phi_2, \phi_3\}$ be the set of assignments to $\mathcal{V}$ as specified in the following table.

Assignment	$\phi_i(x_1)$	$\phi_i(x_2)$	$(X \wedge Y)(\phi_i)$
ϕ_0	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$
ϕ_1	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{0}$
ϕ_2	$\mathbf{1}$	$\mathbf{0}$	$\mathbf{0}$
ϕ_3	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$

The value function for $(X \wedge Y)$ is computed using Equation 2.1. The value function is the symbolic value $x_1 \wedge x_2$.

A simulation loop similar to algorithm 1 is used in performing symbolic simulation. Again, we assume a very basic test bench environment in which the user creates table entries for all inputs and outputs over all time steps. For symbolic simulation, table values can contain symbolic values over a set of symbolic variables that the user creates for the test.

	input values			
v_1, v_0	in$_3$	in$_2$	in$_1$	in$_0$
$\mathbf{0, 0}$	**0**	**0**	**1**	**0**
$\mathbf{0, 1}$	**1**	**1**	**0**	**1**
$\mathbf{1, 0}$	**0**	**1**	**0**	**1**
$\mathbf{1, 1}$	**1**	**1**	**1**	**0**
v_1, v_0	(v_0)	$(v_1 \vee v_0)$	$(v_1 \equiv v_0)$	$(v_1 \oplus v_0)$

Table 2.2: Example of creating symbolic input functions

Symbolic input vectors can be generated by encoding sets of binary input vectors using a *parametric representation* [45, 1]. A set of parametric variables is introduced and a symbolic input vector is created as a function of these variables representing all the Boolean input vectors. Table 2.2 shows an example of encoding four Boolean vectors over four inputs using two parametric variables v_0 and v_1. The symbolic representation of this set of vectors is given in the last line of the table.

Symbolic values are computed at each node over all time steps using Equation 2.1 to do the evaluation of each node value. A symbolic simulation run produces symbolic values for the *fail* and *stop* outputs. A test case failure is indicated when the value function for the *fail* output is **1** for at least one variable assignment. A test case passes if the *fail* output is **0** for all variable assignments and *stop* is **1** for all assignments.

The fact that the *stop* output can be symbolic creates an interesting problem for the simulator. It is possible that, for some time step, the value of *stop* is **0** for some variable assignments and **1** for others. Since a value of **1** means the simulator should stop and **0** means it should continue to the next time step, the simulator is faced with dilemma of whether to stop at the current time step or continue executing.

This is resolved by having the simulator continue, but having it keep track of for which assignments the simulation has stopped. Any *fail* conditions for variable assignments that were stopped in previous time steps are ignored. Thus, the basic simulation loop is modified as shown in Algorithm 5 to handle symbolic *stop* values.

The symbolic simulation loop is augmented with a variable *gstop* which records for which assignments the simulation has stopped for all time steps up until the previous

Algorithm 5 SYMBOLIC_SIMULATION_LOOP

```
1: $t \leftarrow 0$; {t is the current time step}
2: $stop \leftarrow \hat{\mathbf{0}}$;
3: $gstop \leftarrow \hat{\mathbf{0}}$;
4: $Initialize\_all\_node\_values()$;
5: while $gstop \neq \hat{\mathbf{1}} \wedge fail = \hat{\mathbf{0}}$ do
6:    $\{stop, fail\} \leftarrow Simulate(t)$;
7:    $fail \leftarrow fail \wedge \neg gstop$;
8:    $gstop \leftarrow gstop \vee stop$;
9:    $t \leftarrow Next\_time\_step(t)$;
10: end while
```

time step. The *fail* condition is only checked for those assignments in which *gstop* is false.

We are now at a point where we can state and prove the fundamental theorems of symbolic simulation. We want to prove that symbolic simulation is both sound and complete. Soundness means that proving a property to be valid implies that the property is true. When applied to simulation, soundness means that if a test case passes in simulation, it will pass on the real design. Completeness means that any property of the design is provable in simulation.

Symbolic simulation can only be as sound and complete as binary simulation. Binary simulation is sound only if applied to ideal gates, that is, if we ignore timing, noise margin, and other electrical effects. What we want to prove, therefore, is that symbolic simulation is sound and complete when the underlying binary simulation is sound and complete. This is proved in the following two theorems.

Theorem 2.1 (Symbolic Simulation is Sound). *Let Φ be the set of all assignments over a set of variables $\mathcal{V}$. Let $\mathcal{P}$ be a symbolic test consisting of a set of symbolic input vectors over all time steps, a set of expected symbolic output vectors and a symbolic indication of whether each output should be checked at each time step. Then symbolically simulating $\mathcal{P}$ with the result that fail $= \hat{\mathbf{0}}$ implies that for all assignments $\phi \in \Phi$, simulating $\mathcal{P}(\phi)$ will result in fail $= \mathbf{0}$, where $\mathcal{P}(\phi)$ is the binary test resulting from substituting symbolic variables with their binary assignments in ϕ.*

Proof. The proof of soundness is straightforward from the definition of symbolic simulation. Equation 2.1 defines the evaluation of symbolic values as a point wise evaluation over the set of binary values. Symbolic input and output values are encoded point wise over the set of binary input vectors. Thus, if the result of the symbolic simulation results in no failures, then none of the point wise binary simulations must have any failures. $\square$

Completeness means that a pass/fail indication will be determined in all cases. For binary simulation, completeness is guaranteed if the simulation stops. Thus, binary simulation is incomplete only if the simulation never stops. The symbolic equivalent is stated by the next theorem.

Theorem 2.2 (Symbolic Simulation is Complete). *Let Φ, $\mathcal{V}$, and $\mathcal{P}$ be defined as in Theorem 2.1. If for all $\phi \in \Phi$, binary simulation of $\mathcal{P}(\phi)$ stops, then symbolic simulation of $\mathcal{P}$ stops.*

Proof. To prove this, we must analyze Algorithm 5. Since we assume that all binary simulations must stop, there must be some time step for which each binary simulation corresponding to some variable assignment stops.

The value of *gstop* in Algorithm 5 is simply the OR of the value of *stop* over all time steps. Since each binary simulation must stop in a unique cycle, the assignments for which *stop* is **1** for each time step are unique between all time steps. Therefore, each assignment for which *stop* is asserted that is ORed into *gstop* at each time step adds one assignment for which *gstop* is asserted. Thus, when all time steps have been evaluated, the number of assignments for which *gstop* is asserted will be the sum of all assignments for which *stop* was asserted during the simulation run. Since we assumed that all binary simulations must stop, then for all possible assignments, *gstop* will be **1**. This is equal to the constant function $\hat{\mathbf{1}}$. Since one of the the exit conditions for the while loop in Algorithm 5 is $gstop = \hat{\mathbf{1}}$, the symbolic simulation run terminates. $\square$

2.4 Summary

This chapter has presented background material on symbolic simulation. This includes an overview of BDDs and algorithms used to manipulate them and the basic concepts of simulation. These are then combined to produce symbolic simulation. A theory of symbolic simulation was presented that showed that symbolic simulation is as sound and complete as binary simulation.

Chapter 3

Approximate Values

The use of BDDs in symbolic simulation creates the possibility of memory blow-up which prevents the simulation from completing. This is the primary problem that must be overcome in order for symbolic simulation to become a primary verification method. This chapter introduces approximate values as a useful abstraction mechanism in symbolic simulation. Informally, a value is approximated by replacing it with the value X indicating that the value is either unknown or does not matter. Approximation of values results in fewer BDD nodes being created during simulation thereby lessening the probability of memory overflow.

This chapter presents algorithms for creating and manipulating approximate values. It then goes on to show how the simulator can create approximate values dynamically based on variable classification. Symbolic variables in a test are classified as either care or don't care. Functions of care variables are not approximated while functions of don't cares variables are approximated.

3.1 Abstraction

Abstraction, when applied to verification, refers to the suppression of detail that is irrelevant in proving a desired property [63]. There are two potentially useful types of abstraction in symbolic simulation: *circuit* abstraction and *value* abstraction. Circuit abstraction removes irrelevant parts of the circuit while value abstraction keeps the irrelevant logic, but abstracts values on circuit nodes that are not relevant.

The effectiveness of the different types of abstraction depends on the type of verification is being done. There are two basic types of verification: *full* and *partial*. Full methods attempt to verify all functionality in one shot; partial methods verify functionality a piece at a time. Model checking and theorem proving are full methods while simulation, including symbolic simulation, is a partial method.

Circuit abstraction is a common, and in fact, generally necessary, strategy used in model checking to reduce state space. Value abstraction is useful in simulation since each test case verifies only a small piece of the functionality. Since the same circuit is used for all test cases, it is useful to abstract away values on those parts of the circuit that are not relevant to a given test case.

Circuit abstraction is generally more powerful than value abstraction. Value abstraction requires values to be computed for irrelevant nodes; circuit abstraction does not because the nodes have been abstracted away. However, circuit abstraction generally requires more effort than value abstraction. In fact, the large amount of expertise required in finding good circuit abstractions is one of the primary impediments in making model checking practical.

Symbolic simulation can benefit from value abstraction. Since each symbolic vector verifies only a portion of the functionality, limiting the size of BDDs used to represent values on irrelevant nodes can speed up the simulation. An additional benefit in doing this is that a larger number of symbolic variables can be used in a test, allowing more coverage with the same effort as symbolic simulation without value abstraction.

3.2 Approximation

In this thesis, value abstraction is done through *approximation*. Informally, approximation is the process of replacing **1**s and **0**s in the truth table of a value function with Xs. In order to formalize this concept, it is necessary to provide a framework for relating approximate values to their exact values. This is done through a *partially ordered state space*.

3.2.1 Partially Ordered State Spaces

Let $\mathcal{A}, \mathcal{B}, \ldots$, denote sets of values in a domain. A *poset* is a pair $\langle \mathcal{S}, \sqsubseteq \rangle$ in which $\sqsubseteq$ represents a partial order over the set $\mathcal{S}$.

If $\mathcal{A} \subseteq \mathcal{S}$ then $a \in \mathcal{S}$ is an *upper bound* of $\mathcal{A}$ if and only if $b \sqsubseteq a$ for all $b \in \mathcal{S}$. The value a is a *least upper bound (lub)* if $a \sqsubseteq b$ for all upper bounds, b, of $\mathcal{A}$. A lower bound is defined similarly as a value a such that $a \sqsubseteq b$ for all $b \in \mathcal{A}$. The *greatest lower bound (glb)* is also defined similarly as a such that $b \sqsubseteq a$ for all lower bounds, b, of $\mathcal{A}$.

An *upper semi-lattice* is defined as a poset in which a least upper bound exists for all pairs of elements. Similarly, a *lower semi-lattice* is defined as a poset in which a greatest lower bound exists for all pairs of elements. A *lattice* is a defined as a poset that is both an upper and lower semi-lattice. a *complete lattice* is one for which a lub and glb exist for all pairs of the poset. All finite lattices are complete.

Let $\langle \mathcal{S}_i, \sqsubseteq_i \rangle$ be a complete lattice. The poset $\langle \mathcal{S}, \subseteq \rangle = \mathcal{S}_0 \times \mathcal{S}_1 \ldots \mathcal{S}_n$ is a lattice if the orderings $\sqsubseteq_i$ are extended point-wise. The ordering $\sqsubseteq$ is defined such that for all $a, b \in S$, $a \sqsubseteq b$ if and only if for all i, $a_i \sqsubseteq_i b_i$. The state of a circuit is the set of values on all nodes. If node values form a lattice, then, by point wise extension, the state of a circuit forms a lattice.

3.2.2 Approximate Values

A value function f' is an *approximation* of f, written as $f \sqsubseteq f'$, if and only if $\forall \phi . f(\phi) \sqsubseteq f'(\phi)$ where ϕ is a variable assignment. 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 Boolean domain. 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. By point wise extension, a circuit state is approximate if any of its node values are approximate.

Let the *symbolic extension* of an operation be an operation applied point-wise over all possible symbolic variable assignments to a value function. An approximate value for a circuit node can be generated by applying the symbolic extension of the node's operation to the two approximate input values to produce an approximate output value. The correctness of this method is captured in the following formula:

$$f \sqsubseteq f' \wedge g \sqsubseteq g' \Rightarrow (f \ \langle op \rangle \ g) \sqsubseteq (f' \ \langle op \rangle \ g') \tag{3.1}$$

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. This formula holds if the symbolic extension of the Boolean operator $\langle op \rangle$ is defined to be *monotonic*. A function f is monotonic if it obeys the following relationship.

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y) \tag{3.2}$$

3.3 The Ternary Domain

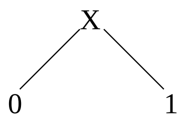
Ternary valued simulation is an example of simulation using approximate values. Ternary values can also be used as the base domain for symbolic simulation.

3.3.1 Ternary Valued Simulation

Let $\mathcal{T} = \{\mathbf{0}, \mathbf{1}, X\}$ 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** as a function of symbolic variables, 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 $a \in \mathcal{T}$.[1] This ordering reflects the semantics of state vectors as representing sets

[1]The work in this thesis is closely related to Symbolic Trajectory Evaluation (STE) [74] and invites comparisons between the two. One of the major theoretical differences is the ordering of the lattice (X above **0** and **1** instead of below). This has only a minor effect on the theory and has no practical significance.



	$\neg$
0	1
1	0
X	X

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

Figure 3.1: Hasse diagram and table for NOT, AND for ternary upper semi-lattice

of binary vectors. Higher elements in the semi-lattice represent more possible sets of binary vectors than elements lower in the semi-lattice.

The functions AND, OR, and NOT are defined over this semi-lattice. 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. Figure 3.1 shows a monotonic implementation of the AND and NOT functions over the ternary domain.

3.3.2 Ternary Symbolic Simulation

Using a partially ordered ternary domain as the base domain for symbolic simulation is one way of using approximate values in symbolic simulation. Beatty and Bryant [4] showed that using ternary vectors instead of binary vectors to exhaustively verify functionality can reduce the number of vectors required. This, in turn, reduces the number of variables required to symbolically encode a set of vectors, thereby reducing BDD sizes and computation time during symbolic simulation.

Symbolic Trajectory Evaluation (STE) [74] is a symbolic simulation method that uses a ternary base domain. STE represents symbolic approximate values using a pair of BDDs to encode the three possible ternary values. An example of this encoding is shown in Figure 3.2 with the encodings $[1, 0]$ representing $\mathbf{0}$, $[0, 1]$ representing $\mathbf{1}$, and $[1, 1]$ representing X.

In STE, the user must determine the best set of ternary values that minimizes BDD size and computation time. This works well for verifying circuits that modify data because partitioning the set of vectors is very regular and can be done algorithmically

index		input values		
decimal	v_1, v_0	y	$y.H$	$y.L$
0	$\mathbf{0}, \mathbf{0}$	$\mathbf{0}$	1	0
1	$\mathbf{0}, \mathbf{1}$	$\mathbf{1}$	0	1
2	$\mathbf{1}, \mathbf{0}$	$\mathbf{1}$	0	1
3	$\mathbf{1}, \mathbf{1}$	X	1	1

$$\begin{aligned} y.H(v_1, v_0) &= v_1 \equiv v_0 \\ y.L(v_1, v_0) &= v_1 \vee v_0 \end{aligned}$$

Figure 3.2: Symbolic indexing of ternary vectors

[45].

Thus, STE and other symbolic simulators based on ternary symbolic simulation have primarily been used in verifying datapath components such as RAMs [67], arithmetic units [51] and instruction decoders [1].

We are interested in increasing the efficiency of verification for control circuits. For these cases, it is not obvious to the user how to partition a symbolic vector into a set of ternary vectors. Instead, the user wants to create sets of binary vectors that represent things such as command types, numbers of requests, and delay values between different transactions. We want the simulator to take these binary symbolic values and approximate them where possible to reduce BDD size and computation time.

3.3.3 Simulator Changes Required to Support Approximate Values

The *stop* and *fail* conditions may need to be handled differently due to the presence of X values on these outputs. STE assumes that any X appearing on the *fail* output indicates a bug. It is up to the user to correct this even if this is due simply to an input vector being too conservative.

STE does not allow reactive tests as defined in Section 2.1.2. Therefore, *stop* is always exact in STE. Approximate *stop* values in reactive tests cause problems that cannot be handled with the SYMBOLIC_SIMULATION_LOOP algorithm given in

Chapter 2. We will deal with this problem in Chapter 5.

3.4 BDD-based Approximate Values

The STE method of encoding ternary symbolic values as two BDDs works well for the ternary domain, but does not extend easily to arbitrary finite base domains. This section describes a general method for representing and manipulating approximate values represented as BDDs over arbitrary base domains. The basis of this technique uses a modified BDD called a *Multi-Terminal BDD (MTBDD).*

3.4.1 Multi-Terminal BDDs

A MTBDD [32] is a BDD which allows terminal nodes other than **0** and **1**. MTBDDs were originally developed to represent algebraic values. In this application, it is useful to think of terminals as pointers to base domain values. Thus, MTBDDs can represent arbitrary approximate base domains. In all cases, however, approximate base domains will contain values for *TRUE* and *FALSE*. These will always be the distinguished values **1** and **0** respectively.

There are minor differences in the algorithms for manipulating MTBDDs compared to BDDs. However, the basic Apply algorithm works without modification. Most of the differences are taken care of by the functions that check and manipulate terminal values. A requirement on the base domain is that the approximate values be canonical. This is necessary in order to maintain the canonicity of BDDs with approximate values.

3.4.2 Ternary BDDs

Ternary BDDs (TBDDs) are MTBDDs having terminal nodes from the ternary base domain. They are created and manipulated in the same way as BDDs over the Boolean domain except that the terminal functions implement ternary operations.

returned			stored	
if	else	return	if	else
X	U	U	U	U
X	C	C	U	U
U	X	U	U	U
U	U	U	U	U
U	C	U	U	C
C	X	C	U	U
C	U	C	U	C
C	C	C	U	U

Table 3.1: BDD Transformations for Complemented Edges

Ternary values create a complication in dealing with *complemented edges*. Complemented edges are an optimization of standard BDDs that allow a smaller representation than BDDs without complemented edges. BDDs with complemented edges have a complement flag for both the *if* and *else* branches of each node. The complement flag indicates that the sub-function for that branch is complemented for this node.

There are four different combinations of values of the complement flags for the *if* and *else* branches. This allows the same function to be represented in two different ways, thereby breaking the canonicity of the BDD. The standard way of resolving this is by selecting one of the representations as the canonical one and transforming the non-canonical one to its canonical representation by complementing the node [58].

When complementation is applied to terminal nodes, the node is required to be complemented. If the terminal node is the X value, its complemented value is also X. This creates another source of non-canonicity when combined with complemented edges. If either the *if* or *else* sub-functions return the value X, then additional rules must be used to ensure canonicity. These rules are described in Table 3.1. The left "if" and "else" columns represent the value of the complement flag returned by the computation of these sub-functions. The "return" column indicates whether this node should be returned as complemented or uncomplemented, and the last columns indicate the setting of the complement flags when creating this node. "U" and "C" indicate complemented and uncomplemented edges respectively and "X" indicates a value of X was returned by the sub-function computation.

3.4.3 Approximate Apply Algorithm

The representation of approximate values using MTBDDs allows the user to create approximate values. But, so far, the algorithms we have seen do not allow the simulator any choice in determining the level of approximation in creating values. We need an algorithm that allows the simulator to increase the amount of approximation on values as they are created. We also need methods to remove this approximation when necessary. This is the subject of the next chapter. For now, a general algorithm for allowing the simulator to approximate values on a case by case basis is presented.

The algorithm used to create approximate algorithms is a modification of the basic Apply algorithm. The algorithm is modified to allow the creation of either an exact value or an approximate value on a node by node basis while computing the BDD. This algorithm, called approx_Apply, is shown in Algorithm 6.

