VALIDATION TOOLS
FOR
COMPLEX DIGITAL DESIGNS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Chian-Min Richard Ho

November 1996

© Copyright by Chian-Min Richard Ho 1996
All Rights Reserved

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 A. Horowitz (Principal Advisor)

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

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.

Oyekunle Olukotun

Approved for the University Committee on Graduate Studies:

Abstract

The functional validation of a state-of-the-art digital design is usually a laborious, ad hoc and open-ended task. Many circuits, and especially modern processors, are too complex to be formally verified in their entirety. Instead, simulation of a register transfer level (RTL) model is used to determine whether the design conforms to the specification for a particular set of test vectors which will hopefully cover all the important and interesting corner cases of the design. Unfortunately, creating a good test vector suite is difficult.

This research explores techniques to make the validation task more systematic, automated and efficient. This can be accomplished by using information embedded in the RTL model to extract the set of "interesting behaviors" of the design, represented as interacting finite state machines (FSM). These behaviors include things like multiple or concurrent events, rare corner cases and long control paths, combinations of which tend to lead to bugs. If all of these interesting behaviors of the RTL could be tested in simulation, the degree of confidence that the design is correct would be substantially higher. This work provides two tools towards this end. First, a test vector generator is described that uses this information to produce a series of test vectors that will exercise all the implemented behaviors of the design in RTL simulation. Secondly, the information can be used as the basis for coverage analysis of a pre-existing test vector suite. The degree to which a test vector suite covers the important tests is known as the *coverage* of the suite. Previous coverage metrics have relied on measures such as the number of toggles on a node in the circuit or code block execution counts. These metrics often give good first order indications of how thorough each part of a circuit has been exercised by test vectors. However, they do not usually give an accurate picture of whether the important test cases involving multiple or concurrent events have been exercised. In this thesis, a new method is proposed of analyzing test vector suite coverage based on projecting a minimized control state graph onto control signals that enter the datapath part of the design, yielding a meaningful metric and providing detailed feedback to the designers about missing tests.

The fundamental problem facing any technique that uses state exploration, as these do, is state space explosion. The exponential growth of the state graph limits the complexity of the models that can be dealt with. Two techniques are proposed to minimize this

problem; first, a dynamic state graph pruning algorithm based on static analysis of the model structure to provide an exact minimization and second, approximation of the state graph with an estimation of the state space in a more compact representation. These techniques help delay the onset of state explosion, allowing useful information to be obtained and utilized by designers, even for complex designs. Results and practical experiences of applying these techniques to the design of the node controller (MAGIC) of the Stanford FLASH Multiprocessor project are given.

Acknowledgments

I arrived at Stanford University in the Fall of 1990 with only a vague notion of wanting to earn a Doctorate Degree, not knowing what exactly I wanted to work on or how to set about it. I thrashed about for a year trying to find direction. Then I persuaded Mark Horowitz to become my advisor, and I think that was probably my best decision at Stanford. Mark has not only been a source of excellent advice, enlightening discussions and excellent questions, but has also been an enormous well-spring of encouragement over the years.

My thanks and appreciation also go out to David Dill, for all his patience and tutorship. My conversations with Dave always helped put my work in context and often led to further ideas and refinements.

I also with to thank Kunle Olukotun for bravely agreeing to serve on my Reading Committee at a time when this dissertation was still a vague jumble of ideas.

This work grew as part of the Stanford FLASH Multiprocessor project and I am indebted to its team members for their friendship and humor. In no particular order: Jeff Kuskin, Dave Ofelt, Mark Heinrich, John Heinlein, Jules Bergmann, Dave Nakahira, Joel Baxter, Ravi Sound, Hema Kapadia, Shankar Govindaraju, Rich Simoni, Ziyad Hakura.

I want to thank my parents and brother Jenson for always believing in me and encouraging me to keep pushing forward with my work.

But most of all, I want to thank my new bride Sandy, whose eternal patience, loving smiles, and tender words made everything possible.

Table of Contents

Abstract iv

Acknowledgments vi

Table of Contents vii

List of Tables x

List of Figures xi

Chapter 1. Introduction 1

1.1 Overview of Thesis 3

Chapter 2. Background 6

2.1 Design Validation through Simulation 6

2.1.1 Validation Test Vectors 11

2.1.2 Measuring Validation. 13

2.2 Formal Hardware Verification 15

2.2.1 Reachability Analysis 16

2.2.2 Equivalence Checking with Logic. 17

2.3 Using Formal Techniques in Validation. 18

Chapter 3. Capturing Interesting Test Behaviors 20

3.1 Focus on Control Logic 20

3.2 Validation Methodology based on Control Interactions. 21

3.3 RTL-to-FSM Translation 24

3.3.1 Semantic Issues 25

3.3.2 Identifying Control-Logic 28

3.3.3 Abstracting the Design. 29

3.4 State-Enumeration Tool 32

3.5 Discussion. 34

Chapter 4. Validation Vector Generation 35

4.1 Paths through the State Graph. 35

4.2 Converting to Test-Vectors. 37

4.3 Undetectable Bugs 40

4.4 Stanford FLASH Memory Controller Example 42
4.4.1 Protocol Processor Test Generation. 44
4.4.2 Protocol Processor Bugs 46
4.5 Related Work 49
4.6 Discussion. 49

Chapter 5. Validation Coverage Analysis 52

5.1 Graph Redundancies 53
5.2 Graph pruning using Static Analysis for Don't Cares 54
5.2.1 Static Analysis of Kill Sets 56
5.2.2 Dynamic Pruning with Don't Cares. 57
5.3 Control Events 63
5.3.1 Coverage Property 65
5.3.2 Control Event Graph 66
5.4 Over-Generalized Environment 68
5.4.1 Case 1: Illegal Sequence from Reset 69
5.4.2 Case 2: No Bubble in Cache Refill 71
5.4.3 Interface Assertions 71
5.5 Incremental Feedback 74

Chapter 6. Coping with State Space Explosion 76

6.1 State Graph Approximation 77
6.2 Approximation from Exact Partitions. 78
6.2.1 Exact Partitions Algorithm 78
6.3 Approximating the Transition Function 83
6.3.1 Choice of Approximation Variables 84
6.4 Related Work in State Graph Approximation. 85
6.4.1 Other Approximation Techniques for Validation 86
6.5 FLASH Coverage Results. 87

Chapter 7. Conclusions 89

References 94

List of Tables

Table 1. Errata Classification of MIPS R4000 21
Table 2. PP Instruction Classes. 44
Table 3. PP State Enumeration Results. 45
Table 4. Test Vector Generation Statistics 46
Table 5. Synopsis of Discovered Bugs 47
Table 6. Dynamic Pruning Results (5MByte Hash Table) 58
Table 7. State variable kill sets comparison 61
Table 8. Constraints in Approximated Instr. Fetch Unit. 81
Table 9. Different Approximated Variables for Inbox 85
Table 10. Coverage Results 87

List of Figures

Figure 1. Processor simulation with Instruction-Level-Simulator 9
Figure 2. Simulation with Self-Checking Vectors and Snoopers 10
Figure 3. Validation Methodology based on Control Interactions 22
Figure 4. Non-deterministic Environment 23
Figure 5. Problematic MPP State Assignment 26
Figure 6. Breaking up state-to-state assignments in MPP 27
Figure 7. Code examples of transparent latch (a) and edge-triggered flop (b). 28
Figure 8. State Annotation Example. 29
Figure 9. Transitive Fan-In Algorithm 30
Figure 10. Murφ State-Space Exploration Algorithm 33
Figure 11. Tour Generation Algorithm. 36
Figure 12. Converting Transition Edge to Test-Vector 37
Figure 13. Sample Transition Condition Mapping 39
Figure 14. Erroneous FSM Implementation with more Behaviors. 41
Figure 15. Erroneous FSM Implementation with fewer Behaviors 42
Figure 16. MAGIC Block Diagram 43
Figure 17. PP Abstract FSM model 45
Figure 18. Bug #5 Timing Diagram (Glitch Masked) 48
Figure 19. Bug #5 Timing Diagram (Garbage written) 48
Figure 20. Don’t Care variable due to code structure. 55
Figure 21. List of RTL Structures where a Kill-Set may be possible. 56
Figure 22. Static Analysis for Kill Sets 57
Figure 23. Dynamic Pruning Algorithm. 58
Figure 24. Relative Pruned State-Space Size 59
Figure 25. Relative Running Times 60
Figure 26. Correlation of kill sets and state reduction 61
Figure 27. Growth of Relative Overhead 62
Figure 28. Example of State Graph Redundancy 64
Figure 29. Independent State Variables 66
Figure 30. Counter fed by Non-Deterministic Input. 68

Figure 31. Illegal Sequence from Reset 70
Figure 32. Illegal Bubble in Cache Refill 72
Figure 33. Converting a Constraint to a Snooper 73
Figure 34. Approximating the State Graph. 78
Figure 35. Approximation of Exact Partitions 79
Figure 36. Conversion of State-Graph to Transition Function 80
Figure 37. Approximating Graph leads to Non-deterministic FSM 81
Figure 38. Non-deterministic FSM 82

Chapter 1. Introduction

Ongoing semiconductor process technology improvements are resulting in progressively smaller feature sizes and continually better yields. This trend has given circuit designers more silicon area on each die in which to pack ever increasing amounts of functionality. As a result, digital designs are progressively becoming faster and more complex. These trends make the task of verifying that the design is free of functional bugs increasingly difficult. Larger designs tend to have more concurrent interactions, all of which require testing. And as the number of these interactions grow, the ability of humans to track and think up tests for them diminishes. At the same time, business considerations such as time-to-market pressure designers to produce working silicon in the fewest number of *tapeouts* (sending the design for fabrication). There is also pressure to avoid a negative public image that results from a serious bug getting into a shipping product [Mer94]. The result of these trends is that functional correctness checking poses an increasingly hard and important challenge to design teams.

One approach to this problem is to use formal verification techniques, discussed in more detail in Chapter 2. These strive to show categorically that a property is true or not true in a design. They show a great deal of promise and have successfully verified many important properties of large designs. However, because of issues of complexity, state-space or representation size, these techniques have tended to work with models of designs that are either relatively small or have been abstracted in order to focus on one aspect of the verification problem. There remain unresolved issues when working with many of the

large designs produced by the semiconductor industry. These unresolved issues have prevented the widespread adoption of formal techniques for functional correctness proofs.

In the absence of a method to completely verify designs, simulation of a detailed register transfer level (RTL) description has been utilized. Simulation cannot realistically be expected to exhaustively test all the operations of a non-trivial design, but with a set of well-designed test vectors, it can provide a certain confidence level. Unfortunately, since simulation is relatively slow, of the order of hundreds of cycles per second or slower, and well-designed test vectors are hard to create, this *validation task* often consumes as much resources as the design process. The introduction of cycle-based simulators and hardware emulation technology [GBC+95] will alleviate this problem to some extent, but achieving exhaustive simulation is impossible. Even at a speed of 100MHz, which is aggressive with existing technology, it would take approximately 325 years to exhaustively simulate a small design of about 300 bits of state.

Current validation techniques rely heavily on test cases that are composed by the designer to stress particular functions of the design. These tests, sometimes called *directed tests*, are meant to exercise the circuit through all its important functionality by directing the simulation towards what are considered interesting situations. These include all variations of the basic functionality of the design as well as combinations of different functions which can occur concurrently. A good suite of validation tests will be able to exercise the design through all its basic functions and many of its more complex, concurrent functions. Unfortunately, the job of thinking up and writing tests for all the combinations of concurrent functions is a difficult and laborious one for a human designer. And quite often, a complex or rare situation that is overlooked by the designer, resulting in incorrect operation, may also be overlooked by the test writer, leaving a bug in the design.

To deal with this possibility, many designers also employ *pseudo-random* test vectors in simulation after basic functionality has been checked with directed tests. The goal is to inject randomness into as many of the inputs of a design as possible to stir up situations missed by the designer and the directed tests. A typical example for processor designs is to randomly assert exceptions. Pseudo-random tests have proven to be very effective at finding bugs, by causing situations to occur in simulation that were missed by the test writer.

Together, directed and pseudo-random tests, run in simulation, form the backbone of the traditional validation methodology. Although they do a reasonable job, the difficulty of making these test vectors effective grows as designs grow more complex. The limitation is that it is infeasible for a human to figure out all the possible interactions that require testing in a non-trivial design. On the other hand, pseudo-random tests provide the ability to stochastically reach lots of the interactions, but cannot guarantee that all will be reached in a reasonable amount of time. It is also difficult to know how many of the interactions have been tested with pseudo-random vectors, making it hard to decide when to stop. There remains a gap between the desired correctness confidence level for many designs and current validation techniques.

1.1 Overview of Thesis

What is needed is some form of instrumentation to guide and measure the validation process so that progress can be judged quantitatively. One such measure is the reachable product state space of the control finite state machines (FSM). In simple terms, the FSMs of a design encode its behaviors. Each individual FSM controls one aspect of the design's behavior. The combination of all the FSMs acting in concert on all the inputs determine the global behavior of a design. Although many errors arise because individual FSMs or the combinational logic derived from them are wrong, these tend to be easy to discover in simulation by directed tests aimed at basic functionality. However, when two or more FSMs communicate or jointly control a behavior, then the task of testing each such interaction manually becomes hard. Apart from the combinatorial explosion of cases, it is frequently difficult to keep track of and create each in a test vector.

Though it is often hard to find all the interactions in a design manually, this information is encoded in the register transfer level (RTL) model of a design, which is a detailed description of the intended implementation. It would be very useful if this information could be utilized automatically with minimal user assistance. Then, validation tools could use this information either to create tests or to measure coverage. It turns out that for designs that try to separate control and datapath sections, such extraction of the control FSMs mostly automatically is possible. In fact, if the RTL is written in a synthesizable subset of a hardware description language (HDL), such as Verilog [TM91], then this extraction is relatively straightforward. The non-trivial aspect is identification of which

logic constitutes control and which datapath. The approach taken in this work is to obtain hints from the designer, and is described in Chapter 3.

Once the control FSMs have been extracted, the global behavior of the design can be found by enumerating all the possible states that the communicating FSMs can reach. This is a technique used in formal verification to prove properties about every state of a design. For validation work, the important information is the reachable state space. Each state in the global state graph represents an interaction between the FSMs. Hence, it provides a map of all the interactions in the design. If the simulated test vectors manage to reach every state and edge of the global state graph, then the confidence level of its correctness would be high. For this work, the FSM enumeration tool is derived from work done in [DDH+92] and is described in Chapter 3.

Using the full global state space of the control FSMs provides a quantitative measure of test vector completeness with respect to implemented FSM interactions. If the state space can be found exactly, this not only provides a way to measure testing progress, but can provide a map of how to test those interactions. If the inputs to the control FSM model correspond to inputs to the RTL design that can be manipulated by test vectors, then it is possible to generate test vectors automatically that would cause the RTL to follow a particular path through the global state space. Chapter 4 discusses an example of this type of test vector generation and its limitations.

Unfortunately, test vector generation is not possible for all designs for several reasons, including explosion of the global state space, the inability to provide correctness checking using these vectors in some designs, and the inability to control some inputs needed by this type of test vector. In cases where these issues cannot be overcome, the global state space can still provide a useful measure of the completeness of functional testing, in other words, coverage analysis.

The experience of using this method of coverage analysis in a real design example, the node controller of the FLASH multiprocessor [KOH+94], demonstrates that coverage information about state space can be overwhelming when presented directly without filtering for relevance. This makes it difficult to use in a practical setting. This observation leads to a method, presented in Chapter 5, that projects the full state space onto the signals that directly affect the datapath portion of the design. Using this projection on the state

space leads to incremental coverage feedback that pinpoints the missing tests in their simplest forms, which is easier to use.

Using the full control FSM state space for test vector generation and coverage analysis is appealing from a completeness viewpoint. However, for many realistic and interesting designs, the global state space of the control FSMs is too large to be found and manipulated with reasonable computing resources. The state space explosion problem plagues all formal verification techniques that attempt to find the reachable states of a model. However, validation uses the state space in a less strict manner than formal verification generally does. It is not always necessary to retain the full global state space to obtain useful coverage information, especially basic coverage information. Hence, it is still productive to approximate the state space with one that requires less memory and runtime but retains properties of the full space. The basic idea is to retain the state variables that the designer believes will interact in interesting ways, and remove variables that have small impacts on the full state space. The problem is that the approximate state graph will either miss real states or create states that do not really exist in the full state graph. Heuristics and issues related to approximation of the state space are discussed in Chapter 6, along with some results of coverage analysis using an approximation of the state space of some of the units of FLASH. These show that despite errors in the approximation, useful information can still be obtained to guide the validation task for complex designs.

The remaining chapters describe in detail the process and the practical experiences of using automatically extracted control state to guide and assist validation.

Chapter 2. Background

This chapter first reviews the most common methods of validation in use today to better understand the nature of the problem and the limitations of current simulation based approaches. To try to provide better validation, some researchers have taken a different approach to the problem and used formal methods to try to definitively prove properties about the design. These are described next. While this group has made enormous progress, there are still a number of problems when trying to prove properties of real implementations and some of these limitations are also discussed. Though both approaches have significant limitations, using techniques from formal verification to aid simulation looks like a promising new area. One such approach is described in the following chapters, using design examples from the FLASH project.

2.1 Design Validation through Simulation

When using simulation for validation, the first question that needs to be answered is how the designer can tell when the simulation is correct, or more to the point, when it is incorrect. This choice of simulation framework can determine the type of validation test vectors that can be used.

There are two common simulation frameworks used for design validation. In the first framework, called the *co-simulation* framework, a second, possibly more abstract, executable model is written without reference to the RTL. This is often called a *golden model*. The two models are then co-simulated using the same set of test vectors and the resulting

operations are compared for equality. The assumption is that a bug in one model will not appear in the other. In other words, bugs will not be correlated in the two models. Hence, every non-equality, or mismatch, is tracked down to discover which model is incorrect. This framework is convenient because the two models check each other's correctness. There is no need to have test vectors that check for specific results, which are difficult to write. Instead, much simpler tests, such as the pseudo-random test vectors that are discussed in the next section, can be used which simply exercise the design. This makes these tests amenable to automatic generation.

However, this framework puts many stringent requirements on the two models being used. The first is that the two models must have state that can be compared for equivalence. This can be difficult to achieve for many designs. If the two models are at different abstraction levels, state that exists in one model may not exist or may not correspond exactly in the other model. For example, an implementation of a design with pipelined operations will likely have state that does not correspond exactly with state in a non-pipelined model of the design.

The second requirement is that there must exist synchronization points at which the corresponding state of the two models can be compared for equivalence. At these points, any transient states of the models have been resolved and comparison can occur. In the simplest cases, a single event can signal such a state, for example, the writeback stage of a processor design. In other cases, the synchronization point is more complex and may involve history of the operations. One natural way to think about synchronization points is to consider them as the end of some abstract operation of the design. The comparison then checks that the operation was performed correctly. This is an especially good perspective if one of the models represents the operation more abstractly than the other. In such a view, complex synchronization points arise because different operations may have different termination events, which can be used to recognize the end of the operation; and they have optimizations that can start the next operation without passing through an idle state. This tends to make the synchronization point a complex condition that needs to examine the history of operations.

The third requirement is that arbitration points in the design match up exactly in the two models for both to perform the same sequence of operations. Specifically, in a design that performs arbitration, the arrival sequence, and possibly arrival times, of data to the

arbitration logic is important to ensure that the two models stay in agreement about the correct operation. If the arbitration point makes different decisions in the two models, all notions of state comparison is lost thereafter. For arbitration logic to make the same decision, at the minimum, it is necessary to preserve the same partial order of data arrival. If the two models are both cycle accurate, this is not too difficult. However, when one model is more abstract than the other, the exact timing of data arrival may be lost in the more abstract model, resulting in different decisions.

With these requirements, making two models agree for all legal input sequences is a difficult task. In many cases, it is tantamount to writing two detailed implementation models. In addition, the two models need to be written without reference to the other for the assumption about uncorrelated bugs to be true. For many designs, creating two models that meet these requirements is prohibitively expensive in terms of resources. So, despite the advantage that test vector creation is simpler with this framework, it is not in widespread use for general circuits. But for circuits where these requirements are met, it can be a very powerful validation tool.

One large class of designs where this framework is feasible and in widespread use is processor designs. For these circuits, instruction-level simulators can be used to compare architectural state, namely the register file, the program counter and the processor status register. The synchronization point is usually when data is written back to the register file. And in general, there are no difficult arbiters in the design. A typical setup of this framework is shown in Figure 1. Such a framework is extremely amenable to pseudo-random and other automatic test vector generation methods.

In the many cases where the co-simulation framework cannot be applied, the second simulation framework, called the *self-checking* framework, can be used, in which test vectors are self-checking. That is, the test vectors are encoded with a way to check for expected results. In this case, only the RTL implementation model is simulated. This removes the need to develop a second model for co-simulation. However, the burden of correctness checking now falls on the test vectors themselves, which must perform checks on expected data values during and at the end of each test to ensure correctness. This has disadvantages: first, it is much harder to write such test vectors that make meaningful checks after complex sequences of operations. These tests are sufficiently hard that it becomes extremely difficult to create an automatic test vector generator. This makes it one

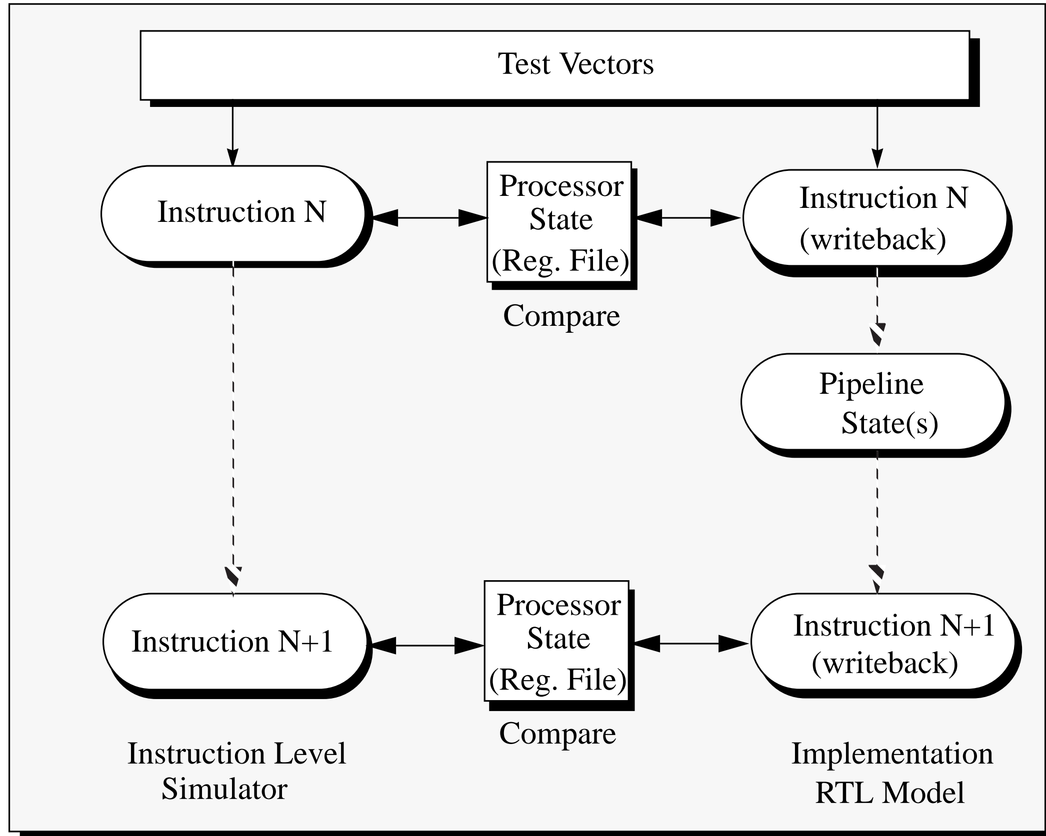

Figure 1. Processor simulation with Instruction-Level-Simulator

of the bottlenecks of current validation efforts. To create a good suite of such self-checking test vectors requires a large team of test writers to work for many months, an expensive, error-prone and time-consuming step in the design process. Secondly, such test vectors will usually require maintenance during the design process to keep up with design changes so that they will continue to provide the correct expected result and that they continue to test the circuit. A third problem is that errors that arise during the test may not show up until long after they occur in the circuit, either because the erroneous result takes a while to show up in state that can be checked, or because tests tend to perform their checks after long sequences of operations. This makes debugging harder. In addition, a related serious disadvantage is that it is possible for errors to arise in the design and not be caught in the checks made by the test vector. In particular, it is difficult to hand-write test vectors that check all possible failure conditions in a non-trivial design. Many errors may actually occur but not get caught with the directed check performed by the test vector.

To address this last disadvantage, RTL simulation in this framework is often supplemented with simulation code, called *snoopers*, that check for error conditions during simulation of the test vectors. Snoopers are simply designer provided checkers that monitor signals and signal combinations in the RTL during simulation. They have the ability to check the detailed operation of the design and can indicate errors close to the source. These are usually not too difficult to write since they are generally localized and small. In addition, unlike a co-simulation model, these checkers do not need to model the expected behavior of the design, they just have to check for obvious errors. Examples are checks for mutual exclusion of signals, absence of unknown, or 'X', values and state consistency checks. Although there is no method to ensure that all possible snoopers have been provided by the designers, they provide valuable additional checks on the design that supplement the self-checks done by the test vectors. A typical set up is shown in Figure 2.

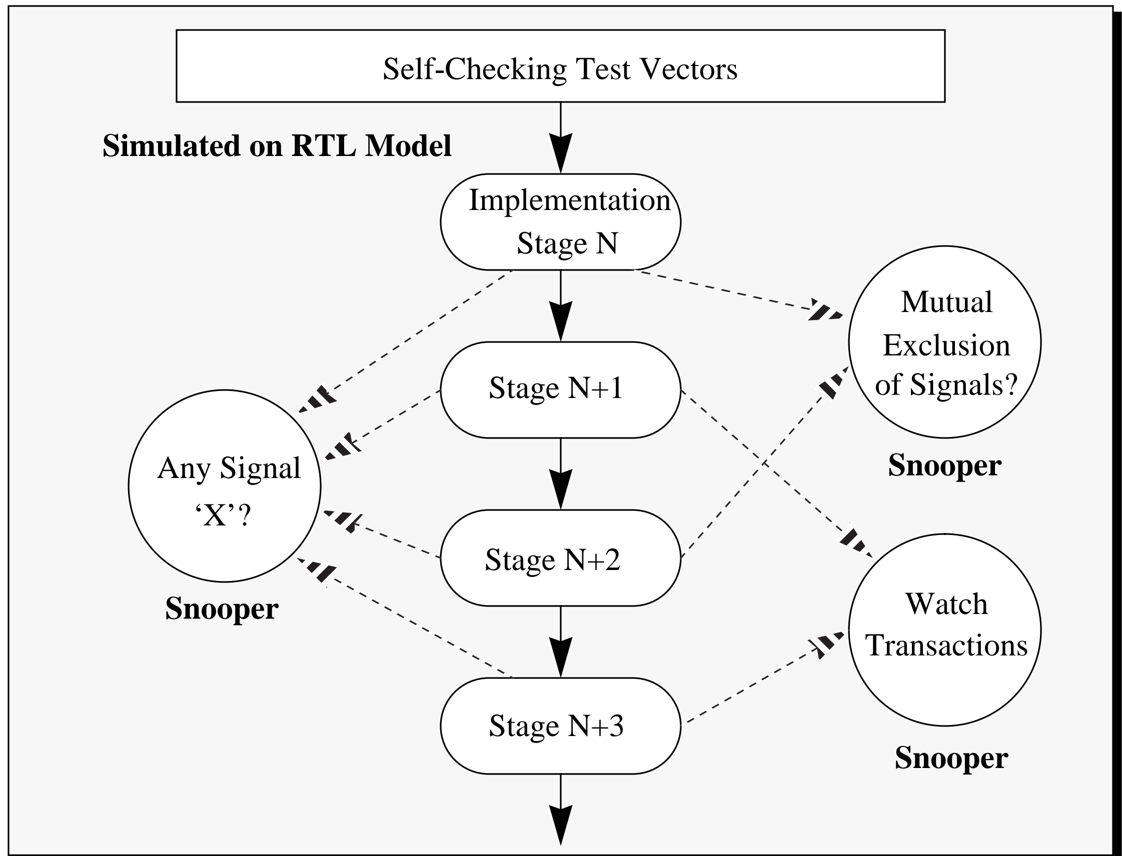

Figure 2. Simulation with Self-Checking Vectors and Snoopers

Given these two general paradigms, the choice of a simulation framework is basically the choice of how to determine correctness of the test vectors that will be simulated. The co-simulation framework generally provides good checking across large parts of a design, but has many difficult requirements that must be satisfied for it to work. If these requirements can be met, this is clearly the better validation framework. An example using this framework is the processor portion of the Stanford FLASH project, which is described in more detail in Chapter 4.

