A Coverage Metric for the Validation of Interacting Processes

Ian G. Harris
Department of Computer Science
University of California Irvine
Irvine, CA 92697 USA
harris@ics.uci.edu

Abstract

We present a coverage metric which evaluates the testing of a set of interacting concurrent processes. Existing behavioral coverage metrics focus almost exclusively on the testing of individual processes. However the vast majority of practical hardware descriptions are composed of many processes which must correctly interact to implement the system. Coverage metrics which evaluate processes separately are unlikely to model the range of design errors which manifest themselves when components are integrated to build a system. A metric which models component interactions is essential to enable validation techniques to scale with growing design complexity. We describe the effectiveness of our metric and provide results to demonstrate that coverage computation using our metric is tractable.

1. Introduction

Simulation-based validation is necessary to verify practical hardware designs today because simulation can be performed for much larger designs than can be considered using formal verification techniques. Validation using simulation requires the development of a test sequence which can reveal design errors which may be present. Test generation may be performed automatically or manually, but in either case there is a strong need for an empirical measure of the effectiveness of a test sequence. A measure of test effectiveness acts as a completion criterion for the test generation process. Such a measure also provides information about which aspects of the design are as yet untested; this information is used to direct the test generation process toward the detection of unrevealed design errors. A measure of test effectiveness is typically refered to as a *coverage metric* and many coverage metrics have been developed for both hardware and software testing [19, 12, 1]. Coverage metrics define a set of criteria which must be satisfied during simulation to ensure detection of design errors.

A range of different coverage metrics have been developed for use at different design abstraction levels, (i.e. gate-level, register-transfer level, state machine, behavioral) and to describe different types of errors (i.e. physical, control-flow, dataflow). We are interested in a coverage metric at the behavioral level because the behavior is the earliest stage at which an executable design description is available which can be evaluated automatically. Detecting design errors early in the design cycle reduces the expense of the redesign needed to correct an error.

All practical system designs are built from a set of interacting concurrent processes, but almost all existing behavioral coverage metrics consider the testing of processes individually. This is problematic because design errors are most likely to be found in the interaction between multiple components, rather than in any single component. A hierarchy is always imposed on the design process in an effort to improve productivity by partitioning the responsibilities of different designers. The use of intellectual property exemplifies this practice by completely separating the design of a component, possibly outsourcing it to a different company.

Partitioning the design provides an abstraction, hopefully allowing the system designer to ignore details of the components. The disadvantage of the use of this abstraction is that it is difficult for one designer to understand the complex interactions between all components. This problem is most acute with the use of intellectual property because the detailed design information is likely to be hidden from the system designer. Design errors which appear as a result of the interaction between components are likely to occur and difficult to detect.

Existing metrics are applied to multi-process designs by first combining all processes into a single, complex behavioral description. For example, state coverage is a state machine metric which requires that all states are entered dur-

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

3-9810801-0-6/DATE06 © 2006 EDAA

ing simulation. State coverage can be applied to a multi-process design by computing the cross-product machine of all of the processes, and then requiring that each state in the cross-product machine is covered. The problem with this approach is not only that the cross-product machine is large, but also that the vast majority of the cross-product machine is redundant in most cases [13]. Use of a cross-product machine implicitly assumes that the individual processes are independent, but this is never true. As a result the cross-product machine will contain many states and transitions which can never be executed. Coverage values for a cross-product machine will be deceptively low because the majority of the state space cannot be explored.

A behavioral coverage metric which focuses on the interaction between processes is needed but coverage computation must be tractable. The number of interactions cannot be too large to enable fast analysis. The set of interactions considered must be pruned to contain only those which are not redundant and which are most likely to reveal design errors.

We present a coverage metric which evaluates the validation of the interactions between processes. We model the behavior of each process as a control-flow graph (CFG) and we assume that executing all control-flow paths in a single process is sufficient to validate that process. An interaction is described by a set of paths in different processes, executed in sequence. In the worst case the set of potential interactions could be as large as the cross-product of the sets of paths in the individual machines. This potential problem is addressed by identifying path pairs which conflict because the signal assignments of the first path violate the control-flow conditions of the second path. Additionally, path pairs are only considered as interactions if the second path is directly dependent on the first path via a shared signal. Our results show that when these restrictions are considered, process interactions can be validated with low time complexity.

