Center for Embedded Computer Systems
University of California, Irvine

A Comparison of Error Detection between Simulation-based Validation and Model Checking

Patricia Lee, Shireesh Verma, and Ian G. Harris

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2620, USA

{leep, shireesh, harris}@ics.uci.edu

CECS Technical Report 13-12
October 9, 2013

A Comparison of Error Detection between Simulation-based Validation and Model Checking

Patricia Lee, Shireesh Verma, and Ian G. Harris
Department of Computer Science
University of California Irvine
Irvine, CA 92697 USA
{leep, shireesh, harris}@ics.uci.edu

Abstract

Design simulation and model checking are two alternative and complementary techniques for verifying hardware designs. This paper presents a comparison between the two techniques based on their detection of design errors, performance, and memory use. While memory and CPU performance gains in simulation-based validation over model checking verification methods are to be expected, the ability of simulation-based validation in detecting errors is comparable to that of model checking. We perform error detection experiments using model checking and simulation to detect errors injected into a verification benchmark suite. The results allow a quantitative comparison of simulation and model checking which can be used to understand weaknesses of both approaches. We see that simulation-based validation is effective, and where it is not, we define test generation goals which make it effective.

1 Introduction

Functional verification is known to be a difficult task accounting for over 70% of the development time and resources [2, 10]. Model checking and simulation-based verification are two vehicles used for this task. Model checking [8, 5] is a well understood technique for verifying finite state machine models of designs under verification which determines whether or not the machine satisfies a given set of properties. If a given machine can violate a property, model checking is guaranteed to detect the violation, given sufficient memory and central processing unit (CPU) time resources. In the worst case, model checking must implicitly explore the entire state space of the machine being verified in order to find a property violation. The requirement of completeness in identifying property violations makes model checking susceptible to what has been called the *state explosion problem*. This describes the fact that the performance and memory requirements of model checking grow at least linearly with the size of the state space of the machine under verification. This means that the resource requirements of model checking may exceed practical limits given the constraints of existing computers.

[1]This research was supported by the National Science Foundation under grant CCF 0437116

Simulation-based verification [11] is known to be more efficient in terms of memory and performance when compared with model checking. For example, it is possible to perform cycle-accurate simulation of a standard processor such as an Intel Pentium processor [4]. However, model checking cannot be attempted for such a complex system and requires that the design be partitioned. One shortcoming of simulation-based verification is that it may not be complete in terms of error detection. The main drawback of simulation is that error detection is dependent on the test sequence and its ability to reveal errors. As a result, it is possible that errors escape detection by simulation while being caught using model checking.

The goal of this research is to compare the use of model checking and simulation based on error detection, memory use, and CPU time. The comparison is accomplished by performing verification of a set of benchmarks, using both simulation and model checking techniques. Design errors are injected into each benchmark, and the error detection ability of both verification techniques is evaluated by the number of erroneous designs detected.

unpredictably. Because an incomplete set of properties can allow design errors to be undetectable by model checking, we perform model checking with all erroneous versions of each design and discount those errors which do not cause model checking to fail. This accounts for any possible missed properties. By focusing on only those errors which cause the given properties to be violated, the error detection ability of model checking is not penalized due to

an incomplete property set. The generation of tests for simulation is performed automatically, using random patterns, with no manual interaction in order not to favorably bias the test generation results.

The remainder of the paper is organized as follows. The overview of our comparison of model checking and simulation-based verification is presented in Section 3. The techniques used for error injection, model checking, and simulation-based verification are described in Sections 4, 3.1, and 3.2 respectively. The experimental results of the comparison are presented in Section 5 and conclusions are summarized in Section 6.

2 Related Work

Although a comparison of the ability of model checking with the ability of simulation to detect errors has not been known to have been made in the hardware realm, a similar comparison is made in software [**?**]. Because no work, to our knowledge, has directly addressed this comparison between model checking with simulation, we present the following work to account for related efforts which have been made.

To investigate the ability of these techniques to detect errors, a standard set of hardware design errors must be defined. In *Hardware Design Errors*, we discuss various efforts in publishing a defined set of errors. *Model Checking Error Detection* and *Simulation Error Detection* categorize two methods that evaluate how errors are detected in model checking and simulation-based validation.

2.1 Hardware Design Errors

We looked at previous work to determine typical design errors found in industry and in the university setting. Through systematic collection of design error data, [**?**, **?**] has compiled a list of design errors and created various error models to capture the core nature of these design errors. [**?**, **?**] compiled the design errors from error data published by industry and from university design projects. The errors range from simplistic to complex. Some of the errors require a rare combination of events before becoming observable.