The self-checking framework is much easier to setup initially than a co-simulation one, but suffers from less complete correctness checking, even with the addition of snoopers. However, for many general circuits, excluding processor designs, this is the more common framework in use because of the difficulty of creating a golden model. Since it is difficult to create an automatic self-checking test vector generator, coverage analysis can be performed instead to assist test writers achieve better validation results with this framework, and this is described in Chapter 5.

2.1.1 Validation Test Vectors

When validating a design with simulation, the quality of the results depends in large part on the test vectors that are run in the simulation. For a bug in the design to be found, the test vectors must exercise the erroneous logic. Good quality directed test vectors are time consuming to produce by hand, but in general, this is the only method of obtaining test vectors that can be used with the self-checking framework. These tests are generally difficult to write because the test writer must have detailed knowledge of the design implementation in order to successfully exercise all the corner cases and subsequently determine what the correct result should be. For complex interactions between multiple control sections of a design, it is even harder to fully consider all possible outcomes and craft tests that can exercise them. The biggest danger is that an important interaction is overlooked and never tested. This is a common mistake that leads to bugs which are only discovered much later in the design process, when it is more expensive to fix. Hence it is important to have a method of determining whether important test cases have been overlooked when using hand-written directed tests.

When a co-simulation framework is used, automatic test vector generators can be utilized to a greater extent. These techniques generally use some form of pseudo-randomization to create many tests from a single test description. There are numerous examples of

test vector generation in this manner: [ABD+91], [ABG+92], [And92], [KN95], [MSY+95] and [WGK90]. These produce random test patterns, which are possibly targeted or *biased* towards certain simulation events. Biasing simply raises the probability of rare events or interactions with the hope of stirring up untested situations. For example, if an input to a design that is being generated has 10 possible values, then a fair distribution would produce each of the possible values with equal likelihood. But if the generator is biased towards 1 of the values, then that value would appear more frequently in the stream of generated values. In this way, the test vectors can be directed towards particular events. These events are normally worry cases identified by the designer as requiring extra testing due to the complexity of the section of logic handling that input value.

Typically, random tests are simulated for hundreds of millions of cycles. When they are usable, that is, when a co-simulation framework is feasible, random generators have the advantage of being relatively simple to set up and can find most of the obvious and many of the complex bugs in a design quickly. A lot of their power stems from the fact that they can generate test interactions not thought of by test writers and designers. This comes about simply because these tests put together lots of random interactions. Over a long period of generation, the probability of exercising many of the complex interactions increases. And in theory, if test generation can proceed indefinitely, all possible interactions should eventually be created. Of course, this is not possible in a real design environment with finite resources. In addition, without some other method to assist, there is no good way to measure the progress of random test vectors in order to estimate when and if certain interactions will get tested. Hence, pseudo-random test vectors, although extremely useful validation tools, are not a systematic method to validate a design.

A second class of automatic test vector generators are constraint based ones, which try to be a little smarter in their choice of random values. The AVPGen system, [CI92], [CIJ+93] and [CIJ+94], uses templates of constraints to create tests that stress a design in corner-cases. The basic idea here is that tests are written in a slightly more abstract form, where exact values of variables and inputs are not fixed in the test description. Instead, constraints are placed on the variables, which are kept symbolically. Test vectors can then be generated from these templates by pseudo-randomly assigning actual values, that meet the constraints, to the symbolic variables. The templates can also be interleaved as directed by the test writer to test event interleavings. A constraint solver is used to resolve constraints from the templates being interleaved to produce a series of new templates

which can then be used for further generation. Typically, various small stress cases are written in the form of test templates. For example, one template may specify a pipelined-microprocessor instruction sequence that forces data-dependencies across cycles. Then, different templates are composed and the constraints of each template are resolved. An actual test sequence is then created by taking the templates and assigning random values to the unconstrained components. This technique can be utilized in both the co-simulation framework and the self-checking one since the basic set of test templates, which are hand-written, may be self-checking.

The power of this technique is that it allows a test writer to think up corner cases and create templates for them. The constraint solver is then responsible for interleaving them and finally assigning actual values to form the test vectors. This late binding of values is powerful because it allows the same templates to be used with many different actual test values. One way to look at this technique is as a method to introduce randomness into hand-written directed tests. The randomness takes the form of different interleavings of templates and the form of different actual values of variables. The shortcoming of this technique is that it still requires design knowledge from humans to create meaningful test templates. It is useful in that the work of each test writer is multiplied into several possible tests, but it still does not provide a method to systematically find all the interactions in a design that need to be tested. This, like all the techniques described so far, leaves open the possibility that important test cases are not exercised with no way to alert the test writer of this fact. The next section talks about techniques to measure validation progress. This is important to validation since the current test vector creation techniques, just described, cannot give any indication of how well they are doing or whether they are testing interesting things.

2.1.2 Measuring Validation

When test vectors are simulated, it is necessary to check whether they really do exercise the design as intended. Sometimes, vectors do not set the necessary conditions, or the design undergoes changes, making the vectors ineffective at testing the interactions they were originally designed for. In addition, the decision to tapeout, though often dependent on market forces, should really be made based on projected completion of the validation task. There are several ways this can be tracked and these are discussed next.

One commonly used method is to track the bug discovery rate and when it becomes flat and low for some length of time, the design is fabricated, [Cla90]. This metric is highly dependent on the quality of the test vectors, since it assumes a constant stream of different, high quality vectors. Only if this is provided will this measure be truly representative of the degree of design completion. The problem is that the rate of creating new, good quality test vectors almost never stays constant. Usually, whenever a new method of test vector creation is tried, a burst of bugs are found since new methods have a good chance of exercising new interactions. As use of the new test creation method matures, the bug rate drops until the next method is introduced. Hence, bug discovery rate is a better productivity measure than a validation completion measure. It certainly does not say anything about which interactions in a design have been tested and which have not.

Other linear metrics, like node-toggle, line, or code-block coverage, used in [And92], [KN95], [WGK90] and [WT95], provide some indication of whether all parts of a design have been tested. Node-toggle coverage measures whether each node in a circuit has been exercised and how often during validation. Clearly, a node that has never toggled indicates logic that has never been tested. In general, this is not good for validation. But even after all nodes have been toggled at least once, this measure can be useful to get a better balance of toggles across nodes in a circuit. This would help ensure that all parts of a circuit received about equal amounts of testing, to a first order.

Line and code-block coverage are similar to node-toggle coverage, except that they measure execution of lines or code-blocks[1] in the RTL description during tests. The information content is fundamentally equivalent to node-toggle coverage. If any line or code-block of the RTL description is not executed, this will generally translate into one or more nodes that do not toggle. One extension to line coverage is branch-taken coverage. This tracks the direction taken at each branch in the RTL code. This provides more information about the behavior of the design. For example, if one code-block can be reached from two different branches, then branch-taken coverage will indicate whether the code-block has been reached from both points, something that line and code-block coverage would not indicate.

[1] A code-block is a basic block of the RTL description.

Although these measures are based on the linear coverage of code rather than the function of the design, they provide a good baseline measure of testing completeness. While they cannot indicate accurately whether logic interactions have been tested, they do ensure coverage of basic functionality before such interactions are considered. If these basic measures do not indicate high coverage, then further, more detailed interaction coverage measures as described in Chapter 5, will not provide much additional useful information. They will indicate a slew of missing cases, simply because one basic test may have been missing. In this situation, it is easier to identify the basic case with the basic coverage metrics, then proceed to the complex interactions involving that basic case.

Once basic measures have indicated good coverage, more probing measures can be used to guide the later stages of validation. One such method attempts to measure validation coverage in a manner analogous to the *stuck-at faults* of manufacturing fault test coverage [KS92]. In this technique, *design errors* are specified which represent types of errors that may occur in RTL, for example, using the wrong gate in a combinational circuit. This is the analog of stuck-at-faults, which model possible manufacturing defects. A design is analyzed to determine where all the design errors may exist, that is, where each of the design errors could possibly be found in the design. Then, analysis measures how many of the design errors can be detected by a test vector suite. Coverage improves with this technique when test suites are capable of uncovering more of the design errors. Conceptually, this approach is appealing because it tries to identify where errors may occur and then grades the test vectors based on their ability to exercise that code. The largest problem with this approach is developing a credible set of design errors. In fault coverage, the stuck-at fault model has a basis in manufacturing defects, lending credence to the methodology. However, design errors do not have a widely accepted corresponding phenomenon which they model. In addition, it is not clear that exercising all such design errors is feasible using a finite amount of resources.

2.2 Formal Hardware Verification

The previous section described the most common approaches to validation currently available using simulation. The biggest problem is that there is no systematic technique to create test vectors that can exercise all the complex interactions in a design. Coverage analysis can guide test vector creation to areas where more testing is required, but current

techniques provide practical information only up to single FSM arc coverage. Complex interactions can still be missed.

One response to these problems is to use formal verification techniques. These attempt to categorically prove or disprove properties about a design. This is in contrast to validation by simulation, which can never categorically declare a correct design, since a complex design can never be exhaustively simulated. If one of these formal verification methods can be applied, it provides a powerful and definitive statement. However, most have limitations, so it is important to understand when each technique can be applied and what results can be expected.

2.2.1 Reachability Analysis

One of the simplest formal techniques is to take a description of a design and find all the possible states that it can ever get into. This state space exploration [BZ83], provides a straightforward method of exploring the range of possible behaviors of a system and checking whether any of the resulting states violates a *safety property*[2]. This technique is powerful since it checks every reachable state in a system for an error. There is no doubt when this technique returns an answer, unlike validation, which can never return a doubt-free answer of “no errors found”. In addition, one of the by-products of the state exploration is usually a sequence from the reset state to any error states, which helps debugging greatly. Such techniques are extremely useful for checking properties of protocols [PD96]. The main drawback is that often, the state space grows too large to handle, a phenomenon known as *state space explosion*. State space explosion occurs because every additional bit added to the state description potentially doubles the state space. There have been techniques developed that can find reductions in the state space that needs to be explored without compromising the property check [IpD93], but these also are best applied to protocol verification.

An alternative to using an explicit representation to store the set of reachable states is to use binary decision diagrams (BDD) [Bry86]. BDDs are a compact and canonical representation of a boolean expression. They can be used to implicitly store the set of reachable states found in state space exploration. When done in this way, reachability analysis is called symbolic because each BDD may represent a large number of states. In many cases,

[2] A *safety property* is an assertion that must hold in every state.

this can extend the size of the state space that can be explored far beyond the size that can be explicitly represented. There are also algorithms that can directly operate on BDDs to formally verify properties, a process known as model checking. Model checking is the verification of properties, some of which may involve paths of the state graph and not just states, against the design. This is utilized with success in [Lon93], [CYF94] and [CGH+95]. In model checking work with BDDs, *temporal logic* properties can also be checked. Temporal logic is a means to express properties which are true over a period of time. These temporal properties look for problems such as *livelock* (no forward progress) and process *starvation* (execution fairness between processes).

Although BDDs have the advantage that they have the potential to represent large numbers of states using very little memory, they too can suffer from a type of explosion. The size of the BDDs needed to represent a design is dependent on the order of the variables in the BDD. With a bad ordering, the size can grow exponentially large. The biggest problem related to this is that it can be very difficult to predict in advance the size of the BDDs required for a particular design. Even relatively small designs can experience a BDD blow-up. This can make the usefulness of BDDs limited unless their use is carefully crafted to preclude this.

2.2.2 Equivalence Checking with Logic

Another formal approach, one that does not attempt to use properties of the state space of a design is the work of [BuD94] and [JDB95]. These attempt to prove the equivalence of two models of a design, one written with implementation details such as pipelines, and one without. Fundamentally, this is a formalization of the co-simulation method of validation, which also compares two models for equivalence. The difference is that this technique attempts to prove it for all possible input values to the design. To reduce the complexity of the task, they use a logic of *uninterpreted functions* to create the two models. An uninterpreted function treats operations, such as add or shift, as black boxes and does not try to evaluate it, which reduces the complexity of the equivalence check. In this way, datapath operations are abstracted, that is, they are not evaluated. Instead, the operation is simply carried along in the logic expression of the operation. A *validity checker,* which decides whether two logic expressions are equal, then proves that the two models are equivalent, thus showing that the implementation (pipelined model) is correct with respect to the specification (unpipelined model).

This work attempts to avoid the state explosion problem of reachability analysis completely. Instead of considering states, it examines possible operations. It is powerful because it can show equivalence for all inputs of a design. However, it is not yet ready for widespread usage for a couple of reasons. First, the explosion problem, though not in terms of states, still arises in checking of equivalence of the logic expressions, which can take a very long time and use a great deal of memory. Secondly, and possibly more importantly, this method requires two models of a design to be written in a special logic. This amount of work seems prohibitively high, especially since some abstraction of the design must occur in order to make the technique feasible. This introduces the possibility of errors being masked or introduced by the abstraction.

Another use of logic has been to utilize *theorem provers* to show that a formal description of the design satisfies a set of properties. A theorem prover takes a description of a design as a set of axioms which state properties of the design. It then accepts queries about other properties which the designer wants to know, for example, whether it is result of an operation can ever be wrong. The theorem prover then uses inference rules to determine the answer to the query. This has been done in [Cyr94] and [MLK96]. These methods are difficult for designers to use because they require familiarity with the logic language and theorem proving techniques. Though powerful, these cannot be utilized in a normal design environment yet.

2.3 Using Formal Techniques in Validation

At the beginning of this chapter, we saw that the hard problem with using simulation for validation is the task of creating good test vectors that can exercise all the complex interactions in a design. None of the current methods can reliably get close to this goal. The first aspect of this problem is just knowing which interactions have been missed by test vectors. Current coverage measures provide basic measures but stop short of being able to accurately identify complex interactions that need to be tested. The end result is that currently, validation tends to be an ad hoc and unreliable method to fully test designs. Formal verification techniques were proposed to solve many of these problems. And although they have made great strides forward, none as yet has shown the capability of handling large design implementations in a general way.

A possible hybrid approach is to try to borrow formal techniques to help the validation task. This holds the possibility that some of the properties of formal verification, such as completeness, can be introduced into validation. Current validation practices lack a good measure of which interactions need testing and which have been tested. One possible approach is to use reachability analysis, or state space enumeration, to quantify the interactions in the control logic of a design. This would provide a definitive and meaningful measure of validation progress. This was done in [HMA95] and [HH96]. In [HMA95], the basic technique is described which extracts the control FSMs from an RTL description. Coverage is then measured in terms of arc coverage of the global control FSM, which is found by symbolic reachability analysis. This provides a comprehensive coverage metric of all the interactions in a design, as will be described in Chapter 3. However, when applied to large designs as in [HH96], this metric turns out to be too conservative. In order to achieve good coverage numbers on this metric, a redundant number of tests need to be simulated. This dilutes the quality of the metric and makes the validation task unnecessarily lengthy. Instead, the common practical implementation of this technique is to measure coverage on single FSMs. Like the basic coverage measures, this is useful and necessary. And like the basic coverage measures, it also does not provide enough information about the complex interactions in a design that need testing. The work in this thesis extends these ideas and attempts to address the issue of deriving a usable, quality coverage metric.

Although the problem of state space explosion arises when state space enumeration is used, validation utilizes the state space information in a less rigid manner than formal verification. This leads to possible reduction and approximation methods that can not be effectively used in formal verification. As will be seen in later chapters, sometimes the exact state space can be reduced and still retain important information for validation. In this way, validation can be guided much more effectively than with any current method. This will be discussed in Chapter 5. But first, the process of extracting useful validation information is described in detail in the next chapter.

Chapter 3. Capturing Interesting Test Behaviors

As the previous chapter showed, the critical problem for simulation based validation is the creation of stimulus vectors that fully exercise the design. This chapter shows one approach for obtaining the needed information for vector generation by extracting the control flow directly from the implementation description. This is done by translating the RTL to a set of cooperating finite-state-machines (FSMs), and consequently finding all the states that are reachable from reset. This global state graph contains *all* the possible behaviors of the design. We focus on trying to exercise the machine through the control state since the control bugs are usually the hardest ones to find, as Section 3.1 shows. To exercise the machine through all possible control interactions is a two step process: extracting the control logic and then finding all possible transitions. These steps are described in Section 3.3 and Section 3.4. Uses of the transition information to generate test vectors and for coverage analysis are described in the following chapters.

3.1 Focus on Control Logic

Common wisdom from practicing hardware designers involved in microprocessor design is that the majority of bugs that were found only after silicon was produced was the result of multiple, rare and unexpected control interactions. The published errata lists for the MIPS R4000 [Mips94] gives some evidence to back this assertion. The errata represent those bugs which not only made it to silicon, but also slipped through to production. These are the bugs that cost the most, both in terms of the cost of an addition spin of silicon and also in terms of delayed time-to-market and are a good target for validation tools.

The bugs can be classified into three categories: the first represent those due to an error in the datapath section of the design; the second due to isolated errors in the control logic; and the third arise when multiple control actions or events occur concurrently. The results of this classification are shown in Table 1.

Table 1. Errata Classification of MIPS R4000

Bug Class	Number of Bugs	% of Total
Pipeline/Datapath ONLY bugs	3	6.5%
Single Control Logic Bugs	17	37.0%
Multiple Event Bugs	26	56.5%
Total Reported Errata	46	100.0%

Multiple event bugs are difficult to find because they require several control interactions to occur in a particular sequence or with some specific timing relation. Creating test vectors that can exercise *all* such situations is difficult with random-generators and nearly impossible when hand-written. Given the large number of difficult bugs that occur due to control interactions, we target our validation tools to try to cover this area more completely.

3.2 Validation Methodology based on Control Interactions

The approach described here is to capture the control logic description from the RTL description. This is then used to find all the possible control interactions in the logic. And finally, this information can be used for validation: either test vector generation or coverage analysis based on control interactions. These steps are shown in Figure 3.

In the first step, the **translator** extracts the pertinent control logic FSMs from an RTL description written in Verilog. The assumption made here is that the Verilog already has a coarse partitioning for synthesis into control and datapath sections. Verilog that does not satisfy this assumption can be translated, but would require more designer input to ensure that a good partition of control and datapath is obtained. Then, to extract the minimal control logic from the control section, the user is asked to annotate the Verilog further with comment-embedded directives that highlight some of the important state variables in the control logic. The translator then applies a *transitive fan-in* algorithm to capture the logic which those state variables depend on. The transitive fan-in algorithm simply finds all the

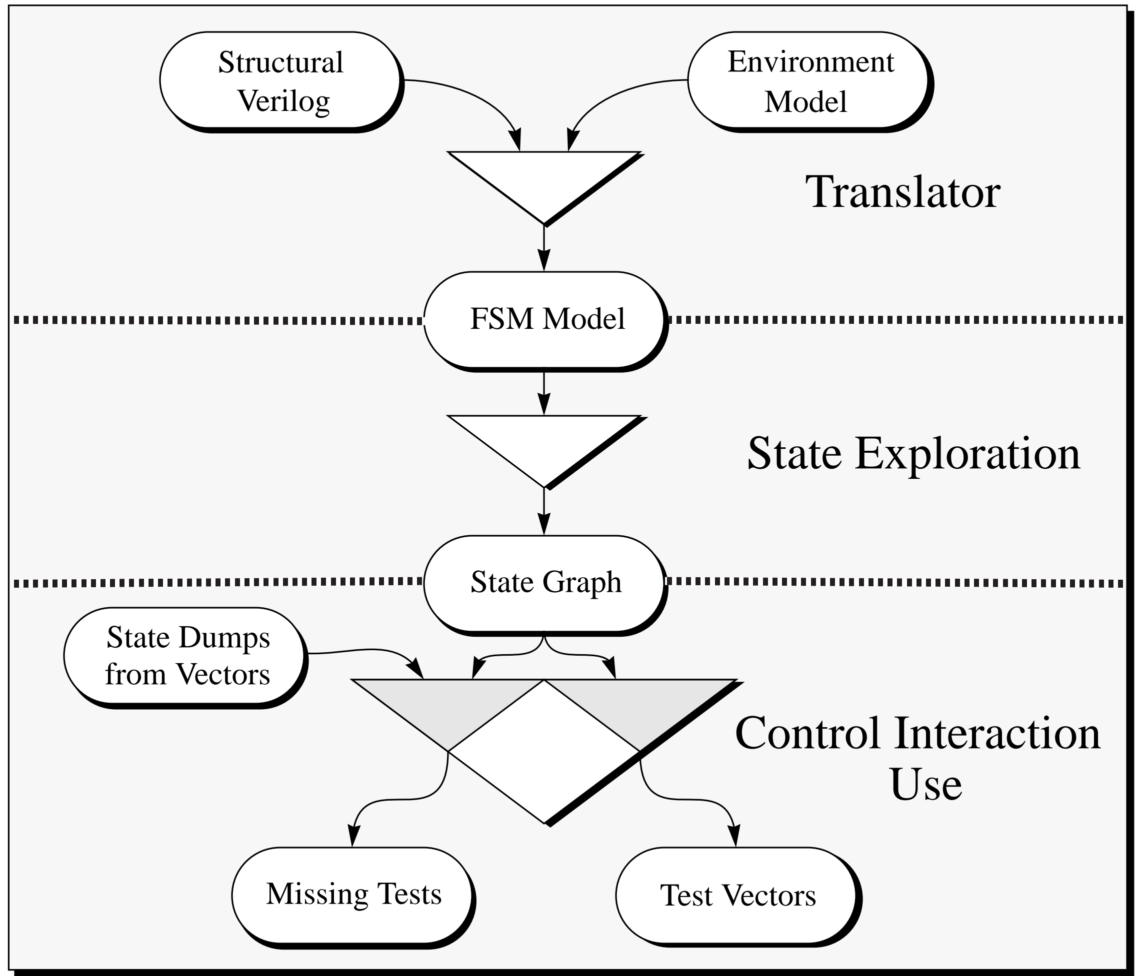

Figure 3. Validation Methodology based on Control Interactions

logic that sets the value of the highlighted state variables. It then iterates, looking for the logic that sets that logic and so forth. The process stops at the module boundary of the control logic, based on the coarse partitioning for synthesis. The extracted logic is then combined with a description of the environment of the FSM model.

The environment acts as an input generator to the FSM model of the control logic. In general, these inputs are either primary inputs to the design, or they represent signals from the datapath, which is not captured by the translator. The most general environment is one which generates any input at any time, known as a *non-deterministic* environment. In the

context of finding the global state-graph, non-determinism means trying all possible input sequences in all cycles, shown in Figure 4.

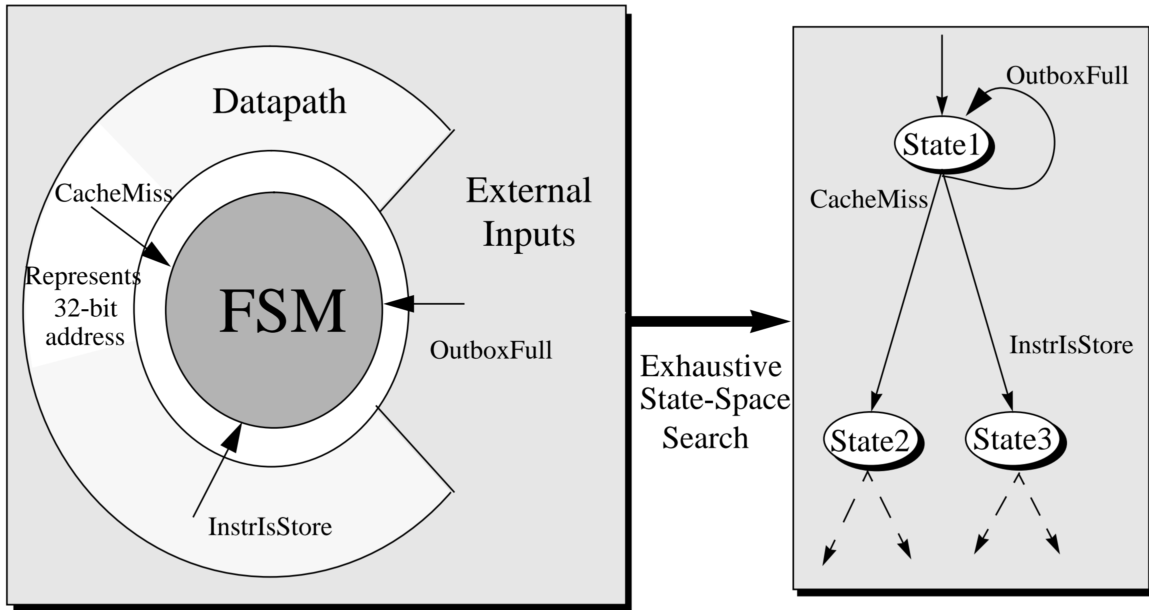

Figure 4. Non-deterministic Environment

The non-deterministic (ND) environment can also provide an abstraction of the interface. For example, a 32-bit address to a cache in a processor design can often be abstractly represented by an ND environment that provides a single bit indicating whether the address will cause a hit or a miss in the cache. This models the control logic effects of the address on the cache by a single bit, as illustrated by the *CacheMiss* signal shown in Figure 4. This often simplifies the control logic model of a design. This is discussed in more detail later in this chapter. The ND environment provides the most generality since it captures all possible behaviors of the FSM model. If a test vector suite manages to exercise all interactions of the FSM model under the full ND environment, we can conservatively assume good coverage of the control interactions. As we will see in Section 5.4, sometimes the ND environment is too general and leads to control behaviors that are not possible in the hardware. When this situation arises, extra constraints can be placed on the environment in the form of input generators that do not take all possible values on every cycle.

The second step of the methodology is state space exploration of the control FSMs. For this, the FSM description language used is a descendant of Murφ [DDH+92] called

MPP (Murφ ++). MPP takes the FSM description of the system and finds all its reachable states from reset. It produces a global state graph and hash table of states which are used in the third step. These first two steps serve to capture the control logic interactions and are described in detail in Section 3.3 and Section 3.4.

The third step in the toolset uses the control logic interactions to either generate test vectors or to perform coverage analysis. To generate vectors, paths through the control graph are found that improve the testing coverage. Each edge of this path results in the generation of a single test vector. This process and algorithm are presented in Chapter 4.

To calculate coverage, the program reads state dumps from a test suite run in simulation on the RTL. These state transitions are marked on the control graph and the individual state variables, giving a coverage metric. Detailed feedback is given to the designers in the form of transition edges not exercised. This process and some of the practical problems encountered applying this algorithm to the models in the FLASH project are described in Chapter 5.

3.3 RTL-to-FSM Translation

One shortcoming of many of today's formal verification techniques is that many require re-writing the design in a particular language. This introduces the possibility that errors are missed or introduced in the translation step. This is especially true where the translation also incorporates an abstraction step. The possibility of error in the translation step can be avoided with the use of an automatic translator, as used in the HSIS and VIS suite of tools [Ber93], [CYB93]. These translate a model written in a hardware description language (HDL), such as Verilog [TM91], to intermediate languages used in further processing. Direct translation offers a more faithful representation of the design, as well as the ability to more easily track changes.

Hence, the first step in capturing control logic interactions is to translate the Verilog description of the design into an FSM modeling language[1]. The translator takes synthesizable Verilog as input, and produces MPP code as output. For the most part, translation

[1]This work also focussed on Verilog as the RTL description language. The other commonly used alternative is VHDL [Ash96]. Conceptually, the only requirements for a VHDL front-end to the methodology is synthesizability of the design.

between structural Verilog and MPP is a straightforward syntactic rewrite. The subset and style of Verilog that is accepted by the translator is similar to the subset that would synthesize well using Synopsys tools. Transparent latches, as well as edge-triggered flops, are recognized. However, there are a few semantic issues that require some care to ensure correct translation of the control logic.