The remainder of the paper is organized as follows. Section 2 describes related work in coverage metrics and interacting machines. Our formal definition of an interaction is described in Section 3 with the criteria for interactions which are important for validation. The system which we have implemented to compute interaction coverage is presented in Section 4. Experimental results are shown in Section 5 and Section 6 summarizes the key points of the coverage metric.

2. Related Work

A coverage metric defines a set of *coverage goals* which must be satisfied during simulation. Ideally, satisfaction of all coverage goals should indicate that all possible design errors are detected. A coverage metric can be used to evaluate a test sequence by determining the fraction of coverage goals which are satisfied when the design is simulated with the test sequence.

Existing coverage metrics have their origins in either the hardware [19] or the software [1] domains. Finite state machines (FSMs) are the classic method of describing the behavior of a sequential system and fault models have been defined to be applied to state machines. State machine coverage metrics assume that a design error impacts the structure of the state machine, the states and the transitions between them. The commonly used fault models [2, 17, 16] are the *state coverage* model which requires that all states be reached, and *transition coverage* which requires that all transitions be traversed. The problems associated with state machine testing are understood from classical switching theory [14] and are summarized in a thorough survey of state machine testing [15].

A number of coverage metrics are based on the traversal of paths through the CFG representing the system behavior. Applying these metrics to the CFG representing a single process is a well understood task. The application of CFG metrics to the behavior of an entire system would require that all component CFGs be merged into one. For this reason, CFG metrics are currently restricted to the testing of single processes. The earliest CFG coverage metrics include statement coverage, branch coverage and path coverage [1] models used in software testing. There are many notable uses of CFG coverage metrics for hardware validation [23, 11, 5, 25]. Many CFG coverage metrics consider the requirements for fault activation without explicitly considering fault effect observability. Researchers have developed observability-based behavioral coverage metrics [6, 8, 21, 22] to alleviate this weakness.

Interacting finite state machines are a computational model which has been used for hardware/software codesign [10, 3]. The majority of research on interacting FSMs has investigated synthesis optimization either using sequential don't cares [24] or by identifying redundant faults [9]. Validation of an FSM network has been investigated in [13]. The problem addressed in [13] is the use of simulation to verify that a network of FSMs is equivalent to a given prototype FSM. The outputs of the prototype machine and the FSM network are compared while acheiving 100% transition coverage on the prototype machine. The reserach in [13] is not applicable if a prototype machine does not exist for comparison.

Research in the field of software engineering has produced coverage metrics for the testing of concurrent software. In [20] a set of coverage metrics are formulated in terms of paths in the *concurrency graph* which represents the communication possibilities between threads. A set of testing criteria for rendezvous communication are presented in [18]. A metric for multi-threaded Java programs is presented [7] which requires that each method be interrupted

by other methods in the same class. The relationship between these concurrency metrics and the detection of bugs in concurrent code has not been demonstrated empirically. In addition, the coverage produced by applying these metrics to benchmark systems has not been published. As a result, it is impossible to determine whether or not these earlier metrics actually correlate to bug detection.

3. Interaction Definition

The set of interactions must describe all of the ways in which the behavior of a set of processes can affect the behavior of another set of processes. For the purposes of this paper we will only consider interactions between pairs of processes. When considering only pairs of processes, the set of interactions must describe all the ways in which the behavior of one process can impact the behavior of another process. The execution of one process may impact the execution of another process through its effect on the global state, which in VHDL is the set of signals connecting the processes. The global state can be seen as the $context$ in which a process is executed. To evaluate the interactions between processes we need to execute each control-flow path of each process in a range of different contexts. An interaction between two processes is a sequence of control-flow path executions, one path in the first process and one in the second. The interaction is of interest if the execution of the first process alters the context of the execution of the second process.