2.2 Model Checking Error Detection

The problem of determining the completeness of a set of properties for a given design has been addressed in the context of model checking via temporal model checking using vacuity testing [**?**]. Model checking determines if a given property holds in a given system by describing the system, stating the property, and then verifying that the property holds in the system [**?**]. This means that model checking's ability to detect errors depends on the ability of the tester to derive a complete set of properties for the system. Without this complete set, there is no guarantee of the complete correctness of the design.

2.3 Simulation Error Detection

The next group of work addresses the ability of error detection by simulation-based validation methods. This is the problem of determining when a testbench is complete using simulation-based validation to show the main correlation to the ability of simulation-based validation to detect errors [**?**]. Many methods offer more accurate assessment of design verification coverage than line coverage [**?**]; however, the focus of this paper is to provide a general comparison of basic methods of model checking with simulation-based verification. Specific analysis of the testbench generation is beyond the scope of this paper.

3 Overview

To compare model checking to simulation-based verification, we perform verification using both techniques on a subset of the benchmarks which are contained in the VIS downloaded benchmarks [3]. These benchmarks were chosen because they could be model checked by the VIS tool [3] and simulated using the Synopsys VCS tool [1] with only minor modification. Each benchmark provided is a design coded in Verilog and is accompanied with a set of Computational Tree Logic (CTL) [9] properties which can be used for model checking.

The comparison process is shown in Figure 1, and the main steps followed to verify each benchmark are listed here.

1. **Model Checking:** Model checking is performed with each erroneous design (as well as the original correct design) and with its set of properties. Based on the completeness of the properties, only a subset of the errors are detected. To account for the possibility of human error in generating a complete set of properties for the designs, we remove errors which cannot be detected by model checking.

2. **Simulation-based Verification:** The subset of the erroneous designs that are detected by model checking are then verified using simulation. Based on the effectiveness of the testbench, only a subset of these errors are detected.

3. **Error Injection:** A set of design errors are inserted into each benchmark design to generate a set of erroneous designs for verification. Each erroneous design contains a single design error.

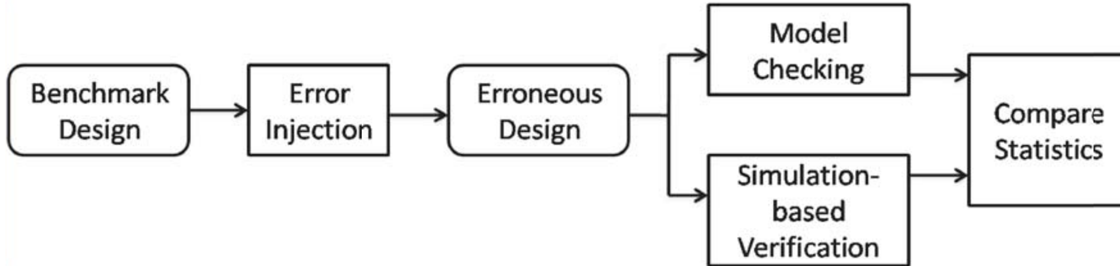

Figure 1. Comparison Between Model Checking and Simulation-based Verification

The error detection, CPU time, and memory use statistics of each verification run are summarized and presented for comparison. Each of the three steps listed above are described in the following sections.

3.1 Model Checking Process

Model checking is performed to verify each of the 100 erroneous versions of each benchmark design. The benchmark suite provides a set of CTL properties for each design. We use these properties for model checking. Each model checking run on an erroneous design ended in one of the following three ways.

1. **Error Detected:** An erroneous design is in this category if model checking completes and at least one of the properties failed. This indicates that the erroneous design violates one of the properties of the design.

2. **Error Not Detected:** An erroneous design is in this category if model checking completes successfully without detecting a property violation. This indicates that the erroneous design does not violate any design property that is provided in the benchmark suite.

3. **Inconclusive:** This indicates that model checking did not complete because of insufficient memory on the workstation running VIS. Given additional computational resources these errors will fall into either the *Error Detected* or the *Error Not Detected* categories.

In the results comparing model checking and simulation, we omit the model checked errors categorized as *Error Not Detected*. It is important to omit errors not detected by model checking in order to simulate a perfect set of properties that detect all errors. This is because a lack of detection is due to the incompleteness of the property set rather than any weakness in the model checking process. Additionally, the errors categorized as *Inconclusive* are not considered in the final results because they can possibly be *Error Not Detected* if additional resources are provided. In summary, only the errors categorized as *Error Detected* are considered in the final results. With respect to these errors, model checking always provides 100% error detection.