3.3.1 Semantic Issues

The most important semantic issue is to ensure that the event-based concurrency model of Verilog is preserved when going to MPP, which has a cycle-based concurrency model. In Verilog, when a variable changes value, an event is posted to the simulator. If any other variables are *sensitized* to, or dependent on, that value, they will be updated and further events posted. In this way, textual ordering of assignments is made irrelevant to the concurrency model. Where this matters is assignment to a state variable. In Verilog, assignment to a `reg` data type is potentially capable of holding state. If coded correctly (without a race condition), state assignments should also be made so that they are independent of textual order[2]. However, in MPP, assignments are evaluated once, in their textual order, every cycle. This may lead to problems if a state-variable is directly used as an input to another state-variable assignment, as shown in Figure 5.

The intended behavior of the Verilog is for simultaneous assignments to *StateA* and *StateC*. A straightforward translation to MPP would result in the wrong concurrency model. However, MPP does give the correct concurrency model for non-state variables, that is, assignments to *wires* are independent of textual ordering, and wire assignments are always evaluated before *any* state assignments are done. This provides the fix for the above problem. If the translator ensures that a *wire* appears between every state-assignment, we get the desired behavior where every state assignment occurs simultaneously using the previous values for other state-variables. Hence, if the translator ever finds a state variable read directly by another state variable in the Verilog, it inserts a wire assignment in between, as shown in Figure 6. Since MPP evaluates its `wire` variables (called `FLOW`) prior to its state variables, this results in all state assignments using the previous values of state variables that appear in the right-hand-side. Although this example is specific to MPP as an FSM enumeration tool, the underlying problem is preservation of an

---

[2] If not, different simulator implementations may result in different results. For transparent latches, this is done by sensitizing the assignment to all variables that it depends on. For edge-triggered flops, this is done by using a non-blocking delayed assign.

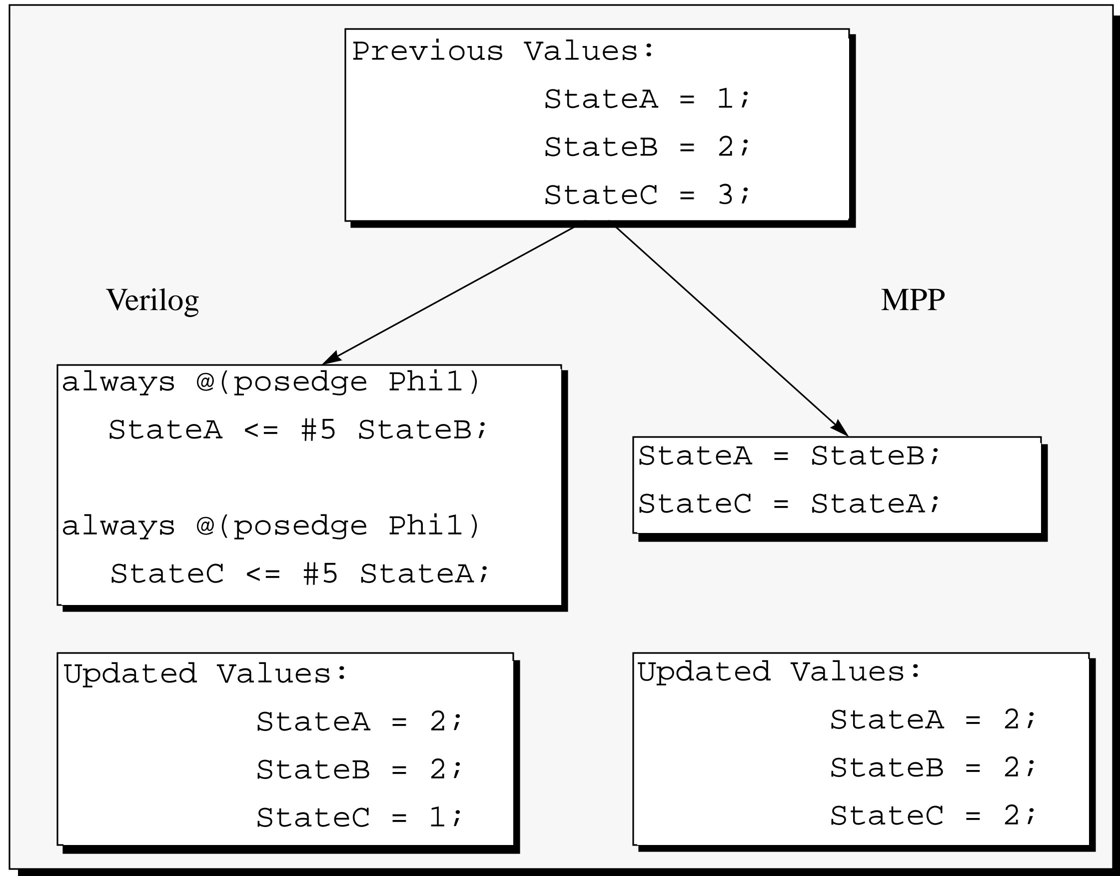

Figure 5. Problematic MPP State Assignment

event-based concurrency model when translating to some other concurrency model. This issue will arise in some form with most FSM enumeration tools.

The translator must also be able to infer all state assignments in Verilog because MPP requires that state variables are explicitly defined. There are several ways in which a variable can be made to hold state. As part of the coding style adopted, all state elements were required to be either:

- transparent latches encoded with a fully-sensitized `always` block, or
- edge-triggered flops encoded with a non-blocking, delayed assignment.

The general semantic issue is identification of state. This is also the primary reason the translator is restricted to a synthesizable subset of Verilog. With this restriction, all state, explicit or implied, can be inferred from the structural equivalent of the description.

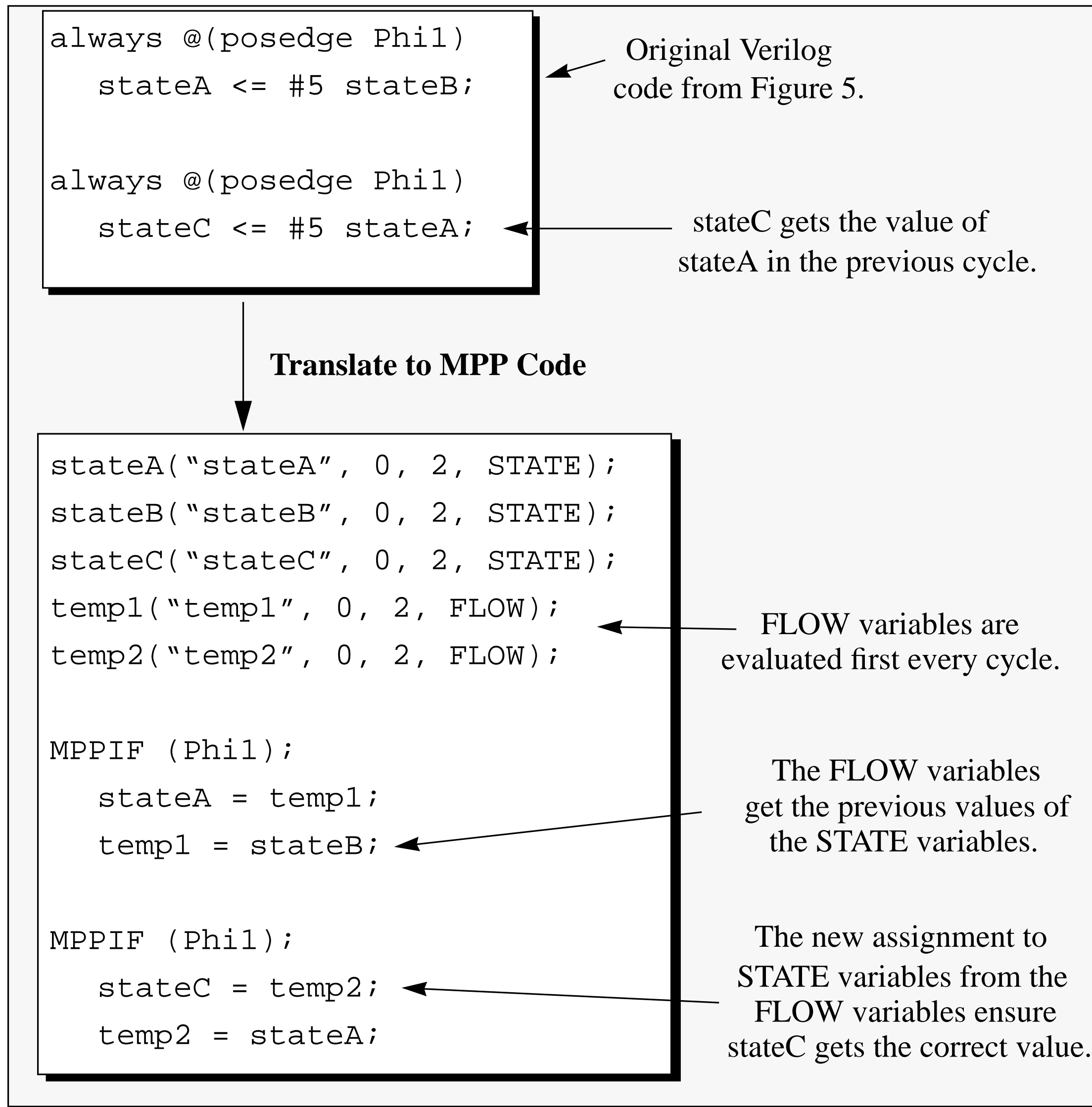

Figure 6. Breaking up state-to-state assignments in MPP

However, an unsynthesizable description may contain implicit state encoded as delays, that is, state is created by placing long delays between assignment statements to the same variable. Such a coding style is described as *behavioral* and can not, in general, be synthesized automatically. For exactly the same reasons, the RTL translator cannot generally infer state in such a description, hence behavioral descriptions cannot be used as input to the methodology.

(a)

```
always @(Phi1 or condition or input)
   if (Phi1 & condition)
      latch = input;
```

(b)

```
always @(Phi1)
   if (condition)
       latch <= #5 input;
```

Figure 7. Code examples of transparent latch (a) and edge-triggered flop (b).

3.3.2 Identifying Control-Logic

A harder task for the translator is determining which parts of the RTL description are required for the control logic FSM modeling. We assume that the RTL we get from the designers already has a coarse level of partitioning between control logic and datapath, which has been done for synthesis. However, even within the control sections, not all the logic present is required to create the FSM model. For example, there may be some logic that takes an output from an FSM and modifies it to control a datapath element. This additional information might not be wanted in the FSM description, which focuses on the logic that determines possible next state values.

To capture only the logic that directly determines the next state function, a *transitive fan-in* algorithm is used. For this, the designer is asked to point out some of the important state variables in the RTL by annotating the Verilog with comment-embedded directives. The annotation is done once for a design and as long as changes to the design are not too drastic, it remains constant for a particular FSM model. An example of the non-intrusive annotation is shown in Figure 8. In general, only the state variables of the major FSM need to be annotated. These are usually assigned values in `switch` statements in code destined for synthesis.

The transitive fan-in algorithm is given in Figure 9. It maintains a stack of variables that are found to be *required* by the FSM logic. Initially, only variables that are named in the annotations are placed in the *required* stack. It also maintains a *provided* set, which

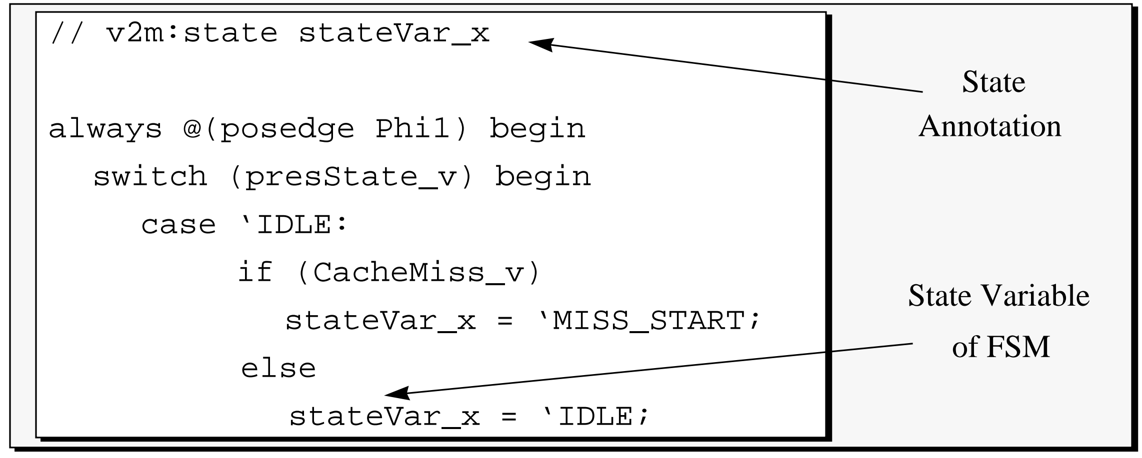

Figure 8. State Annotation Example

holds the variables that have already been included in the translated logic. As each variable is popped from the *required* stack and placed in the *provided* set, the algorithm scans the RTL searching for other variables that help determine its next value. Each of these is compared to the set of variables already found and placed in the *provided* set. Any variable not already in the *provided* set is added to the *required* stack. This continues until the stack is empty. All logic that sets the value of a variable in the *required* set is translated. The net effect of the transitive fan-in algorithm is that all logic that determines the next-state value of the annotated variables is translated, up to the module boundary. The algorithm stops at the module boundary because this acts as a natural *abstraction* interface. Abstraction is discussed in the next section.

If multiple modules are translated together, some cross-module checking of dependences is performed. In particular, if one module requires a signal as an input and it is generated by another module, a dependence is created. The logic in the other module for that signal is labeled *required* even if it was not otherwise needed. In this way, only signals that represent inputs from the datapath or external environment are left to be driven by the environment.

3.3.3 Abstracting the Design

The inputs to the FSM model specified by the transitive fan-in algorithm and not generated by logic within some other module must be driven by the non-deterministic (ND) environment. The ND environment provides an abstract model of the datapath and external signals that is more general than the real hardware. In other words, it generates more

```
/* Transitive Fan-In Algorithm */
initialize REQUIRED queue to NULL;
initialize PROVIDED set to NULL;
do {
    current_var  = POP top of REQUIRED queue;
    /* If we popped a var, mark it as provided. */
    if (current_var != NULL) {
        Add current_var to PROVIDED set;
    }

    for (each RTL statement) {
        switch (statement type) {
            case ANNOTATION: push variable onto REQUIRED queue;
                          /* User specified state variable */

            case CONTINUOUS_ASSIGNMENT:
            case  PROCEDURAL_ASSIGNMENT: {
                /* Procedural assign requires further case analysis of statements allowable
                  in procedural always block to find the assign statement */

                if (assign to current_var) {

                    Mark statement for translation; /* Generate this line of code */
                    for (all variables in RHS of assign) {

                        /* All variables that are read */
                        if (variables not in PROVIDED) {

                            /* This var not in generated code */
                            push variable onto REQUIRED;
                        }

                }
            default: {
                ignore statement;
            }
        }
} while (REQUIRED not empty);
```

Figure 9. Transitive Fan-In Algorithm

possible signal combinations than really occur. This makes the environment much easier to model. For example, if a design has two inputs that have a constraint that they can never both be asserted ‘on’ at the same time, an ND environment can be created by using two inputs that assert ‘on’ independently. While this is easier to write, it leads to the input combination that should not really happen in the real design, namely, both asserted ‘on’. When this happens, if the design has been created with knowledge of the illegal inputs, it is possible that a *false error* can arise. This is where an error is signalled by the validation framework, but is caused by bad input combinations rather than a genuine design problem. Such false errors are distracting and can hide real bugs in a large dump of error reports. They can be removed by adjusting the environment model to be more accurate and model the input constraints. The use of such constraints in the FLASH examples are discussed in Section 5.4.3.

Apart from being simpler to write, an environment model can provide some abstraction of the design. For example, consider the program counter (PC) of a processor design. Let us say this is a 32-bit register. Its impact on the control logic of the processor would normally enter the control module as a 1-bit signal that indicates whether the address in the PC resulted in a *hit* or *miss* in the instruction cache. That is, whether the cache needs to be refilled or not for this address. The actual address is not relevant to this part of the control logic, which controls the cache refill operation. Hence, the single bit acts as a natural abstraction of the 32-bit PC. Instead of requiring a 32-bit input and trying all 2^{32} possible combinations, only the 2 combinations of the single bit need to be tried. This saves the amount of work that must be done on every state and also saves the state space that would be used to store the 32-bit address.

The user must provide the ND-environment as an MPP file that is concatenated to the MPP file created by the translator. In its simplest form, this just drives every input to the FSM with a non-deterministic value. At a later stage, if more constraints on the inputs are needed, this environment file is changed to provide the constraint, which are written as C++ methods. Adding constraints is not difficult and not very time-consuming. The difficult part is determining what constraints need to be added. The most effective method used in the FLASH project was to use no extra constraints initially. All errors were then tracked down. If the error was a false error, the cause was tracked back to the input value that was wrongly generated. A constraint could then be added. This method is the safest since it constrains the inputs to the minimum possible. The experience from FLASH was

that bad inputs quickly manifested themselves as catastrophic errors that were not difficult to track down, if the person tracking down the problem had basic familiarity with the design.

Although some datapath abstractions such as the program counter are usually easy to abstract with the ND-environment, other abstractions require more effort from the user. The most common abstraction is to group signals into equivalence classes. An example of this is given in Section 4.4 where decoded instructions to a processor are grouped into instruction classes. In general, this requires knowledge of the design and must be done manually. The benefit of finding abstractions is that the state space of the control FSMs may be reduced. This can improve running time or memory requirements of the validation tools, and in the best cases, even bring designs that might be too large otherwise, into the realm of the feasible.

3.4 State-Enumeration Tool

The second step in the design validation methodology is the state space exploration of the FSM model. This process attempts to find the set of reachable states of the model from reset. There are several techniques available to do this, using *explicit* and *implicit* methods. The technique used in this work is an explicit method derived from Murφ [DDH+92], called MPP. This decision was a pragmatic one since the state enumeration tool was already under development at Stanford. However, conceptually, an implicit state enumerator utilizing BDDs could also be used in its place. This gives the possibility of manipulating much larger symbolic state spaces.

Murφ, and all its descendents, use an *explicit* hash table to store states found during state enumeration. The FSM model is represented as a set of *guarded rules* that encode the next-state function of the system. A guarded rule is simply a condition followed by an atomic action. If the condition evaluates to true, the action can be executed. These rules are executed, or *fired*, in a non-deterministic fashion, which means that all rules are tried for every state. When a rule is fired, the next state of the system is found and looked up in the hash table. If the state is present, it means that it had been previously found and we can

ignore it now. Otherwise, we add it to the hash table, and put it in the queue of states to be explored.

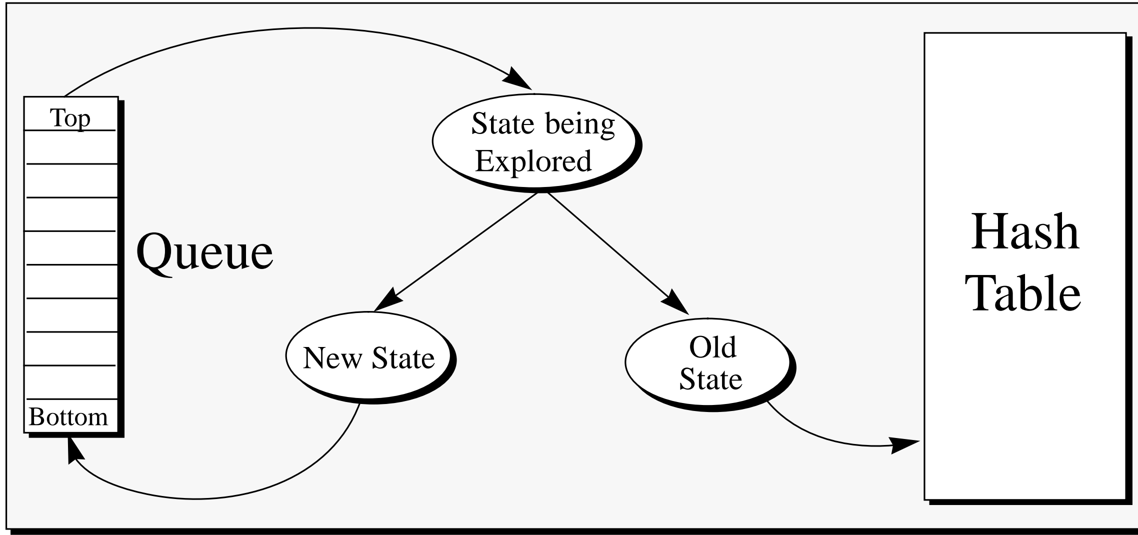

Figure 10. Murφ State-Space Exploration Algorithm

The system starts with just the reset state in the queue and hash table. All the rules are fired and the resulting states are looked up in the hash table. Any that are not present are added to the queue. This process continues until the queue is empty, meaning the entire state space has been found. This process allows modeling of asynchronous systems. Each rule corresponds to a single FSM, so firing every rule at every state leads to trying all possible interleavings of asynchronous events.

To better model synchronous digital systems, a variant of Murφ was created, called Synchronous Murφ and later MPP, that groups rules together to be fired simultaneously. The rules fired together represent the lock-step execution of a synchronous design. By synchronizing the execution of rules, a cycle-based concurrency model is obtained. This synchronous concurrency model provides an implicit edge-triggered clock for the state variables of the design. This needs to be turned into an explicit clock for latch designs with multi-phase clocking.

In the next chapter, a technique for generating test-vectors from the state graph is presented. For this technique to operate, the state enumeration tool must mark the conditions under which a transition edge occurs. For MPP, these conditions are simply the values of

the non-deterministic variables that resulted in that particular edge. This corresponds to the *firing* of a particular *rule* in the Murφ algorithm.

3.5 Discussion

In this chapter, a methodology was given for extracting and using control logic information from the implementation of a design. A procedure was given to mostly automatically extract the FSM description and to explore the state space. This automatic translation step is important to remove possible human errors when creating a model of the design. Though conceptually simple, the translator has hidden within the algorithm, a mini-synthesis step. Fortunately, in this work, the FSM language chosen has a semantics similar enough to synthesizable Verilog that RTL analysis is limited to mostly state inference. However, in general, the RTL must be understood in order to convert it to another format with different semantics. From this requirement, the restriction that the RTL model be synthesizable arises.

As mentioned before, the largest issue for state enumeration is state space explosion. Limiting the logic under consideration to only control removes a large amount of state explosion due to wide bit-width state elements in the datapath, for example, 32 bit registers holding data values. And using an ND environment to abstract components outside the control FSMs also helps. However, for many interesting designs, full state space enumeration will suffer from explosion and be difficult or impossible to complete with finite computing resources. Ways of obtaining useful information for validation despite this will be discussed in Chapter 6. In the next two chapters, the procedures to utilize the control logic interaction information to assist and guide the validation task are fleshed out.

Chapter 4. Validation Vector Generation

One of the two possible operations in the last stage of the validation methodology is generation of test vectors from the control state graph. The goal here is to generate test vectors that cause the simulation to exercise every arc in the graph. Since the graph is created by automatic extraction from the detailed hardware description, the test-vectors generated should be capable of causing the same transitions to occur in the RTL simulation. However, since this generation process is fundamentally a guided random algorithm, it assumes the presence of a correctness checking *golden model*, as described in Section 2.1.1.

There are two parts to test vector generation from the control state graph. First, paths through the graph need to be found that cover each transition edge at least once. Secondly, the path needs to be converted to a test vector that can be run in simulation. These are discussed in the next sections.

4.1 Paths through the State Graph

To maximize the potential for finding bugs in the design, every possible edge in the control graph should be executed at *least* once. This corresponds to trying every interaction of the control logic at least once. A series of arc transitions that traverses every arc at least once is known as a *transition tour*. A transition tour that traverses every arc *exactly* once is known as a *Euler Tour*. However, a Euler Tour can only be found for symmetric graphs that are strongly-connected [Hol91]. The general problem of finding a transition

tour in a non-symmetric strongly-connected graph is called the Chinese Postman Problem [EJ72] and is most frequently encountered in the field of protocol conformance testing, which has developed algorithms for finding such tours in polynomial time [Hol91]. However, the validation methodology does not require a strict transition tour and in fact, there are good reasons not to use a single transition tour. To make concurrent simulation possible and to limit the simulation time needed to reach any bugs found, we break up the transition tour into smaller components that all start from the reset state. In this case, the requirement is that the union of all the arc transitions taken by all the tour components cover all the arcs in the state graph. Since we are generating inputs to a simulator, we try to avoid operations which are costly in simulation, namely backtracking and setting the system to a particular state. Instead, we allow traversing an edge multiple times and prohibit backtracking. This strategy leads to a *depth-first* search of the graph, since this translates naturally to the simulator advancing cycle-by-cycle. We use a *greedy* algorithm which performs a depth-first style search (*ExtendPath*) of the full graph, using each transi-

```
GenerateTours () {
        state = InitialState;
        open output file to write tour;

        do {
                do {
                     /* Depth first traversal generating a vector for every edge traversed. States can be
                      visited multiple times as long as there is an untraversed edge from that state. */

                     state = ExtendPath (state);

                     /* When ExtendPath cannot find a state with untraversed edges, perform a breadth first
                      search looking for any state that has an untraversed edge. If found, generate the vectors
                      to reach this state from the point ExtendPath stopped at. */

                     state = ExploreBFS (state);

                } while (we can find a state with an untraversed edge) &&
                                (number of instr. generated <= MAX instructions per file);

                close output file and open new output file to write new tour;

        /* Explore phase - check whole graph for any remaining untraversed edges. */
        state = ExploreBFS(InitialState);

        } while (there exists a state with an untraversed edge);
        Remove empty last output file;
}
```

Figure 11. Tour Generation Algorithm

tion edge as part of the tour. This proceeds until an untraversed edge cannot be found. At this point, the algorithm goes into an *explore* phase looking in a breadth-first manner for an untraversed edge but without adding edges to the tour. Once found, the shortest path from the point where *ExtendPath* stopped to the untraversed edge is added to the tour and *ExtendPath* can continue. If such a node cannot be found, the algorithm starts a new tour starting from the reset state. This continues until no new tours can be found even from reset. The algorithm used to create this set of partial-tours is shown, as pseudo-code in Figure 11.

4.2 Converting to Test-Vectors

Converting from a transition tour to test vectors requires that the RTL simulation be driven to take the transitions specified in the tour. As we saw in Section 3.4, MPP marks each transition edge with one of the sets of values of the *non-deterministic variables* that resulted in that edge. Hence, the values marked on each transition edge tell us what the datapath and external input values must be in order to cause that edge to be taken in RTL simulation. The task of test vector generation is to cause these values to occur at the appropriate time during simulation. This can be made to happen in a couple of ways.