To formalize the notion of an interaction, we need to enumerate the set of all behaviors of a process. We will assume that the set of all control-flow paths in a process can be used to describe all of the possible behaviors of that process. We assume that there is a set of concurrent processes T, and that each process $t \in T$ has a set of control-flow paths P_t. Each path $p \in P_t$ is defined by a set of predicates encountered along the path in the CFG. The set of conditional predicates which are encountered and satisfied along a path p is C_p. Without loss of generality, each conditional predicate $c \in C_p$ is expressed so that it is satisfied along the path p. For example, the VHDL in Figure 1a contains a path p which involves two predicates which evaluate to FALSE, $b < 3$ and $c > 1$. Since we define C_p to contain only conditional predicates which are positively asserted, both predicates are inverted, so $C_p = !(b < 3), !(c > 1)$.

Each path p contains a set of signal assignments $a \in A_p$, a set of signals $r \in R_p$ whose values are used in the path, and a set of signals $w \in W_p$ whose values are assigned in the path. For example, refer to the path p in Figure 1a defined by predicates $C_p = !(b < 3), !(c > 1)$. This path contains assignments $A_p = (y \Leftarrow in), (x \Leftarrow 5)$, it reads signals $R_p = b, c$, and writes signals $W_p = x, y$. Since we are only interested in interactions between processes, the sets A_p, R_p, and W_p only involve internal signals which are used to communicate between processes. In Figure 1 this means that signals a and b and input and output ports in and out are not considered.

```
     signal a, b, c, x, y : integer;

if (b < 3) then
   a <= b;
else
   y <= in;
end if;
if (c > 1) then
   x <= 1;
else
   x <= 5;
```

(a)

```
if (x > 2)
   out <= y;
else
   out <= 1;
```

(b)

Figure 1. VHDL example, (a) process 1, (b) process 2

We refer to the set of all interactions between a pair of processes t_1 and t_2 as I_{t_1,t_2}. We define an interaction between a pair of processes as a sequence of paths, one in each process, $i \in I_{t_1,t_2} = (p_1, p_2),\ p_1 \in P_{t_1}, p_2 \in P_{t_2}$.

3.1. Path Dependencies

The set of all interactions between a pair of processes t_1 and t_2 is a subset of the cross-product between P_{t_1} and P_{t_2}. The set of all interactions should be a small subset of the cross-product because many path pairs are not interesting for testing purposes. Each interaction captures a functional dependency between the interacting processes. To capture dependencies, the second path involved in an interaction must be $dependent$ on the first path in the sequence via a set of signals. This requirement is stated formally in Equation 1. An example of dependency can be seen between path p_1 in Figure 1a where $C_{p_1} = (b < 3), !(c > 1)$, and path p_2 in Figure 1b where $C_{p_2} = (x > 2)$. Path p_2 depends on path p_1 through their mutual access of signal x.

$$DEP(p_1, p_2) \Rightarrow |W_{p_1} \cap R_{p_2}| > 0 \qquad (1)$$

An interaction is considered to be $covered$ during testing if its two paths p_1 and p_2 are executed in sequence and no path p_3 is executed in between the paths which assigns a value to a signal which is both assigned by p_1 and read by p_2. This can be described using paths in Figure 1a and 1b. Consider two paths in Figure 1a, p_1 and p_3, where $C_{p_1} = (b < 3), !(c > 1)$ and $C_{p_3} = !(b < 3), !(c > 1)$. Both paths p_1 and p_3 assign signal x and therefore form interactions

with path p_2 in Figure 1b where $C_{p_2} = (x > 2)$. If the execution sequence of paths during testing is p_1, p_3, p_2 then the interaction (p_3, p_2) is covered but the interaction (p_1, p_2) is not covered since p_3 was executed closer to p_2 in sequence.

3.2. Interaction Feasibility

In addition to the dependency requirement between the paths of an interaction, the interaction must also be $feasible$ in terms of the possibility of executing the interacting paths in sequence. Consider an interaction involving the path p_1 in Figure 1a described by $C_{p_1} = (b < 3), (c > 1)$ and the path the path p_2 in Figure 1b described by $C_{p_2} = (x > 2)$. This interaction is infeasible because path p_1 cannot be executed immediately prior to the execution of path p_2. The path sequence p_1, p_2 is infeasible because p_1 assigns signal x to 1 while p_2 requires signal $x > 2$.