3.2 Simulation-based Verification Process

Each erroneous design detected by model checking, those categorized as *Error Detected*, is verified using a simulation-based verification process. The simulation-based verification process is depicted in Figure 2.

All of the erroneous versions of a single benchmark design are simulated with the same testbench so that they are all tested with the same test sequence. The simulation results of each erroneous design are compared to the results of the reference model, which is the correct design. An error is classified as being detected by simulation if the simulation results of the erroneous design are different from the results of the reference model. Simulation results are compared only at the primary outputs of the design.

3.2.1 Test Generation

The error detection ability of simulation is determined by the test sequence applied. To ensure that the evaluation is not affected by manual interaction, we generate the testbenches for our benchmark designs automatically. We use VPI (Verilog Procedural Interface) to access the VCS simulator's internal data structures, and we statically analyze them in order to be able to generate the testbench automatically. Based on this analysis, we identify the value domains of the input signals and generate Verilog code to produce values randomly from the value domain. We gereate Verilog code for comparing the primary outputs of the correct and erroneous versions of the design such that an error is flagged in case of any mismatch. The testbench supplies random data to all inputs, with the exception of any *reset*

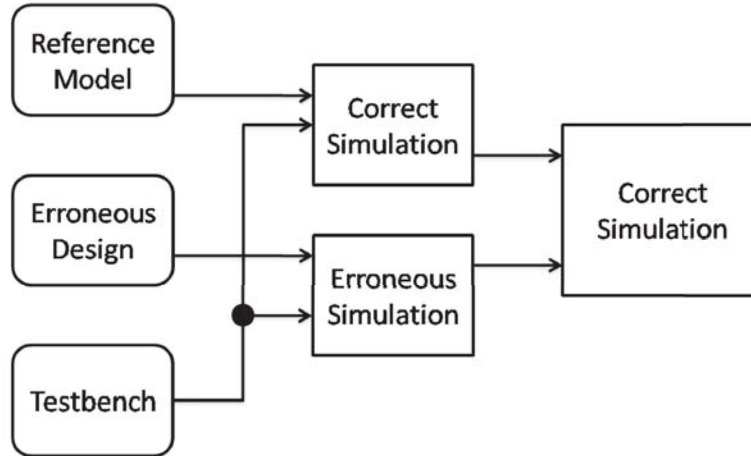

Figure 2. Simulation-Based Verification Process

signal. The *reset* signal is asserted once in the first clock cycle of testing and is de-asserted for the remainder of testing.

An important issue in test generation is that of determining how many tests to apply. The goal of test generation is assumed to achieve high statement and branch coverage. Random test patterns are applied during testing incrementally in the order of tens until statement and branch coverage hit a saturation point, i.e. the point where there is no change in coverage even after applying more test patterns. We obtain the number of patterns required to obtain a saturation coverage value for both statement and branch coverage. This means that we apply the higher of these two for all our simulations so that sufficient coverage is guaranteed.

Figure 3 shows the statement and branch coverage saturation curves for the benchmark example, Nim. The example attains a statement coverage of 86.27% with just 10 test patterns applied. Applying more test patterns does not improve the statement coverage further. However, the branch coverage attains a saturation value of 83.33% after applying 1,000 patterns. Therefore, we use 1,000 test patterns, the higher of the two test pattern amounts, for simulating the design.

Table 1 shows the saturation coverage values of statement and branch coverage for all the designs. The first column shows the design name. The next two columns show the saturating value of statement coverage and the number of patterns required to be applied to attain that value. The final two columns show the saturating value of branch coverage and the number of patterns required to be applied to attain that value.

It is important to mention that the comparison presented in this paper is limited to simulation using random test patterns until high statement and branch coverage are achieved. The use of another test pattern generation approach would potentially cause a fundamental change in the results.

4 Error Injection

Error injection is the process of inserting design errors into a design known to be correct. For each design, the process involves randomly inserting one error from a complete set of errors possible for the design into each newly created erroneous design. We compile a set of realistic errors from various sources [4] for the compiled realistic set of errors in addition to creating our own set of errors.