The most direct method is simply to take control of a signal in the Verilog simulation through the `FORCE/RELEASE` commands, as demonstrated in Figure 12. This allows

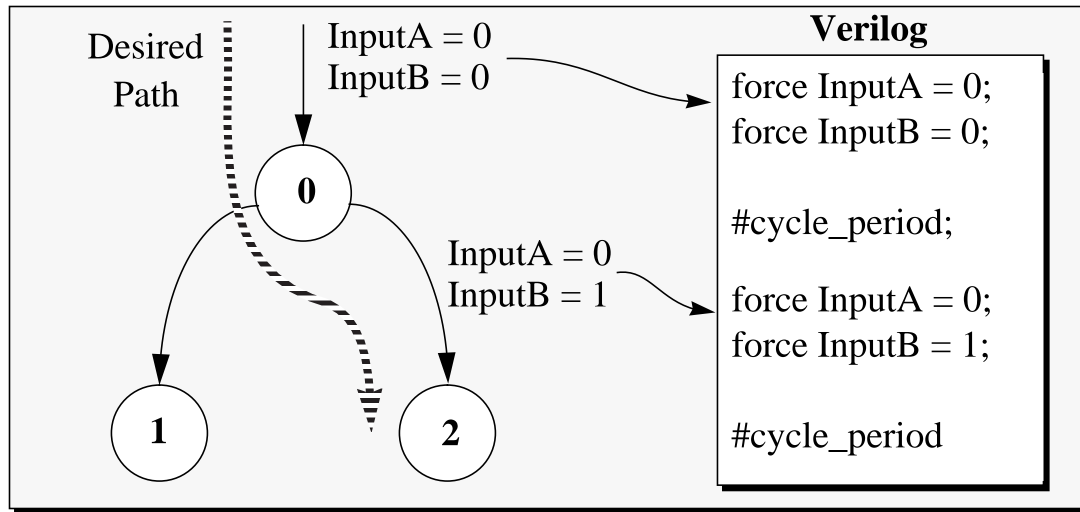

Figure 12. Converting Transition Edge to Test-Vector

direct manipulation of the events simulated. However, care must be taken not to force an illegal operation that would lead to a false error in the simulation. For example, in a processor design, one of the signals that would likely be controlled is an indication from the caches of a *hit* or *miss* of the last access. Forcing this signal to indicate a *hit* would cause a false error in the simulation if the simulated data was not in the cache. There are many examples of situations where the values of signals are constrained in similar ways. Many of these constraints can be accounted for by modeling them in the ND environment of the FSM model. This constrains the inputs to the FSM model with the result that the illegal inputs are never generated since they do not appear in the global state graph. When an input cannot be constrained in the ND environment, extra attention must be paid to setting up the simulation to not produce false errors. For the cache example just given, this can be achieved by pre-loading the cache and limiting each test so that information in the cache is never removed.

The other way to cause particular transitions to happen is to choose the data that is read by the simulation. An example is the simulated instruction in a processor design, which is provided as data to the Verilog simulation. Here, generation of an instruction stream provides the necessary control without direct Verilog intervention. The instructions are read by the simulation and executed. By choosing the sequence of instruction types that are executed, control over signals derived from the instruction can be achieved. With this method, some pseudo-random choices often need to be made. Recall that the ND environment model provides some abstraction of the interface signals. For example, the instructions of a processor can be grouped into equivalence classes that impact the control logic in the same way. Hence, during test generation, when it is time to choose the instruction that is to be used as data for the simulation, any member of an equivalence class can be chosen. Similarly, pieces of the data that do not affect the control logic can be randomly chosen.

In general, the inputs being generated by the FSM environment model need to be assigned values during simulation. During test vector generation, each of these input values must be assigned a value. This correspondence between values in the FSM model and actual wires in the simulation, or data to be simulated, is made in the *transition condition mapping*. This is a per-model file written with information about the design to give a meaningful representation of the test vector in RTL simulation, one that causes the desired

input values to appear at the right control inputs. The mapping also specifies how to choose the random parts of test patterns. A sample mapping file is shown in Figure 13.

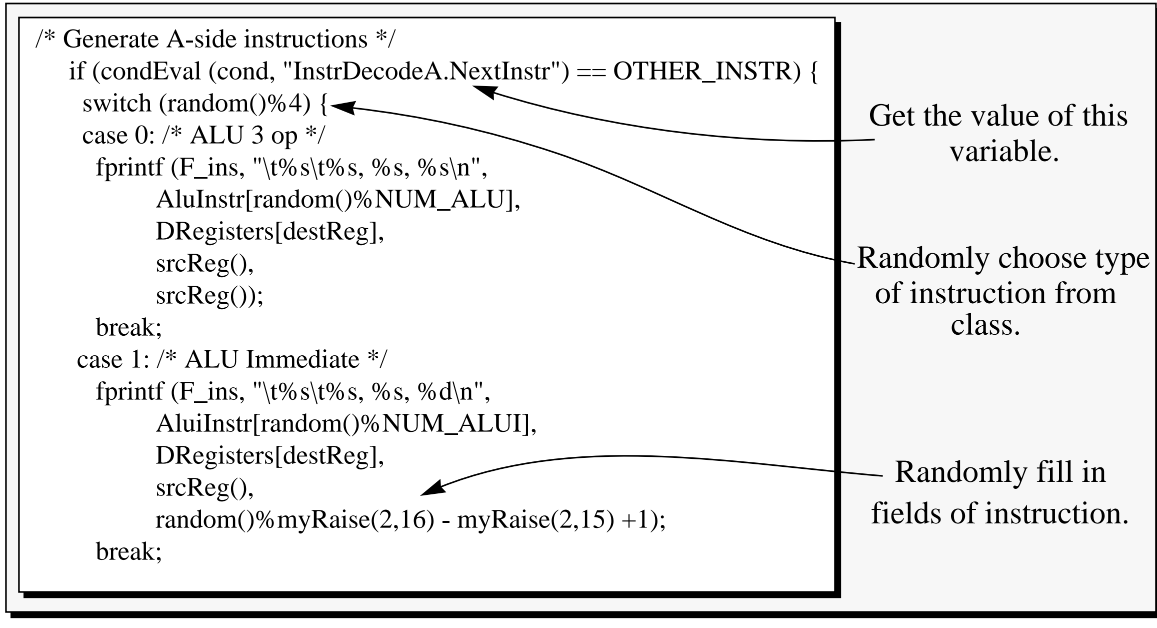

Figure 13. Sample Transition Condition Mapping

For this method of test vector generation to work well, the controlled signals must satisfy a correspondence requirement in the two models that will be co-simulated. That is, the signal must either exist in both models or it must have a simple mapping to appropriate signals in the two models. In the simple case where the signal exists in both models, then the same sequence of forced signal values is applied. In general, this works well for external inputs to a model, such as a databus or communication channel, or some other architectural feature that must appear in both models.

The other simple case is where the signal can be totally ignored in one model because it represents an implementation detail completely abstracted by the other model. An example of this is the cache miss signal of a processor. In an abstract instruction level simulator, a cache miss is treated as a nop (no operation) so the signal only needs to be applied to the detailed implementation model.

The third, and hardest, case is that a signal in one model needs to be represented by different signals in the other model. In this case, the mapping between values of signals in one model and the other needs to be given so that the two versions of test vectors can be

generated. In general, this can be difficult. The most common occurrence of this situation is when the two models to be co-simulated have different boundaries, that is, they enclose different amounts of logic from a design. In this case, the same signal may be present in both models but may be masked by additional logic in one. In particular, one situation where this may occur is when a golden model has been written that encapsulates additional functionality, for example, a design along with an interface bus and the arbiter of the bus. A likely scenario is that the FSM translation from the RTL would stop at the bus boundary of the design. Hence, test vectors would be generated at this interface, but the equivalent vectors for the golden model require additional abstraction to include the bus and arbiter.

Creating test vectors for a model with additional logic is possible by reasoning backward through the logic. The additional logic is incorporated into the transition condition mapping so that the values of signals needed to cause a particular FSM transition is determined in terms of inputs to the additional logic. If the additional logic is purely combinational, this is not too difficult. The harder situation is when there is sequential logic that must be reasoned through. In this case, the state must be modeled and tracked in the transition condition mapping code. This is practical only for small amounts of sequential logic. Fortunately, this is the case in most validation environments.

4.3 Undetectable Bugs

The ability of this technique to detect bugs in the design relies on two things. Firstly, the bugs must manifest as data value differences between the two models being co-simulated. Any behavior that is not modeled in the more abstract model (specification) cannot be checked for correctness. For example, if the specification is not cycle accurate, performance bugs in the implementation might not be found. In addition, errors common to both descriptions cannot be detected.

Secondly, the technique tries to expose bugs by exercising all control edges in the state transition graph. However, an implementation detail of the state enumeration tool leads to the possibility of missing test cases. During state enumeration, MPP only keeps the first set of non-deterministic variable values that results in a new transition edge. Any further sets of values resulting in the same edge is discarded in the standard algorithm. This gives space savings based on the implemented data structures but has the potential of

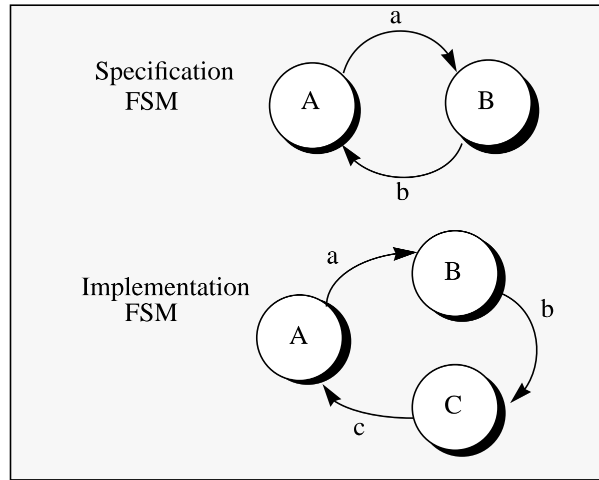

Figure 14. Erroneous FSM Implementation with more Behaviors

overlooking important tests. By enumerating on the implementation FSM, we capture a class of bugs where the implementation has more behaviors than the specification as demonstrated in Figure 14. When the “c” transition of the implementation is simulated, the difference with the specification is exposed and hopefully the error will be detected. However, there does exist a situation in which this methodology might fail to find a bug as shown in Figure 15. For this example, let us assume that the implementation erroneously performs the same state transition for both input “a” and “c”. However, in the state enumeration, each arc is labelled with the first condition leading to a new state, so either “a” or “c” will label the arc depending on which is tried first, but not both. If “a” labels the arc, then the wrong “c” transition will never be exercised to expose the bug. This case can be caught by performing the state enumeration on both the implementation FSM and an abstract model of the specification FSM. However, in many cases, there may not be an easily obtainable specification FSM for a general piece of logic.

This is an example of a more general issue: namely, whenever test vector generation is based on the implementation of a design, there is a chance that the implementation is missing some specified functionality. The most trivial example is when the implementation is completely missing a functional unit, for example the ALU shifter in a processor design. Test vectors generated from the implementation cannot detect that the shifter is missing, but if a shift instruction were to be simulated, this omission would quickly show up. This argument can be applied in both directions. If test vector generation was based on the specified behavior, then implementation corner-cases could not be created. To be abso-

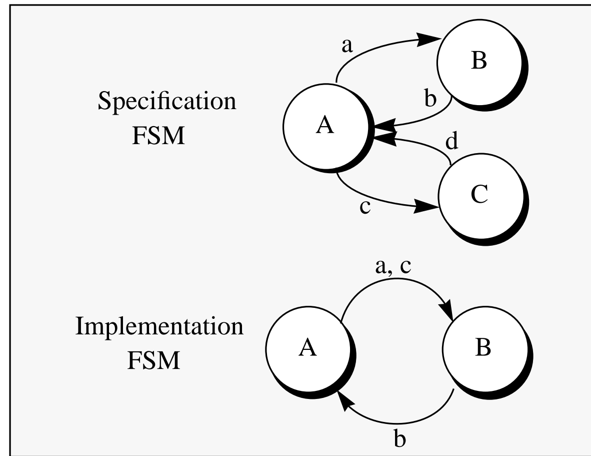

Figure 15. Erroneous FSM Implementation with fewer Behaviors

lutely sure, test vector generation would need to be performed from both directions. However, this is not generally practical as just mentioned because it can be difficult to automatically obtain an FSM description of the specification. The correct way to avoid this problem is to ensure a set of test vectors are also created that test basic functionality of the design. With this, confidence can be obtained that there are no major blocks of logic that are missing.

4.4 Stanford FLASH Memory Controller Example

This research was performed as part of the Stanford FLASH (FLexible Architecture for SHared memory) multiprocessor project [KOH+94]. The FLASH machine was designed to allow the two programming paradigms of shared-memory and message-passing to co-exist on the same hardware. In order to achieve this, a flexible memory controller, called MAGIC (Memory And General Interconnect Controller), was designed and built. MAGIC contains an embedded RISC-processor core; some interfaces that manage and queue requests from several sources including a network and a host processor; an internal scheduler and a DRAM controller. This is the circuit whose validation was the driving force for this work.

To optimize latency and throughput, MAGIC is split into specialized data handling and control logic sections. The control logic performs operations related to memory management, be it directory-based cache-coherence or cache-only-memory-architecture (COMA) or other protocols. These operations follow the same general sequence of

actions: dispatch based on the type of the incoming message; some updating of information; and possibly dispatch of a new message. This sequence forms the basis of a control macro-pipeline which corresponds to the major functional units of MAGIC as shown in Figure 16.

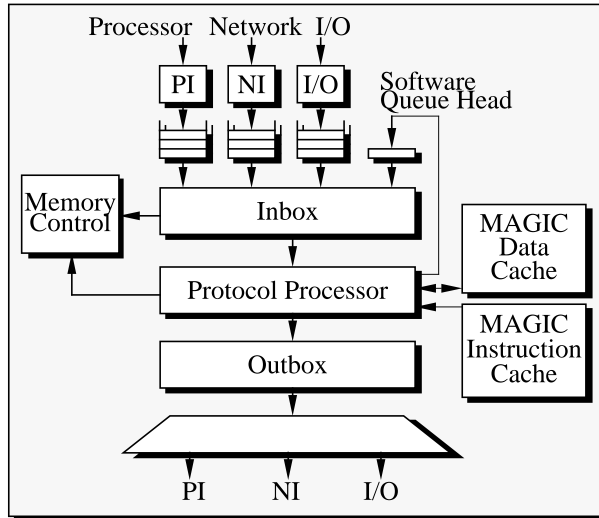

Figure 16. MAGIC Block Diagram

Arguably, the most complex component of the design is the embedded processor core, known as the Protocol Processor (PP). This is a 64-bit, dual-issue, DLX-based [HP90] processor with an aggressive memory system. The chip, which will be fabricated by LSI Logic using a semi-custom gate-array, will be about 250k gate-equivalents.

The Protocol Processor (PP) unit of MAGIC was used an example for this work. The results below were obtained using a version of the PP prior to final functionality freeze and timing optimizations. In general, processors have two classes of stimuli that affect control: the instruction stream and input signals from external sources. As long as both these sources of stimuli are controlled in the RTL simulation, the model should take the same state-transition arcs as the transition tour.

This work focused on modeling the memory system and the stall machine. Several abstractions were necessary to make the PP model state space small enough to handle. As previously mentioned, some natural abstractions existed, such as the 1-bit hit/miss signal for the caches that took the place of 32-bit addresses. Another natural abstraction was to group instructions into equivalence classes. The control logic takes in decoded signals that

represents types of operations, for example, all ALU operations result in the same control signal values. For this work with the PP model, 5 instruction classes were used, shown in Table 2.

Table 2. PP Instruction Classes

Instruction Class	Effect on Control Logic
ALU	Has no effect since there are no exceptions in the PP.
LD	Execution of a load can cause transitions in load/store FSMs.
SD	Execution of a store can cause transitions in load/store FSMs.
SWITCH	A switch instruction executed while the Inbox is not ready causes a pipeline stall.
SEND	A send instruction executed while the Outbox is not ready causes a pipeline stall.

Flow of control instructions, such as branches and jumps, are not explicitly modeled here. For this work, they were introduced randomly into the instruction stream with a known outcome. This gives a random testing of branches while testing all other interactions.

Other abstractions for the PP include the pipeline registers, the caches and the program counter. In this model, one of the properties of the design that helped to keep the state space small is the global stall mechanism. Whenever one of the major FSMs becomes active, a stall signal to the other FSMs keep most of them idle until the operation is complete. The abstract model is shown in Figure 17. In this figure, the functionality that was abstracted is shown as rectangular boxes whereas the FSMs that were modeled are shown as ovals. Although this figure looks complex, the important features are that there are a small number of interacting FSMs (6), with a handful of constrained abstract components generated in the ND-environment. The remaining inputs to the FSM model were left unconstrained. The environment did not require much effort to create and did not need substantive changes while the RTL progressed towards completion.

4.4.1 Protocol Processor Test Generation

This abstract FSM model was run through the tool-set. The state space exploration results are shown in Table 3. One interesting fact to note is that although 98 bits of state

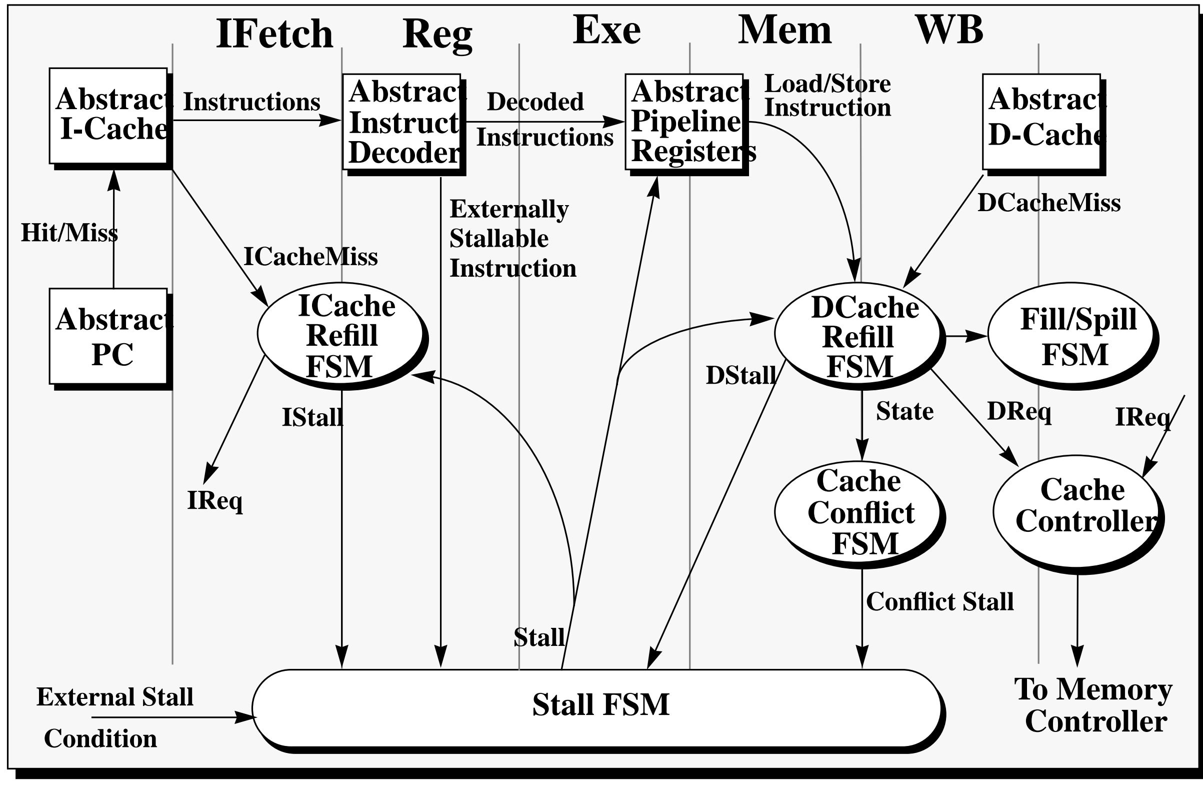

Figure 17. PP Abstract FSM model

Table 3. PP State Enumeration Results

Number of States	229,571
Number of bits per State	98
Execution Time (on DecStation 5000/240)	5.1 cpu hours
Memory Requirement	34 MB
Number of Edges in State Graph	1,172,848

were necessary to hold the state vector, only about 2^{18} states were actually reachable from reset. One of the reasons for this was that the FSMs interlocked to a great extent. In other words, when one FSM was active, many of the others were held in their previous states until that operation was complete.

The results of transition tour generation are shown in Table 4. These results show that simulating the complete set of vectors is achievable and on average a modest number of instructions (7) is needed to test each arc. In addition, breaking up traces into smaller com-

Table 4. Test Vector Generation Statistics

	With no limit	With trace limit (10,000 instructions)
Number of Traces Generated	1,296	1,296
Total number of edge traversals generated	21,200,173	21,252,235
Total number of instructions generated	8,521,468	8,557,660
Generation time (on DecStation 5000/240)	44.8 cpu hours	53.7 cpu hours
Estimated simulation time @ 100Hz (total)	58.9 hours	59.0 hours
Longest Single Trace	21,197,977 edges	144,520 edges
Estimated simulation time @ 100Hz (longest trace)	58.9 hours	24 mins.

ponents does not add much overhead, which is fortunate since it is extremely helpful in reducing the time needed to rerun a simulation to reach a bug.

It might seem strange at first that the same number of traces were generated in both cases. It turns out that the PP model has numerous edges in the graph that can only be reached from reset. These edges represent different initial conditions for the inputs to the model and the traces needed to reach these edges cannot be combined with others. This gives us the lower bound on the number of traces needed to cover all edges in the model (1,296 for the PP model). Without an instruction limit, over 99% of the instructions were generated in the first trace and the remaining 1295 short traces were needed to cover different initial conditions. With the instruction limit, many of the edges in the state graph that were originally covered by trace 1 are spread out among later traces. A total of 853 traces were terminated due to the 10,000 instruction limit which suggests that it took 854 traces to cover most of the edges originally covered in the first trace. The remaining 442 traces covered the remaining initial conditions.

4.4.2 Protocol Processor Bugs

The automatically generated test vectors were used to drive the RTL model of the PP and the resulting register file values were compared to an instruction-level simulator. Several non-trivial bugs were discovered that were not uncovered using hand-written or ran-

domly generated test vectors. The test vectors were run on the PP Verilog model after it had been partitioned for synthesis and was considered fairly mature. All bugs that were found using other methods were found using this technique. In addition, Table 5 shows

Table 5. Synopsis of Discovered Bugs

Bug	Description (Summary of Bug followed by Explanation)
1	Interface miscommunication between PP's cache controller and the Memory Controller. Qualification of an interface signal was needed, but the two units thought that the other would perform it. The bug manifested itself as incorrect data being returned to the I-Cache.
2	Latch not qualified on all stall conditions and lost data. On a simultaneous I & D Cache miss, the latch holding the data that was to be returned after the D-Cache refill was not qualified on the I-Stall and lost its data by the time the I-Cache miss was serviced.
3	Cache conflict stall can cause wrong address to be used on the stalled load. The address used in the load of a conflict stall was not held during the stall. If there was no following instruction that used the address bus of the cache (any load/store instruction), then the correct address from the load remained. However, if the load in the conflict stall was followed by another load/store instruction, then the address of the following load/store was erroneously used.
4	I-Stall fix-up cycle lost if I-Stall condition occurs during Mem-Stall. The I-Cache refill machine takes a cycle to restore the correct values to the instruction registers after an I-Stall. However, it was not qualified on MemStall, so was lost if the I-Stall condition arose after MemStall was asserted. This can happen if a `switch` or `send` is executing in the stalled instruction and the external unit (Inbox or Outbox) signals the PP to wait.
5	Glitch on bus valid signal allows `Z` values to be latched on a load that missed followed by any other load/store instruction interrupted by an external stall condition. This bug is explained in detail in the text.
6	Cache conflict stall with D-Cache hit and simultaneous I-stall results in stale data being loaded. A cache conflict stall occurs because of the split store operation. When the address of the load following a store is the same as the store, a conflict stall is taken to write out the store data before loading it. When there is a simultaneous I-stall caused by an external condition, the load receives the stale data instead of the newly written data.

some bugs found using the generated vectors but not found by other methods over the course of a couple of months of testing and debugging both software and the design.

Bug #5 is illustrative of the improbable interactions that are involved in some bugs which make them hard to detect. The situation leading to the bug was a load that missed in the data cache followed by any other load or store instruction. The first load required retrieval of a line from memory to cache. With critical-word-first restart, the first word returned from memory was driven onto a bus (Membus) leading to the register file. The bug existed as a glitch on the signal that indicated valid data on the Membus, as shown in

Figure 18. The glitch was caused by the presence of a load or store instruction in the pipe

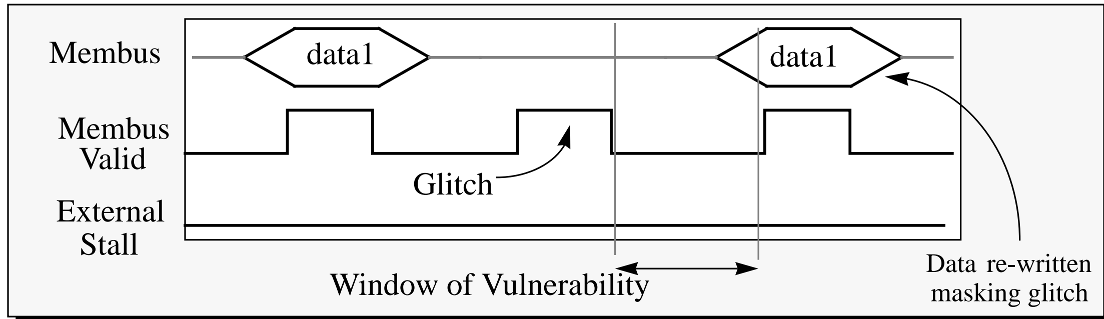

Figure 18. Bug #5 Timing Diagram (Glitch Masked)

following the load that missed and it occurred after the critical word was driven, thereby overwriting it with potential garbage because the bus is at high impedance at this time. However, the logic which implemented the refill was still erroneously implementing an older restart policy. So, it then drove the required data onto Membus a second time, giving the correct result to the register file. This in itself is a performance bug which result comparison does not find. The correctness bug exists only if an external stall condition arose between the time of the glitch and the second write, preventing the second write from occurring, leaving garbage in the register file, as shown in the timing diagram of Figure 18. In actual hardware, it would have occasionally shown up as corrupted data and its

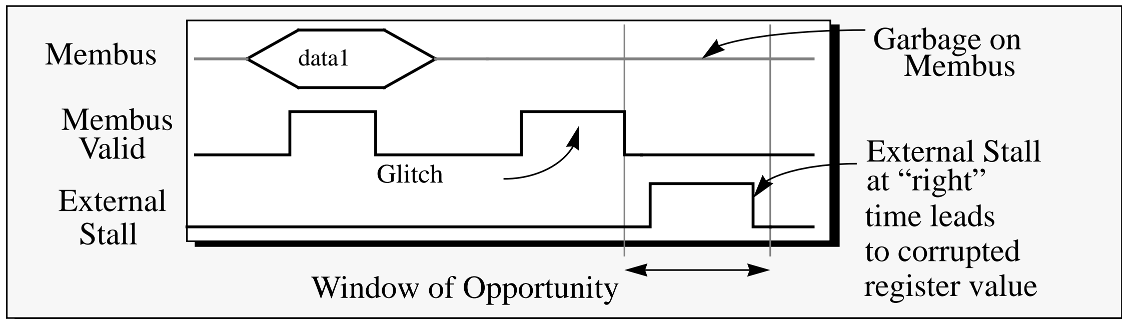

Figure 19. Bug #5 Timing Diagram (Garbage written)

reproducibility would have been limited because of the requirement of a stall arising from another asynchronously operating unit.

This is an example of the types of bugs that are the result of a series of interactions in the control logic that may be overlooked using traditional validation methods. Easily generating this rare interaction is the advantage of using control interactions modeled as FSM.

4.5 Related Work

At the very highest level, using a state graph to map out the testing requirements of a design is not new. This is utilized heavily in the field on *network protocol conformance testing*. Network protocol conformance testing addresses the question of whether an implementation of a network protocol actually satisfies the specification. Quite often, the implementation is provided as a black-box and conformance is tested through a suite of test vectors and expected responses. The protocol will often contain a substantial number of states, all of which must be exercised. To make this task even harder, there is usually limited observability of internal states and controllability is through the external inputs only.

Like the test vector generation technique described in this chapter, network protocol conformance testing uses the idea of a *chinese postman tour* as defined in [EJ73]. One problem encountered in network protocol conformance testing that is not faced in design validation is the *observability* and *controllability* constraints. Special sequences of *unique inputs/outputs* (UIO) need to be developed to determine exactly what state transition was taken for a particular test vector in the implementation [ADL+91]. This is generally not necessary for design validation since we have access to the internal state of the simulators used for validation. However, these techniques may prove useful if validation tests were to be used for conformance testing of hardware parts. Further details and a good overview of how state space enumeration techniques used for protocol testing can be found in [Hol91].