In general, an interaction between two paths p_1 and p_2 is infeasible if the signal assignments A_{p_1} collectively imply the inverse of one or more of the conditional predicates in C_{p_2}. Identifying this condition in the most general way is intractable because the SATISFIABILITY problem can easily be reduced to it. Instead, we simplify the problem in a way which is sufficient to identify infeasible interactions in most practical designs.

A conditional expression can be easily evaluated if all of its signals and/or variables are bound to constant values. If all of the unbound signals of $c \in C_{p_2}$ are assigned to constant values by some assignment $a \in A_{p_1}$ then path p_1 is said to *uniquely determine* conditional expression c. When a conditional is uniquely determined the evaluation of the conditional expression is trivial. We determine if an interaction between two paths p_1 and p_2 is infeasible by substituting the assigned signal values of p_1 into each conditional expression in p_2. If a conditional in p_2 is uniquely determined and evaluates to FALSE then the interaction is infeasible. This computation is stated formally in Equation 2.

$$IF(p_1, p_2) \Rightarrow \exists c \in C_{p_2}, \overline{SUB(c, A_{p_1})} \qquad (2)$$

In Equation 2, the function $SUB(c, A_{p_1})$ evaluates to TRUE iff the conditional expression c is uniquely determined by path p_1 and c evaluates to TRUE upon substitution of its unbound signals with corresponding assignments in A_{p_1}.

4. System Overview

We have implemented a system to compute interaction coverage. The structure of the system is shown in Figure 2. The system takes a behavioral VHDL description and a VHDL testbench as its inputs. Our implementation is currently limited to accept only VHDL which contains no structural constructs in the form of PORT MAP (except in the testbench) and no variable-length loops. The *Path Analysis* block in Figure 2 is the code which extracts the set of all control-flow paths in each VHDL process, and the set of all system interactions which are feasible. The VHDL is simulated with its testbench to generate trace information which indicates the control-flow paths executed at each time step. While we used Synopsys vhdlsim for simulation, any VHDL simulator could have been used. We manually inserted *write* and *writeline* statements into the VHDL description to generate trace information during simulation. The *Trace Analysis* step examines the trace information to determine which paths and interactions were executed during simulation, thereby computing interaction coverage and path coverage for comparison.

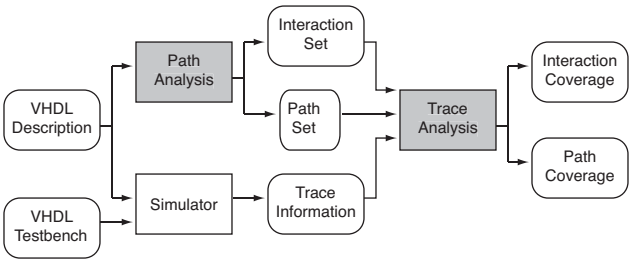

Figure 2. Interaction Coverage System

5. Experimental Results

We evaluate our interaction coverage metric by estimating its ability to model the detection of interaction-related design errors. We also estimate the detection ability of the traditional path coverage metric for the same set of design errors for comparison. We assume that an ideal coverage metric would exactly reflect the fraction of all possible design errors detected. We refer to the fraction of all possible design errors detected by a test sequence as the *Error Coverage*. Our interaction coverage metric can be evaluated by observing how close its coverage value is to the true error coverage for a given test sequence. Error coverage cannot be determined exactly because it would require the enumeration of all possible design errors. Instead, we approximate error coverage by injecting potential errors into a design, one at a time. An error is detected if the output sequence of its corresponding machine is different from the output of the correct machine. By sequentially simulating many erroneous machines we produce an estimate of the error coverage.

The accuracy of the error coverage is dependent on the nature of the errors which are injected into the design. Since

LOC	Processes	Signals
569	4	7

Table 1. Information about $b12$

Interactions	Paths	CPU (sec)
94	76	3.18

Table 2. Result Summary

our metric estimates the detection of error related to interactions between components, we restrict our error insertion to those which impact the signals which pass data between processes. This means that errors are not injected which directly impact the primary inputs, primary outputs, or the variables internal to each process.