The difference between approx_Apply and Apply is the addition of lines 14–15. The function *want_approximate*() implements a set of rules, called *approximation rules* that determine whether a value should be approximated or not. If so, the *approximate*() function computes and returns the correct approximate value. In theory, *approximate*() could return any value that is an approximation of the actual value. However, in practice, normally the terminal value X is returned since approximation is applied to values that are considered to be "don't care" values that should not affect the property being verified. The value X is the least expensive value to return for this case in terms of both time and memory. If no approximation is needed, then the node is returned from the unique table or created and inserted into the unique table as usual.

Figure 3.3 illustrates the operation of approx_Apply in computing the AND of two BDDs. Assume that the approximation rule is that a node is approximated by returning X if both the *if* and *else* functions are non-terminal nodes.

The execution sequence illustrates the series of calls to approx_Apply with the

Algorithm 6 APPROX_APPLY(f,g,$\langle op \rangle$)

```
1: if $terminal\_case(f, g, \langle op \rangle)$ then
2:     return $handle\_terminal\_casef, g, \langle op \rangle)$;
3: end if
4: if $cache\_hit(\{f, g, \langle op \rangle)\}$ then
5:     return $cache\_lookup(\{f, g, \langle op \rangle\})$;
6: end if
7: $top\_var \leftarrow min(var(f), var(g))$;
8: $f_{if}, f_{else} \leftarrow cofactor(f, top\_var)$;
9: $g_{if}, g_{else} \leftarrow cofactor(g, top\_var)$;
10: $t_{if} \leftarrow$ APPROX_APPLY$(f_{if}, g_{if}, \langle op \rangle)$;
11: $t_{else} \leftarrow$ APPROX_APPLY$(f_{else}, g_{else}, \langle op \rangle)$;
12: if $t_{if} = t_{else}$ then
13:     $result \leftarrow t_{if}$;
14: else if $want\_approximate(top\_var, t_{if}, t_{else})$ then
15:     $result \leftarrow approximate(top\_var, t_{if}, t_{else})$;
16: else if $unique\_hit(top\_var, t_{if}, t_{else})$ then
17:     $result \leftarrow unique\_lookup(\{top\_var, t_{if}, t_{else}\})$;
18: else
19:     $result \leftarrow create\_node(top\_var, t_{if}, t_{else})$;
20:     $unique\_insert(\{top\_var, t_{if}, t_{else}\}, result)$;
21: end if
22: $cache\_insert(\{f, g, \langle op \rangle)\}, result)$;
23: return $result$;
```

arguments passed in computing the *if* and *else* branches. The value in parentheses following the pair of operands is the BDD node that is returned by this call. The initial call is with nodes A_1 and B_1. Since both of these nodes have the same top variable, the *else* branch is computed by passing the *else* branches of each of these nodes, A_2 and B_4.

The computation of the sub-functions for the A_2, B_4 node are terminal cases because one of the operands is **1** in both cases. Therefore, the BDDs that are returned in these cases are A_3 and B_3 for the *if* and *else* branches respectively. Since neither of these nodes is a terminal, the value of this node is approximated by the approximation rule and X is returned.

The computation of the *if* branch of the A_1, B_1 node is a terminal case and

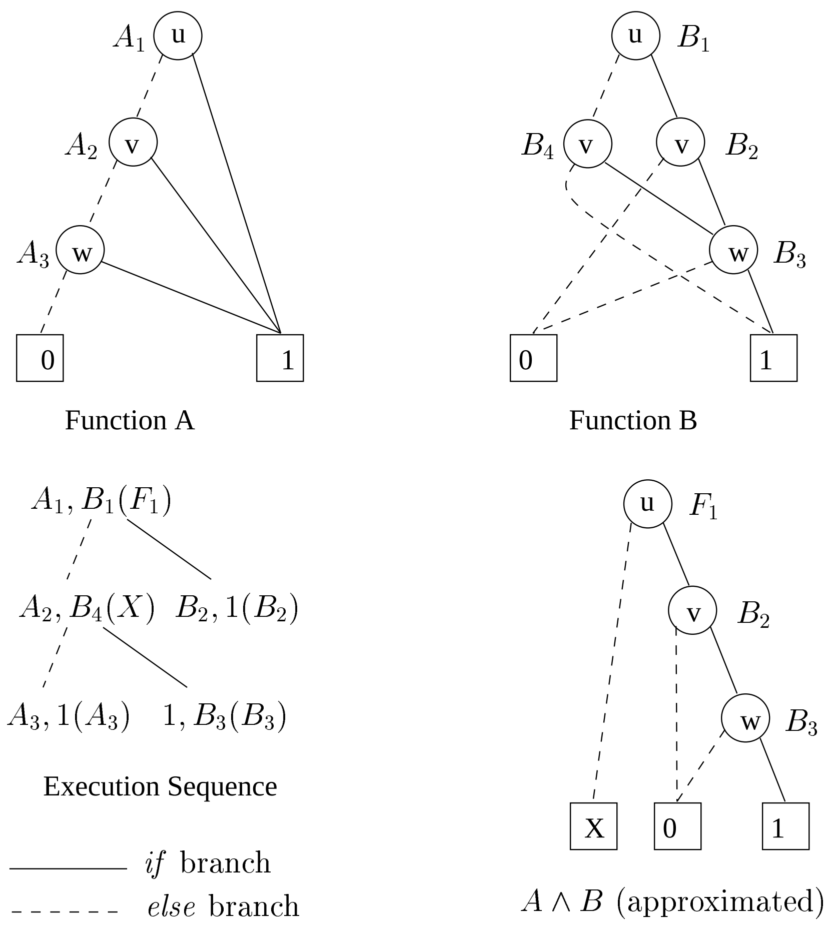


Figure 3.3: An example of approximate apply

returns the node B_2. Since the *else* branch computation returned the terminal value X, no approximation is done and a new node F_1 is created with $if(F_1) = B_2$ and $else(F_1) = X$. For this case, approximation has reduced the size of the BDD by two nodes (A new node for A_2, B_4 and A_3) compared to the exact BDD for F.

3.5 Approximation Based on Variable Classification

We want the simulator to identify nodes irrelevant to proving the property being verified. Nodes may or may not be relevant as a function of the current simulation time step and setting of symbolic variables. For those time steps and variable assignments that are irrelevant, approximate values suffice in order to reduce memory consumption and simulation time. This section presents heuristics that allow the simulator to discriminate relevant from irrelevant nodes.

There is a wide range of possibilities in choosing approximation rules for symbolic simulation. This thesis studies heuristics based on *variable classification*. To understand the motivation for using these heuristics, a generic example is given. This example illustrates the use of symbolic simulation in verifying large systems. This methodology is called *symbolic system simulation* to distinguish it from other symbolic simulation methodologies.

These heuristics are not 100% accurate in identifying when it is appropriate to approximate values and when it is not. There must be some way of dealing with this inaccuracy. This issue will be deferred until the next chapter. The remainder of this chapter will present the motivation for the variable classification heuristic, how it is implemented, and then demonstrate its effectiveness on an industrial design.

3.5.1 Symbolic System Simulation

Figure 3.4 depicts an example of a circuit (labelled DUT) surrounded by structures implementing an environment that constrain the values that can be applied to the circuit. The circuit receives a request that includes data, an address, and a request type that specifies an operation to be performed. Assume there are only four valid request types. The multiplexor constrains the request inputs to only valid types. The multiplexor input selects one of the valid types.

The circuit stores data in a pipeline before being transmitted on the data output. Assume that the DUT transmits data on the output a fixed number of cycles after

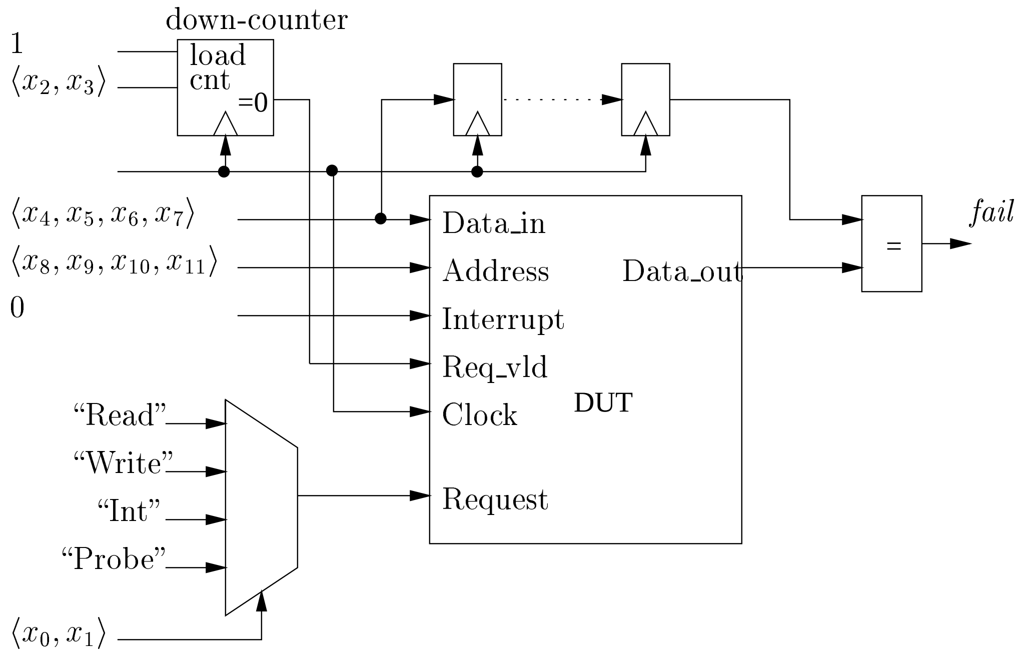

down-counter
1
$\langle x_2, x_3 \rangle$
load
cnt
=0
$\langle x_4, x_5, x_6, x_7 \rangle$
$\langle x_8, x_9, x_{10}, x_{11} \rangle$
0
Data_in
Address
Interrupt
Req_vld
Clock
Request
Data_out
DUT
=
fail
"Read"
"Write"
"Int"
"Probe"
$\langle x_0, x_1 \rangle$

Figure 3.4: Symbolic System Simulation Example

the request arrives. Checking the correctness of the output data requires pipelining the data the same number of cycles as the circuit should. This is done by the registers placed above the DUT in the figure. A comparator at the output determines whether the data matches the expected value and generates the *fail* output.

Since this is a sequential circuit, we want to be able to vary the delay of injecting the request into the circuit in order to explore potential pipeline forwarding in the DUT. This is accomplished by having a programmable down-counter in the environment. A value is loaded into this counter and the counter counts down until it reaches zero, signalled by the output labelled "= 0" in the figure. This output drives the request valid input of the circuit, thus allowing a variable delay for injecting a request into the circuit.

In a directed or random test, we would simply put various binary values on the inputs, issue some clocks, and check the outputs. In symbolic simulation, we would replace some of these scalar values with symbolic values. In the figure, symbolic values are used to select the request type (variables x_0 and x_1) and the request injection delay

(x_2 and x_3). The data is made symbolic (variables x_4–x_7) as is the address (variables x_8–x_{11}). The rest of the variables are left as binary values. Note that the figure shows the state of the test for a single cycle only. In practice, the clock input would oscillate every cycle and the *load* input to the counter would be asserted active only on the first cycle and deasserted for the remainder of the test.

Also, the symbolic values shown would only be asserted on the cycle in which the request is valid. On other cycles, the request, data, and address inputs are don't care inputs, which means values on these inputs should not affect the functionality of the circuit for this test. In binary simulation, it is normal to set these inputs to a benign value, such as all zeroes, once the request has been injected.

Using symbolic simulation, we can do better. After the request is injected, we can put symbolic variables on the don't care inputs since this should not affect the correctness of the test. There are 13 inputs that are don't cares after request injection. Basically, this is all inputs except the *load* input to the counter. Assume that after the request is injected, symbolic variables x_{12}–x_{24} are injected on the 13 don't care inputs.

If we consider all 25 variables, then this symbolic test represents $2^{25} \approx 32$ million binary tests. However, the don't care variables do not necessarily increase the amount of control state covered by the test. Also, the data variables that are checked at the end don't necessarily increase control state coverage either. Because of the *data independence* property [80], a single symbolic data variable is sufficient to distinguish correct from incorrect data. Thus, the set of symbolic variables that search the control space of the design are those that select the request and delay value, x_0–x_3. The address inputs are also don't care inputs since the correct output value does not depend on them. Thus, this test represents 16 significantly different binary tests.

3.5.2 Variable Classification

Based on the above example, we can classify symbolic variables into three types:

- **Control Variables**. These are the variables that significantly differentiate the binary tests represented by a symbolic test. These variables control the operation of the DUT and the timing of events. They are the variables that search the control state space of the DUT. In the example above, x_0–x_3 are the control variables.

- **Data Variables**. These are values that are injected and checked by the test. For many cases, data variables are simply transferred through the DUT unaltered. If they are modified, by an ALU for example, then data variables may rightly be called control variables since one set of data variables can be viewed as steering the other set through the circuit in order to produce an output value. Therefore, we only consider variables transferred unmodified as data variables. Variables x_4-x_7 are the data variables in the above example.
- **Don't Care Variables**. These are values that do not affect the correctness of the test. If there is a bug in either the environment or DUT, don't cares from the simulator's point of view may actually be care variables. Thus, there are two types of don't care variables: *user don't cares* which are what the user intends to be don't cares, and *simulator don't cares* which are true don't cares. Variables x_8–x_{24} are the user don't care variables in the above example.

We only consider control variables to index the set of binary tests encoded by a symbolic test since only these variables cause different control state space to be searched. A single test from the set of indexed tests will have all its control variables assigned to constants, but can have symbolic variables on data and don't care inputs. Since these symbolic variables do not increase coverage of control state in the DUT, it is fair to ask what the value is of using symbolic data and don't care variables, especially if they cause a significant increase in simulation time due to large BDDs being created on don't care nodes.

The value of these symbolic variables is two-fold. First, symbolic don't cares may catch bugs by exercising inputs that were thought not to be relevant. This is usually only a minor source of bugs. Finding these bugs at the expense of slowing down the simulation so much that other bugs may not be found is unproductive. Second, if no bugs are found, the fact that don't care inputs are symbolic gives the user more information about what has been covered by the test. Don't care inputs that were covered by symbolic variables eliminate the need to perform random simulation to cover different combinations of these inputs. But, again, this advantage is not worthwhile if the cost is too high. Thus, having symbolic variables on don't care inputs has value, *but only if the cost of doing so is low.* Using approximate values minimizes the cost of computing don't care values and makes it feasible to put symbolic variables on all don't care inputs during simulation.

3.5.3 Approximation Based on Variable Classification

A node being evaluated by the simulator may be a *care node* or a *don't care node.* A care node is one which affects a checked output value. That is, an incorrect value on a care node causes an incorrect value on a checked output value. A don't care node is one that does not affect a checked output. Nodes may be care or don't care nodes at different time steps during simulation. In a symbolic test, a node may also be either a care or don't care node as a function of the control variable assignments.

A *care value* is a value function that if incorrect causes an incorrect value function to appear at a checked output. A *don't care value* is a value function that does not affect a checked value.

Assume that control and data variables are ordered ahead of don't care variables in BDDs. If for some variable assignment a node is a care node, then all variables on the path from the root of the BDD to a terminal node for this assignment will contain only control and data variables. For all assignments for which the node is a don't care node, control, data, and don't care variables may appear on the path from the BDD root to a terminal node.

Thus, BDD subtrees in which don't care variables appear can be approximated since they represent don't care nodes. However, we do not want the entire subtree approximated since some branches may represent care assignments and others don't care assignments. If a node is labelled with a don't care variable, then the entire subtree rooted by this node must be don't care assignments only. This node can be approximated by the value X. Any subtree that points to this value will also be approximate with some branches pointing to exact values and others pointing to approximate values.

Thus, don't care values can be identified by classifying the variables in the BDDs being created for a particular node. We can exploit this classification using the following approximation rule.

Approximation Rule 1 (Simple Variable Categorization Rule). *if top_var is a don't care variable, then return X as an approximation of this BDD node value.*

This is implemented by having *want_approximate*() return *TRUE* if the variable passed to it (*top_var*) is a don't care variable. This means that the simulator must keep a table classifying each variable as either a care or don't care variable. It is not obvious how such a table is created. The problem of classifying variables will be addressed in the next chapter. Assume for now that each variable has already been classified.

Approximation Rule 1 implies several things. First, variables need only be classified as care or don't care variables. Second, this classification is based on the simulator's view of things, not the users. Third, control variables are classified as care variables, don't care variables are classified as don't cares, and data variables may be classified as either.

It may not be obvious why data variables do not need to be classified as care variables. Normally, if data variables are simply passed through the circuit as a function of the control variables and control variables are ordered ahead of data variables in BDDs, then computed data variable nodes in the BDD are handled by the terminal value functions. In these cases, either a constant is returned or one of the original BDDs. In either case, no new node will need to be created. Approximation only occurs if it is possible to create a new node. Therefore, no approximation is done for these cases.

By treating data variables as don't cares, they will remain exact if they are simply passed through the circuit and will be approximated if they are manipulated in any way. If data variables are manipulated and this manipulation is being verified, then it is appropriate to use exact values for these values. Thus, data variables that are manipulated are considered to be care variables.

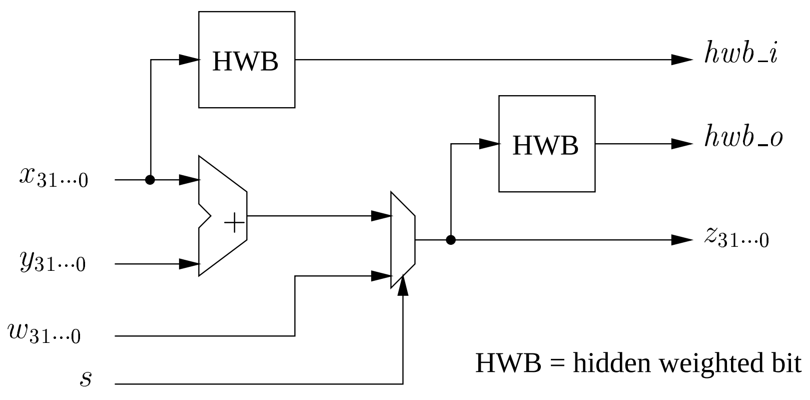


Figure 3.5: Synthetic circuit with don't care logic

3.5.4 Analysis of the Simple Variable Categorization Rule

We would like to understand Approximation Rule 1. Figure 3.5 is an example of a circuit with don't care logic. The primary component in this circuit is an adder. The data paths into and out of the adder go to other places in the circuit and other inputs may be multiplexed with the output of the adder. A CPU, for example, may have an ALU. Its inputs may be connected to other functional units which are selected by the opcode. Its outputs may be multiplexed with the outputs of other units, again depending on the opcode or other control signals. Thus, this circuit is representative of the type of systems we are interested in verifying.

Suppose we are only interested in testing the paths from inputs w, x, and y to output z. In this case paths leading to other outputs are don't care paths. Don't care logic is represented by the blocks labelled "HWB" (hidden weighted bit). In general, we must consider that don't care logic may have any function. Since random functions create exponential sized BDDs in general, the HWB block, which has exponential BDD size for any variable order [7], represents worst case don't care logic.

There are two types of don't care logic: input and output. Input don't care logic is driven by don't care inputs. Output don't care logic is driven by care inputs, but drives don't care outputs. The effect of approximation on these two types of don't care logic is different since approximation is based on the input variable classification and each type receives different classes of variable. The two HWB blocks in the example represent these two types of don't care logic.

If the goal was to verify the adder completely, then variables on the x and y inputs would all be care variables. In this case, all node values would be exact and there would be no benefit in using approximate values over exact symbolic simulation. However, in symbolic system simulation, we are usually interesting in verifying the behavior of the system after verifying the individual components. Thus, if we assume the adder has already been verified at the component level, we only need to verify that it is correctly connected in the system. The only control variable that needs to be exercised in this example is the multiplexor select input s. To verify that this control signal works correctly, it is only necessary to pass data through the adder.