4.6 Discussion

The limiting constraint of this approach is the size of the global state space of the design. In order for the transition tours to be created, the full state space must be obtained first. Our experience with FLASH indicates that as designs mature, the size of the state space grows. This is normally the result of timing requirements: more logic is introduced to pre-compute results, logic is modified for optimal timing and additional registers are

introduced to break timing paths. In most practical designs, the state space quickly grows large. The abstractions used in this chapter can postpone the point when the state space becomes unmanageable, but is not a general solution. Instead, a possible solution is to more closely link the state space enumeration, test generation and simulation so that parts of the state space can be exercised without requiring that the entire state space be found first. In the end, the goal is to validate the design and find bugs. Even if test generation in this manner cannot achieve complete coverage because the state space is too large, it may still be able to uncover errors in parts of the design that might be hard to test with any other technique.

A general limitation of this test vector generation technique that was not encountered to any great extent is the issue of controlling internal nodes of the design. For this technique to work well, it must be possible to take over a signal in simulation without catastrophic results. If the protocols on the interfaces were more complex, the test generation algorithm may not be as straightforward, or the protocols might need to be programmed into the transition condition mapping. The features of a design that make it amenable to this technique are probably the same as those that would make it suitable for pseudo-random testing.

This test generation process can actually be viewed as guided random generation. At each cycle, the algorithm looks for a new path to take that covers a state transition edge that had not previously been taken. If there are multiple candidates, a random choice is made. Also, for many of the abstractions, a choice needs to be made about the particular signal(s) or data from the equivalence class. The consequence of this random generation is that the test vectors require an automatic correctness check, such as a *golden model* simulator. This works well for some models which have a simulation model which is naturally more abstract than the implementation model, such as processors. However, there are many pieces of logic that depend so much on particular signal timings that the task of creating a second simulation model of the design equates to re-writing the RTL.

In the FLASH project, an attempt was made to create an abstract model of the *Inbox* unit to be used as a *golden model*. Our hope was that it would be possible to create a model that could accept a sequence of inputs and produce the same outputs without creating a completely cycle-accurate model. It turns out that within the *Inbox* unit, there is some arbitration between inputs and a general purpose bus which could change internal

state of the unit. In order to produce the same output, the abstract model needed such a degree of cycle-accuracy, that it reproduced the Verilog RTL and in fact became based on it. This contamination makes the *golden model* less effective or even useless as a correctness checker. For designs such as this, the effort of creating the second model may overwhelm the benefit it buys and instead of random test generation, coverage analysis of self-checking tests should be used, as described in the next chapter.

Chapter 5. Validation Coverage Analysis

The previous chapter discussed using control interaction information from state enumeration to generate test vectors. Test vector generation works well when a design has two models that can be co-simulated with the same vectors in order to check correctness. However, there are many designs or components of a design where this is not feasible. In these cases, the control interaction information can still be utilized to provide coverage analysis to guide creation of self-checking test vectors.

The coverage measure described in this chapter is based on state space reachability analysis. The basis for this metric is that bugs in a design cannot be discovered if the control paths leading to it are not exercised in simulation. By reporting a coverage measure based on the state graph of the control logic, the likelihood of creating test vectors that encounter bugs which are the result of rare interactions, should be much higher. The problem when this was tried in practice is that, very often, the coverage reported is very low. In this situation, designers have difficulty relating the coverage numbers to missing tests in the test vector suite. What is required is an incremental infusion of coverage information so that attention can be focussed on the simplest missing tests initially, before progressing to the more complex ones. In fact, it is extremely difficult, if not impossible, to provide a single definitive number that can indicate when validation is complete. Each individual or combination of FSMs has a coverage measure that conveys valuable information to designers. And at the core of validation, the really useful information comes from finding missing test cases rather than reporting numbers which require numerous footnotes to

explain their exact meaning. There are just too many ways to reduce and remove information that significantly change the coverage number reported. With this in mind, an approach is described in this chapter which tries to focus on identifying the useful information. Often this means taking smaller pieces of the coverage measure and making these high before taking the next piece.

The basic coverage metric is obtained simply by logging the state transitions observed in the RTL model during simulation with test vectors. The test vectors can be generated by pseudo-random techniques or be hand-written directed tests. The coverage is then represented as a state-coverage metric and an edge-coverage metric:

$$\boldsymbol{SCM} \text{ (State Coverage Metric)} = \frac{\text{Number of States Visited in Tests}}{\text{Total Reachable States in State Graph}}$$

$$\boldsymbol{ECM} \text{ (Edge Coverage Metric)} = \frac{\text{Number of Edges Visited in Tests}}{\text{Total Reachable Edges in State Graph}}$$

5.1 Graph Redundancies

This state space calculation of coverage is more useful than more simplistic measures, such as node-toggle coverage because it takes interactions into account. However, it often produces an excessive testing requirement for the design. Taking the state graph from a model that has been directly translated from RTL turns out to have a great number of redundancies, which are states that do not necessarily need to be tested, resulting in low coverage numbers. This in itself is not a bad thing, but the consequence is that the analysis points out a very large number of state interactions that may not have been tested. If many of the tests are redundant, that is, they have little potential for finding bugs, then the task of finding the truly missed and needed tests among the large number of unneeded tests becomes a hurdle to getting useful feedback from the analysis.

Detailed examination of the FLASH design examples revealed three sources of state graph redundancies. Firstly, the structure of the RTL code leads to some bits being *don't*

care values in certain situations. The value of these bits does not affect the next state and they can be used to reduce the size of the state graph. Secondly, an important property of test vectors is that they attempt to exercise conflicts for resources in the design, and bugs arise when a resource conflict actually occurs. When considered in terms of their ability to exercise different resource requirements, many sequences through the state graph result in the same testing potential. In other words, they exercise the same resource contentions. These can be coalesced to reduce the testing obligation that the coverage analysis imposes on the design. Thirdly, in many cases, the environment of the FSM, which is really the input generator, is too general and results in illegal states being reached in the FSM model. These illegal sequences are prevented from occurring in the real hardware by constraints on other components, so the real hardware cannot reach some of the states which eventually get reported as uncovered by the analysis. In these instances, the environment needs to be constrained to produce legal sequences only.

Together, these three problems conspire to produce a mountain of interaction tests that are reported by the straight-forward state graph coverage analysis. Buried among these are the tests that have high potential for discovering bugs. To find these important tests, each of these three problems need to be addressed and a solution found that removes the redundant information. Each problem is discussed further in the following sections and solutions proposed, with supporting data from the FLASH examples. One pragmatic conclusion of this work with the FLASH design was that incremental coverage information is the most valuable. Not only does it allow the designer to deal with smaller amounts of information at a time, it also aids in identifying redundancies early, when they are easier to spot.

5.2 Graph pruning using Static Analysis for Don't Cares

In many large designs, it is often true that portions of the state space are equivalent, meaning that pairs or groups of states can be represented by a single state without loss of information. Many techniques [KVB+95] have been proposed to find such equivalences and hence reduce the state space. Most of these have used the original graph as a starting point for finding equivalences, making them ineffective at dealing with the state explosion problem. However, when dealing with state graphs derived automatically from RTL descriptions, many equivalences can be traced back directly to the structure of the RTL. As a consequence of the logic structure, some sections of the resulting state graph will

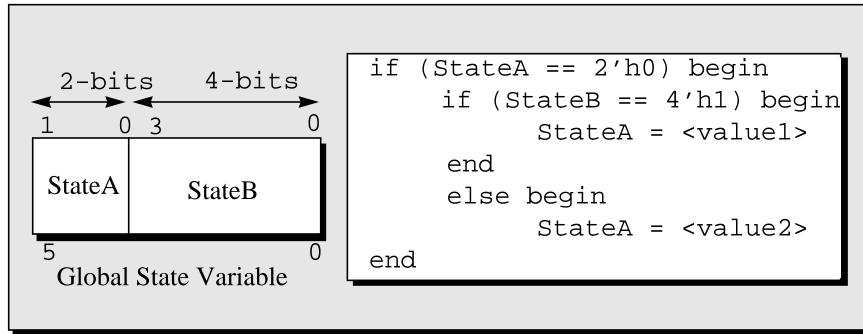

Figure 20. Don't Care variable due to code structure

inevitably be equivalent. For example, in Figure 20 we have a global state variable that is composed of two component variables and we show a portion of the RTL code that sets the next state. Assume *StateB* is set every cycle from an input. In this situation, the value of *StateB* is irrelevant when *StateA* is not equal to zero and can always safely be set to zero without losing any information. This is true since *StateB* is not even looked at when *StateA* is not zero and since *StateB* is set every cycle from the input, the next global state is the same no matter what value *StateB* is. So the sixteen possible values of *StateB* can be coalesced into a single representative state, reducing the overall state space.

This minimization can be done with any RTL construct where evaluation of a state variable is dependent on the value of a different variable and it can be shown that the state variable will receive a new value before it can next be used. This is analogous to the kill set in compilers [ASU88], where a register is considered *dead*, or unused, after it is last read and before it has a new value written to it. For state space exploration, different values in that dead register show up redundantly as multiple states in the state graph, whereas it is sufficient to just zero it without losing any information.

It would be possible to analyze the RTL structure very carefully to figure out dependencies and generate a complete kill set for each variable. However, to be of real help in managing state explosion, pruning the state graph must occur dynamically and with minimum overhead. Hence, we introduce slightly stronger constraints on the kill set to make recognition of pruning situations easy. Instead of doing multi-cycle analysis to figure out when a variable is written, we will impose the constraint that the variable to be pruned must be written every cycle. With this restriction, it is only necessary to determine if the variable to be pruned is read on any particular cycle. If not, it can be zeroed for that cycle.

5.2.1 Static Analysis of Kill Sets

Analysis of the RTL for kill sets occurs once the state variables in the design have been determined. For each of the state variables, we look for structures where the variable would not get read. A partial list of these structures is given in Figure 21. For every vari-

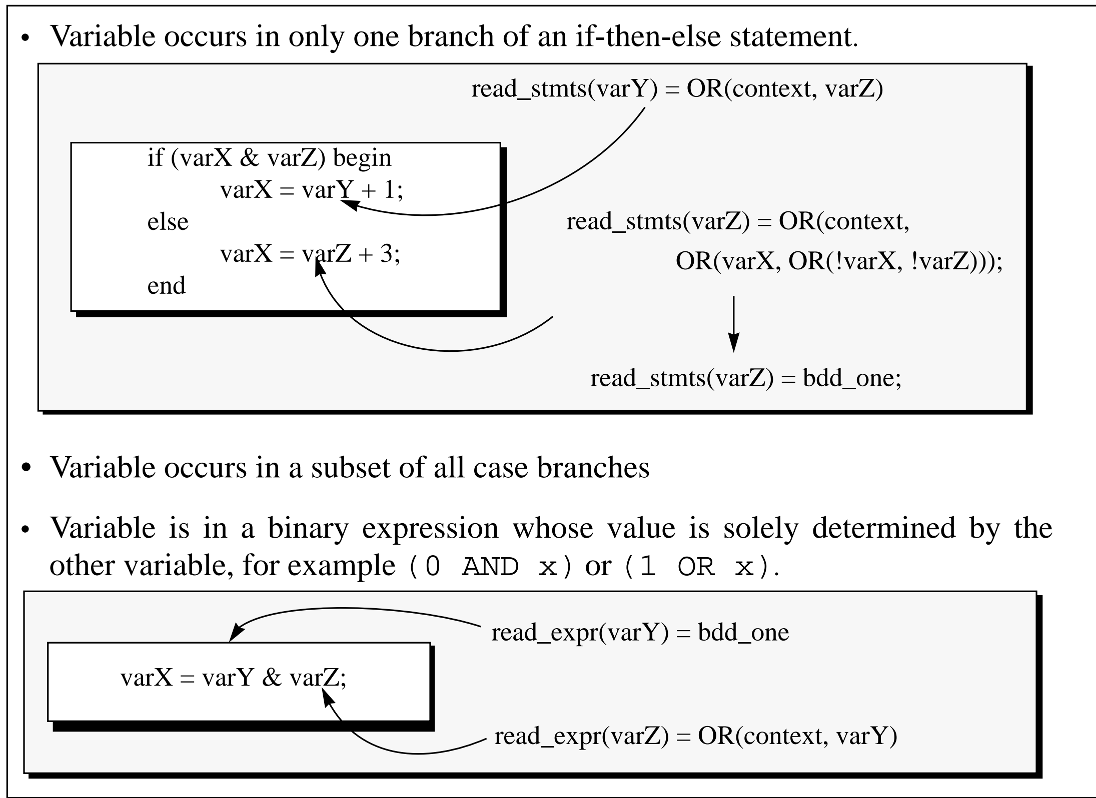

Figure 21. List of RTL Structures where a Kill-Set may be possible

able that is not read on occasion, we check the RTL to see if that variable gets a new value on every cycle. If so, we can construct a binary decision diagram (BDD) to represent the set of conditions where it can be ignored. The BDD simply encodes the values of the other state variables that allow us to treat the variable being considered a *don't care* value. The example in Figure 22 demonstrates this analysis.

The BDD in Figure 22 (b) represents the contexts in which *StateC* can be ignored, based on the RTL shown in Figure 22 (a). This basically says that *StateC* can be ignored unless *StateA* takes the value 2 and *StateB* takes the value 1. In all other situations, *StateC* is a *don't care*. This analysis is done for every state variable independently, except in the

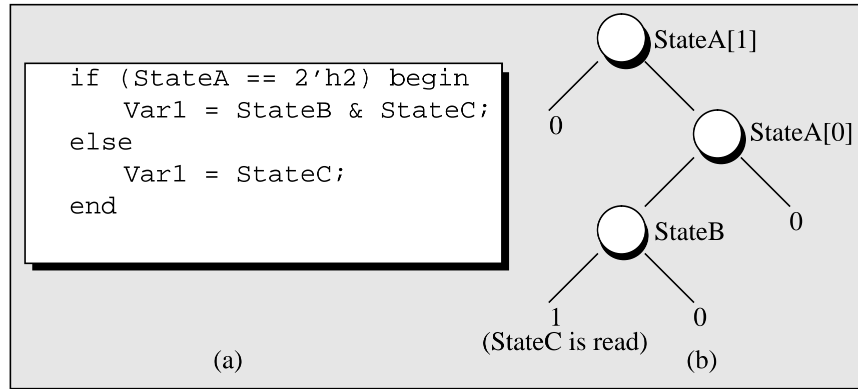

Figure 22. Static Analysis for Kill Sets

case of binary expressions, where one must not turn both arguments into a *don't care* on the basis that the other will determine the value of the expression. A BDD is created for each variable, which is then used in the dynamic pruning.

To keep the dynamic portion of the algorithm simple, hence minimizing its overhead, the variables that form the support of the BDDs of the kill sets need to be state variables. The support of the BDD is the set of variables that compose the BDD. In other words, the value of the BDD when evaluated depends on these variables. If any of these variables are not state, they need to be iteratively replaced until only state variables form the support. This allows the dynamic algorithm to operate based on the value of the current state alone, without requiring any information about the transition function. Otherwise, the pruning algorithm would be required to evaluate portions of the state function for each state, which is a slow operation.

5.2.2 Dynamic Pruning with Don't Cares

Once the kill sets have been determined for every state variable, we simply modify the state enumeration tool to check the kill sets whenever it finds a state and is about to check its hash table. The algorithm, shown in Figure 23, checks the current state to see if it is one of the situations in which a state variable is a *don't care*. If so, it zeroes out that variable before the hash table lookup, making all states that differ only in that state variable equivalent. As shown in the figure, this algorithm is quadratic in the number of bits in the state variable. This overhead plays an increasingly larger part of the running time as the number of bits increase. For some models, the state space savings will compensate for the overhead to reduce the running time. In other models, this overhead will be much larger

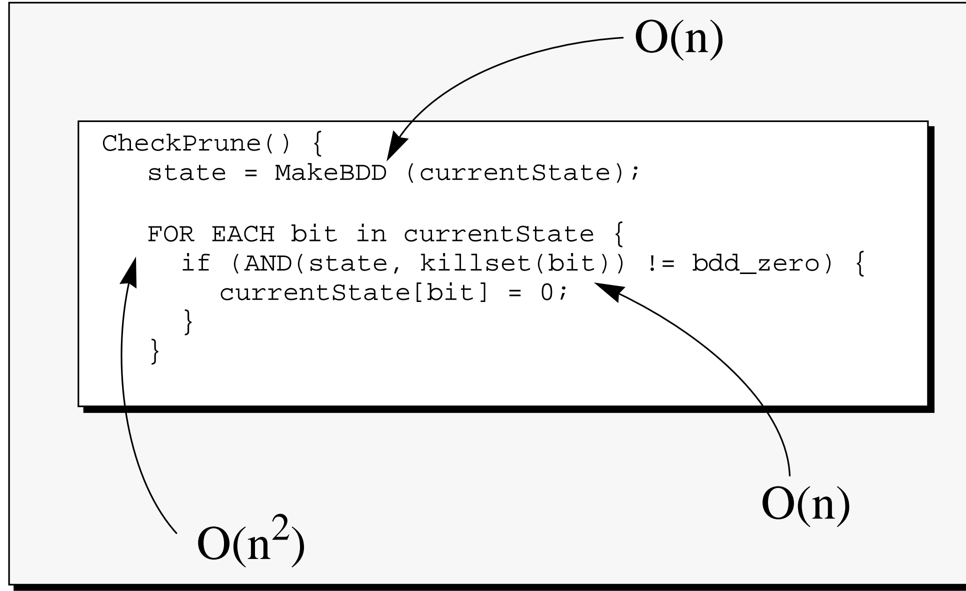

Figure 23. Dynamic Pruning Algorithm

than the running time of the unpruned model. However, running time alone is not the deciding factor in determining the benefit of pruning. By reducing the state space, pruning serves to focus attention on the core states of the model that need to be tested. This issue will be revisited later in this section with the benefit of empirical data.

Even with the more stringent conditions for kill set creation that we imposed for efficiency reasons, we found that this dynamic pruning could provide good results with examples from the FLASH project, shown in Table 6.

Table 6. Dynamic Pruning Results (5MByte Hash Table)

Unit of Design	State Bits	Input Patterns	Full Graph Size	Size with Dynamic Pruning	Reduction	Full Graph Running Time	Pruned Graph Running Time
Protocol Processor	120	1,536	36,826	27,258	26.0%	1,229.4s	7,712.9s
InstrFetch	50	3,072	193,320	159,530	17.5%	9,482.5s	13,111.8s
PP LoadStore	72	128	Space Out >277,570	244,937	>11.8%	24,556.0s (6.8 hrs.)	53,039.2s (14.7 hrs.)
Inbox	24	524,288	4,960	484	90.2%	831.3s	93.7s
Outbox	10	65,536	164	70	57.3%	274.8s	112.7s
IO	86	$> 2^{32}$	4,896	3,209	34.5%	917.2s	4449.8s

These results are also plotted as graphs to illustrate the relative state space reductions, shown in Figure 24 and as the relative running times, shown in Figure 25.

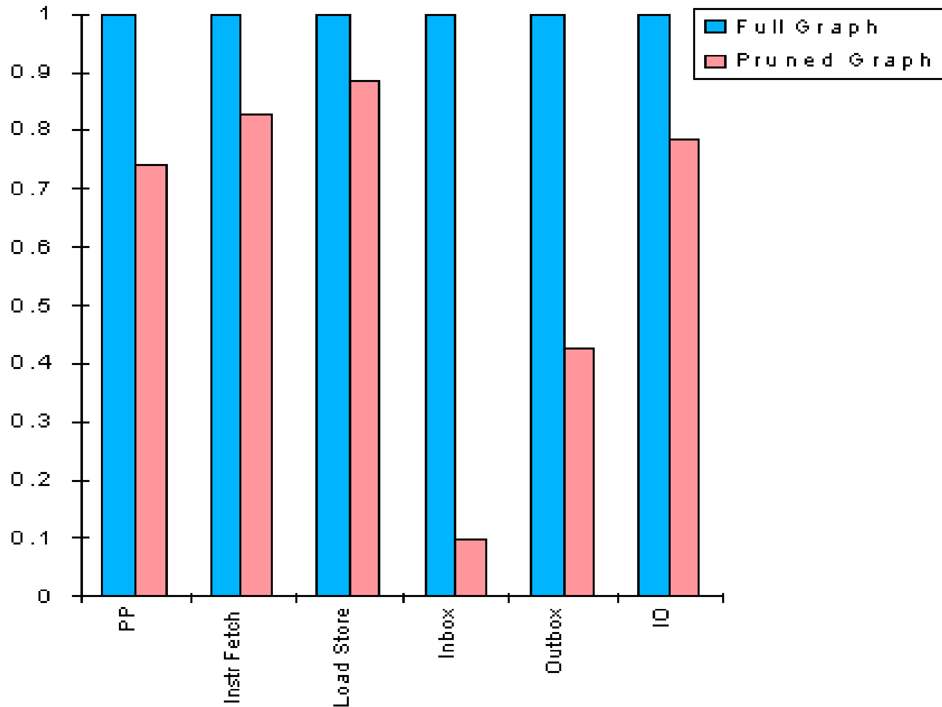

Figure 24. Relative Pruned State-Space Size

These results show that for some models, such as the *Inbox*, pruning resulted in a substantial decrease in the state space size and running time. Clearly, this is a case where the state space reduction offset the running-time overhead and came out ahead. In the median case of the *Instruction Fetch* unit, we see a more modest decrease in the state space, but this is not sufficient to offset the increased running time. In the worse case, the *Protocol Processor* unit, we observe only a modest savings from pruning and incur a large runtime overhead cost.

One encouraging observation from these results is that there was some pruning reduction in all units. This suggests that kill sets arise in RTL somewhat independently of the coding style since these four units represent four different designers. It is also important to note that it is acceptable to incur a slightly higher running time during state space exploration to secure a smaller state graph. The smaller state graph will have fewer redundancies leading to higher quality validation results based on it. However, it would be preferable to

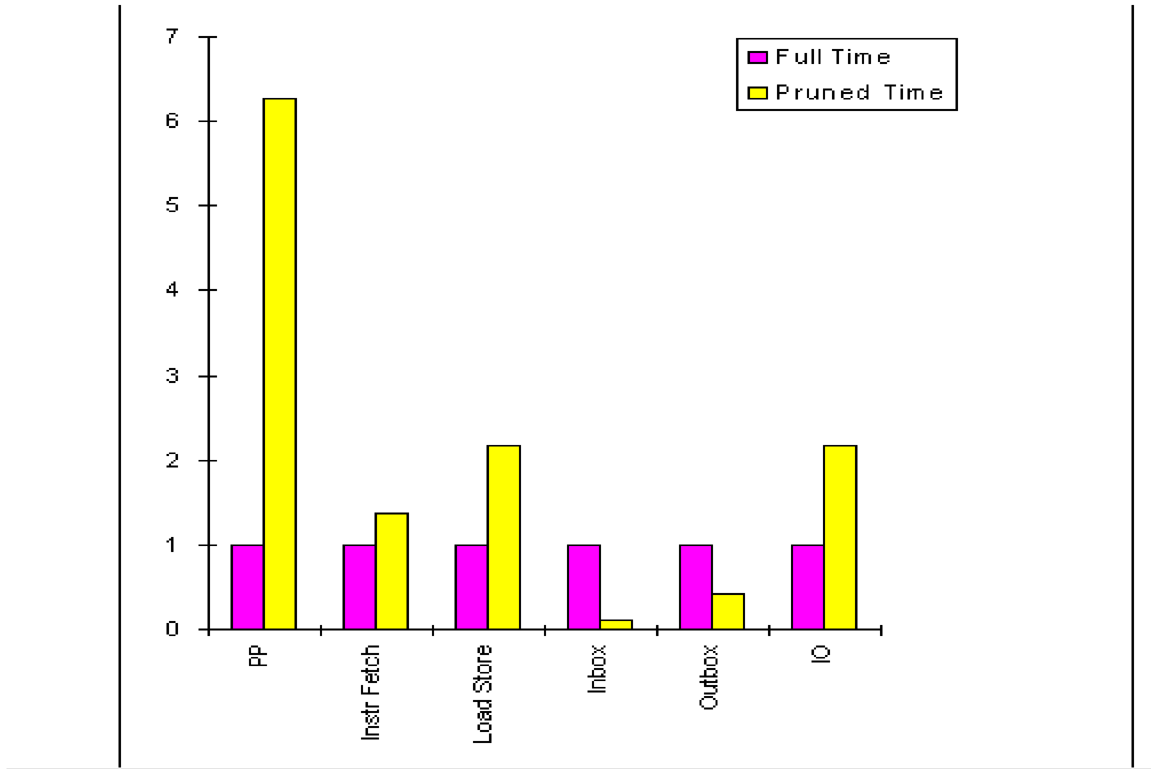

Figure 25. Relative Running Times

know the degree of reduction possible and the expected overhead for a model prior to running it. In some cases, pruning may be too costly in terms of runtime overhead and expected to achieve only a small state space reduction. The *PP* model shown above falls into this category.

There are two factors that affect how much state space reduction pruning achieves. The first is the *pruning potential* of a design. This is the amount of reduction that might be reasonably expected given the number of kill sets that have been found. The size of the kill sets is determined by how often the design sets up situations in which a don't care state bit occurs. This can be measured to some degree by the number of state variables for which the kill set is non-empty. This information is available after the translation analysis

step, before the state enumeration step. Table 7 lists the percentages of state variables that

Table 7. State variable kill sets comparison

Unit	Total State-Variables	with Kill-Sets	% with Kill Sets	% State Reduction Observed
Protocol Processor	85	17	20.0%	26.0%
PP Instr. Fetch	28	7	25.0%	17.5%
PP LoadStore	57	10	17.5%	>11.8%
Inbox	22	16	72.7%	90.2%
Outbox	7	4	57.1%	57.3%
IO	57	11	19.3%	34.5%

have non-empty kill sets for the FLASH examples. The data indicates a strong correlation between the number of state variables with a non-empty kill set and the amount of state reduction that can be expected. The correlation data is also presented in graphical form in Figure 26.

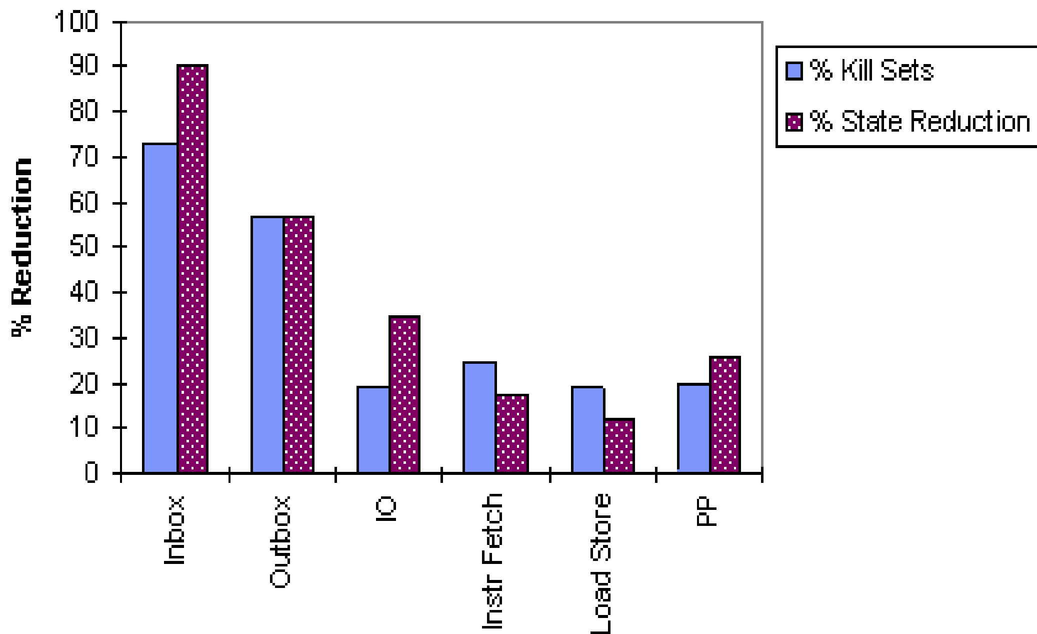

Figure 26. Correlation of kill sets and state reduction

Although the correlation is not perfect, for the three units that could be explored with all possible inputs (*Inbox*, *Outbox* and *IO*), the amount of state reduction was at least as much as the percentage of state variables with a kill set. For the *PP*, *InstructionFetch* and *LoadStore* units, only a fraction of the possible inputs were used in order to keep the state space manageable. In these cases, in particular the *InstructionFetch* and *LoadStore* cases, the full reduction potential of pruning does not seem to be realized.