The errors which we inject are signal assignments of the form $sig \Leftarrow val;$, where sig is an internal signal and val is a value in the domain of that signal. Insertion of these errors models the impact of performing system design without complete and correct understanding of the operaton of a component. An error reflects some operation performed by a process which was not considered during system design. These errors are inserted randomly but their insertion is restricted to those which produce correct VHDL; errors are inserted between the BEGIN and END of a process, and sig is a signal which is assigned by the process in which the error is inserted.

5.1. Benchmark Information

We have evaluated our coverage metric by applying it to the $b12$ example which is part of the ITC99 benchmark suite [4]. The $b12$ example was chosen because it is composed of multiple concurrent processes, unlike the benchmarks earlier in sequence, $b01$ through $b11$. The $b12$ benchmark also does not contain structural constructs (i.e. PORT MAP) as do the largest benchmarks in the suite.

Information about the $b12$ benchmark is presented in Table 5.1. In the Table 5.1, **LOC** refers to the number of lines of VHDL code, **Processes** is the number of processes, and **Signals** is the number of signals in the example. Only signals which are used to communicate between processes are considered, so input and output ports are not counted.

5.2. Interaction Coverage Results

Table 5.2 presents a summary of the results of interaction coverage computation and path coverage computation for comparison. In the table, **Paths** is the total number of control-flow paths in the the individual processes and **Interactions** is the number of interactions which were found to be feasible and involved dependent paths. **CPU** is the CPU time required to perform *Path Analysis* and *Trace Analysis* on a 867MHz PowerPC G4 processor machine with 640MB RAM, running Mac OS X version 10.3.9.

Figure 3 shows the interaction coverage, path coverage, and error coverage results. The horizontal axis of the graph is the number of test patterns applied, and the vertical axis is the coverage percentage. Three curves are shown in Figure 3 which show interaction coverage, path coverage, and error coverage. Error coverage was computed by injecting 50 random errors. A total of 20,000 test patterns were applied. All input bits were random except for the $reset$ and $start$ input signals whose probabilities were reduced to prevent the system from being reset too frequently. After all test patterns are applied, the Interaction Coverage achieved is 56.4%, Path Coverage is 85.5%, and Error Coverage is 38.0% . The relative positions of the three curves in Figure 3 shows that interaction coverage consistently provides a much better estimate of error coverage than path coverage does.

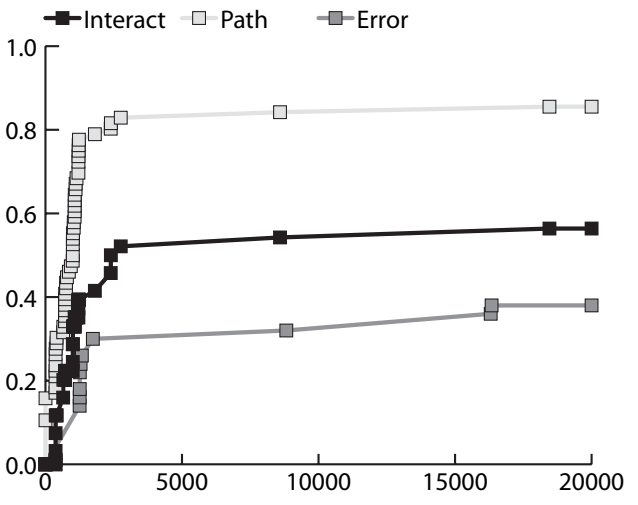

Figure 3. Interaction Coverage and Path Coverage Results

6. Conclusions

We have presented a coverage metric to model the interactions between multiple concurrent processes. Interactions between complex components are difficult for any one designer to understand, making design errors related to component interaction difficult to detect. Our coverage metric

models the meaningful interactions between components, while ignoring those interactions which are infeasible or unlikely to reveal errors. In this way, the number of interactions for evaluation is reduced, making coverage computation tractable. The work presented in this paper is restricted to interactions between process pairs but it is easily extended to consider larger sets of interacting processes at greater computational expense.