Thus, a simple system level test would be to put a set of symbolic data variables on x inputs, zeroes on y inputs, and a different set of symbolic data variables on the w inputs. If we put a symbolic control variable on the s input, we need only check that the correct symbolic variables passed through the circuit as a function of the symbolic control variable. We can repeat this test with the inputs on x and y swapped to check all paths through the circuit. The following table lists the results of running these two tests using both approximate and exact symbolic simulation.[2]

	approximate			exact		
		BDD nodes			BDD nodes	
test case	time	peak	final	time	peak	final
add_dc2	0.03s	196	99	256s	7.7×10^6	48,477
add_dc3	0.03s	196	99	258s	7.7×10^6	48,477

Approximation has radically reduced simulation time on the don't care portion of the logic compared to exact symbolic simulation. The final BDD size shows a large improvement also, but a more realistic measure of improvement is the difference between peak BDD node usage which more accurately reflects the amount of computation done during each simulation run.

It may not be apparent why this dramatic improvement is possible. Although the circuit state is highly approximate, the pass/fail indication for the property being

[2]Simulations run on a Pentium III, 800MHz with 512Mbytes of memory.

verified is not. But this is only a necessary condition since the test case passed. If there was a bug and the test case failed, it is quite likely that the pass/fail output would be approximate. Since approximate values are conservative, additional work may need to be done in order to produce an exact result that indicates that bug really exists. In fact, it may require more work than using exact symbolic simulation. In practice, bugs are triggered easily and producing an exact enough value for these cases is generally a small amount of overhead.

The variable classification that was used for these tests marked the select input as a care variable and all data variables as don't care variables. Because the data was simply passed through the circuit, all BDD manipulation involving data variables on the care path through the circuit was done by the *handle_terminal_case*() function. As soon as the data variables entered the HWB blocks which tried to manipulate them, these values were immediately approximated.

This case is a worst case for exact symbolic simulation. The worst case for approximate symbolic simulation occurs when both inputs to the adder are symbolic and the output value of the adder is checked. In this case, these data values would have to be made care values. Consequently, exact values would be generated in the HWB blocks and approximate symbolic simulation would have the same performance as exact symbolic simulation.

This leaves open the question of what the behavior might be on a realistic example which has a mixture of symbolic care and don't care variables. The next section presents results of applying symbolic simulation with approximate values on a realistic example.

3.5.5 Symbolic Simulation with Approximate Values on an Industrial Example

This section examines the behavior of symbolic simulation with approximate values on a real industrial design. The design, called the *NI*, is a block in a traffic management chip which is part of a large networking system. The primary function of this chip is buffering packets into and out of the system.

Simulator Implementation

The implementation of the algorithms in this thesis is embodied in a program called *qsym*. This program accepts hierarchical gate level Verilog input. It is compatible with the gate level format generated by Synopsys' Design Compiler. The test circuit is in Verilog RTL format. Code is synthesized with no constraints and low compile effort to generate a gate level netlist quickly.

The simulator supports a custom behavioral test language which is bit-level, but allows behavioral constructs including *assign, thread, guard (if-else,) wait, symbolic variable creation, data check*, and *stop*. This language allows most of the functions available in behavioral Verilog. A custom language was chosen because Verilog does not support some of the constructs necessary for symbolic simulation (namely, symbolic variable creation) and for ease of implementation.

The simulator uses David Long's BDD package *cmubdd* [57]. This package was extended to support the APPROX_APPLY algorithm in addition to being heavily modified[3] to improve the performance of the package in this application. It was found that symbolic simulation with approximate values tends to produce many small BDDs. Profiling of the simulator revealed that up to 80% of the execution time of the simulator was spent in the BDD routines performing error checking. These routines were optimized by stripping out all error checking. This gave a five-fold improvement in BDD package performance and a two to three times improvement in simulator performance when there are less than 10 symbolic control variables.

Circuit Description

One of the main functions of networking circuits is to buffer data. To maximize the utilization of buffering RAM, packets are stored in fixed sized cells in memory. Therefore, there needs to be a circuit that breaks incoming packets into fixed sized cells. The NI block performs this function. The basic functions of the block are listed below.

- Add an eight byte header to the incoming packet.

[3]read:*hacked*

- Break the resulting packet into 112 byte cells.
- Create cell headers identifying which packet the cell belongs to.
- Handle packet lengths between 64 and 16K bytes.
- Pad out end-of-packet cells to the correct length.
- Convert a 128 bit wide packet interface to a 160 bit wide cell interface.
- Check for errors and mark cells as bad if any word within the cell is in error.

Structurally, the block consists of three main components: an incoming 128-bit word FIFO which feeds a re-alignment buffer to convert the 128-bit interface to 160-bits. The re-alignment buffer also breaks packets into cells and performs the necessary padding and insertion. The output of the re-alignment buffer is a cell FIFO which drives the output of the block. The block implementation requires approximately 100K gates and 3500 state holding elements. The main difficulty in implementing this unit is in handling the various boundary conditions in packing 128-bit words into a 160-bit interface and padding cells to the correct length at the end of the packet.

The NI consists mostly of data path elements. The control state consists of the depth of the incoming word queue, the depth of the cell FIFO, and the state of the re-alignment buffer.

The Test case

The strategy for verifying this unit using symbolic simulation is to inject a stream of packets and then check that each is properly converted to cells. The test case takes advantage of packet symmetry by checking only a single packet within the stream. Symbolic variables are used to select the target packet within the stream. The role of the symmetric packets within the stream is to reach all the possible initial states that the target packet may encounter. If all possible target packets are exercised, then all possible states are tested. Symmetry reduces the number of checked data variables and increases the number of don't care variables allowing more opportunity for approximation. Symmetry does not reduce the number of control variables that are care variables.

In order to explore the entire reachable state, the following parameters are controlled by symbolic variables in the test case: target packet length, symmetric packet length, inter-packet delay for each packet, and intra-packet delay (one delay “bubble” is inserted within each packet, with the place within the packet selected by symbolic variables) for both the target packet and symmetric packets.

The test case has 66 control variables, 32 data variables, and up to 565 don’t care variables. The test simulates between 56 and 76 machine cycles, depending on the the control variable assignments.

This test was run on the NI shortly after the block was designed. The designer had done extensive, but not complete, verification of the block using directed tests. At the time this symbolic test was run on the block, many bugs had already been found and fixed in the design.

The symbolic test found two bugs, one trivial and the other simple. The first bug was that two packet header bits were not being passed through to the cell headers. This was due to an ambiguity in the specification and so would easily have been caught by directed testing if the designer had interpreted the specification differently.

The second bug was that one particular data word was not transferred properly when the packet length was between 76 and 79 bytes. This was discovered easily by the symbolic simulator since it specifies the packet length using a symbolic value. The bug was easy to diagnose since the symptom was a don’t care data variable on a different word in the previous packet being propagated through instead of the correct word. This bug could have been found by directed testing without too much effort. This is exactly the type of bug that is targeted by this research.

Results

Figure 3.6 plots the execution time and memory usage of both exact symbolic simulation and symbolic simulation with approximate values for the NI.[4]

[4]Simulations run on a Pentium III, 800MHz with 512Mbytes of memory.

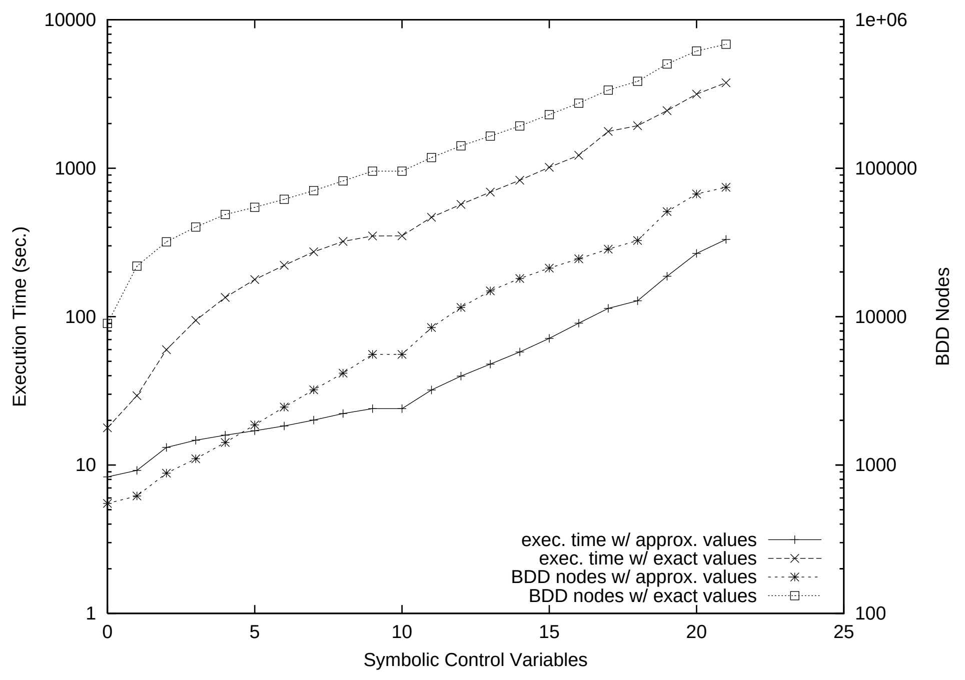


Figure 3.6: Comparison of approximate and exact symbolic simulation

The test case code identifies control variables and allows the user to individually specify whether each control variable is symbolic or is either the constant **0** or **1**. The figure plots execution time and the maximum number of BDD nodes used during the test as a function of the number of symbolic control variables. A number of runs were performed in which the number of control variables made symbolic is varied from zero to 21. Control variables that are not symbolic are set to either **1** or **0**.

The plot shows that both exact and approximate symbolic simulation scale exponentially with the number of symbolic control variables. In both cases, increasing the number of symbolic control variables by one increases the number of BDD nodes by approximately 25%. Run time in both cases is proportional to the number of BDD nodes created, indicating that run time is dominated by BDD creation time.

Exact symbolic simulation creates more than ten times the number of nodes than approximate symbolic simulation with commensurately slower execution time. Therefore, for this example, symbolic simulation with approximate values improves simulation efficiency compared to exact symbolic simulation. However, because both methods scale exponentially at the same rate, this advantage only allows approximate symbolic simulation to handle a fixed number of additional symbolic control variables before memory runs out. For this example, approximate symbolic simulation allows tests with 11 more control variables to be used with the same BDD node limit.

3.6 Summary

This chapter introduced approximation and presented algorithms for creating approximate values on a case-by-case basis. A strategy for deciding how much to approximate values based on variable classification was described. An experiment on a realistic test case showed that approximate values based on variable classification improve the performance of symbolic simulation by reducing the BDD sizes.

Chapter 4

Improving the Approximation

The previous chapter presented algorithms for approximating values and for using variable classification to direct the creation of approximate values. The result was a reduction of BDD computation time compared to performing exact symbolic simulation.

There is a price that is paid for this: the user must manually classify variables as either control, data, or don't care variables. This has two problems. First, there is the additional, though generally small, effort in doing this. Second, the user may incorrectly classify variables leading to conservative answers if too few variables are classified as care variables or excessive simulation time if too many are classified as care variables.

This chapter addresses these problems by introducing methods to automatically classify variables. The simulator starts with an optimistic classification which may cause the simulator to incorrectly approximate values. This may cause the final simulation result to be approximate. The simulator then re-classifies variables to "improve" the approximation and re-simulates the design using the improved variable classification. Multiple iterations of re-classification and simulation may be performed until an exact result is produced by the simulator. The goal is to improve the approximation just enough to create an exact result and no more in order to minimize simulation time and memory usage.

4.1 Adaptive Variable Classification

There are three steps in performing automatic variable classification. First, an initial classification for each variable is chosen. Next, simulation is performed, and finally, variables are re-classified based on the result of the simulation run. The last two steps are repeated until an exact result is produced. The initial classification may either be specified by the user or be some default classification.

During simulation with a particular classification, every node value will have its own unique approximate value function that is dependent on the node, simulation time, and current variable classification. If re-simulation is required due to an overly conservative result, then there is a set of node values that must be made more exact to produce an exact final result. This could be done either by explicitly identifying those node values and making only these values more exact or by identifying the variables that those node values depend on and re-classifying these as care variables.

Identifying the set of nodes that needs improvement entails storing information about how all computed node values were approximated. The amount of information required to be stored can be prohibitive, limiting design and test scalability. Since our goal is to replace random and directed tests, it is necessary to be able to scale design and test sizes as well as they do. Node-by-node approximation improvement does not allow that, consequently this thesis considers improvements using variable re-classification only.

Iteratively simulating and then re-classifying variables until an exact final result is produced can be done using a single extra word of storage per node in the circuit (see Section 4.2). Since a given variable may fan-in to many nodes over many simulation cycles, variable re-classification may cause more values than is necessary to be made exact. Thus, variable re-classification-based improvement is scalable, but at the cost of node values potentially being more exact than is necessary.

4.1.1 Related Work

Approximation improvement by re-classifying variables is an instance of automatic abstraction refinement. As far as is known, the work described in this thesis is the only work describing the use of automatic abstraction refinement on symbolic simulation.

Automatic abstraction refinement has been studied in the context of model checking [69, 68, 56]. Although the goals and algorithms are quite different between abstraction applied to model checking and symbolic simulation, at a high-level, there is some commonality. The approximation improvements in this thesis are iterative. Several methods proposed for refining model checking abstractions iteratively have been proposed [2, 21, 38, 54, 3, 47]. The general methodology is to partition the state space, abstract each partition, and then incrementally improve one partition at a time.

Some of these methods select a partition to refine based on the abstract counterexample generated [21, 38, 54, 3, 2]. A counterexample in model checking is a sequence of states, while in symbolic simulation it is a variable assignment. Thus, the algorithms for extracting the necessary information from the counterexample are different from the methods used in this thesis.

Of the methods using counterexample guided refinement, some base the refinement on variable analysis [54, 3, 21]. These methods generally use more sophisticated analysis techniques than the simple variable selection heuristic used in this thesis.

4.1.2 Example of Approximation Improvement

Consider the circuit shown in Figure 4.1. This circuit has three inputs: a, b, and c. We want to show that the output f is $\mathbf{0}$ for all possible input values. To prove this, we apply symbolic variables x_1, x_2, and x_3 to each of these inputs respectively. Let the variable order be $x_1 < x_2 < x_3$.

Analyzing the circuit, we decide that x_1 is a care variable and x_2 and x_3 are don't cares. Therefore, we set the approximation rule to return X when a BDD with variables x_2 or x_3 is being created, and an exact value when x_1 is the BDD variable. The first line of Table 4.1 shows the computed values for each node.

The result that is produced, $x_1 \wedge X$, is conservative. To improve this result, the simulator re-classifies one of the variables from don't care to care. For now, assume that the simulator arbitrarily chooses from amongst the don't care variables. We will

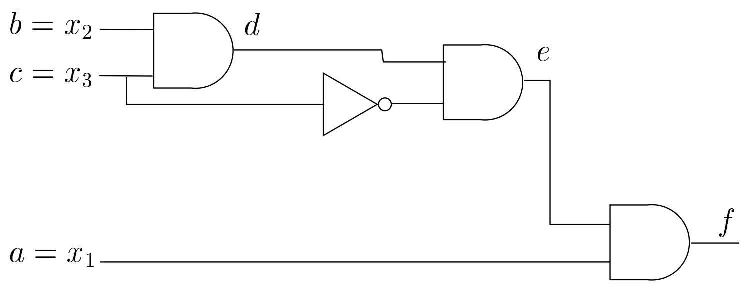


Figure 4.1: Example circuit for approximation improvement

x_1	x_2	x_3	d	e	f
C	D	D	X	X	$x_1 \wedge X$
C	D	C	$X \wedge x_3$	$\mathbf{0}$	$\mathbf{0}$

Values for variables x_i indicate whether the variable is a care (C) or don't care (D) in the approximation rule.

Table 4.1: Node Values for Approximation Improvement Example

see better algorithms for doing this shortly.

In this case, let x_3 be the variable that is chosen to be promoted from don't care to care status. The second line in Table 4.1 shows the values computed for each node using this new classification. The result is the exact value $\mathbf{0}$ on the output.

4.2 Re-Classification Variable Selection

In the test cases we are studying, it is common to have hundreds or even thousands of symbolic variables injected into the test. Most of these variables are don't care variables that we want the simulator to keep as approximate as possible. Consequently, re-classifying don't care variables by random selection is very inefficient. Instead, we want the simulator to intelligently select variables to be re-classified.

To do this, a set of variables that are potential re-classification candidates are associated with each node value computed during simulation. This set is created dynamically for each node when a new value is computed for that node. Similarly to how the actual node value is computed, the set of re-classification candidates is created from the re-classification candidate sets for the input values to this node. The re-classification candidate set for the final output of the simulation specifies which variables are to be re-classified for the next simulation run.

There are different strategies in computing these sets that have implications on memory usage and the number of simulation runs required to classify all variables correctly. Possibilities range from computing the largest possible set of variables to re-classify to the minimum number of one variable to re-classify per run. Computing the largest set means that only one re-simulation run is required, but it is possible that many more variables than necessary are re-classified.

In order to avoid memory explosion caused by re-classifying too many variables, this thesis only studies methods that re-classify a single variable per run. This requires only a single extra word of storage per circuit node. The tradeoff is that a large number of runs may need to be performed in order to re-classify a sufficient number of variables.

However, the additional runs do not add as much overhead as might be apparent. Assume that all variables are initially classified as don't care variables. This will cause all computed node variables to be ternary values. Simulation time, therefore, will be short. As variables are re-classified to be exact, more simulation time is required as more exact node values are generated. When all variables are classified correctly, simulation time is much longer than the initial re-classification runs. Thus, the total fraction of time the simulator spends re-classifying variables is less than the total number of runs indicates.

To minimize the number of re-classification runs required, a good heuristic is needed to choose a candidate variable at each node. The goal of the heuristic is to select a variable that is a potential fan-in of the node value; as a requirement, it should not select variables that it can determine are not fan-in variables. Algorithm 7 illustrates a simple heuristic that always selects a don't care variable that the output value is sensitive to. This algorithm assumes a node value data structure which has two components: *val* which contains the value, and *var* which contains the identifier for the re-classification candidate. The values f and g are inputs and *node* is the output value. The output node value function is assumed to have already been

Algorithm 7 SIMPLE_VAR_SELECT(f,g,node)

```
1: if $is\_non\_controlling(g.val)$ then
2:     $node.var \leftarrow f.var$;
3: else if $is\_non\_controlling(f.val)$ then
4:     $node.var \leftarrow g.var$;
5: else if $is\_controlling(f.val)$ then
6:     $node.var \leftarrow f.var$;
7: else if $is\_controlling(g.val)$ then
8:     $node.var \leftarrow g.var$;
9: else if $g.var = \{\}$ then
10:    $node.var \leftarrow f.var$;
11: else if $f.var = \{\}$ then
12:    $node.var \leftarrow g.var$;
13: else if $\neg is\_care(g.var) \wedge is\_care(f.var)$ then
14:    $node.var \leftarrow g.var$;
15: else if $\neg is\_care(g.var) \wedge g.var < f.var)$ then
16:    $node.var \leftarrow g.var$;
17: else
18:    $node.var \leftarrow f.var$;
19: end if
20: return $node$;
```

computed. Initial re-classification candidate sets for values injected by the test case are selected according to Table 4.2.

Lines 1–8 select a candidate variable for those cases in which at least one of the inputs is a constant. The function *is_non_controlling*() returns true if the value is a non-controlling constant for the Boolean function of this node. For the AND function, **1** is a non-controlling value. Similarly, *is_controlling*() returns true if its argument is a controlling constant value with **0** being controlling for the AND function.