We would expect the runtime overhead of dynamic pruning to track the $O(n^2)$ cost of the algorithm. The growth in the relative overheads is plotted in Figure 27. The numbers

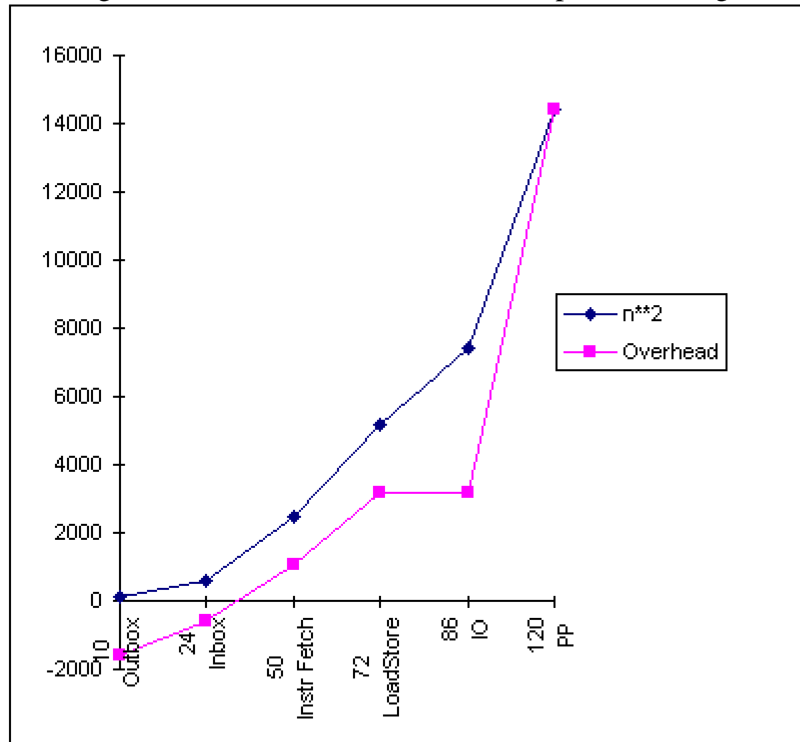

Figure 27. Growth of Relative Overhead

in this graph were normalized to the PP model for comparison purposes. The outlying data point for the IO was due to a particularly small state-space given the relatively large number of state bits. This seems to be a result of the IO design. This graph shows the expected quadratic running time of the algorithm as predicted in Figure 23. However, the more important fact is that it is possible *a priori* to guess the likely state space savings and the anticipated runtime overhead of using dynamic pruning immediately after the static analysis phase. This provides a method of deciding whether to proceed with dynamic pruning or not for a particular design.

From the empirical data of six design models, it seems that dynamic graph pruning has the potential to find and remove a substantial number of equivalent states in the state graph. Intuitively, its effectiveness depends on the number of times a kill set is activated in the course of state enumeration. Consequently, if some of the inputs to the model are held at fixed values, the amount of pruning observed drops, sometimes reaching the point where no reduction occurs. The explanation for this is that the fixed inputs prevent the graph search reaching the situations where the kill sets get activated. However, when all the inputs are allowed to take all values, the data presented indicates that there is, to a first degree, a correlation between the number of kill sets discovered in static analysis and the number of times a kill set gets activated. This is probably because *don't care* situations in the RTL code are randomly distributed.

The correlation between the number of state bits in the model and the runtime overhead can be traced back to the dynamic algorithm. Together, these two properties give a good indication of the amount of state savings and the overhead that can be expected for a given FSM model. These can be used to decide after the static analysis but prior to the state space exploration whether pruning should be utilized. This is important since the state space exploration is the more time-consuming process and a good educated guess about the outcome of pruning can optimize the use of computing resources.

5.3 Control Events

The previous section talked about the first of three sources of redundancies in the control state graph that acted to dilute the coverage measurement, making it harder to extract the important information. The second source of redundancies in the state graph arise when the graph is examined with validation in mind. With this viewpoint, some of the testing requirements of the full state graph turn out to be equivalent.

Since the *full* control state graph of a design is a comprehensive representation of its control behaviors, the natural procedure would be to require that every edge in the graph is exercised. However, our experience in using control state graphs for coverage analysis has been that it is often difficult to obtain test vectors from designers and test writers that exercise every transition edge, except for the simplest of circuits. One reason is that in many designs, not all control transitions that appear in the full control state graph need to be exercised to fully test functionality. A typical example is shown in Figure 28.

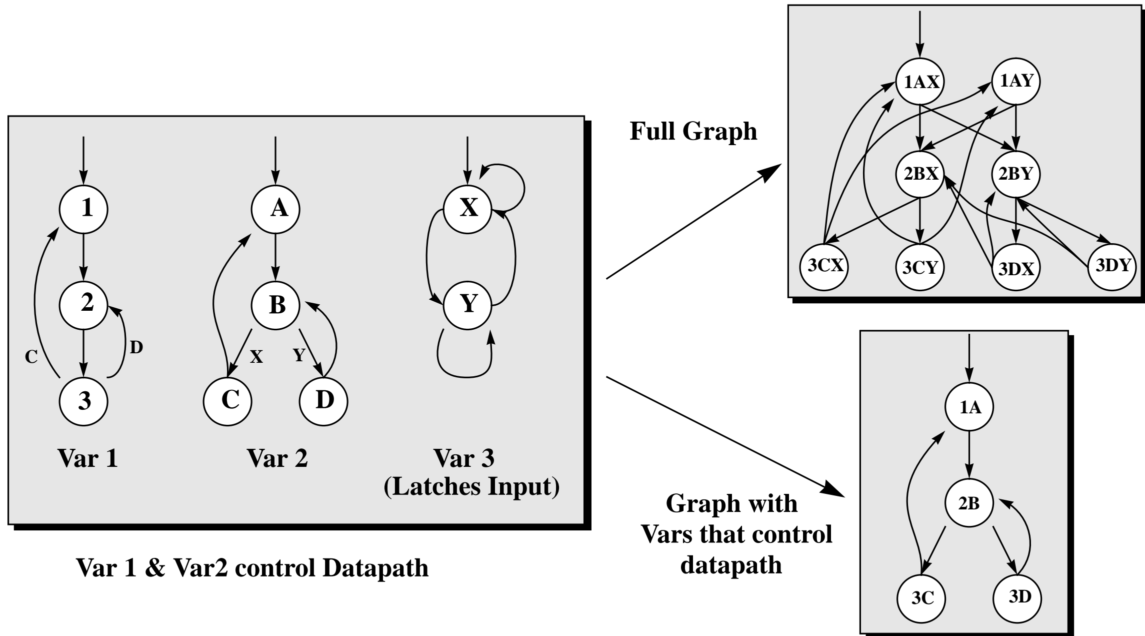

Figure 28. Example of State Graph Redundancy

In this example, there are three variables, two of which control the datapath (*Var1* and *Var2*), with the third read only by the FSM representing *Var2*. If we take the full control state graph, we have 8 states and 16 edges that need to be exercised in tests. However, some of these edges represent redundant tests. For example, the edge $2BX \rightarrow 3CX$ is equivalent to $2BX \rightarrow 3CY$ from a testing viewpoint since the datapath observes the same sets of commands from both edges. *Var3* can be ignored since its contribution to the behavior of the design has been made *explicit* by the state space exploration that composed the individual FSMs. It does not directly control the datapath, so it is sufficient to exercise the edge $2B \rightarrow 3C$. This simply says that one or the other of those edges need to be tested, but not necessarily both. By applying this principle to the entire state graph, we obtain the reduced graph shown in the lower right of Figure 28. This graph contains only the variables that directly or through some combinational logic control the datapath, called *datapath control variables*, and has just 4 states and 5 edges that need to be exercised.

We can generalize this observation by redefining the important tests in terms of possible *control events*.

Definition:

*A **control event** is an equivalence class of states which have identical values for the datapath control variables.*

In other words, a control event represents a particular set of commands to the datapath. Intuitively, this gives a better measure of an interesting event than a simple cross-product of all control state variables since it takes into account which variables actually control the datapath actions and focuses our attention on the control-datapath interface.

5.3.1 Coverage Property

Using the control event graph allows us to focus on the interface between the control logic and the datapath. It makes the assumption that the sequence of operations seen by the datapath is the important information, not the timing of signals to the datapath. For example, if the datapath is capable of tracking the number of occurrences it sees of a particular control signal and taking action only after it sees some number, then this sequence-only assumption is violated. In other words, control events assume that the datapath performs the same operation given the same control signals. In effect, we treat some of the control edges in the full state graph as equivalent and claim that testing any one of those edges is sufficient to cover all of them. The assumption this makes is that the datapath does not hold any control state, so that only the sequencing of datapath commands matter, not their timing and duration.

The state variables that are not included in the control event graph also need to be accounted for in some way for coverage. These variables, which we call *independent state variables*, have had their entire contribution to the state space made explicit by the full state graph. However, it is still necessary to check that each individual variable has been exercised. The example in Figure 29 demonstrates this. Here we have two *independent* state variables, *Var2* and *Var3* and a control event graph consisting of *Var1*. The transition from state *B* to state *C* in the control event graph is taken if either *Var2* is in state *Y* or *Var3* is in state *3*. Hence, the control event graph might be fully covered even if one of the independent variables does not make all of its individual transitions. By requiring that each state variable, independent or otherwise, is fully exercised through its individual state graph, a better coverage metric is obtained without incurring the cost of a state graph

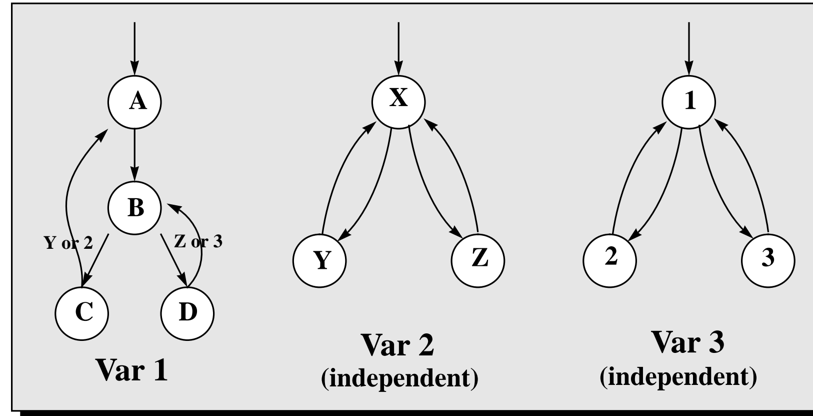

Figure 29. Independent State Variables

cross-product. This decreases the testing requirements while keeping all interaction information that affects the datapath.

Putting this together, we see that the full coverage property is comprised of two conditions:

- Each individual state variable is transition-covered.
- The control event graph is transition-covered.

As a matter of pragmatics, the first condition should be satisfied before attempting the second since if the individual state variables are not transition-covered, then it is likely that there are large sections of the control event graph that will not be covered also. The degree to which these conditions are satisfied serve as the coverage metrics.

5.3.2 Control Event Graph

A graph of control events can be created by projecting the full control state graph onto the set of datapath control variables. This graph represents all possible sequences of control events as given by the RTL description.

A precise definition of the control event graph requires introducing some additional mathematical definitions. We assume that there is a set of state variables V in the original design. Each state in the original state graph can be regarded as a function that maps each variable in V to a value (for simplicity, we assume that all values are drawn from a single domain, which we will call D). Hence, the type of a state s in the original graph is $V \rightarrow D$.

The set of datapath control variables V' is a subset of V. We wish to define the *projection* of the original state graph onto a state graph whose state variables are restricted to V'. Let S be a set of states mapping V to D. The projection of S onto V' (written *proj(S, V')*) is a set S' of states $s' : V' \rightarrow D$ where $s' \in S'$ iff there exists an $s \in S$ such that $s(v) = s(v')$ when $v \in V'$.

With the projection function, we can formally describe the control event graph as a projection from the full control state graph. Assume that the full control state graph, G, is represented by the standard 6-tuple (Σ, O, S, s_0, δ, λ), where Σ is the input alphabet, O is the output alphabet, S is the finite set of states, s_0 is the start state, δ is the next state function ($\delta: S \times \Sigma \rightarrow S$) and λ is the output function ($\lambda: S \times \Sigma \rightarrow O$). To get the control event graph, we apply the *proj* function to the set of states of G using the set of datapath control variables for the projection. We also map the transition and output functions to this projected set of states by simply applying *proj* on the arguments. For example, if in the full control state graph there is a transition $s_n \times input_1 \rightarrow s_m$, then in the control event graph, the projected transition would be $proj(s_n) \times input_1 \rightarrow proj(s_m)$. Similarly for the output function. Note that the *proj* function creates a non-deterministic FSM [HU79], which coverage analysis uses to track when transition edges are taken without reference to the transition conditions.

So, the control event graph, G_e, which is the projection of G onto the set of datapath-visible state variables, ρ, is the 6-tuple (Σ, O, S_e, s_{e0}, δ_e, λ_e), where S_e is the set of projected states of S, described formally as ($\forall s \in S, s_e = proj(s, \rho) \Leftrightarrow s_e \in S_e$); the start state is the simple projection $s_{e0} = proj(s_0, \rho)$; and the transition and output functions are derived as above giving (δ_e: $S_e \times \Sigma \rightarrow 2^{S_e}$) and (λ_e: $S_e \times \Sigma \rightarrow 2^O$), where 2^X represents the power set of X, which is the set of all subsets of X [HU79].

The analysis of the RTL to determine which state variables are visible to the datapath occurs in the translator also. This analysis determines which variables need to be part of the control event graph. Any variable that directly, or through some combinational logic, is an output of the module needs to be included. This analysis is a simplified version of the transitive fan-in algorithm of Section 3.3.2.

This section described a method to isolate and focus attention on the control logic-to-datapath interface. By using this as a coverage measure instead of just the control state

variables, many validation redundancies can be removed from the state graph. This helps to identify the state variables that can be tested independently, which reduces the state cross-products that need to be tested. The third source of redundancy over general environments is discussed in the next section.

5.4 Over-Generalized Environment

When an environment is under-constrained, or over-generalized. In other words, the inputs are allowed to follow sequences that are not possible in the real hardware. This has the potential of pushing the FSM model into states that cannot really be reached in hardware. These illegal states, which may actually be an entire sub-tree, then get reported as uncovered interactions, which only serve to conceal the interactions which really do need to be tested. A good example is the counter shown in Figure 30.

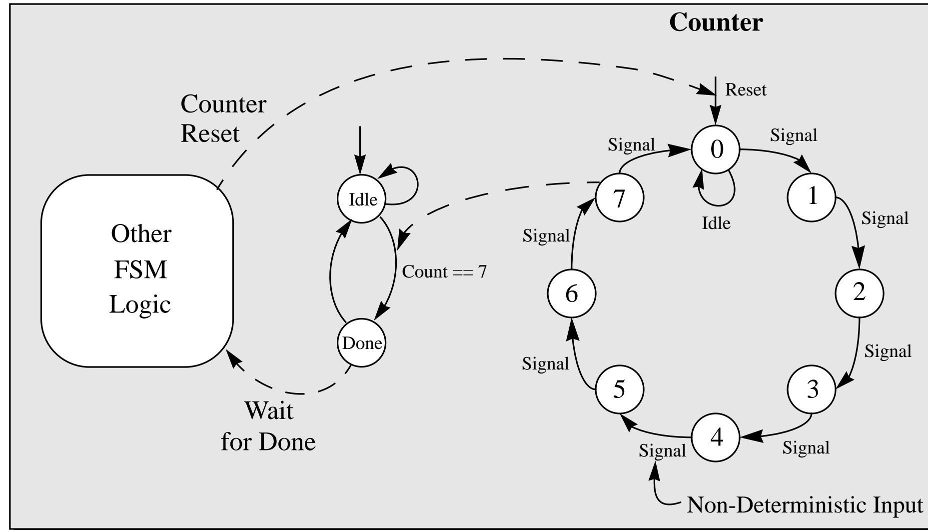

Figure 30. Counter fed by Non-Deterministic Input

When the input to the counter is a non-deterministic variable, the counter will step through every value for every state in the rest of the design because the *signal* event will be tried in all possible states. However, in the real hardware the FSM logic might be trying to refill a cache line and the counter is keeping track of how many datawords have been retrieved so far. In this case, it only makes sense for *signal* to occur during a cache miss refill and not at any other time. The result of the ND-modeling is that the counter will increment even during non-cache miss periods, making the state space several times larger

than it really is. The state space explosion problem is already very severe and this exploration of unreachable states exacerbates it unnecessarily. Once identified, problem areas like this can be dealt with on an individual basis. In the case of the counter, the ND-variable that produces *signal* can be replaced with a more accurate model that produces it only during periods where it makes sense.

The experiences of applying the coverage toolset to the instruction fetch (ifetch) unit of the PP illustrates how these illegal sequences arise in real designs, how they can be found and can be removed from the FSM model. Two examples are presented.

5.4.1 Case 1: Illegal Sequence from Reset

The *InstructionFetch* FSM model was approximated and partitioned from the full PP as described in Section 6.1 to create a manageable state graph. The model started off with a fully general non-deterministic environment where all inputs were allowed to take on all their possible values every cycle in all combinations. When the state graph and coverage information were presented to the designer, several unreachable product states were quickly identified. The states of the individual FSMs in question were encoded into the RTL, but it was believed that the product state was impossible to reach. There are two possible explanations: the state was an artifact of the approximation process, which is described in Section 6.1; or the environment was too general and produced a path through the FSM that was not possible in the real hardware.

To determine which explanation was responsible, the sequence of events that led to the illegal state was derived from the state graph. The sequence indicated that a path from reset was possible, shown in Figure 31.

Initially, `Reset_v` is asserted, keeping both the `IFetch` and `CacheCtrl` FSMs in their initial states. However, if an instruction cache miss was indicated during this time, the latch `ICReq_v` would transition from `0` to `1` and hold that value. On reset, the lines of the instruction cache hold invalid data, so this condition is the normal action. When `Reset_v` is dropped, the FSMs act on the cache miss and transition. Unfortunately, `ICReq_v` is not normally expected to contain a `1` unless the `IFetch` FSM has transitioned to state `WaitOK`. This condition is true, enforced by the transition function, except for the case where `Reset_v` is asserted. The end result is that the combined FSMs reach a state with `WaitOK` and `CC_I`, which eventually leads the `cacheCtrl` FSM to state

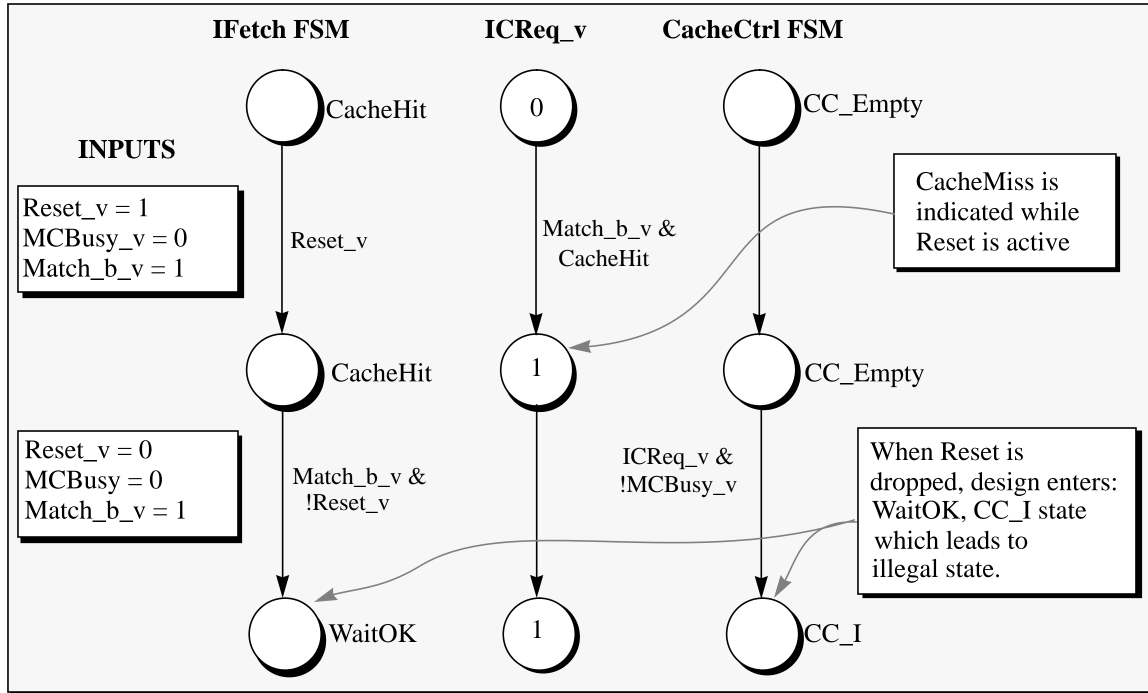

Figure 31. Illegal Sequence from Reset

CC_II which is "illegal". The state CC_II represents an instruction cache miss taken while servicing an instruction cache miss, which is impossible for this particular design.

When this sequence was analyzed and compared against what the RTL simulation really did, it was discovered that the signal MCBusy_v, from the memory controller unit of MAGIC, would always be asserted when Reset_v was asserted and remain asserted for a large number of cycles after Reset_v was dropped. As shown in the sequence, this prevents the cacheCtrl FSM from transitioning to CC_I. By the time MCBusy_v was dropped, the danger was over because the *InstructionFetch* FSM was in state WaitOK, ready to move on to its next state, and never visiting the illegal state combination.

The analysis of this illegal sequence led to two benefits: firstly, the environment model for the FSM model was constrained so that MCBusy_v would be asserted for the duration of Reset_v plus a few cycles. This removed the illegal sequence and helped to reduce the state space a little. Secondly, analyzing the sequence provided insight into the working of the design in a rare corner case that the designers were not fully aware of. It also made explicit an assertion about the interface properties of the MCBusy_v signal that

was implicit in the design. This is potentially useful as future changes for timing or functionality could change the method of generating this signal and nullify this assertion.

5.4.2 Case 2: No Bubble in Cache Refill

After fixing the environment to enforce the `MCBusy_v` assertion, the illegal state in the state graph still showed up, with a different event sequence. This sequence is shown in Figure 32.

In this sequence, during an instruction cache refill sequence, the *InstructionFetch* FSM determines that the refill is complete when the 15th word is returned from main memory. The *CacheCtrl* FSM, on the other hand, waits for the 16th word before signaling its completion. Hence, if the final word of the cache refill is delayed for some reason, the FSMs get into the illegal states where the *InstructionFetch* FSM is `IDLE` and the cacheCtrl FSM is not.

The constraint provided by the hardware in this case is that once a refill starts, the memory controller guarantees that all sixteen words of a cache line will be returned in consecutive cycles without any bubbles. In an older version of the design, this was not always true, because main memory in this design is protected with a 1-bit error-correcting, 2-bit error-detecting code (ECC). If error-correction was required, some words could be delayed. The important point here is that implicit assumptions made about interfaces between units can change over the course of design. These assumptions often need to be made explicit in the environment model, which is useful for the design process as well as these validation tools.

5.4.3 Interface Assertions

These two examples from FLASH illustrate the types of interactions that get highlighted by using control logic state graphs. They also show how using state graphs can make explicit some of the assertions implicitly used in an interface. In both of the examples, the correct operation of the design relied on an understanding of the interface between units designed by different people that was not written down formally or informally in any documentation or code comment.

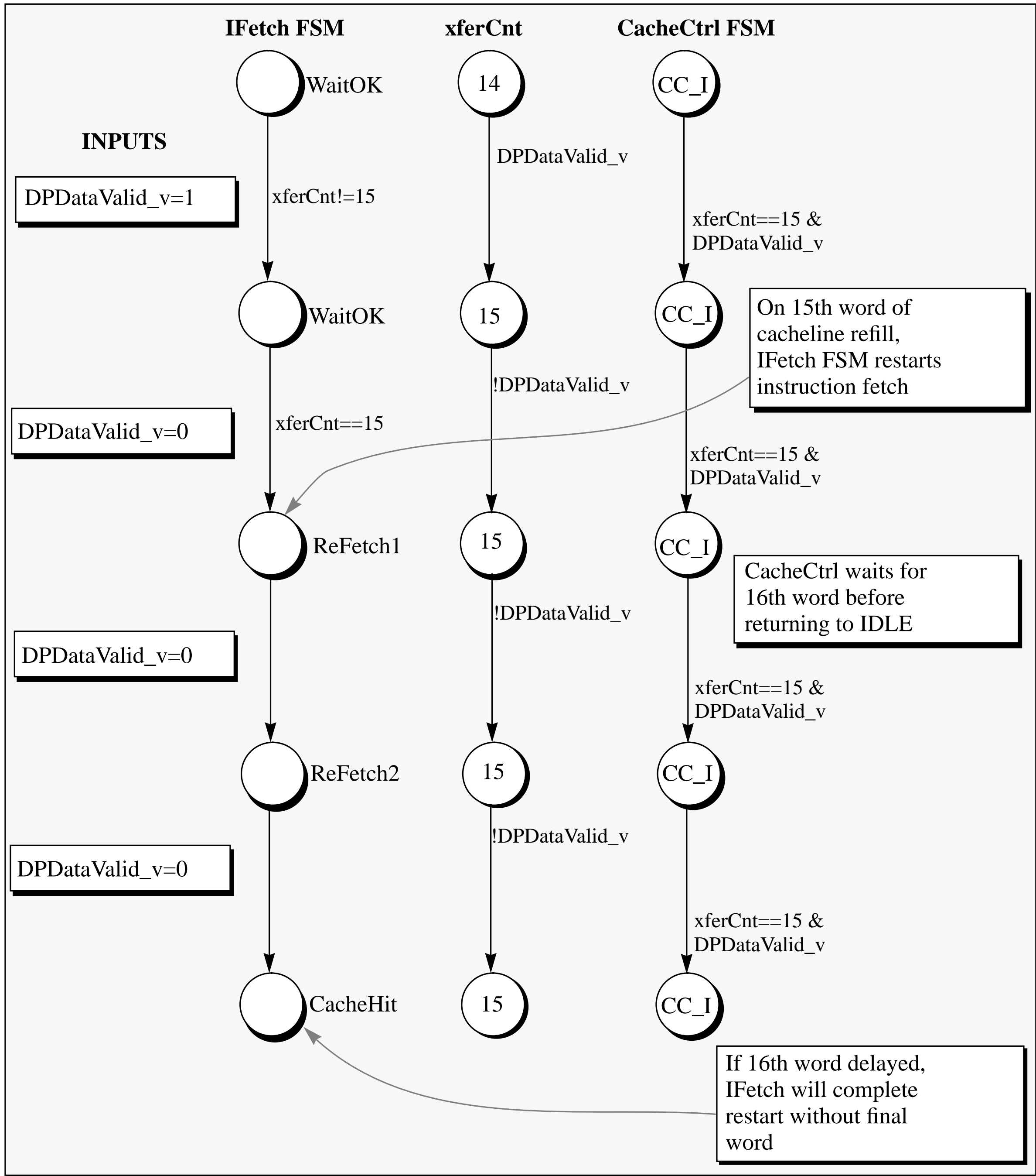

Figure 32. Illegal Bubble in Cache Refill

By starting the FSM model with the most general environment and constraining it only when it generates clearly illegal sequences, we stress the assumptions made about interfaces. Each constraint added to the environment makes explicit an assertion about the

interface. Even if these assertions are clearly understood by the designers, making them explicit helps when changes or timing modifications are done at a later stage.

The interface assertions can also be encoded as *snoopers*, described in Section 2.1, that are run with the RTL in simulation. This adds another correctness check to the RTL simulation. The automatic conversion of an environment constraint to a *snooper* is shown in Figure 33.