4.1 Textual Errors

Textual errors are applied directly to the original textual behavioral description [**?**] and are a type of error that is well modeled as mutants. Mutation errors come from mutation analysis which has been studied previously in software testing and hardware validation [6, 7]. In mutation analysis terminology, a mutant is a version of a behavioral description which differs from its original by a single potential design error. An operator is a function which is applied to the original program to generate a mutant. A set of operators describes all expected design errors. An important feature of mutation errors is that they are limited to a single line of code in a behavioral description. The errors created by these operators include control-flow errors which involve an en-

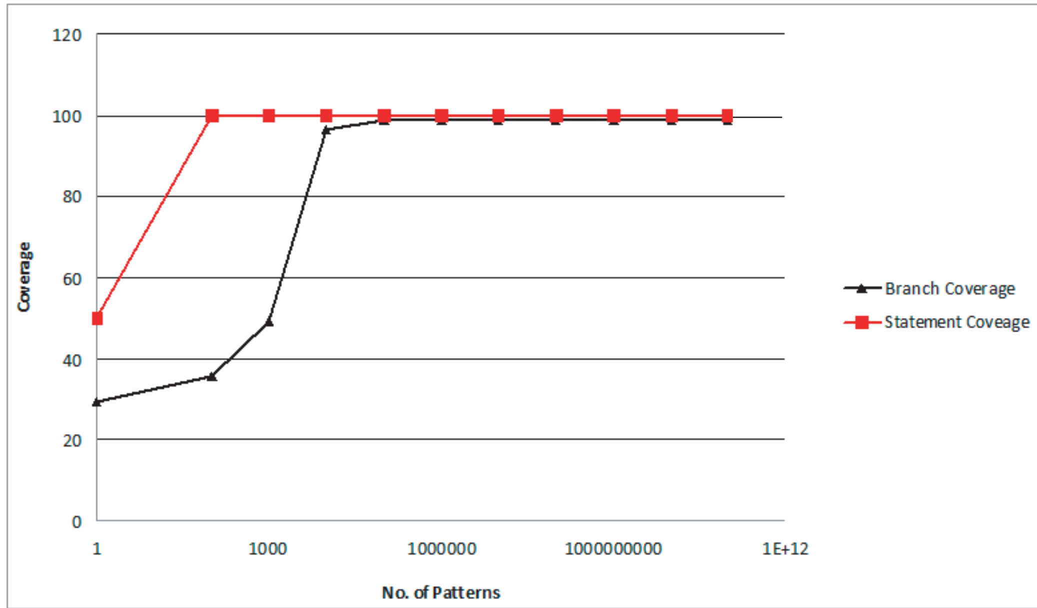

Figure 3. Plot of Statement/Branch Coverage vs. No. of test patterns for benchmark example Nim

Design	Statement Cvg. Saturation %	Branch Cvg. Saturation %	No. of Patterns
am2901	100	100	10^2
am2910	100	100	10^3
bpb	100	100	10^3
bufa	100	100	10^5
counter	100	100	10
fifo	100	100	10^2
gcd	98.31	93.75	10^5
huffman	98.77	100	10^4
microwave	100	100	10
miim	96.19	89.06	10^5
nim	86.27	83.33	10^3
nullmodem	72.46	47.37	10^2
palu	100	100	10^2

Table 1. Saturation Values of Statement and Branch Coverage for all Benchmark Designs

tire control-flow branch or control-flow construct removal error. For each of the correct designs, a copy of the program with the insertion of a one-line or one-operation mutation is inserted to create a set of designs each with only a single mutation inserted. Each of these errors model a type of misinterpretation of the specification on the part of the designer.

4.1.1 Arithmetic Operator Replacement

This type of error involves a replacement of an arithmetic operator in the expressions with other arithmetic operators such that the valuation of expressions differs from their correct valuation while maintaining the syntactic and semantic correctness of the design. The operators include $+, -, *, /$. Each of the other operators replace the correct opperator to create the replacement error. One main reason this type of replacement error, as well as all the other replacement mutant errors, may occur is that a designer may have just created a "goof" error where the designer simply placed one operation in place of another because of a mistake in typing or misunderstanding of a behavior stated in the design specification. Also, such an error may occur with fatigue of the designer, a designer's hast in writing a code excerpt in the interest of a deadline, or possibly in the designer's confusion because of the code excerpt being embedded in some complex code design. We insert this specific operator replacement, as well as all the other replacement mutant errors, into the Verilog code by replacing each of the alternate and erroneous operators with its original correct operator.

Figure 4 (a) shows a code fragment. Replacing the $+$ operator in line 1 of Figure 4 (a) with a $-$ operator as shown in Figure 4 (b) causes a wrong value to be computed for signal x which causes the output z to evaluate to a wrong value creating a data flow error. On the other hand, replacing the $+$ operator in line 2 of Figure 4 (a) with a $-$ operator as shown in Figure 4 (c) causes the wrong control flow path to be taken which creates a control flow error.