If one input is non-controlling, the relevant re-classification candidate is the re-classification candidate value on the other input. If one of the inputs is controlling, then the other input cannot affect this output value. Thus, the candidate value from the controlling input is selected. Note, that the output value will be a binary constant in this case, which means that this re-classification candidate is really a don't care. However, for debugging purposes, it is useful to propagate the variable from the

value	re-classification candidate set
0	$\{\}$
1	$\{\}$
X	$\{\}$
care variable x_i	$\{\}$
don't care variable x_i	$\{x_i\}$

Table 4.2: Re-classification Candidate Set Creation Rules

run	**class.**			**node values{re-classification candidate}**					
	x_1	x_2	x_3	a	b	c	d	e	f
1	D	D	D	$x_1\{x_1\}$	$x_2\{x_2\}$	$x_3\{x_3\}$	$X\{x_2\}$	$X\{x_2\}$	$X\{x_1\}$
2	C	D	D	$x_1\{\}$	$x_2\{x_2\}$	$x_3\{x_3\}$	$X\{x_2\}$	$X\{x_2\}$	$x_1 \wedge X\{x_2\}$
3	C	C	D	$x_1\{\}$	$x_2\{\}$	$x_3\{x_3\}$	$x_2 \wedge X\{x_3\}$	$X\{x_3\}$	$x_1 \wedge X\{x_3\}$
4	C	C	C	$x_1\{\}$	$x_2\{\}$	$x_3\{\}$	$x_2 \wedge x_3\{\}$	$0\{\}$	$0\{\}$

Values for variables x_i indicate whether the variable is a care (C) or don't care (D) in the approximation rule.

Table 4.3: Node Values for Re-classification Example

controlling input.

Lines 9–12 catch those cases in which the value X was applied as an input value. The function *is_care*() returns true if the variable passed to it has been marked as a care variable. Lines 13–14 cause a don't care variable to be selected if either of the input candidates is a don't care variable.

Lines 15–16 select a don't care variable when both input candidates are don't care variables. There are a lot of possible heuristics that could apply in this case. However, using only local information, it is impossible to guarantee that the best variable will be selected in all cases. The simple heuristic of selecting the lowest ordered variable allows the user to control re-classification candidate selection when there is a choice. This heuristic works well in most circumstances.

To illustrate this algorithm, we can apply this to the example shown in Figure 4.1. Assume that the initial classification of all variables is as don't care variables. Table 4.3 lists the values of each node and the re-classification candidate set for each node. After each run, the re-classification candidate at output f is marked as a care variable and the simulation re-run. The table lists each run performed by the simulator after re-classification until an exact value is produced.

In this example, the simulator had to classify all variables as care variables in order to get an exact result at the output. Contrast this with the re-classification done in Table 4.1. In this case, only one re-classification was required to produce an exact result. However, in order to determine the correct variable to re-classify in this case, it is necessary for the simulator to know about the reconvergence of x_2. This requires non-local knowledge of the circuit when computing the value of node d. In the worst case, the effort required to determine this is the same as simply making all variables exact. Thus, we must accept some amount of inefficiency in finding variables to re-classify. This can be a problem in realistic cases. Section 4.5 will address this problem in more detail and propose some solutions.

4.3 The Search and simulate Algorithm

The simple algorithm suggested in the previous section suffers from significant overhead in performing re-classification runs before a final result is produced. This section describes an algorithm that improves the efficiency of searching for re-classification candidates. The basic idea of this algorithm is to have two simulation modes, one for searching for variables to re-classify and the other to simulate with a particular variable classification.

It is not necessary to mark variables as care variables and create BDDs in order to discover which variables are care variables using our variable selection heuristic. Instead, the simulator can operate in a mode in which it is “searching” for care variables. Each time it discovers a variable, the simulator sets it to a constant for the next run to remove it from consideration and reruns the test. It continues doing this until the output value is a constant.

When the final output is constant during search mode, the simulator can assume that it has discovered all care variables. It can then re-run the simulation with all variables symbolic and care variables marked as such so that exact values are generated where needed. This is called “simulate” mode to emphasis that the simulator expects that the test case will be completely simulated.

An added bonus of the search and simulate method is that setting symbolic variables to constants can very quickly find simple bugs. This is especially important when debugging test cases in which the test may fail due to incorrect errors on non-symbolic inputs. Simulating with all variables marked as don't cares will find these basic errors much quicker than even approximate symbolic simulation with all care variables classified correctly. Experience has shown that once the simulator produces a passing exact value while searching, the test usually will pass completely. Thus, at this point it makes sense to restore all symbolic variables to their original state and re-simulate using the correct classification to make values exact where necessary.[1]

Algorithm 8 lists the SEARCH_AND_SIMULATE algorithm. The simulation starts by applying an initial classification to all symbolic variables. This could either be supplied by the user or be a default classification of don't care for all variables.

The variable *mode* in the algorithm determines whether the algorithm is searching or simulating. In simulate mode, it assumes that the current classification is correct. The outer while loop performs iterations of searching and simulating. Initially, the loop assumes that the current classification is correct and so it sets the mode to simulate (line 3). The inner while loop first simulates the test case (line 5) and a symbolic value for *fail* is returned. A symbolic value is said to be *satisfiable* if there exists some variable assignment for which the value *true*. If the *fail* value returned is satisfiable, then a bug has been discovered and the algorithm immediately returns a **FAIL** indication (lines 6–7).

If a symbolic value is **0** for all assignments, it is said to be *unsatisfiable*. If the value of *fail* is unsatisfiable, then an exact value has been produced. In simulate mode this means the test completed with no errors. Therefore, the algorithm returns with a **PASS** indication (lines 8–10). If not in simulate mode, then the simulator must have set some variables to constants. Therefore, at this point, the simulator breaks out of the inner while loop (line 12) in order to restore the discovered variables to symbolic values.

If the *fail* output was neither satisfiable nor unsatisfiable, then it is approximate

[1]In fact, it is this observation that inspired the search and simulate method in the first place.

Algorithm 8 SEARCH_AND_SIMULATE(f,g)

```
 1: $apply\_initial\_classification()$;
 2: while (1) do
 3:    $mode \leftarrow$ SIMULATE;
 4:    while (1) do
 5:       $fail \leftarrow simulation\_loop()$;
 6:       if $is\_satisfiable(fail)$) then
 7:          return FAIL;
 8:       else if $fail = \mathbf{0}$ then
 9:          if $mode =$ SIMULATE then
10:             return OK;
11:          end if
12:          break;
13:       end if
14:       $mode \leftarrow$ SEARCH;
15:       $var\_id \leftarrow get\_selected\_var(fail)$;
16:       $var[var\_id] \leftarrow \{\mathbf{0}, \mathbf{1}\}$;
17:       $pushvar\_id$;
18:       $x\_satisfy(fail)$;
19:    end while
20:    while stack not empty do
21:       $var \leftarrow pop()$;
22:       $make\_symbolic\_care\_variable(var)$;
23:    end while
24: end while
```

(assignments are either to **0** or X). The simulator, therefore, switches to search mode (line 14) gets the variable id returned by the selection heuristic for the *fail* output, and randomly sets this variable to one of the constants, **0** or **1** (lines 15–16). Each variable that is discovered in this way is pushed onto a stack to allow restoration to symbolic values later on (line 17).

The last while loop (lines 20–23) restores those variables that were set to constants back to being symbolic values. At this point, the simulator potentially has all variables correctly classified. Since the outer loop sets the mode to simulate, if they are classified correctly, simulation will produce an exact value. If not, the simulator reverts to search mode and continues searching for more variables.

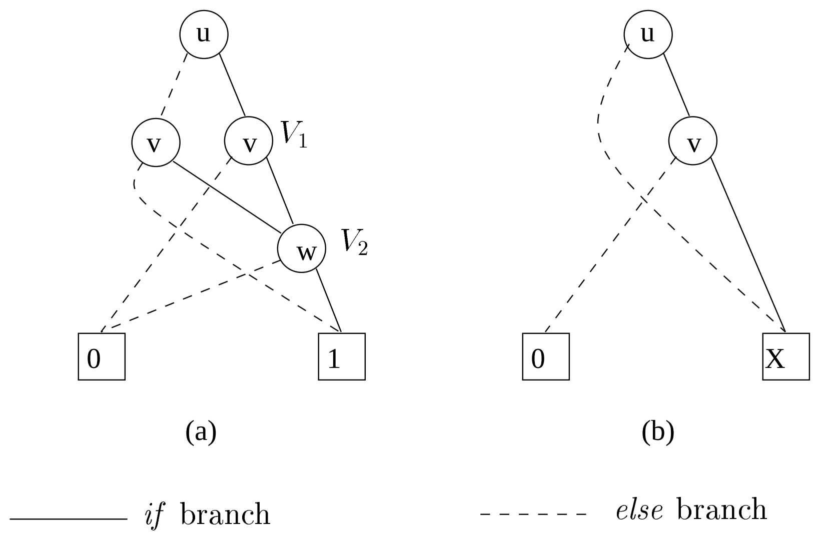


Figure 4.2: Example BDD for search and simulate algorithm

This algorithm does not obviously terminate. The issue is: what is to prevent the simulator from re-discovering the same set of care variables from one simulate/search iteration to another? For example, suppose the output of exact simulation was the BDD depicted in Figure 4.2a. Assume the SEARCH_AND_SIMULATE algorithm discovers variable u in the first iteration and sets this variable to **1**. In the next iteration it finds v and sets this variable to **0**. The third iteration will produce the exact value **0** since this path in the BDD leads to the terminal node **0**.

The SEARCH_AND_SIMULATE algorithm will then switch to simulate mode and mark u and v as care variables and re-simulate. The result of this simulation will be the BDD in Figure 4.2b. However, if it starts searching again, the algorithm may follow the same path as before and never discover variable w, which is necessary in order to produce an exact result.

To solve this problem, whenever an approximate *fail* value is returned, the algorithm calls the *x_satisfy*() function (line 18) which traces a path from the X terminal node back to the root node. For each variable encountered along this path, it pushes that variable onto the stack and sets the variable to the constant value required to traverse the edge from that node to the terminal X. In this way, the algorithm is forced to discover at least one new variable that caused an approximate result to be produced during each search.

This guarantees that SEARCH_AND_SIMULATE will always be complete assuming that the set of underlying binary tests is complete. The is true because calling *x_satisfy*() always forces the simulator to improve the approximation at least some amount on each run.

4.4 Analysis of the Search and simulate Algorithm

The overhead introduced by automatic approximation improvement is the runs spent searching and simulating before the last run. Overhead, therefore, is measured as the time spent simulating all runs before the final one. This section analyzes the amount of overhead on an example of using symbolic simulation to verify a simple adder.

In the best case, the overhead of the SEARCH_AND_SIMULATE algorithm is only the scalar runs that must be performed; one for each variable that must be re-classified as a care variable. In the worst case, there is a symbolic run for each re-classified variable in addition to the scalar re-classification runs. The variance in overhead is unpredictable in general: the number of variables requiring re-classification is unknown and the number of simulate mode runs is unknown. In this section, we will analyze a 32-bit adder. For this case, the number of scalar re-classification runs is predictable in advance, but the number of simulate mode runs is not.

The adder is a simple ripple-carry adder with 64 inputs and each one must be marked as a care variable in order to completely verify the functionality of the adder. This experiment uses the SEARCH_AND_SIMULATE algorithm with all variables initially classified as don't care variables.

The test case uses a method called *self consistency checking* [52] to verify the functionality of the adder. Self consistency checking uses the circuit itself as its specification. The laws of addition given below can be used to verify the functionality of the adder without having to model a separate reference adder.

Identity $X + 0 = X$.

Successor $X + 1 = succ(X)$.

	runs		time		BDD nodes	
approx. rule	search	sim	search	sim	peak	final
simple	63	3	0.8s	17.5s	569K	159K
exact	-	1	-	7.1s	312K	82K

Table 4.4: Results from self consistency test of a 32-bit adder

Commutativity $X + Y = Y + X$.

Associativity $X + (Y + Z) = (X + Y) + Z$.

All of these except the Successor property can be verified without having to perform any complex operations. Let $\vec{f}$ and $\vec{g}$ be vectors of symbolic variables. Commutativity can be verified by applying $\vec{f}$ and $\vec{g}$ to inputs X and Y of the adder respectively, simulating, and saving the symbolic output of the adder away. Then $\vec{f}$ and $\vec{g}$ can be applied to the opposite inputs, Y and X respectively, and simulating to produce a new symbolic output. If this value is equal to the previous value, then the commutativity property holds.

Verifying all of these properties does not guarantee that the adder has no bugs. However, it does provide better coverage than thorough directed and random testing can provide.

Table 4.4 shows the result of running a single test that verifies the Identity, Successor, and Commutativity properties on a 32-bit adder.[2] The variable order used for this test was $X_0 < ...X_7 < Y_0 \ldots Y_7 < X_8 < \ldots X_{15} < Y_8 \ldots Y_{15} \ldots$. This is neither a best case nor worst case order.

Although 63 runs in search mode were required, the total time taken by all of these runs is a small fraction of the total simulation time. However, three runs in simulate mode were required before all variables were classified correctly The first run is always in simulate mode, but its run time is equivalent to a search mode run since all variables are initially classified as don't care variables. The first set of iterations found all variables except for one. Because all variables but one were classified as care variables, the second simulate mode run had a run time roughly equal to the final

[2]Simulations run on a Pentium III, 800MHz with 512Mbytes of memory.

run using exact simulation. A single subsequent search mode run was required after the second simulate mode run in order to find the remaining care variable. Thus, the simulate time is the sum of two runs with most or all variables classified as care variables.

One thing to note is that the BDD node usage is higher using approximate values than exact values. The final run using approximate values is identical to the exact symbolic simulation and the BDD node usage for this run is identical to exact simulation. The large BDD node usage occurs in the second simulate mode run. Approximation has increased the size of BDDs rather than decreased them. This is due to BDD node sharing being destroyed by approximation.

In conclusion, performing approximation improvement incurs some overhead. This section has shown that this overhead is not as large as may appear on the surface. However, there is potentially a large amount of variance in the amount of overhead induced by automatic approximation improvement. The next section analyzes this in more depth and explores some optimizations to reduce overhead.

4.5 Different Approximation Rules

This section shows that reconvergent paths in the circuit and test case are the primary cause of the large variance in approximation improvement overhead. Weakening the approximation rule is explored as a way of dealing with this problem.

In symbolic system simulation, different settings of care variables can cause different data variables to be propagated to checked outputs. The value function of a checked data output is usually represented as a multiplexor function in this case. Suppose the circuit we wanted to verify was exactly a multiplexor. Figure 4.3a illustrates a multiplexor being compared to a reference multiplexor as a test case. The test case has a single control variable c and two data variables x_1 and x_2. Assume we used the SEARCH_AND_SIMULATE algorithm starting with all variables classified as don't cares on to verify this circuit. Table 4.5 lists the values computed for each run of the SEARCH_AND_SIMULATE algorithm.

The simulation starts in simulate with all variables classified as don't cares. The

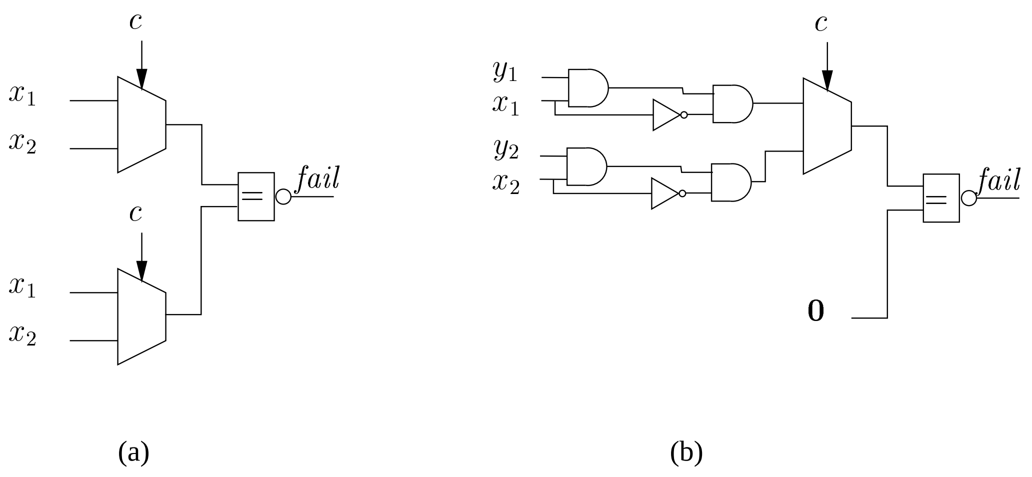


Figure 4.3: Multiplexor circuit and multiplexor circuit with reconvergent inputs

mode	c	x_1	x_2	*fail* value{re-class. candidate}
simulate	D	D	D	$X\{c\}$
search	0	D	D	$\mathbf{0}\{x_1\}$
simulate	C	D	D	$\mathbf{0}\{x_1\}$

Values for variables indicate whether the variable is a care (C), or don't care (D) if symbolic, constants otherwise

Table 4.5: Node values for multiplexor verification example

resulting simulation is conservative with variable c as the re-classification candidate. A simulation in search mode with c set to **0** causes an exact value to be generated since only the single data variable x_1 is passed through the circuit. Approximation is not done in this case because all BDD operations are handled by the terminal handling function. The last run in simulate mode has the control variable classified as a care variable. This will cause exact BDDs to be generated since operations on data variables on all branches of the BDD will be handled as terminal functions.

Now assume that the circuit we were verifying was a multiplexor, but with each data input having a reconvergent don't care path such as is shown in Figure 4.3b. The outputs of the reconvergent data inputs are always **0**. Therefore, this test case verifies that the output of the multiplexor is **0** for all variable assignments. Table 4.6

mode	c	x_1	x_2	y_1	y_2	*fail* value{re-class. candidate}
simulate	D	D	D	D	D	$X\{c\}$
search	**0**	D	D	D	D	$X\{x_1\}$
search	**0**	**0**	D	D	D	$\mathbf{0}\{y_1\}$
simulate	C	C	D	D	D	$(c \wedge X)\{x_2\}$
search	**1**	D	**0**	D	D	$\mathbf{0}\{y_2\}$
simulate	C	C	C	D	D	$\mathbf{0}\{x_2\}$

Values for variables indicate whether the variable is a care (C), or don't care (D) if symbolic, constants otherwise

Table 4.6: Node values for multiplexor with reconvergence verification example

lists the runs performed by the SEARCH_AND_SIMULATE algorithm on this test case.

Three iterations of simulating then searching are required. The first simulate mode run discovers the care variable c plus the first data variable x_1 which must be made a care variable because of the logic in front of the data input. The second simulate mode run produces an approximate result because x_2 has not been discovered as a care variable yet. The subsequent search finds the second data variable and the final simulate run produces an exact result.

The pattern here is that each simulate-then-search iteration discovers a single data variable only. If we extended the size of the multiplexor, there would be one of these iterations per data variable. Since the number of data variables is exponential in the number of control variables for a mux, there can be an exponential blowup of search and simulate runs compared to the multiplexor example without fan-in logic on the data inputs.

The Simple Approximation Rule (Approximation Rule 1) always approximates values whenever it is invoked. The fan-in logic causes the rule always to be invoked resulting in the need to mark data variables as care variables. Different approximation rules that are less conservative may reduce the need to mark data variables as care variables. This section explores two different possibilities for approximation rules and analyzes their performance on the NI example (see Section 3.5.5 on page 53 for details on this example).

if(v)	*else*(v)	BDD size	total nodes
terminal	terminal	1	N
terminal	non-terminal	N	2^N
non-terminal	terminal	N	2^N
non-terminal	non-terminal	2^N	2^{2^N}

N is the number of variables below this node

Table 4.7: BDD Sizes for different cases of returned *if* and *else* branch values

4.5.1 The Data and Linear Approximation Rules

The Simple Approximation Rule approximates a node value solely on the basis of whether *top_var* is a care variable or not. Suppose that the *if* and *else* nodes returned were the terminals **1** and **0** respectively. In this case, the BDD rooted at this node is just this single node. No BDD blow up is possible for these cases. Thus, if we relax the approximation rule to allow an exact value to be returned if the *if* and *else* branches are both terminals, then APPLY can return exact values in some cases that the Simple Approximation Rule cannot. BDD node usage may increase somewhat, but the amount should be small.