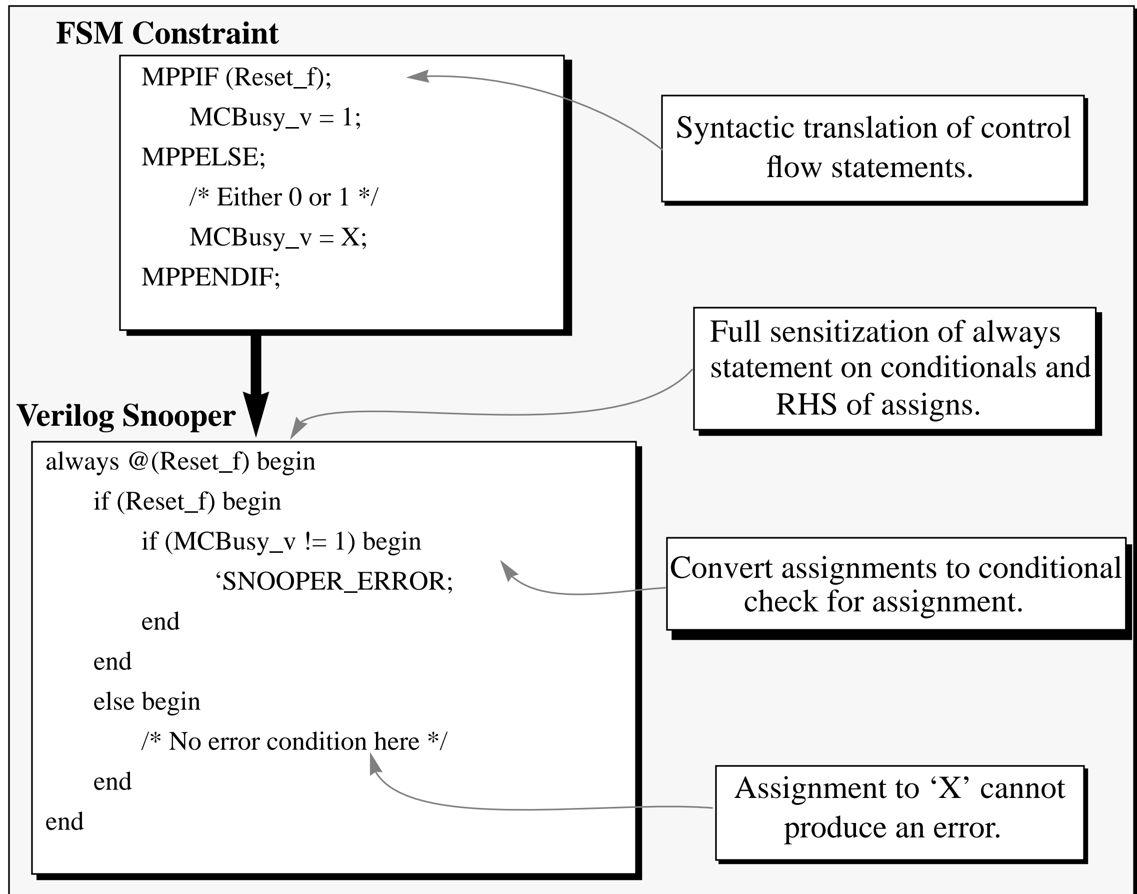

Figure 33. Converting a Constraint to a Snooper

The simple FSM constraint that forces `MCBusy_v` to be `1` whenever `Reset_v` is asserted is shown. When `Reset_v` is not asserted, `MCBusy_v` is allowed to take on any value, indicated by the `X`. In the FSM model, the `X` is a *non-deterministic variable*. The equivalent Verilog snooper demonstrates the conversion, annotated with the translation rules.

5.5 Incremental Feedback

After these three sources of redundancies were addressed, the coverage numbers for the FLASH units were more in line with the expectations of the designers. The missed interactions that the coverage tool reported actually corresponded to real test situations that were not exercised. This was important for tool credibility in a design team setting. By providing feedback that could be understood and related back to the RTL, it was possible to figure out where the missing tests were.

In practical terms, this means that the best feedback to give to designers is in terms of simple measures first, then progessing to more complex measures. So, the first thing to do is to give coverage information about the individual FSMs. This has multiple advantages. From a user's viewpoint, this is the simplest to understand and the most effective at pointing out problems in the test vector suite. If a state or a transition remains untested on an individual FSM, this is an important missing test. It is also relatively easy to repair since many states and transitions on a single FSM represent basic functionality. From a coverage viewpoint, identifying a missing state on a single FSM translates to finding the common factor in a set of missing states that span several FSM. In other words, if a state is missing from a single FSM, then when several FSM are considered together, there will be many missing product states that include the missing single state. By building the coverage measure up, the multiplication of missing states does not dilute the later measures of product control states.

A coverage metric should not be considered a single static measure. It should be treated as a guide to further improve validation. As such, it can be used to progressively consider more complex interactions up to and including control events. In this way, coverage analysis should always present a horizon of information just a little beyond the current validation task so that the most pertinent information is identified at each stage rather than being lost in a mound of derivative information, and the user can more easily digest and act upon it.

Up till now, these techniques describe a method to analyze coverage with an exact state graph. Unfortunately, for many designs, the state space is inherently large and finding the full state graph becomes a problem. Ideally for these models, some useful information can still be obtained from the control logic FSMs and utilized to guide validation. This is

an important issue to ensure the validation tools are not restricted to small design examples and is the subject of the next chapter.

Chapter 6. Coping with State Space Explosion

The previous chapter described a method of using the control state graph of an implementation to obtain concise coverage information. This technique works for designs where the control logic can be captured and fully enumerated. But, as with any method that utilizes state space exploration, the *state space explosion* problem limits the size of the designs that can be dealt with. State space explosion is the name given to the exponential increase in the number of states with respect to the number of state bits. Experience with the FLASH project has been that state space blow-up can strike rapidly with even small changes in the RTL. In general, the design increases in complexity as it undergoes timing tweaks: logic is moved around to improve critical paths, some functions become pre-computed with a select, and some moved into other units. Overall, interfaces become less clean and more state is introduced as part of the design process.

For the validation tools to be useful for practical designs, they need to be able to deal with state space explosion in a graceful manner. Simply declaring that a design is too large and cannot be handled would not be an acceptable outcome of the methodology. One approach is an approximation method that lets designers get an approximation of the state space of a large design using a smaller representation. This allows coverage analysis and code generation to proceed on the approximate graph, which may point out areas of a large design which require more validation effort.

6.1 State Graph Approximation

Dynamic graph pruning, described in Section 5.2, helps to keep the state space from growing unnecessarily. It finds situations in which states are equivalent and can be represented by any of the states of the equivalence class. However, for many large designs, the actual state space is extremely large and graph pruning alone cannot keep the state graph manageable. For these cases, it is still important to obtain information to aid designers, but exact information may not be practical.

Using control events as the measure of interesting behaviors in a design can be viewed as taking an approximation of the full state graph. In particular, it is an approximation based on the heuristic that the important state variables are those that directly control the datapath. However, since the control event graph is projected from the full state graph, this approximation has a high correlation to the original graph. Every state in the control event graph represents *at least one* state in the original graph. The drawback is that for some models, the control event graph may be manageable in size, but the intermediate full state graph is large.

One method of approximation is to reverse the order of operations to obtain the control event graph. Namely, project the state variables in the transition function first, and then find the set of reachable states, as shown in Figure 34. The approximation step consists of projecting the states onto a subset of the state variables. The variables that are not in the projection are said to be *approximated*. Although any state variable can be approximated, the selection of these variables can have an impact on the quality of the approximation process, resulting in the introduction of unreachable states. This choice of approximation variables is discussed further in Section 6.3.1.

Reversing the sequence of operations avoids generating the full state graph. However, since the approximation is no longer based on the exact graph, there is a danger that it will be incorrect in significant ways. The error depends on how the approximation of the transition function is made. Two techniques for approximation are compared in the following sections.

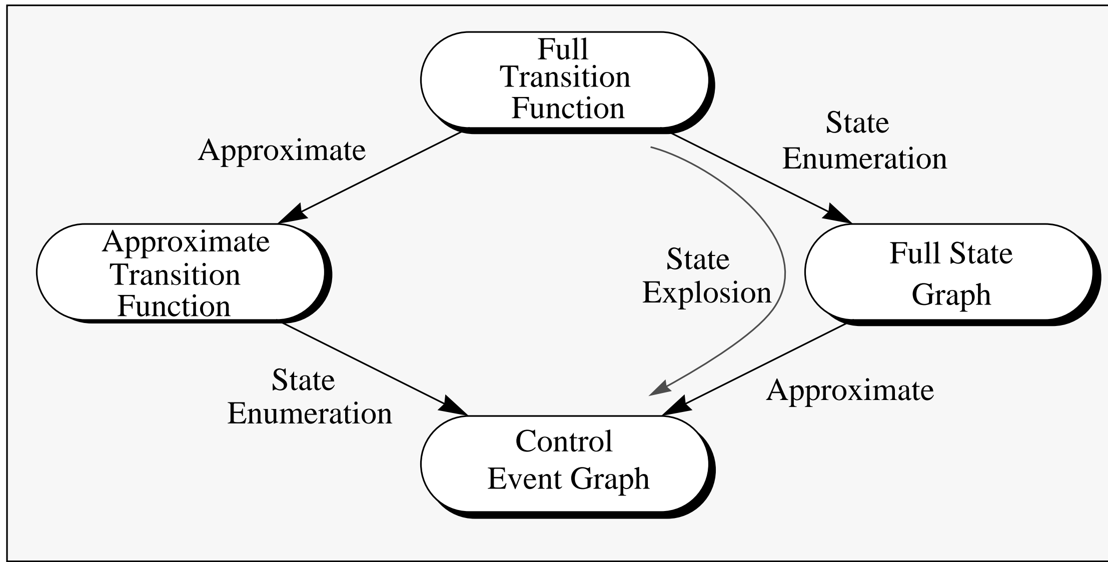

Figure 34. Approximating the State Graph

6.2 Approximation from Exact Partitions

One method of forming the approximation is to divide the design into a number of partitions, each of which can be fully enumerated in a reasonable amount of time and memory. An approximation can then be made of each partition and an approximate transition function created from this. The global approximation is then generated by combining the transition functions of each of the partitions. This process is shown in Figure 35.

This approach to approximation is intuitively appealing because it breaks the design into smaller components. Since the approximation is taken on these smaller, fully enumerated components, it is likely to be more accurate. This accuracy takes the form of only generating approximate states that correspond to at least one real state found in the enumeration.

6.2.1 Exact Partitions Algorithm

The algorithm for approximating based on exact partitions has three steps. First, the state graph of each partition is enumerated, treating state variables from other partitions as primary inputs. As mentioned while describing the environment of the FSM model in Chapter 3, primary inputs are treated as non-deterministic variables that take all their pos-

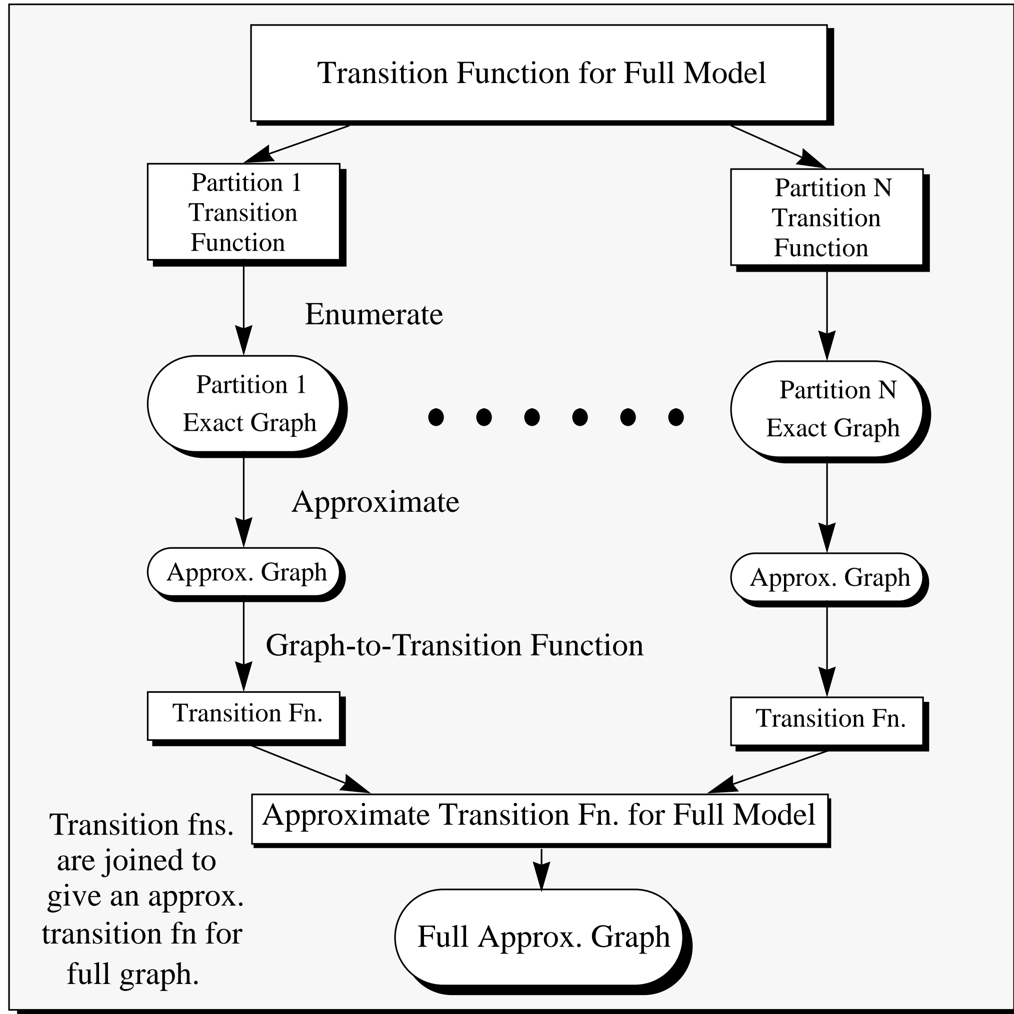

Figure 35. Approximation of Exact Partitions

sible values on every cycle. In other words, the constraining information provided by state variables in other partitions is discarded during the state enumeration. This results in state graphs for each partition which are over-approximations of the actual behaviors.

The approximation step is then performed on each individual partition graph. This is just a projection of the state vector onto a subset of the variables. During the projection operation, the approximated variables are compared against the approximated variables of the other states in the equivalence class. If any of the approximated bits are the same, these are kept. This provides additional constraints when the individual partitions are recombined, which help to keep the approximation close to the actual graph.

Once each partition state graph has been approximated, they need to be brought together to form the approximate graph of the full model. This can be achieved by converting the individual graphs back into a transition function. This process is mechanical, as demonstrated in Figure 36. A single state variable is used to represent all the states of

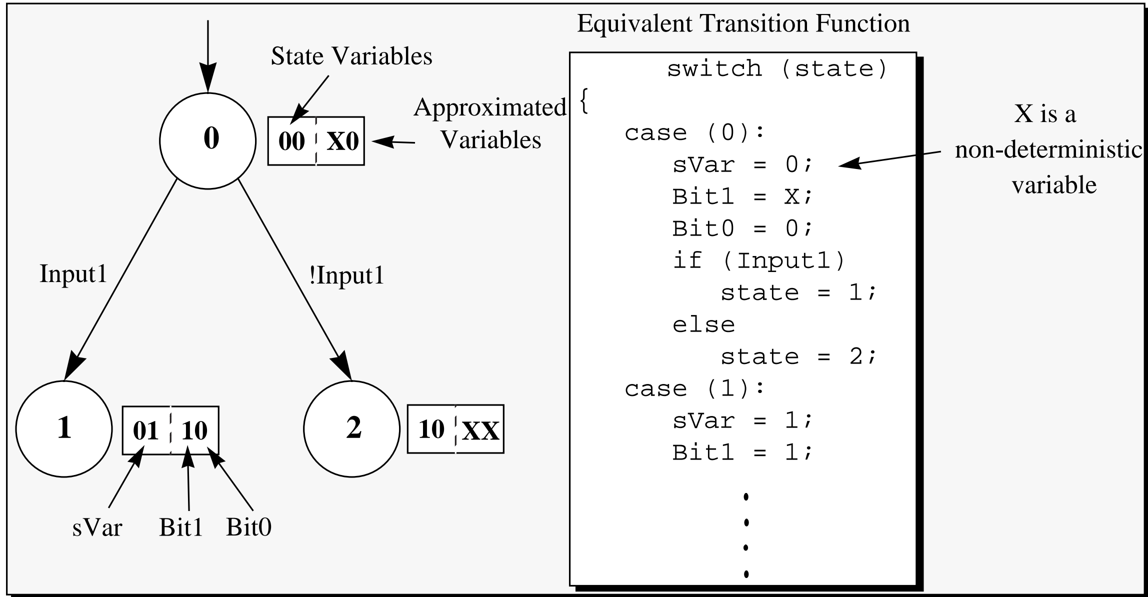

Figure 36. Conversion of State-Graph to Transition Function

the graph. The transition function is then a straightforward translation of the edges out from each state. The FSM can be regarded as a Moore machine where the outputs are the state variables. The approximated variables are also generated as outputs for the other partitions to use as constraints. If an approximated variable holds a definite value in one of the approximated states, this information is kept. Otherwise, the variables is treated as an *unknown*[1] and takes all its possible values on every cycle.

After the transition functions are generated for each partition, they are combined into a single transition function for the full model. This can then be explored by MPP to generate the full approximate graph. The *accuracy* of the approximation is enhanced if more constraints are present between the partitions. This corresponds to how many states in each partition graph have definite values for the approximated variables. If there are more definite values, then the full transition function is more constrained, leading to a more

[1]Represented as "X" in Figure 36

accurate graph. The number of states in the approximated graph for one section of the PP Instruction Fetch unit with definite values for the approximated variables is shown in Table 8, indicating that such constraints do exist for many variables.

Table 8. Constraints in Approximated Instr. Fetch Unit

Variable	States with x	States with 0	States with 1	% with Constraints
AInstr	72	2	4	7.7%
Reset_v	23	53	2	70.5%
Stall_x	72	4	2	7.7%
rICStall_v	38	1	39	51.3%
xferCnt_v	44	30	4	43.6%

The problem with this method of forming the approximate graph is that the transition function created by joining the transition functions of the partitions is a *non-deterministic FSM* (ND-FSM). An ND-FSM is generated because each approximated state represents an equivalence class of states. Some of these states will have transition-edges with overlapping conditions that lead to different approximated states. This requires a non-deterministic FSM to model the graph. An example is shown in Figure 37. Two states, A and B are

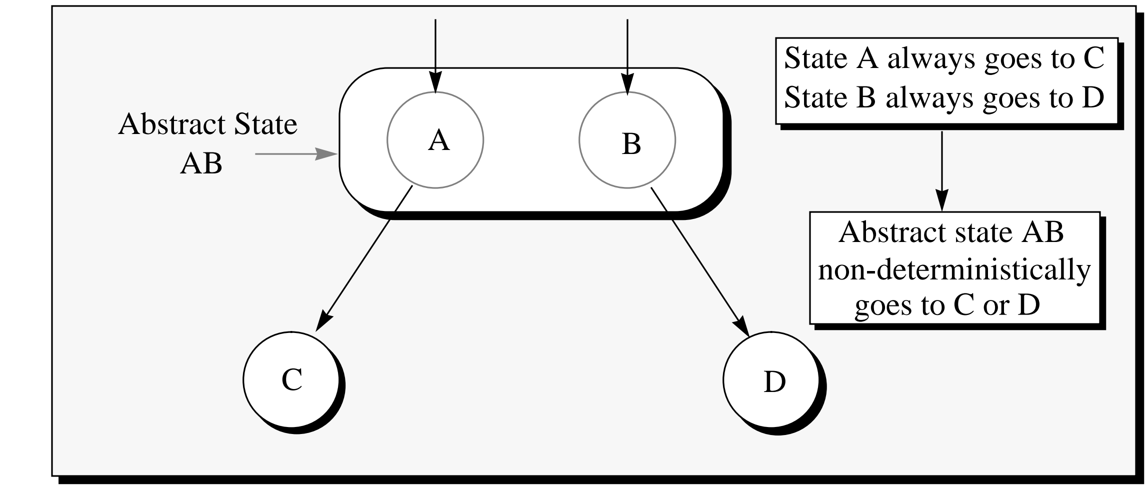

Figure 37. Approximating Graph leads to Non-deterministic FSM

approximated with a single state, abstract state AB. On any input condition, state A goes

to state C while state B goes to state D. So, the abstract state AB can go to either C or D on any input condition.

In general, a non-deterministic FSM [HU79] is one where the conditions on the transition edges from a state are not mutually exclusive. In other words, given a source state and a set of inputs, there may be more than one valid destination state, as demonstrated in Figure 38. From state 1, if input A and input B are observed, the next state may be either

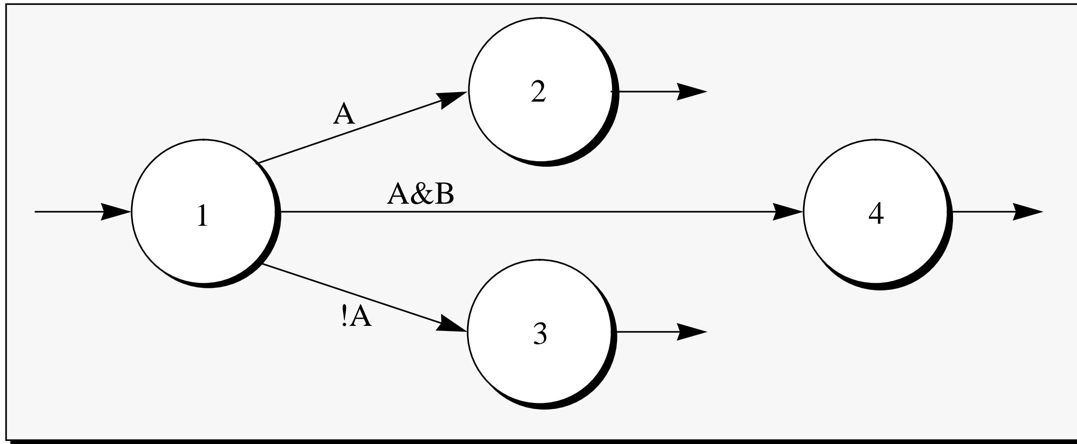

Figure 38. Non-deterministic FSM

state 2 or state 4. A non-deterministic FSM merely specifies the possible destination states and the conditions *necessary* to get there. The expressive powers of a non-deterministic FSM and a deterministic FSM are equal, as shown in [HU79]. However, a deterministic FSM may require more states to represent the same functionality.

The non-determinism in ND-FSMs should not be confused with the non-deterministic variables given as inputs to the *deterministic FSM*s modeled by MPP. MPP provides *non-determinism* in the form of variables that are assigned values based on the rule being fired. These special variables (which cannot be `STATE` variables), are declared with a range of possible values. Each rule takes a different combination of values for these variables. This results in each of these variables taking all of their possible values in every cycle. This provides non-determinism in the sense that at each cycle, these variables are completely unconstrained.

In order to find the approximation of the full graph from the combined transition function, a mechanism for state enumeration of non-deterministic FSMs is required. MPP

cannot handle such a model directly. It can be simulated by introducing a further variable into the model which provides the non-determinism. This variable (non-deterministic in the MPP sense), chooses the *set* of transition rules that will be fired in a particular cycle. The transition rules in each *set* are guaranteed to be mutually exclusive, allowing MPP to operate correctly. A straightforward implementation of this policy is to assign a *set* to each state in the equivalence class. This guarantees the mutual exclusiveness of each transition edge in a *set*. This policy was implemented for the FLASH examples. Unfortunately, the transition functions became too large to compile with GCC version 2.6.3. To effectively pursue this method of approximation, a state exploration tool capable of handling non-deterministic FSMs is required.

6.3 Approximating the Transition Function

Although there is support for the intuition that the previous method of approximation would give a good representation of the state space, the difficulty of efficiently exploring the state space of a non-deterministic FSM forced exploration of other algorithms.

An alternative method of obtaining the approximate graph of the full model without first generating the exact full graph is to approximate the transition function of the full model. This can be done by replacing the approximation variables with non-deterministic variables. Instead of using the transition function to set those values, they are allowed to take all their possible values every cycle. This provides an over-approximation of the transition function, in other words produces a state graph that is a superset of the real graph.

The advantage of this method is that it does not require a state enumerator that is capable of handling non-deterministic FSMs, as the previous method did. The disadvantage is that all constraints on the approximated variables is lost. However, as the number of states in an equivalence class grows, the likelihood of the approximated variables having a definite value decreases. For models with a lot of approximation, there is likely to be no difference in the final graph between approximating the transition function and approximating the exact partition graphs, as described in the previous section.

Creating the approximate transition function is a straightforward replacement of the variables that are to be approximated with a non-deterministic variable. This occurs in the translator, which knows about code dependencies and can remove statements that assign

values to the approximated variable. The list of variables to be approximated is given by the user in a file.

6.3.1 Choice of Approximation Variables

The choice of variables to approximate can have a large impact on the accuracy of the resulting graph. There are a few heuristics which seem to help keep the important information in the graph. The first is that many of the crucial state variables are multi-bit, whereas many of the less consequential ones are single-bit variables. The exception to this are multi-bit counters, many of which can be approximated.

A second heuristic that has worked well is to approximate variables that depend on the primary inputs of the model. This approximation has the effect of moving the model boundary inwards, putting some state outside. The greater the dependency on the inputs, the better this heuristic works. In the optimal case of a state variable that merely latches a primary input, this gives an exact reduction of the state space without losing information about the remaining state variables. If there is some dependency on other state variables, the resulting state graph may be under-constrained and lead to illegal states or edges.

A corollary to the second heuristic is that state variables that connect multi-bit variables should not be approximated. If they are, constraint information between the multi-bit variables is lost and we may degenerate to a simple cross-product of states. Quite often, if constraint information between the multi-bit variables is kept, the combined state is smaller than the cross-product. To illustrate the effect of this heuristic, the Inbox unit was approximated in different ways and the resulting graphs compared to the exact graph. These results are presented in Table 9.

Table 9. Different Approximated Variables for Inbox

Variable	Approximate States	Actual States	Approximate Edges	Actual Edges	% Wrong
choseSomeNIQueue_s	424	424	6,250	5,275	18.5%
forceJTMiss_s	322	322	4,943	4,887	1.2%

The variable `forceJTMiss_s` latches an input to the module, while `choseSomeNIQueue_s` connects up state variables from one portion of the design to another. For each of these variables, the state graph was found, based on approximation of that variable. The number of states and edges are shown in the columns labelled "approximate states" and "approximate edges". The exact full graph representation was also found and then projected onto the state variables used in the approximate graphs. This provides a comparison of what the projected states should be and what the approximation found. The number of states and edges from the exact graph are shown in the columns labelled "actual states" and "actual edges".

The encouraging news is that for the small Inbox model, the approximate graphs had the same number of states as the projections from the real graphs. However the approximate graphs found more edges than the projections showed should exist. This is a result of the under-constraining of the model, resulting in unreachable edges being found in the approximate graph. This empirical data point suggests that if an approximation is taken for a variable that connects other state variables, the resulting over-approximation is much worse than approximations of variables near the boundary of the design.

6.4 Related Work in State Graph Approximation

Graph approximation has been used in other formal methods work. However, the needs of approximation for validation are different from the needs of approximation for formal verification. In [AIK+95] and [DH95], real-time infinite state spaces were approximated by finite state spaces representing time intervals. Though effective in their domain, the concept of interval states does not translate directly to validation. The work of [CHM+93] and [CHM+94] bears more similarities to the graph approximations described here. In this work, a large state space was partitioned into smaller sub-graphs with state-variables from other sub-graphs being treated as primary inputs[2]. This gives an over-approximation of the true state space, but allows verification to occur on each sub-graph.

If no error is found in the over-approximation, then there is no error in the true state graph. This approximation technique of treating some state variables as primary inputs is also used in the techniques described here.

6.4.1 Other Approximation Techniques for Validation

Another method to approximate the state graph is to partition a large model by breaking it apart and using a non-deterministic environment for each of the subsequent submodules. The non-deterministic environment is less constrained than the real component which it replaces. Hence, it generates all the possible output sequences of the original, but contains less state. Unfortunately, it will likely generate many illegal sequences too, which may then lead to illegal or phantom states in the global model. This has the impact of diluting the quality of the coverage metric, but it does not miss any states and edges that need testing. The advantage of partitioning is that it allows more details of each sub-module to be kept. This may help for places in a design where more focussed attention is desired. The disadvantage is that constraints from the other sub-modules are lost and so illegal input sequences may be generated, leading to numerous false errors.