4.1.2 Relational Operator Replacement

This error involves replacement of a relational operator in the expressions with other relational operators such that the valuation of expressions differs from the correct valuation while maintaining the syntactic and semantic correctness of the design. The operators involved are $<, >, \leq, \geq, =, \neq$, all of which could be replaced by each of the other operators. This operator can cause error in the data as well as control flow of the design.

4.1.3 Logical Operator Replacement

This error involves a replacement of a logical operator in the expressions with other logical operators such that the valuation of expressions differs from the correct valuation while maintaining the syntactic and semantic correctness of the design. The operators involved are $\&\&$, $||$, both of which could be replaced by the other operator. This operator also can cause error in data as well as control flow of the design.

4.1.4 Reduction Operator Replacement

This error involves a replacement of unary reduction operators like $|$ or $\sim$ from expressions with other reduction operators such that the valuation of expressions differs from their correct valuation while maintaining the syntactic and semantic correctness of the design. This operator also can cause error in data as well as control flow of the design.

4.2 Behavioral Errors

Behavioral errors are those errors which are made based on the structure of the CFG (control flow graph) of a design. We have modeled common errors made based on the CFG of a design by omission of conditional constructs and branch omission within the code.

4.2.1 Conditional Construct Omission

This error involves removal of a conditional construct completely from the design e.g. an *if* or a *case* statement. These type of errors may result when a block of logic is omitted by the designer. Natural language specifications often specify causality between events. This translates directly to control-flow constructs in a hardware description. Also, this operator can cause an error in the data as well as control flow of the design. In the Verilog code, this conditional construct omission is inserted into the code by deleting the entire conditional construct whether that be in form of a "case" statement or an entire "if/else" construct.

4.2.2 Branch Omission

This error involves removing a branch of a conditional construct such as that of a branch of an *if* or *case* statement. This type of error results when a clause of logic is omitted by the designer. This arises out of misinterpretation of specification on the part of a designer. This type of error also causes error in data as well as control flow of the design. In addition to being a possible "goof" type error, this type of error arises out of misinterpretation of specification on the part of a designer with respect to the proper functionality of a component or when the designer overlooks one block of logic in the specification due to the complexity of the design specification. Often, a design is built from an FSM (Finite State Machine) into code using a "case" statement construct. This means that there is a possibility of an entire state with edges omitted from the design. Additionally,

1. x = a + 1;	1. x = a - 1;	1. x = a + 1;
2. if (a + b > 0)	2. if (a + b > 0)	2. if (a - b > 0)
3. y = a + 3;	3. y = a + 3;	3. y = a + 3;
4. else	4. else	4. else
5. y = b + 4;	5. y = b + 4;	5. y = b + 4;
6. z = x + y;	6. z = x + y;	6. z = x + y;
(a)	(b)	(c)

Figure 4. (a) Arithmetic Operator Replacement Example Code Fragment, (b) Data Flow Error, (c) Control Flow Error

there can be a case where just the state is omitted while the edges are left behind in the case of maintenance of legacy code or legacy functions and the possibility of an incomplete cleaning of the code with this update or maintenance process. In the Verilog code, this branch omission was inserted into the code by deleting a single case/state (or multiple cases forming a portion) of the entire "case" statement or portion of the "if" or "else" part of a complete "if/else" construct.

4.3 FSM-Based Errors

FSM (Finite State Machine) models are adept at capturing the control and data flow of a design. Error models based on FSMs well represent design errors made by designers during the design implementation process. The state machine example described in Verilog HDL (Hardware Description Language) code shown in Figure 5 illustrates these error models. A graphical representation of the state machine is shown in Figure 6 to futher illustrate these error models.

4.3.1 State Elimination

State elimination involves removing a complete state from the state machine. In Verilog, this is synonymous to a branch omission, as mentioned in the textual errors section, and is accomplished by removing a branch of a conditional construct, such as that of an *if* or *case* statement. If state *C* is removed from Figure 6, the resulting state machine is erroneous. A side effect of this removal is that all the transitions originating and ending at state *C* become invalid. In terms of the Verilog description shown in Figure 5, branch *C* of the case statement corresponding to lines 22 through 26 are removed to simulate this error.

4.3.2 State Change

State change involves exchanging a label of a state in a state machine with that of another. For example, if the label of state *A* in Figure 6 is changed to *B* and that of state *B* is changed to *A*, the properties and the operations inside states *A* and *B* are interchanged. This results in an erroneous state machine, and additionally, this results in many of the transitions becoming invalid. In terms of the Verilog description shown in Figure 5, this type of error is denoted by replacing the value A on line 13 with that of B and replacing the value B on line 17 with that of A. In this case, the design is stuck in the new state *B* since there are no outgoing transitions from it. Alternatively, this error may cause different behaviors in other cases. For the sake of completnesss, we include the description of these types of errors although they arenot included in our set of inserted errors in the design examples used in this work.