This can be generalized by considering all combinations of the *if* and *else* branches returning either a terminal or non-terminal BDD. There could be either zero, one, or two terminal values returned. For each of these cases, it is possible to determine the maximum size of the BDD rooted by the node being created and the maximum number of nodes created in the unique table for this variable. This is summarized in Table 4.7.

It obviously doesn't make sense to return an exact value if the *if* and *else* branches are both non-terminals if *top_var* is a don't care variable. However, since for all other cases, the BDD size that can be created is quite restricted, we can consider two new approximation rules.

Approximation Rule 2 (Data Approximation Rule). *If top_var is a don't care variable and either the if or else branches is a non-terminal value, then return X as an approximation of this BDD node value.*

Function Label	Function	Approximation Rule		
		Simple	Data	Linear
multiplexor	$(c \wedge x_1) \vee (\neg c \wedge x_2)$	exact	exact	exact
data	$x_1 \wedge (\neg x_1 \wedge x_2)$	X	$x_1 \wedge X$	exact
linear	$(x_1 \wedge x_2) \vee (x_1 \wedge x_2)$	X	X	exact
non-linear	$(x_1 \oplus x_2) \vee (x_1 \oplus x_2)$	X	X	X

c is a marked care variable, x_1, x_2 are don't care variables.

Table 4.8: Computed values for different approximation rules

Approximation Rule 3 (Linear Approximation Rule). *If top_var is a don't care variable, then if both the if and else branches are non-terminal values, then return X as an approximation of this BDD node value.*

The effect of these rules is apparent only when there is reconvergence present in the circuit. Table 4.8 shows the values computed for these two approximation rules and the Simple Approximation Rule for four functions with different types of reconvergence.

The differences between the functions in the table are the complexity of the reconvergence. The multiplexor function has no reconvergence. Consequently, the data variables are only ANDed with constants and, thus, are not approximated. If data variables reconverge as in the *data* case, then the approximation rule will be invoked when computing the outer AND. Since the Simple Approximation Rule always returns an approximate value, it will approximate this case to X. However, the other approximation rules will produce more exact values since the *if* and *else* branches are constants in this case.

The *linear* case will be approximated by the Data Approximation Rule during the computation of $x_1 \wedge x_2$ since a non-constant node will be returned for the *if* branch for the root node of this BDD. Thus, this value will be approximated and the reconvergent value will also be approximated. The Linear Approximation Rule will not approximate $x_1 \wedge x_2$ since one branch is always constant for simple product terms. When computing the reconvergent function, since both the *if* and *else* branches are equal, APPROX_APPLY will return the *if* branch before getting to the approximation

	runs		time		BDD nodes	
approx. rule	search	sim	search	sim	peak	final
exact	-	1	-	197s	738K	61K
simple	22	2	148s	20s	8.7K	2.5K
data	22	2	196s	75s	122K	28K
linear	6	2	41s	65s	153K	24K

Table 4.9: Results from different approximation rules on the NI test case

rule and, thus, an exact result is returned. For the *non-linear* case, the XOR function is approximated even for the Linear Approximation Rule and so it returns an approximated value for this function.

4.5.2 Analysis of Different Approximation Rules on the NI Test Case

Table 4.9 lists the results of running SEARCH_AND_SIMULATE with different approximation rules on the NI test case.[3] For this test, six control variables were made symbolic and all other symbolic variables were set to constant values. All data and don't care variables are symbolic. The initial variable classification was to mark all variables as don't care variables.

The minimum number of search mode runs we would expect is six, which is the number of symbolic control variables. The Simple and Data Approximation Rules require 16 additional variables to be marked as care variables. In both cases these are data variables. The fact that these must be made care variables indicates that these functions are not simple multiplexor functions. However, the test case is simply passing data through the circuit and checking it. Thus, it must be the case that the data paths are affected by reconvergent don't care logic. Determining the reconvergence present by analyzing the circuit is difficult because the large design size means there are many paths feeding a single data output. Also, the reconvergence takes place over many cycles because the test case injects data at the beginning of the test and

[3]Simulations run on a Pentium III, 800MHz with 512Mbytes of memory.

then checks at the end after many cycles have been simulated. The fact that only the Linear Approximation Rule need classify only the six control variables indicates that the don't care reconvergence is at least as complex as the *linear* convergence function given in Table 4.8.

The total execution time (search time plus simulate time) for the Simple and Linear Approximation Rules is less than the execution time for exact symbolic simulation despite the number of simulation runs required. The final run with properly classified variables requires substantially less execution time than the exact symbolic simulation time. The Data Approximation Rule requires more execution time than exact symbolic simulation because it needs 22 search mode runs and the execution time per run is longer than using the Simple Approximation Rule due to the more exact BDD values being created.

Comparing BDD node usage, we see that the Simple Approximation Rule generates considerably fewer nodes than any other method. This is to be expected because the more relaxed rules allow more exact values to occur on don't care nodes, while the Simple Rule almost always approximates all don't care values to X. The only reason that the Linear Rule requires less simulation time is because it requires fewer search mode runs. Thus, these two approximation rules have a time/memory tradeoff. Neither one is better in any sense. The Data Rule has no advantage for this test case, however, for other test cases, it may represent a point that is between the Simple and Linear Rules. Based on this, a good strategy to follow might be to use the Linear Approximation Rule initially and switch to the Data Rule if memory capacity becomes a problem.

4.6 Summary

The problem introduced by using variable classification to identify values that can be approximated is that it requires additional work by the user to classify variables. This chapter addressed this problem by presenting methods to improve a default and/or incorrect classification such that an exact result can be produced while maintaining the maximum amount of approximation.

The primary problem in doing this is in selecting variables to re-classify. A re-classification candidate selection algorithm was presented that finds a single variable per run to be re-classified. It may require many runs to re-classify sufficient variables using this heuristic. Therefore, the SEARCH_AND_SIMULATE algorithm was presented which can efficiently find variables with a minimum of overhead.

In the worst case, it was shown that the SEARCH_AND_SIMULATE algorithm could increase simulation time beyond that of exact symbolic simulation. To alleviate this problem, different approximation rules were explored that allowed trading off the number of runs for memory consumption.

In conclusion, it is probably best to treat approximation improvement analogously to dynamic variable ordering in BDD packages. Dynamic variable ordering requires a lot of computational effort and while it is being performed, forward progress on the task at hand stops. Approximation improvement has the same flavor in that it is time consuming and no forward progress is being made on simulation while it is being done. Therefore, it is probably best to use approximation improvement similarly to how dynamic variable ordering is used: typically, find a good variable order by dynamically ordering, then re-use it on subsequent runs of the application, since normally only minor changes are made between runs that do not affect the variable order.

Symbolic simulation with approximate values can be done the same way. Start with a default classification, perform approximation improvement, and then save the resulting classification for re-use in subsequent runs. Even if SEARCH_AND_SIMULATE is always applied from the saved classification, usually the classification is correct so a single run suffices to return an exact value.

Chapter 5

Reliable Symbolic Simulation

The previous two chapters presented methods for improving the performance of symbolic simulation on system level tests. Automatic approximation of node values using variable classification to guide the amount of approximation reduces both execution time and memory usage compared to using exact symbolic simulation without automatic approximation. However, these gains are still insufficient to fulfill our original goal of using symbolic simulation as a primary verification method. In order to do this, symbolic simulation must be made as easy to use as other primary methods such as directed and random testing.

Whether or not it is the most efficient verification method in terms of bugs found for the effort spent, directed testing is generally held to be the easiest to use. This makes it the most widely used verification method, so naturally it finds the majority of bugs in most designs. The following factors contribute to making directed testing easy to use.

- Directed tests can be written quickly.
- Directed tests have predictable run times.
- Directed tests provide good feedback if an error is detected.
- Each test case adds some coverage which grows predictably with time.
- There is no memory blow up.

- Run time scalability is linear in the size of the design.

The main themes in this list are predictability, feedback, and scalability. As a whole, we call these characteristics *reliability.* It is not necessary to completely replace directed testing, however, so it is not necessary to meet all the reliability requirements of directed testing. Random simulation, for example, has lower predictability, lacks good feedback, and requires more up front work than directed testing, but is still considered sufficiently reliable to be a primary verification method.

Model checking fails to meet our reliability requirements because of its poor design size scaling, unpredictable memory blowup and lack of feedback about the source of the blowup. Exact symbolic simulation is more reliable than model checking because it allows the user to control the number of variables in a test. This allows design sizes to be scalable given a fixed number of symbolic variables. Symbolic simulation with approximate values improves upon this by allowing the number of symbolic variables to be scalable if they are don't care variables. So far, however, we have seen no way to scale the number of control variables without unpredictable memory blowup occurring.

As long as BDDs are being used to represent values, memory blowup cannot be eliminated. It is sufficient, however, for the simulator either to continue in a degraded mode after blowup occurs or blowup in such a way that some coverage proportional to execution time is generated. This behavior is called *graceful degradation.* Graceful degradation of control variable scaling is a sufficient condition to meet the reliability requirements we are looking for.

This thesis addresses this problem using *satisfiability (SAT)* solving techniques. Instead of creating BDD variables, each discovered care variable will be case split such that two runs are performed, one for each setting of the discovered care variable to the constants **0** and **1**. The case splitting is accomplished using the *Davis-Putnam (DP)* algorithm, a well known SAT solving algorithm. The DP algorithm can prove a test to be satisfiable or unsatisfiable using only a small, fixed amount of memory, the trade-off being a run time that is potentially exponential in the number of split variables. However, this method provides coverage proportional to run time if the user terminates the run before completion and thus has the desired property of graceful degradation.

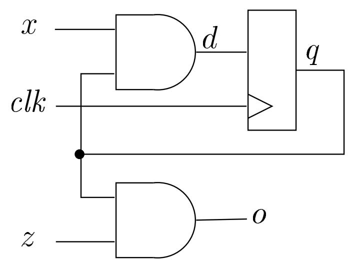
After presenting background material on SAT and the DP algorithm, this chapter will show that symbolic simulation can be cast as a SAT problem. It will then show that the DP algorithm is equivalent to performing symbolic simulation with approximate values using automatic approximation improvement. *Quasi-symbolic simulation*, a method for performing symbolic simulation based purely on this idea is presented. Finally, it will be shown that BDD-based symbolic simulation can be made reliable by using DP to handle memory overflow.

5.1 Symbolic Simulation as a SAT problem

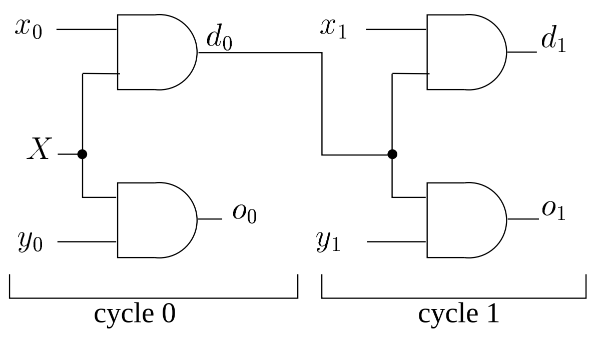
An instance of the satisfiability problem consists of a Boolean formula, normally in *conjunctive normal form (CNF)*. A CNF formula consists of a set of *clauses*, each of which is a disjunction of literals. The problem is to find an assignment to all variables in the formula such that the formula is true. SAT is NP-complete and thus, for the best known algorithms, it may require time exponential in the size of the formula to find a satisfying assignment.

It is straightforward to convert a CNF formula to a circuit. Each variable is a primary input to the circuit. Each clause corresponds to an OR gate with each input being driven by either a primary input or its complement. All OR gate outputs are ANDed together to form a single primary output. Evaluation of the formula is done by applying different symbolic variables to each primary input and symbolically simulating. If the result is satisfiable, then the formula is satisfiable.

It is also possible to convert an instance of symbolic simulation to a SAT problem. Consider a combinational circuit and test. The circuit consists of combinational gates. The test specifies a value, which can be either a constant or a symbolic variable, for each primary input. The test also specifies some relation to be checked over the outputs that produces either a PASS or FAIL indication. This check also can be implemented in gates. The goal of symbolic simulation of this test is to find a variable assignment for which FAIL is asserted. This is a satisfiability problem if we interpret FAIL to mean satisfiable. Since it is possible to convert one to the other and vice



(a) Sequential Circuit



(b) Unrolled Combinational Circuit

Figure 5.1: Unrolled sequential circuit example

versa, then symbolic simulation of a test case and SAT are equivalent problems.[1]

A sequential circuit with a test case that requires multiple cycles is also an instance of a SAT problem. This is demonstrated by treating a sequential circuit simulated for N time steps as a combinational circuit that has been replicated N times. For each primary input and output of the sequential circuit, there are N primary inputs and outputs in the unrolled circuit, one for each time step. State holding elements, such as registers, are eliminated by connecting the input to the register in one cycle to the output of the register in the next cycle. Figure 5.1 illustrates a sequential circuit and the equivalent combinational circuit unrolled for two time steps. Note that the initial state of the register is assumed to be X.

Most research in SAT solving algorithms focuses on CNF formulas. Algorithms that work with CNF formulas are called *clausal* SAT solvers. Symbolic simulation formulated as a SAT problem is not in clausal form. To apply clausal SAT solving algorithms to symbolic simulation, it is necessary to convert the symbolic simulation problem to a clausal form. There are two ways to do this. First, the circuit can be flattened to a two level representation directly. However, this can cause an exponential blowup in the number of gates in the circuit. For large circuits, this is usually too costly.

[1]This does not imply that all uses of symbolic simulation are instances of SAT problems. Image computation is one example of an application of symbolic simulation which is not a SAT problem.

In the second method, a new variable is created for each signal in the circuit [55]. Given that each gate is connected only to a set of input signals and an output signal, a set of clauses can be generated that constrains the input and output signals to be legal values. For example, for the gate $Z = X \wedge Y$ in the circuit, we could create symbolic variables x, y, and z and the set of clauses $\{(\neg x, \neg y, z), (x, \neg z), (y, \neg z)\}$ which constrain the inputs and output to legal combinations of values. The conjunction of the set of clauses for all gates and the test case results in a clause database that is satisfiable if and only if the circuit and test case are satisfiable.

The number of variables and clauses in this database is proportional to the number of gates in the circuit. For large circuits, this can lead to thousands of variables in the database. A clausal SAT solver has the freedom to set these variables any way it chooses. Since most combinations of variables are illegal, this can lead to a lot of wasted time searching invalid state space. For circuit properties that can be proved unsatisfiable by setting some internal signal to a particular value, clausal SAT solvers tuned particularly for circuit-like structures can be very effective [61]. We are interested in verification methods that scale to design sizes of millions of gates and tests of hundreds of cycles. Unrolling a circuit this large may lead to a database that does not fit in available memory, meaning that clausal SAT solving methods do not meet our scalability requirements.

Symbolic simulation can be viewed as a type of non-clausal SAT solving. Non-clausal SAT solving algorithms have been studied, with a variety of techniques being proposed [66, 35]. Non-clausal SAT solving has some similarities and differences with clausal SAT solving. In both cases, case splitting is the primary method of searching the state space. The difference is in how formulas are evaluated as a result of the variable settings decided upon by the search algorithm. The most widely used complete search algorithm for SAT solving is a variant of the *Davis-Putnam (DP)* algorithm. Before discussing how SAT solving is used in symbolic simulation, the DP algorithm will be presented.

Algorithm 9 DPLL(S)

```
1:  $S' \leftarrow unit\_propagate(S)$
2:  $truth \leftarrow evaluate(S')$;
3:  **if** $truth = TRUE$ **then**
4:      **return** $TRUE$**;**
5:  **else if** $truth = FALSE$ **then**
6:      **return** $FALSE$**;**
7:  **else**
8:      $l \leftarrow select\_split\_var(S')$;
9:      $struth \leftarrow \text{DPLL}(S'[l \leftarrow TRUE])$;
10:     **if** $struth = FALSE$ **then**
11:         $struth \leftarrow \text{DPLL}(S'[l \leftarrow FALSE])$;
12:     **end if**
13:     **return** $struth$**;**
14: **end if**
```

5.1.1 The Davis-Putnam Algorithm

The DP algorithm is a complete algorithm based on searching in the space of partial assignments. The most widely used variant of this algorithm is due to Davis, Putnam, Logemann, and Loveland [25] and is called DPLL. A partial assignment allows a variable to be assigned to TRUE, FALSE, or to be unassigned. The search uses case splitting and backtracking to search the entire space of assignments. Algorithm 9 lists the DPLL algorithm including some heuristics designed to speed up the search.

DPLL is passed the set of current variable assignments S which specifies for each variable, whether it is TRUE, FALSE, or unassigned; the initial assignment has all variables unassigned. The *evaluate*() function evaluates the formula for a given variable assignment. If *evaluate*() returns TRUE or FALSE, then DPLL immediately returns the truth value (lines 2–6).

If the truth value using the current variable assignment set is unknown, the variable assignment is refined by splitting it into two cases. A variable is chosen to split on using heuristics that try to minimize the number of case splits. The function *select_split_var*() (line 8) selects this variable. DPLL is recursively called with the variable assignment modified to set the selected variable to FALSE. This recursive call must return either TRUE or FALSE. If TRUE is returned, a satisfying partial assignment has been found and the function can return at this point (lines 9 – 10). If FALSE, then the other variable assignment is recursively checked (line 11). If both cases yield a FALSE result, then this subtree of the search is unsatisfiable and FALSE is returned, otherwise TRUE will be returned (line 13).

A standard optimization is a procedure called *unit propagation* (line 1). A *unit clause* is a clause containing a single unassigned literal. In order for the formula to be TRUE, the literals in all unit clauses must be made TRUE. After setting all unit literals to TRUE, it is possible that more unit clauses will be created. The *unit_propagate*() function iteratively sets unit literals to TRUE and re-evaluates the formula until no unit clauses remain. The *unit_propagate*() function returns a modified variable assignment that has all variables that were unit propagated set to TRUE or FALSE as necessary.

5.1.2 Quasi-Symbolic Simulation

Symbolic simulation with approximate values can be viewed as a non-clausal SAT solving algorithm. A partial assignment is a variable classification; assigned variables are care variables and unassigned variables are don't cares, the values TRUE and FALSE are mapped to the constants **1** and **0**. Evaluating a formula by simulating it with the Simple Approximation Rule will produce a result that is directly equivalent to the evaluation of a formula by the SAT solver. A result of **1** or **0** maps to TRUE or FALSE respectively, and an approximate result indicates an unknown satisfiability result.

Ignoring unit propagation for the moment, the DP algorithm can be viewed as an approximation improvement algorithm. In DP, the goal is to find the best variable to split on. In symbolic simulation, the goal is to find the best variable to mark as a BDD variable. The *select_split_var*() function in DPLL can be implemented using the Simple Variable Selection heuristic (Algorithm 7 on page 64). Since the selected variable is always split, meaning that it is set to a constant, BDDs larger than a single node will never be created. At the same time, DP is complete and, thus, maintains the completeness of symbolic simulation. Consequently, using DP to perform approximation improvement produces the same result as symbolic simulation with BDDs, *but with no memory blowup!*

The tradeoff is that the number of runs required to prove a test case unsatisfiable may be exponential in the number of care variables. However, each of these runs is a scalar run and so completes quickly. For comparison, the SEARCH_AND_SIMULATE algorithm generally requires a number of runs that is linear in the number of care variables, although the runs in simulate mode may run much slower than the scalar valued runs used in search mode.