Another similar technique is to replace a single, state-intensive module with an approximation. For example, an *n-bit, 1-hot* component may be replaced with a non-deterministic generator than produces just *n* different values randomly, with no state. This would give a superset of the full behavior while removing state from the state graph. This replacement is useful in the situations where it can identified and a simple approximate model exists. This was used to a small degree in the FLASH examples.

The two approximation techniques described earlier and those mentioned above provide an over-approximation of the full state space, meaning they represent more states than really exist. However an under-approximation can also be taken. There are a couple of methods to do under-approximation of the state space. The simplest method to get an under-approximation of the full state space is to restrict some of the inputs to the model to fixed values. This acts to prune portions of the state space, giving a subset approximation of the full state space. This is useful to probe certain behaviors of the design, where only a subset of the possible inputs are wanted. For example, if we want to generate test vectors from an exact portion of a large state graph. However, when used for general coverage

---

[2] A primary input is treated as a non-deterministic value that takes all its possible values every cycle.

analysis, it suffers from not having a mapping for all exact states. When the state dumps from logging a test vector are coverage marked on an under-approximated graph, there can be many states in the dumps that have no corresponding state in the approximate graph. In other words, the diagnostic value of the coverage analysis is reduced since many of the test vectors may end up being thrown away if they are not in the portion of the state graph explored.

This under-approximation technique, though useful for reducing the amount of state that needs to be handled, suffers from not having an indication of the full state space. For coverage, the important information is often found in the interactions between different parts of the design. Under-approximation throws much of this information away. Under-approximation was attempted with the FLASH design examples. However, the results were poor. Whenever the simulated test vectors wandered outside the under-approximated state space, the coverage numbers would drop dramatically and the output would be filled with states that could not be marked for coverage in the state graph. In contrast, over-approximation uses all the simulated test vector information and overall provides a better picture of the validation status.

6.5 FLASH Coverage Results

Four design examples from FLASH were analyzed for validation coverage. These designs were run through the full methodology as described in Chapter 3. For the units of the Protocol Processor (Instruction Fetch and LoadStore), approximation using approximated transition functions (Section 6.3) was needed to keep the state space manageable. For all the units, graph pruning, control events and removal of illegal input sequences were employed to remove redundancies in the state graph. More attention and effort was focussed on some units due to manpower constraints. As can be seen from the results in Table 10, these units (Instruction Fetch, Inbox and Outbox), have a much higher coverage

Table 10. Coverage Results

	Protocol Processor	Instruction Fetch	Load Store	Inbox	Outbox	IO
Graph States	22,080[a]	1,586[a]	12,192[a]	426	52	3,209
Graph Edges	2,189,553	14,455	1,106,688	3,968	506	70,211

Table 10. Coverage Results

	Protocol Processor	Instruction Fetch	Load Store	Inbox	Outbox	IO
Control Event[b] States	76	17	132	28	14	142
Control Event Edges	184	47	532	236	62	1,778
Simulated Test Cycles	794,342	391,967	685,683	249,336	101,756	67,828
Full State Coverage	< 0.1%	< 0.1%	< 0.1%	19.3%	53.6%	< 0.1%
Full Edge Coverage	< 0.1%	< 0.1%	< 0.1%	3.0%	13.8%	< 0.1%
Control Event State Coverage	30.3%	94.1%	28.8%	82.1%	92.9%	16.3%
Control Event Edge Coverage	26.6%	59.6%	12.4%	28.0%	54.8%	6.7

a. Graph approximated.
b. Designers narrowed variables in control events in some cases to remove independent variables.

measure than the other units. For comparison, the coverage of the full state graph is also shown. The same design and set of test vectors were applied and the resulting coverage calculated using the two methods.

The value from the coverage results shown here is in the underlying benefits felt by the designers in having missing test cases brought to their attention in a manner they could act upon without excessive additional effort. Using incremental coverage measures in concert with the other techniques presented allowed the vital information to percolate to the surface. Additional tests were written in the course of this work that were identified by these incremental state space coverage metrics.

Chapter 7. Conclusions

Fundamentally, circuits are complex entities, hard to completely understand and consequently hard to validate. Hence it is easy to see why functional validation for complex digital designs is such a difficult task that absorbs a large amount of resources. Until recently, there have been few tools or accepted methodologies that could assist designers. Existing validation techniques suffer from problems with creating and evaluating a good test vector suite that can be used in simulation. On the other hand, formal verification techniques still require time and effort to mature into general purpose tools. The combination of the two approaches seems to provide a promising method of dealing with large implementations by using simulation to verify the design, and using techniques from formal methods to guide the simulation into meaningful areas.

The approach described in this thesis attempts to extract control logic information from an implementation model of the design. It then uses this information to discover the range of possible behaviors of the design, which maps out the required validation testing. In some cases, it can even generate the required test vectors directly. A number of issues and obstacles arose in the course of investigating how the control logic information could be used to guide validation. These will be discussed in this chapter.

When simulation is used to verify a design, the first issue that must be addressed is how the user can tell whether the design did the right thing during simulation. One good solution is a more abstract golden model that can be co-simulated with the RTL imple-

mentation. This can automatically perform state-comparison checks throughout the simulation, catching errors close to their source. However, for the reasons described in Chapter 3, creating a useful golden model for general designs is difficult. The alternative is to use self-checking test vectors, which are much more limited. These tend to only check for certain operations to be completed correctly, those targeted by the test itself. Sometimes, other errors may arise in a test but not get caught because the test was not checking all possible errors in the design. And when they do catch bugs, it can be difficult to isolate the exact cause because the error check is often far separated in time from the error source. Hence, although the self-checking validation framework is commonly used, it may be a weak correctness check. This is a problem because even the most comprehensive suite of test vectors are of limited use if the validation framework allows bugs to be exercised and not caught.

One technique of addressing the shortcomings of self-checking test vectors is to introduce snoopers into a simulation. Snoopers are a good method of finding errors that operate with any validation framework. These low level checks give much greater visibility of errors in a design, allowing them to be caught close to their source. If enough snoopers are used in simulation, automatic test vector generation can be utilized even without a golden model. The problem with snoopers is that it is difficult to know when enough have been placed in the simulation. Since these are hand-written using design knowledge, they are also not trivial to derive. On the other hand, since they tend to be localized and small, they are not as hard to create as a golden model. An open question is whether good snoopers can be created automatically so that low-level, comprehensive checks can be performed on a model. This would open up the possibility of more automatic test vector generation without requiring the effort of writing a golden model.

The second major issue encountered in this thesis is the exponential state space explosion problem. This work started out with the guess and hope that the state space of a particular class of designs, namely processors, would be manageable with careful abstractions. We set out to test this hypothesis with the processor section of the FLASH node controller. The results given in Chapter 4 show that for a particular design, at a particular stage in the design process, the state space can be manageable. And when this is the case, good validation results can be achieved by automatically generating test vectors to exercise all the interactions. However, further investigation showed that the state space continues to grow as the design matures towards tapeout. In particular, as timing analysis

is performed, logic is changed to reduce cycle times. This takes the form of more pre-computation, introducing registers to break multi-cycle timing paths, and simply changing the way the logic performed its computation to shorten paths. The general conclusion is that for any realistic design that is meant to be fabricated, the state space will ultimately explode and techniques that can operate in spite of that are required.

The results from Chapter 4 also show how useful automatic test vector generation from the control state graph can be at uncovering bugs due to interactions. The biggest problem with the algorithm presented is that the three steps in the process (translation to an FSM model, state-enumeration, and use of the state graph for test generation or coverage analysis), are disjoint, meaning that each step must terminate before the next step can start. This is due partly to the experimental nature of the work at the time and partly due to practical constraints on tool development, namely, using a pre-existing state enumeration tool. This makes the method vulnerable to state space explosion, which in this case prevented further exploration of the technique. However, if the steps of the method could be overlapped, so that the full state space does not need to be found before test generation can start, then perhaps progress can be made towards finding bugs even if the full state space is never fully enumerated. Such an approach would provide automatic test vector generation despite large state spaces.

Since the test vector generation method could not be pursued, given the lack of a golden model and the onset of state space explosion, this thesis went on to use the interaction information from the control FSMs for coverage analysis. A rather unexpected conclusion from the investigation is that FSM product state information is difficult to use as a coverage measure. A critical issue is knowing how the coverage information actually corresponds to missing tests. Control events represent a measure of validation effort that is almost complete with respect to control interactions in the datapath. However, the pragmatic experience of this effort is that much of the useful information comes from the simple interactions of small numbers of FSMs that are built up to form products. Control events give a maximal product, but in the process of increasing coverage to that level, the single FSM and small product FSM coverage measures provide valuable feedback. Single FSM transition coverage, in general, ensures that basic design functionality is exercised. At the other extreme, product states composed of a large (greater than five) number of FSMs have the problem of being difficult to interpret. Our experience was that designers had difficulty understanding the significance of such large product states, unless the

machines were closely related in function. What this means is that with large product states, even if a missing transition was presented to the designer, the test vector needed to exercise that transition could not always be created or even formulated. Clearly, this is the exact situation where automatic test vector generation would have been useful. The end result was that the most value was obtained from coverage information on a small number of FSMs, since these could easily be translated back into missing tests.

From a usability viewpoint, one of the problems of the methodology presented in this thesis was the occurrence of false errors. A false error is an error indicated by the method, but is actually an artifact of the technique and not representative of a genuine design problem. False errors arose from two sources; improperly constrained inputs to the FSM model from the ND environment and approximation of the state space, with the larger problem being the illegal input sequences from the ND environment. The examples in Chapter 5 showed that it is usually not difficult to provide the necessary input constraints once they are identified. The effort comes from identifying the correct constraints. This involves detailed knowledge of the timing and behavior of the interface signals. As a result of this requirement for design expertise, and also partially because an unconstrained input tends to catch more errors, this work took the approach of finding all the possible errors and subsequently identifying the false errors and adding constraints only when they became necessary. This initially appeared to be the more conservative and easier approach. However, the practical reality of false errors is that they rob credibility from a methodology in the eyes of designers. The lesson I believe this points to is that extra effort at the front-end of a methodology that can reduce the rate of false errors repays itself rapidly in terms of usability. If the necessary input constraints had been introduced much earlier, less effort would have been required following false errors.

The final issue brought up in this thesis is whether approximation is a good way to handle the state space explosion problem for validation work. This question can be answered in two ways. If one assumes or requires that the full state space be found, then I believe that approximation is the best method of handling state space explosion. The most obvious need for the full state space is in coverage analysis which wants to know how much of the total state space transitions have been exercised and how much remains. Although BDDs provide a good way to represent large numbers of states in a compact representation, BDDs are also subject to an explosion if they have a bad variable order. The question is really whether there is a compact representation of large state spaces. An

approximation is a representation that throws away some of the information about the state graph, hopefully only a small amount of the core FSM interactions. At this time, there does not appear to be any robust *exact* representation of large state spaces. The second way to answer the question is to remove the premise that the full state space needs to be found. As mentioned before, validation uses the state space as a map for testing requirements. That means that useful work can still be performed without the full state graph. This is an alternative to approximation and promising alternative for future work.

There remains much work to be done to create validation tools that can readily assist designers uncover all the hard bugs in designs. The work presented in this thesis puts forward the use of the control state space graph as a guide to the control interactions present in a circuit. Using this information, test vectors can be generated or coverage analysis performed. While much work remains to be done, the techniques proposed in this thesis present promising new tools for validation for complex digital designs.

References

[ABD+91] A. Aharon, A. Bar-David, D. Dorfman et. al., "Verification of the IBM RISC System/6000 by a dynamic biased pseudo-random test program generator", In *IBM Systems Journal*, Vol. 30, No. 4, 1991.

[ABG+92] Ali M. Ahi, Gregory D. Burroughs, Audrey B. Gore et al., "Design Verification of the HP 9000 Series 700 PA-RISC Workstations", In *Hewlett-Packard Journal*, August 1992.

[ADL+91] Alfred V. Aho, Anton T. Dahbura, David Lee and M. Umit Uyar, "An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours." In *IEEE Transactions on Communications*, Vol. 39, No. 11, November 1991.

[AGL+95] Aharon Aharon, Dave Goodman, Moshe Levinger, Yossi Lichtenstein, Yossi Malka, Charlotte Metzger, Moshe Molcho, Gil Shurek, "Test Program Generation for Functional Verification of PowerPC Processors in IBM", In *Proceedings of Design Automation Conference*, June 1995.

[AIK+95] R. Alur, A. Itai, R. P. Kurshan, and M. Yannakakis, "Timing Verification by Successive Approximation". In *Information and Computation*, Vol. 118, No.1 April 1995, pages 142-157.

[AIK+92] R. Alur, A. Itai, R. P. Kurshan, and M. Yannakakis, "Timing Verification by Successive Approximation". In *Proceedings of Computer Aided Verification* 1992, June 1992.

[And92] Walker Anderson, “Logical Verification of the NVAX CPU Chip Design”, In *Digital Technical Journal*, Vol.4 No.3, Summer 1992.

[AK95] David P. Appenzeller, Andreas Kuehlmann, “Formal Verification of a PowerPC Microprocessor”, In *Proceedings of the International Conference on Computer Design* 1995, October 1995.

[Ash96] Peter J. Ashenden, “The designer's guide to VHDL”, Morgan Kaufmann Publishers, 1996.

[ASU88] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*, Addison-Wesley Publishers, 1988.

[BeB94] Derek L. Beatty and Randal E. Bryant, “Formally Verifying a Microprocessor using a Simulation Methodology.” In *Proceedings of Design Automation Conference*, June 1994.

[Ber93] Berkeley CAD Group, “Revisiting BLIF-MV, An Intermediate Format for Verification and Synthesis of Hierarchial Networks of FSMs”, *HSIS Distribution*, 1993.

[BhD94] Vishal Bhagwati and Srinivas Devadas, “Automatic Verification of Pipelined Microprocessors.” In *Proceedings of Design Automation Conference*, June 1994.

[BM83] D. L. Bird, C. U. Munoz, "Automatic Generation of Random Self-Checking Test Cases", In *IBM Systems Journal*, Vol. 22, No. 3, pages 229-245, 1993.

[BRS93] U. Bruning, G. Radke and J. Sladky, “State-Machine-Development-Tool for High-Level-Design Entry and Simulation”, In *Proceedings of International Conference on Computer Design*, October 1993.

[Bry86] Randal E. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation”, In *IEEE Transactions on Computers*, C-35(8). pages 677-691. August 1986.

[Bry90] Randal E. Bryant, “A Methodology for Hardware Verification Based on Logic Simulation”, *Carnegie-Mellon University Technical Report* CMU-CS-90-122. March 1990.

[BuD94] Jerry R. Burch and David L. Dill, “Automatic Verification of Pipelined Microprocessor Control.” In *Proceedings of Conference on Computer-Aided Verification*, June 21-23 1994.

[BZ83] D. Brand and P. Zafiropulo, "On Communicating Finite State Machines", In Journal of the ACM, Vol. 30, No. 2, pages 323-342, 1983.

[CAB+95] A. Cao, A. Adalal, J. Bauman, "CAD Methodology for the Design of UltraSPARC-I Microprocessor at Sun Microsystems Inc", In P*roceedings of Design Automation Conference*, June 1995.

[CGH+95] Edmund M. Clarke, Orna Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, L. A. Ness, "Verification of the Futurebus+ Cache Coherence Protocol", In *Formal Methods in System Design*, Vol.6, No.2, pages 217-32, March 1995.

[CGL91] Edmund M. Clarke, Orna Grumberg, David E. Long, "Model Checking and Abstraction", In *Proceedings of ACM Symposium on Principles of Programming Languages*, January 1992.

[Cho78] Tsun S. Chow, "Testing Software Design Modeled by Finite State Machines," In *IEEE Transactions on Software Engineering*, Vol. 4, No. 3, pages 178-187, May 1978.

[CHM+93] Hyunwoo Cho, Gary D. Hachtel, Enrico Macii, Bernard Plessier, Fabio Somenzi, "Algorithms for Approximate FSM Traversal", In *Proceedings of Design Automation Conference* 30, June 1993.

[CHM+94] Hyunwoo Cho, Gary D. Hachtel, Enrico Macii, Massimo Poncino, Fabio Somenzi, "A State Space Decomposition Algorithm for Approximate FSM Traversal", In P*roceedings of the European Design and Test Conference. EDAC 94*. February 1994.

[CI92] Ashok K. Chandra and Vijay S. Iyengar, "Constraint Solving for Test Case Generation", In *Proceedings of the International Conference on Compter Design*, 1992.

[CIJ+94] A. K. Chandra, V. S. Iyengar, R. V. Jawalekar, M. P. Mullen, I. Nair and B. K. Rosen, "Architectural Verification of Processors Using Symbolic Instruction Graphs", *IBM Research Report,* September 1994.

[CIJ+93] A. Chandra, V. Iyengar, D. Jameson, R. Jawalekar, I. Nair and B. Rosen, "AVPGEN - A Test Generator for Architecture Verification", In IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 3, No. 2, pages 188-200, 1993.

[Cla90] Douglas W. Clark, "Bugs are Good: A Problem-Oriented Approach to the Management of Design Engineering." In *Research Technology Management*, Vol. 33, No. 3, pages 23-27, May-June 1990.

[CP88] Paolo Camurati and Paolo Prinetto, "Formal Verification of Hardware Correctness: Introduction and Survey of Current Research" In *Computer*, July 1988.

[CQC95] Gianpiero Cabodi, Stefano Quer, Paolo Camurati, "Extending Equivalence Class Computation to Large FSMs", In *Proceedings of the International Conference on Computer Design*, October 1995.

[CS95] David A. Cyrluk and M. K. Srivas, "Theorem Proving: Not an Esoteric Division, but a Unifying Framework for Industrial Verification", In *Proceedings of the International Conference on Computer Design*, October 1995.

[CSC93] Pinhong Chen, Jyuo-Min Shyu and Liang-Gee Chen, "Hardware Verification Using Symbolic State Transition Graphs", In *Proceedings of the International Conference on Computer Aided Design*, November 1993.

[CYB93] Szu-Tsung Cheng, Gary York and Robert K. Brayton, "VL2MV: A Compiler from Verilog to BLIF-MV", *HSIS Distribution*, 1993.

[CYF94] Ben Chen, Masami Yamazaki and Masahiro Fujita, "Bug Identification of a Real Chip Design by Symbolic Model Checking", In *Proceedings of the European Design Automation Conference, EDAC* 1994.

[Cyr94] David Cyrluk, "Microprocessor Verification in PVS", Unpublished.

[DDH+92] David L. Dill, Andreas J. Drexler, Alan J. Hu and C. Han Yang, "Protocol Verification as a Hardware Design Aid", In *Proceedings of International Conference on Computer Design*, October 1992.

[DH95] David L. Dill and Howard Wong-Toi, "Verification of Real-Time Systems by Successive Over and Under Approximation", In *Proceedings of Computer Aided Verification* 1995, June 1995.

[Dil88] David L. Dill, "Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits", In *Advanced Research in VLSI, Proceedings of the Fifth MIT Conference*, March 1988.

[EJ73] Jack Edmonds and Ellis L. Johnson, "Matching, Euler Tours and The Chinese Postman." In *Mathematical Programming*, Vol. 5, pages 88-124, 1973.

[Gat94] James Gateley, "Logic Emulation Aids Design Process", In *ASIC & EDA*, July 1994.

[GBC+95] James Gateley, Miriam Blatt, Dennis Chen , et. al. "UltraSPARC-I Emulation", In *Proceedings of Design Automation Conference*, June 1995.

[GDN92] Abhijit Ghosh, Srinivas Devadas and A. Richard Newton. *Sequential Logic Testing and Verification*, Kluwer Academic Publishers, 1992.

[Goe81] Prabhakar Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits", In *IEEE Transactions on Computers*, Vol. C-30, No. 3, March 1981.

[HB95] Ramin Hojati, Robert K. Brayton, "Automatic Datapath Abstraction In Hardware Systems", In *Computer Aided Verification*, June 1995.

[HeP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA, 1990.

[HH96] Richard C. Ho and Mark A. Horowitz, "Validation Coverage Analysis for Complex Digital Designs", In *Proceedings of the International Conference on Computer Aided Design*, San Jose, California, November 1996.

[HMA95] Yatin V. Hoskote, Dinos Moundanos, Jacob A. Abraham, "Automatic Extraction of the Control Flow Machine and Application to Evaluating Coverage of Verification Vectors", In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, October 1995.

[HMK96] Anoosh Hosseini, Dimitrios Mavroidis, Pavlos Konas, "Code Generation and Analysis for the Functional Verification of Microprocessors", In *Proceedings of the Design Automation Conference*, June 1996.

[Hol85] Gerald J. Holzmann, "Tracing Protocols", In *AT&T Technical Journal*, Vol. 64, No. 10, December 1985

[Hol91] Gerald J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall Software Series, 1991.

[HP90] John L. Hennessy and David A. Patterson, C*omputer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, 1990.

[HU79] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley Publishing Company, 1979.

[IKN+94] Hiroaki Iwashita, Satoshi Kowatari, Tsuneo Nakata and Fumiyasu Hirose, "Automatic Test Program Generation for Pipelined Processors", In *Proceedings of the International Conference on Computer Aided Design*, November 1994.

[IpD93] C. Norris Ip and David L. Dill, "Efficient Verification of Symmetric Concurrent Systems", In *Proceedings of the International Conference on Computer Design*, November 1993.

[JDB95] Robert B. Jones, David L. Dill, Jerry R. Burch, "Efficient Validity Checking for Processor Verification", In *Proceedings of the International Conference on Computer Aided Design*, November 1995.

[JMF95] Jawahar Jain, Rajarshi Mukherjee, Masahiro Fujita, "Advanced Verification Techniques Based on Learning", In *Proceedings of Design Automation Conference*, June 1995.

[JP96] Kevin D. Jones, John P. Privitera, "The Automatic Generation of Functional Test Vectors for Rambus Designs", In *Proceedings of the Design Automation Conference*, June 1996.

[KLM95] D. Knapp, T. Ly, D. MacMillen, R. Miller, "Behavioral Synthesis Methodology for HDL-Based Specification and Validation", In *Proceedings of Design Automation Conference*, June 1995.

[KN95] Michael Kantrowitz and Lisa M. Noack, "Functional Verification of a Multiple-issue, Pipelined, Superscalar Alpha Processor - the Alpha 21164 CPU Chip", In *Digital Technical Journal*, Vol. 7 No. 1, Fall 1995.

[KNS96] Michael Kantrowitz, Lisa M. Moack, Will Sherwood, "I'm done Simulating; Now what? Functional Coverage/Correctness Analysis of the DECChip 21164 Alpha Microprocessor", In *Proceedings of the Design Automation Conference*, June 1996.

[KOH+94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. "The Stanford FLASH Multiprocessor". In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, April 1994.

[KS92] Sungho Kang and Stephen A. Szygenda, "Modeling and Simulation of Design Errors", In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, October 1992.

{KS94] Sungho Kang and Stephen A. Szygenda, “Design Validation: Comparing Theoretical and Empirical Results of Design Error Modeling” In *IEEE Design and Test of Computers*, Vol. 11 No.1 March 1994.

[KSF+95] Jainendra Kumar, Noel Strader, Jeff Freeman, Michael Miller, “Emulation Verification of the Motorola 68060”, In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, October 1995.

[KVB+95] Timothy Kam, Tiziano Villa, Robert Brayton and Alberto Sangiovanni-Vincentelli, “Implicit State Minimization of Non-Deterministic FSMs”, In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, October 1995.

[LO95] Jeremy Levitt and Kunle Olukotun, “An Inductive Formal Verification Methodology for Pipelined Microprocessors”, In *Proceedings of Design Automation Conference*, June 1996.

[LTN90] Bill Lin, Herve J. Touati, A. Richard Newton, “Don’t Care Minimization of Multi-Level Sequential Logic Networks”, In *Proceedings of the International Conference on Computer Aided Design*, November 1990.

[MD95] Charles H. Malley and Max Dieudonne, “Logic Verification Methodology for PowerPC Microprocessors”, In *Proceedings of Design Automation Conference*, June 1995.

[MDW+89] Hi-Keung Tony Ma, Srinivas Devadas, Ruey-Sing Wei, Alberto Sangiovanni Vincentelli, “Logic Verification Algorithms and Their Parallel Implementation”, In *IEEE Transactions on Computer-Aided Design*, Vol. 8 no. 2 February 1989.

[Mer94] Dean Takahashi, “Pentium’s flaw may have been hard to prevent”. In *San Jose Mercury News*, Thurday December 1, 1994, pages 1A & 20A.

[Mips94] Mips Technology Incorporated, “R4000PC/SC, Processor Revision 2.2 and 3.0 Errata.”. At http://www.mips.com/HTMLs/R4000_PC_bugs.html

[MPS92] Enrico Macii, Bernard Plessier, Fabio Somenzi, “Verification of Systems Containing Counters”, In *Proceedings of the International Conference on Computer Aided Design*, November 1992.

[MSY+95] Carlos Montemayor, Marie Sullivan, Jen-Tien Yen, Pete Wilson, Richard Evers, “Multiprocessor Design Verification for the PowerPC 620 Microprocessor”, In *Proceedings of the International Conference on Computer Design*, October 1995.

[NJK94] B. E. Nelson and R. B. Jones and D. A. Kirkpatrick, "Simulation event pattern checking with PROTO", In *Proceedings of the International Conference on Simulation and Hardware Description Languages (SHDL)*, January 1994.

[PoD92] Fong Pong and Michel Dubois, "The Verification of Cache Coherence Protocols", In *Proceedings of 5th Annual ACM Symopsium on Parallel Algorithm and Architecture*, 1992.

[PoD94] Fong Pong and Michel Dubois, "Formal Verification of Complex Coherence Protocols Using Symbolic State Models", *University of Southern California Technical Report* CENG-94-01. January 1994.

[RPG+94] Elizabeth M. Rudnick, Janak H. Patel, Gary S. Greenstein and Thomas M. Niermann, "Sequential Circuit Test Generation in a Genetic Algorithm Framework", In *Proceedings of Design Automation Conference*, June 1994.

[RS95] Kavita Ravi and Fabio Somenzi, "High-Density Reachability Analysis", In *Proceedings of Design Automation Conference*, June 1995.

[SG96] Sanjay Sanwant, Paul Giordano, "RTL Emulation: The Next Leap in System Verification", In *Proceedings of the Design Automation Conference*, June 1996.

[Sid90] Deepinder P. Sidhu, "Protocol Testing: The First Ten Years, The Next Ten Years", In *Protocol, Specification, Testing and Verification X*, 1990.

[SrB90] Mandayam Srivas and Mark Bickford, "Formal Verification of a Pipelined Microprocessor", In *IEEE Software*, September 1990.

[TaK93] Sofiene Tahar and Ramayya Kumar, "Towards a Methodology for the Formal Hierarchical Verification of RISC Processors", In *Proceedings of the International Conference on Computer Aided Design*, October 1993.

[TM91] Donald E. Thomas and Philip R. Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishers, 1991.

[VTA+93] P. Vishakantaiah, T. Thomas, J.A. Abraham and M.S. Abadir, "AMBIANT:Automatic Generation of Behavioral Modifications for Testability", In *Proceedings of the International Conference on Computer Aided Design*, October 1993.

[VWK95] Peter Vanbekbergen, Albert Wang and Kurt Keutzer, "A Design and Validation System for Asynchronous Circuits", In *Proceedings of Design Automation Conference*, June 1995.

[Wes89] Colin H. West, "Protocol Validation in Complex Systems", In Computer Communication Review, Vol. 19, No.4, pages 303-312, September 1989.

[WGK90] David A. Wood, Garth A. Gibson, Randy H. Katz, "Verifying a Multiprocessor Cache Controller using Random Test Generation", In *IEEE Design & Test of Computers*, August 1990.

[WT95] Tsu-Hua Wang and Chong Guan Tan, "Practical Code Coverage for Verilog", In *International Verilog HDL Conference*, March 1995.

[YGM+95] Lawrence Yang, David Gao, Jamshid Mostoufi, Raju Joshi, Paul Loewenstein, "System Design Methodology of UltraSPARC-I", In P*roceedings of Design Automation Conference*, June 1995.