4.3.3 Transition Elimination

Transition elimination involves the removal of any transition between two states. In Figure 6, the transition from state *A* to *B* and the transition from state *B* to *A* are removed. Removing these transitions results in an erroneous state machine. In terms of the Verilog description shown in Figure 5, multiple code changes can cause this error type of error to emerge. For instance, line 14 can be deleted to eliminate the possibility of a transition from state *A* to *B*, or the *else* branch on line 19 can be omitted to eliminate the transition from state *B* to *A*. Again, just as with the state elimination error, a branch omission represents this type of error occurrence. Other ways of denoting this error are not included in this work.

4.3.4 Transition Change

Transition change involves changing the destination of a transition. In Figure 6, changing the transition from state *A* to *B* into a transition from state *A* to *C* results in this type of error. In terms of the Verilog description shown in Figure 5, this error is shown by replacing the value B on the right side of the statement on line 14 with value C.

```
1.  module state_machine (clock, in, out);
2.  input  clock, in;
3.  output reg out;
4.  reg state;
5.  initial
6.  begin
7.     state = A;
8.     out = 0;
9.  end
10. always @ (posedge clock)
11. begin
12.   case (state)
13.     A: begin
14.            state = B;
15.            out = 0;
16.          end
17.     B: begin
18.           if (in == 0)  state = C;
19.           else  state = A;
20.           out = 0;
21.         end
22.     C: begin
23.           if (in == 1)  state = A;
24.           else  state = B;
25.           out = 1;
26.          end
27.   endcase
28. end
```

Figure 5. Verilog Description for a Finite State Machine

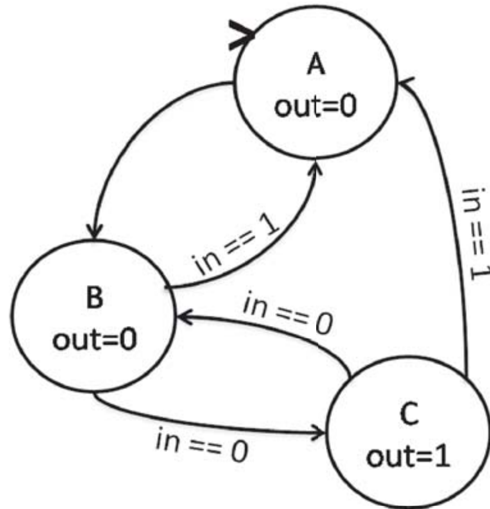

Figure 6. Grahical Description of Finite State Machine

4.3.5 Transition Label Change

Transition label change involves changing the pre-condition of a transition if there is one or introducing a pre-condition if there is not one. In Figure 6, removing or changing the the label $(in == 0)$ for the transition from state *C* to state *B* causes an erroneous machine. In terms of the Verilog description shown in Figure 5, this error of replacing value 1 in the expression on line 23 by the value 0 causes the pre-condition for the transition to now become $(in == 1)$ rendering the design erroneous.

4.3.6 Output Change

An output change involves altering the computed correct output value of the state machine. In Figure 6, if instead of setting out to 0 in state *B* it is set to 1, an incorrect output value is computed by the state machine rendering the design erroneous. In terms of the Verilog description shown in Figure 5, this error can be caused by replacing value 0 on the right hand side of the assignment statement $out = 0$ on line 20 by the value 1.

5 Experimental Results

Experimental results on the benchmarks are shown in Tables 4 and 5. We perform all model checking and simulation experiments using a 1.5 GHz Sun UltraSPARC CPU running SunOS version 5.9.

5.1 Error Distribution

We inject 100 design errors into each benchmark design for analysis. For almost all benchmarks, we are able to insert more design errors but instead limit the number of design errors to 100 for space and complexity reasons.

The Table 2 shows the maximum possible errors in the benchmark design by applying various operators. The first column represents the benchmark name. *CCO* and *BO* depict conditional construct and branch omission. *AOR*, *LOR* and *ROR* signify arithmetic, logical and relational operator replacements respectively. *Reduction* represents the errors associated with changes applied to reduction operators.

The Table 2 shows the distribution of errors injected in the benchmark design by applying various operators.