At the same time, DP uses depth first searching. Each leaf node in the search will be found in a number of runs that is linear in the number of split variables. Leaf nodes will be found at a rate that is proportional to the number of runs, which is proportional to the total simulation time. Each leaf node is a complete scalar test and so contributes some amount of coverage. Consequently, coverage increases linearly with simulation time using DP-based approximation improvement. This means that, even if the DP algorithm is aborted early, some amount of coverage is generated and this amount is proportional to the amount of simulation time. This is exactly the reliability property we are looking for.

Symbolic simulation with approximate values using DP as the approximation improvement algorithm behaves identically to symbolic simulation using BDDs. However, it uses only scalar values internally and has different reliability characteristics. Therefore, to distinguish it from BDD-based symbolic simulation, we call this method *quasi-symbolic simulation.*

5.1.3 Quasi-Symbolic Simulation with Unit Propagation

Unit propagation is a necessary optimization when dealing with clausal formulas. In most cases there are many dependencies between variables. When a variable is case split by setting it to a constant, unit propagation finds those variables dependent on the case split variable that must be set to a particular constant in order to satisfy the formula. This reduces the size of the search tree by eliminating branches for variables that are unit propagated. For example, internal signal values in a circuit are highly dependent on fan-in and fan-out signal values. If the input to an AND gate is set to **0**, then the output must be set to **0** in order for the circuit state to be legal. Setting the output to **0** may force other signals to be fixed values. Thus, unit propagation is a necessary optimization for solving the satisfiability of circuits by conversion to clausal form.

Unit propagation does not provide any advantage when all split variables are independent. This is generally the case in symbolic system simulation. For example, in multiplexor type circuits, there are no dependencies between the control variables that select different data values. Since these circuit types are common, the value of unit propagation is not clear in symbolic system simulation. In fact, if the user writes a test case perfectly, all control variables are independent and unit propagation is not useful. However, if a bug is present due either to a test case error or circuit error, unit propagation can, in certain cases, greatly reduce the amount of searching required to find the bug. For example, if a bug only occurs when certain symbolic values are set to particular values, then often when one of these variables is set to a constant, the rest can be unit propagated. In fact, this is more likely to occur with less probable bugs. Thus, unit propagation can be beneficial when expecting bugs to occur.

The description of the DPLL algorithm in Section 5.1.1 described unit propagation in terms of clausal formulas. Since quasi-symbolic simulation is an implementation of the Davis-Putnam algorithm over non-clausal formulas, a method of performing unit propagation on non-clausal formulas is needed. Dalal [23] describes such a method using additional data structures for each circuit node value. This algorithm is shown in Algorithm 10.

Each node in the circuit has two sets associated with it: a *C-set* and *D-set*. These sets contain literals that list the unit propagation candidates for that node. C-sets and D-sets are computed for each node based on the C-sets and D-sets of the fan-in nodes and the Boolean operation of the node. The C-set for the PASS/FAIL output of the simulator specifies which literals can be unit propagated. A non-empty D-set at the PASS/FAIL output is an immediate indication of satisfiability.

The C-set of a node value is the set of literals that are conjoined with this value. If a literal is conjoined with a value, then when the literal is **0**, the node value must

Algorithm 10 DALAL'S ALGORITHM(op,f,g)

```
1: if $op = new$ then
2:     $l = create\_literal()$;
3:     $C = \{l\}$, $D = \{l\}$;
4: else if $op = f \wedge g$ then
5:     $C \leftarrow C_f \cup C_g$, $D \leftarrow D_f \cap D_g$;
6: else if $op = f \vee g$ then
7:     $C \leftarrow C_f \cap C_g$, $D \leftarrow D_f \cup D_g$;
8: else if $op = \neg f$ then
9:     $C \leftarrow \{\neg l | l \in D_f\}$, $D \leftarrow \{\neg l | l \in C_f\}$;
10: end if
11: if $l \in C \wedge \neg l \in C$ then
12:     $val(op) \leftarrow \mathbf{0}$;
13: else if $l \in D \wedge \neg l \in D$ then
14:     $val(op) \leftarrow \mathbf{1}$;
15: else
16:     $val(op) \leftarrow X$;
17: end if
```

be **0**. When the literal is **1**, knowing nothing else about this node value, we must assume that it is unknown. Thus, a value with C-set $\{l\}$ represents the value $X \wedge l$. D-sets list literals disjoined with a value. A value with D-set {l} represents the value $X \vee l$. If a value has both a C-set and D-set with the same literal, then that value must be equal to the literal. This is the only case in which a value can have both a non-empty C-set and a non-empty D-set.

The inputs to Dalal's Algorithm are an operation and up to two input values. At a primary input, the only possible operation is to create a new variable. This is done by creating the new variable and then assigning this literal to the C-set and D-set of the resulting value (lines 1–3). If the operation is the AND or OR Boolean functions, the C-set and D-set are computed as functions of the input C/D-sets. For the Boolean AND function the C-set is the union of the input C-sets and the D-set is the intersection of the input D-sets (lines 4–7). This is easy to see by computing the AND of $X \wedge a$ and $X \wedge b$ which have C-sets {a} and {b} respectively.

$$V = (X \wedge a) \wedge (X \wedge b)$$

$$\begin{aligned} V &= X \wedge a \wedge X \wedge b \\ V &= X \wedge a \wedge b \end{aligned}$$

Thus, the resulting value has C-set $\{a, b\}$. Negating a value with C-sets and D-sets follows DeMorgan's rule; the literals in the C-sets and D-sets are swapped and each literal is negated (lines 8–9).

The computed C-set or D-set may contain a literal and its complement after performing a Boolean operation. Such as set is called *basic inconsistent*. Values with basic inconsistent sets can be replaced by constants. If a C-set is basic inconsistent, the value is equal to **0** since $l \wedge \neg l = \mathbf{0}$ and if a D-set is basic inconsistent, the value is equal to **1**. Basic inconsistent sets are detected and handled by lines 11–17 of the algorithm.

5.1.4 Reactive Tests

There is one more issue that must be dealt with in using DP-based symbolic simulation that does not occur with BDD-based symbolic simulation. The addition of unit propagation to the basic DP algorithm causes reactive tests to become incomplete even if the set of underlying binary tests is complete. A reactive test may inject a request and then wait for the circuit to respond before asserting *stop* to the simulator. The amount of time the simulator has to wait may be unbounded. Reactive tests can be made complete by making three changes to the DPLL algorithm. The complete quasi-symbolic simulation algorithm with these changes is shown in Algorithm 11.

First, the BASIC_SIMULATION_LOOP (Algorithm 1 on page 15) is used as the DPLL *evaluate*() function instead of the SYMBOLIC_SIMULATION_LOOP algorithm (line 1). The difference between these algorithms is that the BASIC_SIMULATION_LOOP stops simulation at the first time step in which *stop* is non-zero while the SYMBOLIC_SIMULATION_LOOP waits until the last possible time step to stop. Next, DPLL is modified to case split on *stop* if it is approximate before case splitting on *fail* (lines 2,11–13). This is to prove that the simulation really stops at this time step

Algorithm 11 QUASI_SYMBOLIC_SIMULATION(S)

```
1: $\{stop, fail\} \leftarrow$ BASIC_SIMULATION_LOOP($S$);
2: if $stop = \mathbf{1}$ then
3:    if $fail = \mathbf{1}$ then
4:       return $TRUE$;
5:    else if $fail = \mathbf{0}$ then
6:       return $FALSE$;
7:    else
8:       $S' \leftarrow unit\_propagate(S)$
9:       $l \leftarrow select\_split\_var(fail, S')$;
10:   end if
11: else
12:    $S' \leftarrow S$
13:    $l \leftarrow select\_split\_var(stop, S')$;
14: end if
15: $struth \leftarrow DPLL(S'[l \leftarrow \mathbf{1}])$;
16: if $struth = FALSE$ then
17:    $struth \leftarrow DPLL(S'[l \leftarrow \mathbf{0}])$;
18: end if
19: return $struth$;
```

since *stop* may be asserted prematurely if it is an unknown value. Once *stop* is proven to be satisfiable ($stop = \mathbf{1}$) at a particular time step, then *fail* can be split if necessary (lines 3–10). To support this, *select_split_var*() is modified to select the variable to split on from either *stop* or *fail* as necessary (lines 9,13).

The final modification is to only allow unit propagation when splitting on *fail* (line 8) to prevent the algorithm from cutting off potential valid stop times. To illustrate this, consider the following example. A four bit counter is initially loaded with a value and then decremented. The counter asserts *stop* when the count reaches zero. For any binary vector loaded into the counter, *stop* will always eventually be asserted. Thus, if a test consists of loading the counter and waiting for *stop* to be asserted, the test case will always eventually terminate.

Now consider a symbolic vector representing the set of all possible count values being loaded into the counter. A portion of the unrolled circuit with the *stop* generation logic for each time step is shown in Figure 5.2. The solid lines indicate the signals

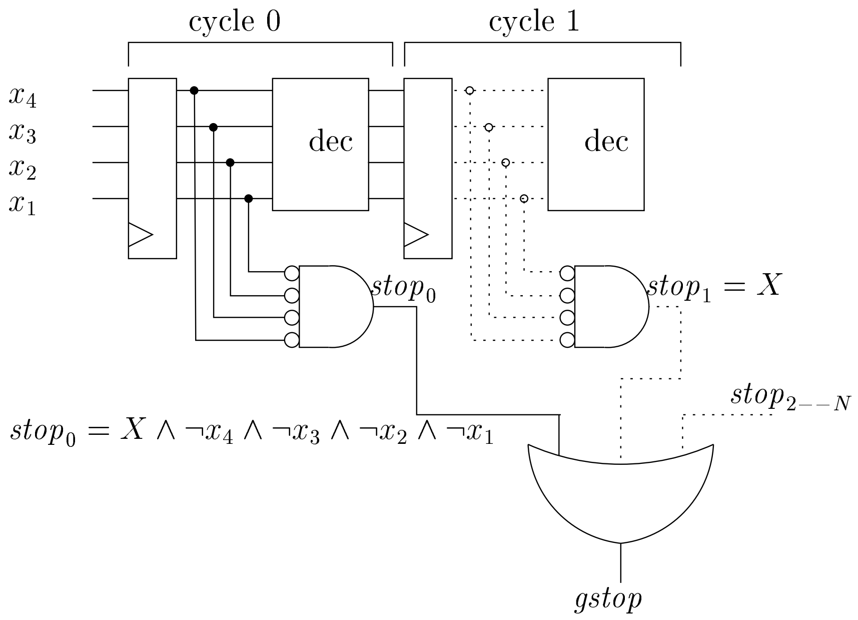


Figure 5.2: Unrolled example of decrementing counter with reactive *stop*

evaluated after simulating for one cycle. The dotted lines indicate values generated after simulating for two or more cycles. Exact BDD-based symbolic simulation would stop when *gstop*, which is the OR of *stop* for all cycles, is **1**. The quasi-symbolic simulation algorithm, however, will stop at the first time step at which *stop* becomes non-zero. For this example, *stop* becomes non-zero in time step 0 as shown in the figure as the value of $stop_0$. The approximation rule causes the value of $stop_0$ to be X with the C-set $\{\neg x_4, \neg x_3, \neg x_2, \neg x_1\}$.

For the next run, all variables would be set to **0** as indicated by the stop condition C-set for $stop_0$ if unit propagation was allowed. In the next evaluation $stop_0$ is **1** since a value of all zeroes is loaded into the counter. This means the test case has been proved to actually stop in the first cycle. The algorithm now tries to backtrack. Since all variables were unit propagated, no case splitting will be performed. The simulator will think that all cases have been explored. Therefore, the search would terminate immediately and would have tested only the case in which the test stops after one cycle. But, clearly, the test can stop at all lengths up to the maximum counter value since the count was specified to be a symbolic vector representing all possible counts.

Since the test case should stop eventually for all case split values, by not allowing unit propagation of *stop*, the entire search tree affecting the time step at which the simulation stops will be enumerated. Unit propagation can still be done when splitting on the *fail* condition. At the point that this is being split, the current time step has been proven to be a valid stopping point. Thus, unit propagation of *fail* cannot prevent the simulator from stopping in other time steps.

5.1.5 Analysis of Quasi-Symbolic Simulation

In system-level tests, control variables specify independent parameters such as request types, delays between requests, and addresses. The lack of dependencies between control variables means that little or no unit propagation occurs when performing quasi-symbolic simulation on these cases. Consequently, $\Omega(2^N)$ runs are required to prove unsatisfiability for these cases where N is the number of control variables. For satisfiable test cases, quasi-symbolic simulation can prove satisfiability of a test case (find a bug) with the number of runs being linear in the number of variables that must be made constant to sensitize the error in the best case; each of these runs requires constant time. In the worst case, the number of runs required to prove satisfiability is $O(2^N)$ for N control variables.

The run time for BDD-based symbolic simulation has been observed to be insensitive to whether the test is satisfiable or not, even when using approximate values. If the run time is exponential in the number of symbolic control variables for a given circuit and test case using BDDs, this run time is independent of whether bugs are present and how hard or easy the bugs may be to sensitize. Thus, BDD-based symbolic simulation usually performs better on unsatisfiable test cases while quasi-symbolic simulation can perform better on satisfiable test cases, especially for simple bugs.

This section presents the results of two experiments using quasi-symbolic simulation to illustrate the difference between BDD-based and quasi-symbolic simulation

ctrl.	quasi-symbolic		exact BDD		approx. BDD	
vars	evals	time	BDD nodes	time	BDD nodes	time
6	57	191s	313K	225s	6.2K	10.6s
9	57	193s	526K	2214s	22K	80s
11	59	199s	-	-	48K	316s
19	68	432s	-	-	197K	1277s
28	72	430s	-	-	-	-
37	76	457s	-	-	-	-
46	79	486s	-	-	-	-
55	82	501s	-	-	-	-

Table 5.1: NI bug test case results for quasi-symbolic, exact BDD, and approximate BDD-based symbolic simulation

on both satisfiable and unsatisfiable test cases. The first experiment compares quasi-symbolic simulation to BDD-based symbolic simulation on the NI design (see Section 3.5.5 on page 53). The second experiment uses a different design to demonstrate the reliability of quasi-symbolic simulation in developing a test case to show that quasi-symbolic simulation can find simple bugs quickly.

Experiment 1: A Satisfiable NI Test case

This experiment compares the execution time of quasi-symbolic simulation and BDD-based symbolic simulation with approximate values on a test case that detects a bug. One of the bugs found in this design causes certain words of a packet to be corrupted whenever the frame length of the target packet is between 76 and 79 bytes. This is independent of any other variable in the test. The target frame length is specified by eight symbolic variables allowing packet lengths to range from 0 to 255 bytes. The probability of this bug occurring in a binary directed or random test, therefore, is one in 64. The variables specifying the frame length are sequential in the variable order and appear as variables 51–59 in the order.

The experiment consisted of making the frame length variables symbolic and then performing a number of runs, each with a larger number of additional control variables set to symbolic values. Non-symbolic control variables were set to **0**. Table 5.1 lists the results of performing these runs.[2] The first column in the table lists the total number of symbolic control variables for each run. For quasi-symbolic simulation, the number of evaluations (simulation runs) and time required before the bug was found are given. For BDD-based symbolic simulation, the number of BDD nodes created and the total simulation time are given, including any time required for approximation improvement. Results are given for both exact values and approximate values using Approximation Rule 1. For the runs using approximate BDD-based values, a variable classification was supplied by the user and no improvement was necessary to produce an exact result for any of the runs.

The results show that exact BDD-based symbolic simulation blows up quickly as the number of symbolic control variables is increased. BDD-based symbolic simulation with approximate values fares better, but still blows up quickly. Quasi-symbolic simulation finds the bug with little variance in execution time as the number of symbolic control variables is increased.

We can characterize the efficiency of finding a given bug using quasi-symbolic simulation from analyzing this bug. The case splitting required to find this bug has the following pattern. First, some number of control variables ordered ahead of the frame length variables are split before the simulator starts splitting the frame length variables. The simulator only needs to explore one path for these variables since they don't affect the sensitization of the bug. The simulator then has to enumerate 32 cases of the six frame length variables before finally finding the failing branch. It then case splits an additional 12 data variables along a single branch before producing an exact satisfiable value. The variance in the number of evaluations for the test cases with different numbers of symbolic control variables is due solely to the number of control variables ordered ahead of the frame length variables that need to be split. Since these only need to be split once, the number of evaluations is linear in the number of symbolic control variables.

If the frame length variables are ordered ahead of all other control variables, then the case splitting of other control variables would be eliminated, minimizing the number of evaluations required. On the other hand, if the frame length variables are

[2]Simulations run on a Pentium III, 800MHz with 512Mbytes of memory.

interleaved with the other control variables, then the amount of case splitting required could be exponential in the total number of control variables. This is because for each assignment to the frame length variables, for all variables interleaved with the frame length variables, all assignments to these variables must be enumerated.

Thus, performance of quasi-symbolic simulation on satisfiable test cases is a function of the number of variables that a bug is sensitive to and the distance between them in the variable order. A bug with few sensitizing variables close together will require few runs and generally will require much less simulation time than BDD-based symbolic simulation to find the same bug. A bug with many or widely spread sensitizing variables will require many runs and will require more simulation time than BDD-based symbolic simulation.

Experiment 2: MCU Test case Development

This experiment explores the performance of quasi-symbolic simulation in debugging test cases. Test case bugs are more common than hardware bugs; for every hardware bug found, there may be one or more test case bugs that must be found and debugged. From the simulation algorithm's point of view, test case bugs are no different than hardware DUT bugs. Thus, being able to reliably detect test case bugs is an important part of system-level testing.

A test case was written to find a known bug in an existing design. The design, called the MCU [78], was taped out and a bug was subsequently found that escaped detection during both simulation and bringup. This bug is considered to be an instance of a hard bug. The test written was based on a loose description of the problem. This test was run using quasi-symbolic simulation, then debugged and modified until the hardware bug was found. Each test version had eight symbolic control variables, four symbolic data variables, and between 411 and 1,334 symbolic don't care variables. The run time of each version of the test case created as the test was debugged was recorded. The results are summarized in Table 5.2.[3]

There were 51 versions of the test corresponding to 50 test case bugs found before the hardware bug was found. Each version of the test is categorized into one of four

[3]Simulations run on a Sun Ultra-5, 271MHz with 128Mbytes of memory.

case	no.	evals	time(sec.)
SAT	19	3.8	31.4
TIMEOUT	22	1.6	49.0
LEAK	1	78	863.0
UNSAT8j		1	7.1
UNSAT8k		53	392.3
UNSAT11		111	886.64
UNSAT23		111	1025.21
UNSAT28		53	420.4
UNSAT27		53	432.3
UNSAT32		37	352.0
UNSAT33		37	353.6
UNSAT34		15	143.5

Table 5.2: MCU Test Case Development Performance

types. Tests which were satisfiable due to a test case bug are labelled *SAT* bugs. Test cases that exceeded a specified simulated cycle limit are labelled *TIMEOUT* errors. The test case that found the target hardware bug is labelled *LEAK*. Test cases that were correct based on an interpretation of the bug description, but for which this interpretation was wrong resulted in the test reporting no errors. These cases are labelled *UNSATxx*. There is a much wider variance in execution time between the UNSAT cases, so each of these is listed separately. For each class, the number of test cases of that class is listed, along with the average number of evaluations and average execution time.

The results show that the satisfiable test error cases uniformly require a small number of case splits. Also, for these cases, no unsatisfiable branches were found; the first exact value produced by case splitting detected a bug in all cases. The hardware bug caused incorrect data to be transmitted through the chip. The effect of this is that a large number of symbolic don't care variables "leaked" through to a checked output. A large number of case splits were required to resolve this to an exact value. However, this represents only a single branch in the search tree. The fact that symbolic simulation could definitively rule out incorrect interpretations of an ambiguous specification was useful in finding the bug. The longer run times and wide variance of these cases shows the inefficiency of quasi-symbolic simulation in proving tests unsatisfiable compared to using BDD-based symbolic simulation.