References

[1] B. Beizer. *Software Testing Techniques, Second Edition*. Van Nostrand Reinhold, 1990.

[2] K.-T. Cheng and J.-Y. Jou. A functional fault model for sequential machines. *IEEE Transactions on Computer-Aided Design*, 11(9):1065–1073, September 1992.

[3] M. Chiodo, P. Giusto, A. Jurecska, H. Hsieh, A. Sangiovanni-Vincentelli, and L. Lavagno. Hardware-software codesign of embedded systems. *IEEE Micro*, pages 26–36, August 1994.

[4] F. Corno, M. S. Reorda, and G. Squillero. Rt-level itc 99 benchmarks and first atpg results. *IEEE Design and Test of Computers*, pages 44–53, July-August 2000.

[5] F. Corno, M. S. Reorda, G. Squillero, A. Manzone, and A. Pincetti. Automatic test bench generation for validation of RT-level descriptions: an industrial experience. In *Design Automation and Test in Europe*, pages 385–389, 2000.

[6] S. Devadas, A. Ghosh, and K. Keutzer. An observability-based code coverage metric for functional simulation. In *International Conference on Computer-Aided Design*, pages 418–425, November 1996.

[7] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, July 2003.

[8] F. Fallah, S. Devadas, and K. Keutzer. Occom: Efficient computation of observability-based code coverage metrics for functional verification. In *Design Automation Conference*, pages 152–157, June 1998.

[9] F. Ferrandi, F. Fummi, E. Macii, M. Poncino, and D. Sciuto. Symbolic optimization of interacting controllers based on redundancy identification and removal. *IEEE Transactions on Computer-Aided Design*, 19(7):760–772, July 2000.

[10] D. D. Gajski and F. Vahid. Specification and design of embedded hardware-software systems. *IEEE Design and Test of Computers*, 12(1):53–67, 1995.

[11] A. Hajjar, T. Chen, and A. von Mayrhauser. On statistical behavior of branch coverage in testing behavioral vhdl models. In *High Level Design Validation and Test Workshop*, pages 89–94, 2000.

[12] I. G. Harris. Hardware-software covalidation: Fault models and test generation. *IEEE Design and Test of Computers*, 20(4):40–47, July-August 2003.

[13] Z. Hasan and M. Ciesielski. Functional verification & simulation of fsm networks. In *VLSI Test Symposium*, pages 326–332, April 1993.

[14] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw Hill, 1978.

[15] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. *IEEE Transactions on Computers*, 84(8):1090–1123, August 1996.

[16] N. Malik, S. Roberts, A. Pita, and R. Dobson. Automaton: an autonomous coverage-based multiprocessor system verification environment. In *IEEE International Workshop on Rapid System Prototyping*, pages 168–172, June 1997.

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

[18] T. K. Shih, C.-M. Chung, Y.-H. Wang, Y.-F. Kuo, and W.-C. Lin. Software testing and metrics for concurrent computation. In *Third Asia-Pacific Software Engineering Conference*, pages 336–344, 1996.

[19] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design and Test of Computers*, 18(4):36–45, July/August 2001.

[20] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, March 1992.

[21] P. A. Thaker, V. D. Agrawal, and M. E. Zaghloul. Validation vector grade (VVG): A new coverage metric for validation and test. In *VLSI Test Symposium*, pages 182–188, 1999.

[22] S. Verma, K. Ramineni, and I. G. Harris. An efficient control-oriented coverage metric. In *Asian-Pacific Design Automation Conference*, 2005.

[23] A. von Mayrhauser, T. Chen, J. Kok, C. Anderson, A. Read, and A. Hajjar. On choosing test criteria for behavioral level harware design verification. In *High Level Design Validation and Test Workshop*, pages 124–130, 2000.

[24] H. Y. Wang and R. K. Brayton. Exploitation of input don't care sequences in logic optimization of fsm networks. In *International Conference on Computer-Aided Design*, pages 728–735, November 1995.

[25] Q. Zhang and I. G. Harris. A data flow fault coverage metric for validation of behavioral hdl descriptions. In *International Conference on Computer-Aided Design*, November 2000.