5.2 Error Detection Results

Table 4 shows the error detection results of model checking. The first three columns are labeled **Design** (representing the design name), **LOC** (the number of lines of code), and **Inputs** (the number of inputs). The **ED** (errors detected) column contains the number of design errors of the 100 injected that are detected using model checking. The remaining design errors evaded detection for one of two reasons: either the model checking process could not complete within the memory limit or model checking completed but the design error failed to violate a property. The mem-

Design	CCO	BO	AOR	LOR	ROR	Reduction
am2901	15	57	28	122	161	12
am2910	9	27	24	329	357	6
bpb	17	45	40	11	0	114
bufa	19	54	16	64	42	0
counter	2	6	8	6	0	0
fifo	5	13	32	82	91	0
gcd	7	29	12	106	105	0
huffman	5	47	8	0	21	0
microwave	5	20	0	0	0	0
miim	40	109	12	312	91	0
nim	1	19	52	50	140	0
nullmodem	6	23	16	16	63	192
palu	6	19	12	16	21	0

Table 2. Error Space

Design	CCO	BO	AOR	LOR	ROR	Reduction
am2901	4	15	7	32	40	2
am2910	3	8	8	40	40	1
bpb	16	30	20	2	0	32
bufa	10	25	10	33	22	0
counter	2	6	8	6	0	0
fifo	2	6	16	38	38	0
gcd	6	10	12	40	32	0
huffman	5	47	8	0	21	0
microwave	5	20	0	0	0	0
miim	8	20	0	55	17	0
nim	1	8	20	20	51	0
nullmodem	6	21	10	10	20	33
palu	6	19	12	16	21	0

Table 3. Error Distribution

Design	Tot. Undet. Err.	Low Cov.	Data Aliasing
miim	75	49	26
nim	26	16	10
nullmodem	64	57	7

Table 6. Undetected Error Distribution

ory and performance measurements of model checking for the detected errors are shown in the remaining columns. Columns **Mem. Avg** and **Mem. WC** show the average and worst case memory use values, and columns **CPU Avg** and **CPU WC** show the average and worst case CPU times. The memory use and CPU time requirements vary widely across examples.

Simulation-based validation requires less CPU time and memory than verification methods using model checking, with simulation memory requiring consistently 2 to 3 orders of magnitude less than model checking memory requirements.

5.3 Analysis of Error Detection

Based on our analysis of the error detection results, low coverage and data aliasing are the *only* two causes for the errors which are not detected. These two causes can be addressed to account for these errors by using test generation techniques that have already been presented [**?**].

Table 5 shows that there are only three benchmarks with errors that are not all detected by simulation: *miim, nim*, and *nullmodem*. We have manually examined the each undetected error to determine it source cause. We have found that all of the undetected errors are caused by either low coverage or by data aliasing. The Table 6 shows the detailed distribution of undetected errors between the two categories.

Low Coverage The completely random stimulus generation that is used in simulation has its weaknesses in not being able to attain a 100% statement and branch coverage for a few benchmark examples. Especially in the examples Miim, Nim and Nullmodem, some injected errors are not detected because the statements or branches where these errors are embedded are never executed. The example, Nullmodem, attains a maximum statement coverage of 72.46% and a branch coverage of 47.37%. In this case, 64 out of the total 100 errors injected are not detected. The reason is that these errors are embedded on complex control flow paths which are not activated by the random stimulus generation. Out of the total of 165 undetected errors over all the benchmark examples, 122 fall under this category.

Figure 7 shows an assign statement that describes a heavily nested control flow that is very difficult to cover completely. An error on one of these paths is difficult to detect if stimulus does not activate that path. For example, if $busy[10]$ on line 7 in Figure 7 is changed to $busy[10]$, it does not result in an erroneous output from the design if line 7 is not executed. This injected error is not detected.

Data Aliasing Data aliasing occurs in case of conditional expressions which are assigned to other signals. If an error is injected on the expression of a conditional predicate and this error does not cause the conditional expression to evaluate to a value different than its correct value, then this kind of error is never detected. The expression may be based on a very specific value of the signal under question, such that it becomes a corner case to exercise. Out of the total of 165 undetected errors over all the benchmark examples, 43 fall under this category. This category does not include the errors which are associated with a branch causing the error to never be exercised. Such errors lie under the low coverage category because of low branch coverage.

The assign statement shown in Figure 8 (a) is checking if the value of *count* is 16. An error is injected in this predicate as shown in Figure 8 (b) by replacing the $==$ operator by a $>$ operator. If the value of *count* being 16 or more is a corner case such that it does not occur during the simulation, the conditional predicate will evaluate to the same value in both the cases (a) and (b).