In summary, quasi-symbolic simulation can be effective when bugs are present in either the test case or the DUT. It is potentially more scalable in the number of control variables for satisfiable test cases than BDD-based symbolic simulation. For unsatisfiable cases, in which no bug is present, quasi-symbolic simulation is generally less effective than BDD-based symbolic simulation. However, if memory blowup occurs, BDD-based symbolic simulation cannot prove anything about the design. Quasi-symbolic simulation can always produce a partial result. This means that it meets our requirement for reliability. The next section will explore ways of combining SAT and BDD-based symbolic simulation in order to improve performance on unsatisfiable cases while preserving sufficient reliability.

5.2 Combining SAT-Based and BDD-Based Symbolic Simulation Methods

The difference between SAT and approximate BDD-based symbolic simulation is in how each method handles a symbolic variable that has been identified as a care variable. In approximate BDD-based symbolic simulation, variables marked as care variables become BDD variables. In the SAT-based quasi-symbolic method, care variables are case split such that two runs are performed, one for each setting of the care variable to the constants **0** and **1**. It is possible to combine these two approaches by deciding on a variable-by-variable basis to either case split or create a BDD variable.

This strategy is implemented by modifying algorithms we have already seen. There are many ways to do this; exploring all the options is beyond the scope of this thesis. Instead, this section will explore two options with the goal of improving the reliability of unsatisfiable test cases. Both options require the same modifications to the lower level algorithms presented in this thesis. The remainder of this section will describe these necessary modifications before presenting the two high-level strategies

Algorithm 12 APPROX_APPLY_MO(f,g,$\langle op\rangle$)

```
 1: if $terminal\_case(f, g, \langle op\rangle)$ then
 2:     return $handle\_terminal\_case f, g, \langle op\rangle)$;
 3: end if
 4: if $cache\_hit(\{f, g, \langle op\rangle)\}$ then
 5:     return $cache\_lookup(\{f, g, \langle op\rangle\})$;
 6: end if
 7: $top\_var \leftarrow min(var(f), var(g))$;
 8: $f_{if}, f_{else} \leftarrow cofactor(f, top\_var)$;
 9: $g_{if}, g_{else} \leftarrow cofactor(g, top\_var)$;
10: $t_{if} \leftarrow$ APPROX_APPLY_MO$(f_{if}, g_{if}, \langle op\rangle)$;
11: $t_{else} \leftarrow$ APPROX_APPLY_MO$(f_{else}, g_{else}, \langle op\rangle)$;
12: if $t_{if} = t_{else}$ then
13:     $result \leftarrow t_{if}$;
14: else if $want\_approximate(top\_var, t_{if}, t_{else})$ then
15:     $result \leftarrow approximate(top\_var, t_{if}, t_{else})$;
16: else if $unique\_hit(top\_var, t_{if}, t_{else})$ then
17:     $result \leftarrow unique\_lookup(\{top\_var, t_{if}, t_{else}\})$;
18: else if $BDD\_no\_memory\_available()$ then
19:     $result \leftarrow X$;
20: else
21:     $result \leftarrow create\_node(top\_var, t_{if}, t_{else})$;
22:     $unique\_insert(\{top\_var, t_{if}, t_{else}\}, result)$;
23: end if
24: $cache\_insert(\{f, g, \langle op\rangle)\}, result)$;
25: return $result$;
```

for combating memory explosion.

5.2.1 Memory Overflow Handling

We want the simulator to use a fixed amount of memory to create BDD nodes. However, there is no a priori way of knowing exactly how much memory will be required based on the number of BDD variables in the test. The only way to determine this is to build BDD nodes and then detect when overflow occurs. In order to not throw away work already done, there must be a strategy for dealing with overflow when it happens.

Overflow occurs in the APPROX_APPLY algorithm when it tries to create a new BDD node and there is no memory space available. A simple way of handling this is to simply return X in this case. Thus, memory overflow is handled using another approximation rule. Algorithm 12 shows a version of APPROX_APPLY that returns an approximate value when memory overflow occurs. The only modification to this algorithm is the addition of lines 18 and 19. The BDD package is assumed to have a function which returns a flag indicating if there is sufficient memory available to create a node. Normally this is implemented by allowing the user to set a threshold for the maximum number of nodes that can be created. This function simply checks that the current node count is less than the specified threshold.

The *approximate*() function can create new BDD nodes. However, the memory available check is done after the approximation check. Thus, it may be necessary to lower the node threshold to account for these nodes. Because of the nature of approximated nodes, this is only a small percentage of the total number of BDD nodes when the number of nodes is near the maximum and usually does not cause a problem.

5.2.2 Variable Selection Heuristic

If there is insufficient memory to create a new BDD node during simulation, APPROX_APPLY_MO will return an approximate value. If the node is a don't care node, then it is still possible that the final result will be exact. If the node value for which overflow occurred was a care value, then the final result will be approximate. Since in symbolic system simulation there are many more don't care values than care values usually, we don't want the simulator to abort simply due to memory overflow. Instead, each internal value that is approximated due to overflow is marked as an overflowed value. If the final satisfiability result returned by the simulator is marked as an overflow value, the selected variable will be case split resulting in two simulations with smaller BDDs.

A heuristic is needed to choose an appropriate care variable to be split when there is insufficient memory to create a new node. This is done by modifying the variable selection heuristic (Algorithm 7 on page 64) to propagate a case split candidate instead of a re-classification candidate when overflow occurs during the creation of a node value. To facilitate this, APPROX_APPLY_MO can be modified to return an overflow indication and a case split candidate. A simple heuristic is to return *top_var* as the case split candidate.

Algorithm 13 on page 109 illustrates a modified version of the Simple Variable Selection heuristic that selects case split candidates when overflow occurs. The data structure for a value expression is modified to add a flag, called *ovflw*, indicating that the *var* field is a case split candidate due to overflow instead of a re-classification candidate. The overflow indication and case split variable returned by APPROX_APPLY_MO are specified in global variables *ovflw_flag* and *ovflw_var*.

If overflow occurred in the current node, then SIMPLE_VAR_SELECT_MO sets the overflow flag and sets the case split candidate to the overflowed variable returned by APPROX_APPLY_MO (lines 2–4). Lines 5–16 are the normal cases when a constant value is returned. Lines 17–29 are additional lines to handle the case that overflow did not occur on this node, but did on one of the input values. Overflow values have precedence over non-overflow values. If both inputs overflowed, then the lower indexed variable is chosen (lines 17–23). Otherwise, if either input overflowed, that value is chosen (lines 24–29). The remainder of the code is unmodified from the original algorithm.

5.2.3 CD-MTBDDs

When using DP-based case splitting with C-sets and D-sets to implement unit propagation, the value function for each node is extended to include a pointer to a C/D-set. BDDs and C/D-sets can be combined. C/D-sets can be implemented using approximate BDDs with an approximation rule that limits the possible BDDs to be representations of C/D-sets only.

This approximation rule lies between the Data Approximation Rule and the Linear Approximation Rule in the amount of approximation it creates. The *if* and *else* branches represent either C-sets or D-sets. In the case that the AND function is being performed and the *if* and *else* branches represent C-sets, an exact result will be returned, which is the same as that returned by the Linear Approximation Rule. If both branches are D-sets, the value X would be returned which is the same as if the Data Approximation Rule had been applied.

The *qsym* program implements this approximation rule using a different approach. Instead of implementing the approximation rule directly, terminal nodes represent C-sets and D-sets. Thus, the approximation rule is implemented in the terminal node handling functions. C-sets and D-sets are maintained in a canonical data structure and the terminal node values are pointers to these data structures. The primary advantage in using this method is that it allows the simpler C/D-set data structure to be used if DP-based case splitting only is used.[4]

5.2.4 DP Based Search and BDD Based Symbolic Simulation

Given a design and a test case, a fixed amount of memory is required to perform scalar simulation and DP-based approximation improvement. The basic idea in combining DP and BDD-based symbolic simulation is that any remaining memory can be used to create BDDs. This reduces the amount of case splitting exponentially in the number of variables that are made BDD variables. From the simulator's point of view, there are now three types of variables: care BDD variables, care case split variables, and don't care variables.

There are two basic strategies that can be used in determining which variables to case split and which to make BDD variables.

Option 1 Mark all discovered care variables as BDD variables and only re-mark them as case split variables if memory overflow occurs.

Option 2 Mark all discovered care variables as case split variables. Variables are then incrementally re-marked as BDD variables until memory overflow occurs.

In both cases, once the right number of BDD variables is found that just fits in the available memory, case splitting will occur over the remaining care variables.

[4]The actual C/D-set data structure is a trie.

Neither one of these algorithms is better than the other, but they work differently in different circumstances. Option 1 works best when the number of BDD variables that does not cause memory overflow is close to the total number of care variables. This is because the number of simulation runs that require re-marking a variable is equal to the difference between the total number of care variables and the number of BDD variables that allow the simulation to complete without overflow. Option 2 requires a number of runs equal to the number of BDD variables.

For example, if there are 36 total care variables and 32 of them can be marked as BDD variables without memory overflow, then Option 1 would require four re-markings while Option 2 would require 32. On the other hand, if the total number of care variables is much larger than the number of BDD variables that allow no overflow, then Option 2 requires fewer re-markings. If there are 100 total care variables instead of 36 for example, then Option 1 would require 68 simulation runs to re-mark variables while Option 2 would still only require 32. Further, each re-marking run using Option 1 will not produce any coverage and will take the maximum simulation time. Option 2 will produce shorter re-marking runs and each run will provide some amount of coverage which grows larger with time. Thus, Option 2 is the more reliable algorithm.

Option 1 can be implemented by using SEARCH_AND_SIMULATE as the evaluation function for DPLL. In this case, SEARCH_AND_SIMULATE is modified to return immediately if the final result is approximated due to overflow. DPLL will detect this and case split the specified variable.

Option 2 is implemented by modifying DPLL. The evaluation routine is still the basic symbolic simulation loop. Thus, DPLL gets the final result directly. As variables are case split, no BDDs will be created until a leaf node of the search is reached and backtracking occurs. Backtracking occurs one step at a time. At each step, the variable that was backtracked is marked as a BDD variable. This results in the entire sub-tree for the second case split value of each variable to be explored at one time using BDDs. Each sub-tree that is explored as each variable is backtracked adds one more BDD variable. If overflow occurs, the variable that is returned will be case split.

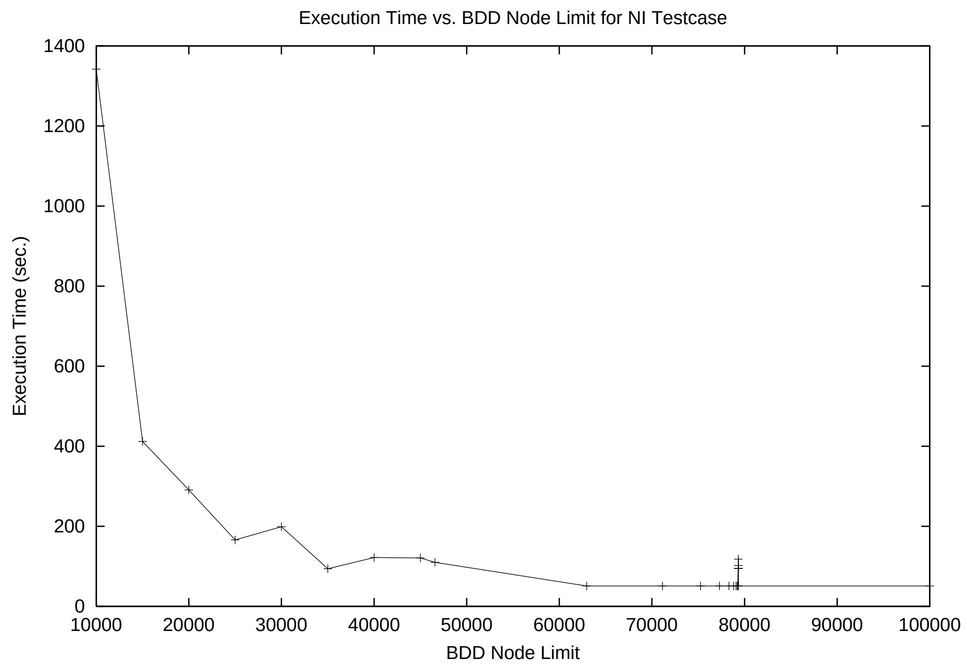


Figure 5.3: Execution Time vs. BDD Node Limit for Unsatisfiable NI Test case

5.2.5 Analysis of Combined SAT and BDD-based Symbolic Simulation

This section illustrates the effect of memory overflow handling in simulating an unsatisfiable test case. In this experiment the NI design is used with an unsatisfiable test case that has a fixed number of symbolic control variables. The BDD node limit is then varied to simulate different overflow conditions. The execution time required to prove the test unsatisfiable is plotted as a function of the BDD node limit. The goal of this experiment is to show the graceful degradation of execution time when there is insufficient memory for the test case and design being simulated.

This experiment implements Option 1 by using SEARCH_AND_SIMULATE as the simulation loop for DPLL. The simulator uses CD-MTBDDs and the variable selection selection routine modified for memory overflow as described above. The test used a user supplied variable classification which did not need improvement in the case that there was sufficient memory.

A number of runs were performed in which the BDD node limit was set to different values. The range of values was from 10,000 nodes to 200,000 nodes. The execution time required to prove the test case unsatisfiable as a function of the node limit is plotted in Figure 5.3. The maximum number of nodes needed with no overflow is 79,342 nodes. Below this number, BDD overflow occurred, but did not necessarily affect the execution time of the test.

The simulator performs a garbage collection whenever overflow occurs. For cases with node limits above 30,000 nodes, this usually resulted in a large number of nodes being freed. Consequently, case splitting only occurred for these cases if overflow occurred when computing a care value. This is the case for the small peak between 79,310 and 79,341 nodes in the graph. The randomness of whether overflow occurs on a care or don't care value causes the number of case splits required to vary randomly as a function of the BDD node limit. This accounts for all of the non-monotonicity in execution time seen in the graph.

Below a limit of 30,000 nodes, garbage collection frees few if any nodes. This causes the amount of case splitting to increase inversely proportional to the node limit and the number of BDD overflows to increase hyper-exponentially as the node limit is reduced below 30,000 nodes. Since each overflow causes a garbage collection to occur, the execution time comes to be dominated by the garbage collection time. This means that this algorithm will be inefficient for those cases in which the node limit is small compared to the node limit required for no overflow to occur. For contrast, quasi-symbolic simulation requires 424 seconds and 60 evaluations to prove this test case unsatisfiable using only 8,248 BDD nodes. This compares to 1,342 seconds using BDDs with a node limit of 10,000 nodes.

5.3 Related Work

The idea in semi-formal verification is to make symbolic methods reliable. While symbolic simulation tries to increase functional coverage, many other semi-formal methods try to increase state coverage using automated methods. Typically, a three step process is used. First, state machines are either extracted from the design [34, 65, 83] or are created manually [42], then symbolic methods are used to find all reachable states and, finally, tests that visit each state or transition are generated. Another semi-formal technique starts with existing binary tests and attempts to "expand" them to increase state or transition coverage [44, 33]. These methods improve reliability because reachability and test generation can be done incrementally. The reliance on state exploration, however, limits scalability and does not provide sufficient reliability to be a primary verification method, although they may work well for finding bugs.

Innologic's commercial symbolic simulator has the ability to detect and handle BDD overflow. Apparently, their method is to abort the simulation as soon as overflow is detected on an internal node value. Their method then randomly sets one of the input symbolic values to a constant value in order to reduce BDD sizes for the next run. It is not clear that there is any attempt to make the method complete by performing two simulations for each variable set to a constant. Even if this were the case, since their method does not distinguish between care and don't care variables, a lot of don't care variables may need to be set to constants before memory usage is sufficiently reduced.

The application of SAT methods to verification has become common in recent years. The methods described in this thesis directly incorporate the SAT decision procedure into the simulator. Other SAT-based verification 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) [6]. BMC requires the circuit to be unrolled for however many cycles are being simulated, using memory proportional to the product of design size and the number of cycles being unrolled. Performing SAT solving directly in the simulator eliminates the need to perform explicit unrolling and, thus, is more scalable.

Sequential ATPG [20] is similar to symbolic simulation in that it is trying to produce a particular value on a particular output in a sequential circuit. Sequential ATPG as a model checking algorithm has been studied [8]. Sequential ATPG has been shown to be inefficient when there are few justifying states and may timeout trying to justify a state and so does not meet our reliability goals.

Combining BDDs and SAT has also been a topic that has gained a lot of attention recently. Combining BDDs and SAT is most widely used in combinational equivalence checking. Gupta and Ashar [40] describe a method that initially uses SAT solving to compare nodes and then switches to using BDDs. The basic strategy is similar to Option 2 described in the previous section. Instead of deciding on a variable-by-variable basis whether to case split or use BDDs, their method decides on a node-by-node basis. Nodes are partitioned into two halves, BDDs are used on one half and SAT on the other. Reda and Salem [72] describe a similar method but use BDDs and SAT on opposite sides of the partition than Gupta and Ashar. It is not clear if these methods can be extended to work on sequential circuits.

5.4 Summary

The inability to produce useful results when memory overflow occurs is the primary reliability feature lacking in BDD-based symbolic simulation. This chapter addressed this problem using SAT-based methods. SAT-based methods were shown to be an instance of using symbolic simulation with approximate values and approximation improvement. The primary difference between SAT-based symbolic simulation and BDD-based symbolic simulation with approximate values is in how each method handles variables that have been re-classified as care variables. BDD-based symbolic simulation marks the variable as a BDD variable and causes exact BDDs to be built. SAT-based symbolic simulation case splits the variable, requiring multiple runs to produce a result.

A symbolic simulation method called quasi-symbolic simulation which uses purely SAT-based approximation improvement was described. This method was shown to be effective when there were bugs present in the design. It was also shown to be more scalable and reliable than BDD-based symbolic simulation for these cases. However, it was shown to be inefficient compared to BDD-based symbolic simulation on unsatisfiable test cases.

Combining SAT-based and BDD-based approximation improvement resulted in improved performance on unsatisfiable cases compared to quasi-symbolic simulation. It also handles memory overflow in a reliable way. That is, if overflow occurs, it allows resorting to case splitting to maintain completeness of the simulation. This also allows a partial result to be returned if memory overflow occurs and the coverage obtained is proportional to the execution time. Thus, this method potentially meets our reliability requirement.

There are tradeoffs involved in combining SAT and BDD-based symbolic simulation. The amount of memory available versus the amount needed dictates the best heuristics to use to get the best efficiency. There is still a lot of room left for research in combining SAT and BDD-based methods to achieve optimal results.

Algorithm 13 SIMPLE_VAR_SELECT_MO(f,g,node)

```
1: $node.ovflw \leftarrow false$;
2: if $ovflw\_flag$ then
3:     $node.ovflw \leftarrow true$;
4:     $node.var \leftarrow ovflw\_var$;
5: else if $is\_non\_controlling(g.val)$ then
6:     $node.var \leftarrow f.var$;
7: else if $is\_non\_controlling(f.val)$ then
8:     $node.var \leftarrow g.var$;
9: else if $is\_controlling(f.val)$ then
10:    $node.var \leftarrow f.var$;
11: else if $is\_controlling(g.val)$ then
12:    $node.var \leftarrow g.var$;
13: else if $g.var = \{\}$ then
14:    $node.var \leftarrow f.var$;
15: else if $f.var = \{\}$ then
16:    $node.var \leftarrow g.var$;
17: else if $f.ovflw \wedge g.ovflw$ then
18:    $node.ovflw \leftarrow true$;
19:    if $g.var < f.var$ then
20:        $node.var \leftarrow g.var$;
21:    else
22:        $node.var \leftarrow f.var$;
23:    end if
24: else if $f.ovflw$ then
25:    $node.ovflw \leftarrow true$;
26:    $node.var \leftarrow f.var$;
27: else if $g.ovflw$ then
28:    $node.ovflw \leftarrow true$;
29:    $node.var \leftarrow g.var$;
30: else if $\neg is\_care(g.var) \wedge is\_care(f.var)$ then
31:    $node.var \leftarrow g.var$;
32: else if $\neg is\_care(g.var) \wedge g.var < f.var)$ then
33:    $node.var \leftarrow g.var$;
34: else
35:    $node.var \leftarrow f.var$;
36: end if
37: return $node$;
```

Chapter 6

Conclusions

The verification process normally consists of two phases. The first phase, referred to as the primary verification, systematically tests all the functionality in the design. The second phase augments the primary verification using different methods with the goal of finding bugs that were missed in the first phase. The primary methods of finding bugs today are directed and random testing. Most conventional approaches to improving verification try to find better ways of augmenting random and directed testing. This thesis has argued that this does not provide sufficient gain to significantly improve the current verification bottleneck. Instead, this thesis has taken the approach of trying to find a better method than directed or random testing for primary verification.

Evaluating the success or failure of a primary verification method uses different criteria than an augmenting method. The goal of methods that augment the primary verification is to find bugs that were missed. Therefore, it is usually necessary to show that a proposed augmenting method actually found a bug on a real design for the method to be considered successful. For primary methods, however, the ability to find bugs is a given. The success of a primary verification method is measured by how much effort is required to execute the test plan in terms of both human and computer time. It is assumed that successfully passing all tests in the plan entails finding and fixing all bugs.

The effort in executing a test plan includes the time to write the test, CPU time to run the test, test debug time, and any time required to debug hardware bugs. Symbolic simulation provides speedups over directed and random testing in both test writing and debugging effort because of the symbolic encoding of multiple binary tests into a single symbolic test and the deterministic nature of symbolic tests. The success of a symbolic simulation tool, then, is measured by how much CPU time speedup it has over binary simulation for running an equivalent set of tests. This speedup can vary from a billion or more in the ideal case to zero in the case that the symbolic simulation blows up due to memory limitations. Thus, a more important criteria is the reliability of this speedup. We would rather have a lower best case speedup if the symbolic simulator is no slower than binary simulation in the worst case.

Thus, the primary focus of this thesis has been on improving the reliability of symbolic simulation. This is done using three primary ideas. First, the simulator is given the ability to automatically approximate values on a node-by-node and cycle-by-cycle basis. Approximation minimizes BDD sizes, increasing speed and lowering the probability of BDD overflow. Second, variable classification is an effective heuristic for determining which node values during simulation are don't care values. Values that the simulator knows to be don't care values based on the variable classification are set to X. Third, SAT-based case splitting can mitigate BDD overflow such that some useful coverage is always obtained.

This produced a symbolic simulation method that had reliability characteristics similar to directed and random testing. These reliability characteristics are not exactly the same as random and directed testing. However, random and directed testing do not have the same reliability characteristics either, yet they are both considered to be primary verification methods capable of finding all bugs. Directed and random testing are considered to complement each other very well. Using the methods introduced in this thesis, symbolic simulation with approximate values potentially can take its place as a third primary verification method that complements both random and directed testing.

6.1 Future Work

There is much work that can be done to follow-on from this thesis. There are three main areas that need to be addressed.

- Improvements to SAT and BDD-based symbolic simulation with approximate values.
- Coverage analysis of symbolic simulation.
- Methodology improvements to support symbolic simulation.

6.1.1 Improvements to Symbolic Simulation with Approximate Values

The first item simply is to carry forward the work of this thesis. Many of the algorithms presented in this thesis are heuristics. There are certainly many variants of these heuristics that need to be explored before knowing which is best.

In particular, the variable selection heuristic is very simplistic. The variable selection heuristic is of primary importance when bugs are present to minimize the amount of work needed to zero in on the bug.

Secondly, only a few options for combining SAT and BDD-based symbolic simulation were explored. There is certainly more work that could be done here. It would also be good to explore incorporating other SAT optimizations that are standard optimizations in off-the-shelf complete SAT solvers. Exploring how to incorporate local search methods may also be fruitful.

6.1.2 Coverage Analysis

Symbolic simulation using classified variables opens up a range of coverage analysis possibilities that is not possible with conventional binary simulation. In particular, variable classification allows the detection of necessary conditions for bug sensitization.

In the system-level tests we are considering, a bug is detected when incorrect data appears at a checked output. This incorrect data is often a don't care value. Assume the correct value was injected as a data value. If a bug occurs, then it must be the case that, at some point, the don't care value was propagated through the circuit while the care value was blocked. Thus, data being blocked is a necessary condition for a bug to occur.

Suppose the test passed, meaning that the data was correctly propagated through the circuit. Suppose further that the data was blocked at some node in the circuit. This implies that there is some other path in the circuit for this data to follow and that this path is not being exercised by this test. In other words, there is a lack of coverage.

Conventional metrics such as line coverage (including observability based coverage [31]) and state coverage measure neither necessary nor sufficient conditions for bugs to occur. Therefore, lack of coverage using these metrics can only be determined after all tests have been written and passed, if at all. Measuring coverage of necessary bug conditions can be done on each test case and, so, can find potential bugs earlier and more completely.

6.1.3 Methodology Improvements

Experience using symbolic simulation with approximate values using the experimental implementation *qsym* showed that this methodology required more work than normal directed and random testing to setup and debug. This resulted in this method being less efficient when finding bugs.

However, the primary culprit was the primitive user interface to the tool. *qsym* required a gate level netlist instead of RTL simply because this greatly simplified the coding of the tool. This made the turnaround time when changing RTL very long. When a bug occurred, the primary debugging tool was gdb. Binary simulation enjoys a plethora of support tools such as test bench automation tools, waveform viewers, graphical debuggers, coverage analysis tools, and hardware accelerators. There is also a lot of expertise and accumulated wisdom on the best methodologies for maximizing the efficiency of directed and random testing. There is very little accumulated wisdom on using symbolic simulation, especially for doing system-level verification.

Symbolic simulation introduces problems in all of these peripheral areas. What is the best way to write tests incorporating symbolic values? How are symbolic values represented on a waveform viewer? What is the best way to debug a symbolic test?

It may be that during the early stages of verification when there are many bugs, it is the process of writing and debugging a test that is the bottleneck in verification. The actual verification engine may not be the bottleneck at all, no matter whether it is a binary or symbolic engine. Thus, for symbolic simulation to succeed as a primary verification method, it is vital to address these methodological issues.

6.2 Conclusion

This thesis has taken a different approach to solving the verification bottleneck. Instead of trying to augment existing binary simulation methods, the goal has been to try to devise a better primary verification method. It is not clear if symbolic simulation with approximate values is the ultimate solution given this approach. A lot of work is required to build the necessary infrastructure to determine if these methods are indeed capable of meeting their goals. However, it is my hope that other researchers will take this same approach to trying to break the verification bottleneck.

Bibliography

[1] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Formal verification using parametric representations of boolean constraints. In *Proceedings of the 36th Design Automation Conference*, pages 402–407, June 1999.

[2] R. Alur, A. Itai, R. Kurshan, and M. Yannakakis. Timing verification by successive approximation. In *Proceedings of Computer Aided Verification. Fourth International Workshop, CAV '92*, pages 137–150, 1992.

[3] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Proceedings of Computer Aided Verification. Fifth International Workshop, CAV '93*, pages 29–40, July 1993.

[4] Derek L. Beatty and Randal E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proceedings of the 31th Design Automation Conference*, pages 596–602, 1994.

[5] Bob Bentley. Validating the Intel Pentium 4 microprocessor. In *Proc. of 38th Design Automation Conf.*, pages 244–248, 2001.

[6] Armin Biere, A. Cimatti, Ed 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.

[7] Beate Bollig, Martin Lobbing, Martin Sauerhoff, and Ingo Wegener. On the complexity of the hidden weighted bit function for various BDD models. *Informatique Theorique et Applications*, 33(2):103–116, 1999.

[8] Vamsi Boppana et al. Model checking based on sequential atpg. In *Proceedings, Computer Aided Verification (CAV'99), LNCS 1633*, pages 418–430, 1999.

[9] Melvin Breuer. A note on three-valued logic simulation. *IEEE Transactions on Computers*, C-21(4):399–402, April 1972.

[10] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS: A compiled simulator for MOS circuits. In *Proceedings of the 24th Design Automation Conference*, pages 9–16, 1987.

[11] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Proceedings of the 28th Design Automation Conference*, pages 397–402, June 1991.

[12] Randal E. Bryant. Symbolic verification of MOS circuits. In *1985 Chapel Hill Conference on VLSI*, pages 419–438, May 1985.

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

[14] Randal E. Bryant. Symbolic simulation — techniques and applications. In *Proceedings of the 27th Design Automation Conference*, pages 517–521, June 1990.

[15] Randal E. Bryant. Formal verification of memory circuits by switch-level simulation. *IEEE Transactions on Computer-Aided Design*, 10(1):94–102, January 1991.

[16] Randal. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[17] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *CAV 94: Computer Aided Verification, LNCS 818*, pages 68–80, 1994.

[18] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10e20 states and beyond. In *Proc. of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[19] J. L. Carter, B. K. Rosen, G. L. Smith, and V. Pitchumani. Restricted symbolic evaluation is fast and useful. In *1989 IEEE International Conference on Computer-Aided Design*, pages 38–41, 1989.

[20] K.-T. Cheng. Gate-level test generation for sequential circuits. *ACM Trans. Design Automation of Electronic Systems*, 1(4):406–442, October 1996.

[21] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guide abstraction refinement. In *CAV'00: International Conference on Computer Aided Verification*, July 2000.

[22] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[23] Mukesh Dalal. Efficient propositional constraint propagation. In *Proc. of the Tenth National Conf. on Artificial Intelligence (AAAI-92)*, pages 409–414, 1992.

[24] J. A. Darringer. The application of program verification techniques to hardware verification. In *Proceedings of the 16th Design Automation Conference*, pages 375–381, 1979.

[25] M. Davis, G. Logemann, and D. Loveland. Machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[26] David L. Dill. What's between simulation and formal verification? In *Proceedings of the 35th Design Automation Conference*, 1998. Embedded Tutorial.

[27] David L. Dill and John Rushby. Acceptance of formal methods: Lessons from hardware design. *Computer*, 29(4):16–30, April 1996.

[28] E. B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9(2):90–99, 1965.

[29] Asgeir P. Eiriksson. The formal design of 1M-gate asics. In *Formal Methods in Computer-Aided Design. Second International Conference, FMCAD '98*, pages 49–63, 1998.

[30] E.M.Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244– 263, 1986.

[31] Farzan Fallah, Srinivas Devadas, and Kurt Keutzer. OCCOM: Efficient computation of observability-based code coverage metrics for functional verification. In *Proc. of 35th Design Automation Conf.*, pages 152–157, 1998.

[32] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design: An International Journal*, 10(2/3):149–169, April 1997.

[33] Malay Ganai, Adnan Aziz, and Andreas Kuehlman. Enhancing simulation with BDDs and ATPG. In *Proc. of 36th Design Automation Conf.*, pages 385–390, 1999.

[34] Daniel Geist et al. Coverage-directed test generation using symbolic techniques. In *Formal Methods in Computer-Aided Design (FMCAD '96)*, pages 143–158, 1996.

[35] Allen Van Gelder. A satisfiability tester for non-clausal propositional calculus. *Information and Computation*, 79(1):1–19, October 1988.

[36] A. Goel and W.R Lee. Formal verification of an IBM CoreConnect^{TM} processor local bus arbiter core. In *Proc. of 37th Design Automation Conf.*, pages 196–200, 2000.

[37] Richard Goering. Bleeding-edge chip designers seek EDA transfusions. *Electrical Engineering Times*, 2000. 6/12/2000 Issue.

[38] Shankar G. Govindaraju and David L. Dill. Counterexample-guided choice of projections in approximate symbolic model checking. In *Proceedings of ICCAD 2000*, November 2000.

[39] David Greve. Symbolic simulation of the JEM1 microprocessor. In *Formal Methods in Computer-Aided Design (FMCAD '98)*, pages 321–333, 1998.

[40] Aarti Gupta and Pranav Ashar. Integrating a boolean satisfiability checker and BDDs for combinational equivalence checking. In *Proceedings of the Eleventh International Conference on VLSI Design: VLSI for Signal Processing*, pages 222—224, 1998.

[41] Pei-Hsin Ho et al. Smart simulation using collaborative formal and simulation engines. In *2000 IEEE International Conference on Computer-Aided Design*, pages 120–126, November 2000.

[42] Richard Ho and Mark Horowitz. Validation coverage analysis for complex digital designs. In *1996 IEEE International Conference on Computer-Aided Design*, pages 146–151, 1996.

[43] Alan Hu, Masahiro Fujita, and Chris Wilson. Formal verification of the HAL S1 system cache coherence protocol. In *IEEE International Conference on Computer Design*, pages 438–444, 1997.

[44] C. Norris Ip. Simulation coverage enhancement using test stimulus transformations. In *2000 IEEE International Conference on Computer-Aided Design*, pages 127–134, November 2000.

[45] Prabhat Jain and Ganesh Gopalakrishnan. Efficient symbolic simulation based verification using the parametric form of boolean expressions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):1005–1015, 1994.

[46] Rajesh Raina James Monaco, David Holloway. Functional verification methodology for the PowerPC 604 microprocessor. In *Proceedings of the 33th Design Automation Conference*, pages 319–324, 1996.

[47] Jae-Young Jang, In-Ho Moon, and Gary Hachtel. Iterative abstraction-based CTL model checking. In *Meeting on Design Automation and Test in Europe*, pages 502–507, March 2000.

[48] Jae-Young Jang, Shaz Qadeer S, M. Kaufmann, and Carl Pixley. Formal verification of FIRE: a case study. In *Proc. of 34th Design Automation Conf.*, pages 173–177, 1997.

[49] J. S. Jephson, R. P. McQuarrie, and R. E. Vogelsberg. A three-value computer design verification system. *IBM Systems Journal*, 8(3):178–188, 1969.

[50] R.B. Jones and D.L. Dill. Efficient validity checking for processor verification. In *IEEE International Conference on Computer-Aided Design*, pages pages 2–6, San Jose, California, USA, 1995.

[51] Robert B. Jones. *Applications of Symbolic Simulation to the Formal Verification of Microprocessors.* PhD thesis, Computer Systems Laboratory, Stanford University, August 1999.

[52] Robert B. Jones, Carl-Johan H. Seger, and David L. Dill. Self consistency checking. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1166 of *Lecture Notes in Computer Science*, pages 159–171. Springer-Verlag, November 1996.

[53] Michael Kantrowitz and Lisa M. Noack. I'm done simulating: Now what? verification coverage analysis and correctness checking of the decchip 21164 alpha microprocessor. In *Proc. of 33rd Design Automation Conf.*, pages 325–330, 1996.

[54] R. P. Kurshan. *Computer-Aided Verification of Corrdinating Proceses.* Princeton University Press, 1994.

[55] Tracy Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):4–15, January 1992.

[56] W. Lee, A. Pardo, J.-Y. Jang, G. Hachtel, and F. Somenzi. Tearing based automatic abstraction for CTL model checking. In *1996 IEEE International Conference on Computer-Aided Design*, November 1996.

[57] David E. Long. CMU BDD package, 1993.

[58] J.C. Madre and J.P. Billon. Proving circuit correctness using formal comparison between expected and extracted behavior. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 205–210, 1988.

[59] S. Malik, A. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *International Conference on Computer-Aided Design*, November 1988.

[60] Yossi Malka and Avi Ziv. Design reliability - estimation through statistical analysis of bug discovery data. In *Proceedings of the 35th Design Automation Conference*, pages 644–649, 1998.

[61] J. Marques-Silva and Karem Sakallah. GRASP—a new search algorithm for satisfiability. In *Proceedings of the Int'l Conf. on Computer-Aided Design*, pages 220–227, 1996.

[62] Ken L. McMillan and J. Schwalbe. Formal verification of the Gigamax cache consistency protocol. In *International Symposium on Shared Memory Multiprocessing*, pages 242–251, 1991.

[63] Tom Melham. Abstraction mechanisms for hardware verification. In *VLSI Specification, Verification and Synthesis*, pages 267–291, January 1987.

[64] J. Strother Moore. Symbolic simulation: An ACL2 approach. In *Formal Methods in Computer-Aided Design*, pages 334–350, 1998.

[65] Dinos Moundanos, Jacob A. Abraham, and Yatin V. Hoskote. Abstraction techniques for validation coverage analysis and test generation. *IEEE Transactions on Computers*, 47(1):2–14, January 1998.

[66] J. Otten. On the advantage of a non-clausal davis-putnam procedure, 1997.

[67] M. Pandey, R. Raimi, D. L. Beatty, and R. E. Bryant. Formal verification of PowerPC(TM) arrays using symbolic trajectory evaluation. In *Proceedings of the 33rd Design Automation Conference*, pages 649–654, June 1996.

[68] Abelardo Pardo. *Automatic Abstraction Techniques for Formal Verification of Digital Systems.* PhD thesis, University of Colorado at Boulder, Dept. of Computer Science, August 1997.

[69] Abelardo Pardo and Gary Hachtel. Incremental CTL model checking using BDD subsetting. In *Proceedings of the 35th Design Automation Conference*, pages 457–462, 1998.

[70] Viresh Paruthi and Andreas Kuehlmann. Equivalence checking using a structural sat-solver, bdds, and simulation. In *Proc. of Intl. Conf. on Computer Design*, 2000.

[71] Vaughan Pratt. Anatomy of the Pentium bug. In *TAPSOFT '95: Theory and Practice of Software Development*, pages 97–107, 1995.

[72] S. Reda and A. Salem. Combinational equivalence checking using boolean satisfiability and binary decision diagrams. In *Proceedings Design, Automation and Test in Europe Conference 2000*, pages 122–126, 2001.

[73] R. Rudell. Dynamic variable ordering for binary decision diagrams. In *Intl. Conf. on Computer-Aided Design*, pages 42–47, November 1993.

[74] Carl-Johan Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.

[75] Jeffrey Su, Laurent Arditi, Satyaki Das, Jens Skakkebæk, and David Dill. Formal verification of the TORCH microprocessor RTL design. Unpublished, available at http://citeseer.nj.nec.com/su98formal.html, 1998.

[76] Scott Taylor, Michael Quinn, Darren Brown, et al. Functional verification of a multiple-issue, out-of-order, superscalar alpha processor - the DEC Alpha 21264 microprocessor. In *Proceedings of the 35th Design Automation Conference*, pages 638–643, 1998.

[77] Miroslav Velev, Randal E. Bryant, and Alok Jain. Efficient modeling of memory arrays in symbolic simulation. In *Proc. of the 9th CAV, Springer LNCS 1254*, pages 388–399, 1997.

[78] Wolf-Dietrich 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.

[79] Alexander Wolfe. Intel fixes a Pentium FPU glitch. *Electrical Engineering Times*, 1994. 12/7/1994 Issue.

[80] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 184–192, January 1986.

[81] Jen-Tien Yen and Qichao Richard Yin. Multiprocessing design verification methodology for Motorola MPC74XX PowerPC microprocessor. In *Proceedings of the 37th Design Automation Conference*, pages 718–723, 2000.

[82] Steve Young, Chris Malachowsky, and Luis Aldaz. Case studies: Chip design on the bleeding edge. In *Proc. of 37th Design Automation Conf.*, page 648, 2000. Panel Session.

[83] Jun Yuan, Jian Shen, Jacob A. Abraham, and Adnan Aziz. On combining formal and informal verification. In *Proceedings of Computer Aided Verification. Ninth International Workshop, CAV '97*, pages 376–387, 1997.