An important observation of this paper is that there are only two causes for errors being undetected by simulation, low coverage and data aliasing. For our experiments, we have used random test generation, but many approaches have been presented in previous work to address these two problems. Several test generation techniques exist to achieve high coverage [**?**] and to avoid data aliasing [**?**]. If these existing test generation techniques are employed, then simulation would have detected all of the errors detected by model checking.

6 Conclusions

We present a comparison of the error detection abilities of model checking and simulation-based verification and restrict our comparison to simulation using random pattern generation and high line coverage as a termination condition. Manual interaction in the verification process is minimized in our experiments by restricting the design errors examined and using an automatic test generation process. The results reveal a weakness in the error detection ability given the test generation process we use for simulation.

Design Information			**Model Checking Results**				
Design	LOC	Inputs	ED	Mem. Avg (MB)	Mem. WC (MB)	CPU Avg. (sec)	CPU WC (sec)
am2901	140	10	0	∞	∞	∞	∞
am2910	111	7	9	17.97	26.54	2.2	3.4
bpb	111	6	3	13.14	13.14	0.9	0.9
bufa	86	3	67	47.22	69.28	18.8	122.1
counter	42	0	20	6.21	6.64	0.1	0.1
fifo	147	3	4	64.68	66.01	14.3	18.7
gcd	147	3	35	35.36	70.44	48.6	397.1
huffman	227	1	92	18.85	29.86	1.8	3
microwave	42	4	57	6.45	7.27	0.1	0.1
miim	841	10	98	13.51	39.92	1.9	53
nim	121	2	33	57.17	87.82	233.2	574.3
nullmodem	262	2	100	10.91	11.27	0.3	0.4
palu	115	5	16	9.49	9.78	0.3	0.4

Table 4. Benchmark Information and Model Checking Results

Design	Mem (MB)	CPU (s)	SC	BC	EC (%)
am2901	0.01	0.28	100.00	100.00	-
am2910	0.01	0.28	100.00	100.00	100.00
bpb	0.01	0.27	100.00	100.00	100.00
bufa	0.01	0.19	100.00	100.00	100.00
counter	0.03	0.24	20	100.00	100.00
fifo	0.01	0.22	100.00	100.00	100.00
gcd	0.01	0.29	98.31	93.75	100.00
huffman	0.01	0.26	98.77	100	100.00
microwave	0.01	0.21	100	100	100.00
miim	0.01	0.22	96.19	89.06	23.47
nim	0.01	0.26	36.27	83.33	21.21
nullmodem	0.01	0.22	72.46	47.37	36.00
palu	0.01	0.21	100.00	100.00	100.00

Table 5. Simulation Verification Results

```
1. assign alloc_adder =
2.         ~busy[0] ? 0 : ~busy[1] ? 1 :
3.         ~busy[2] ? 2 : ~busy[3] ? 3 :
4.         ~busy[4] ? 4 : ~busy[5] ? 5 :
5.         ~busy[6] ? 6 : ~busy[7] ? 7 :
6.         ~busy[8] ? 8 : ~busy[9] ? 9 :
7.         ~busy[10] ? 10 : ~busy[11] ? 11 :
8.         ~busy[12] ? 12 : ~busy[13] ? 13 :
9.         ~busy[14] ? 14 : ~busy[15] ? 15 :
10.        0;
```

Figure 7. Complex Control Flow

```
assign nack = alloc & (count ==  16);
```

(a)

```
assign nack = alloc & (count >  16);
```

(b)

Figure 8. Data Aliasing Example

References

[1] Vcs. http://www.synopsys.com.

[2] D. Abts and M. Roberts. Verifying large-scale multiprocessors using an abstract verification environment. In *Design Automation Conference*. ACM Press, 1999.

[3] R. Alur and T. Henzinger. Vis: A system for verification and synthesis. In *International Conference on Computer Aided Verification*, pages 428–432, July 1996.

[4] B. Bentley. Validating the intel pentium 4 microprocessor. In *Design Automation Conference*, 2001.

[5] E. J. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[6] G. A. Hayek and C. Robach. From specification validation to hardware testing: A unified method. In *International Test Conference*, pages 885–893, October 1996.

[7] K. N. King and A. J. Offutt. A fortran language system for mutation-based software testing. *Software Practice and Engineering*, 21(7):685–718, 1991.

[8] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[9] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[10] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on multivalued ar-automata. In *Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 2001.

[11] B. Wile, J. Goss, and W. Roesner. *Comprehensive Functional Verification : The Complete Industry Cycle*. Morgan Kaufmann Publishers, 2005.