Automatic Generation of Functional Coverage Models from Behavioral Verilog Descriptions

Shireesh Verma
shireesh@ics.uci.edu

Ian G. Harris
harris@ics.uci.edu

Kiran Ramineni
kiran@ics.uci.edu

Department of Computer Science
University of California Irvine
Irvine, CA 92697, USA

ABSTRACT

As an industrial practice, the functional coverage models are developed based on a high-level specification of the Design Under Verification (DUV). However, in the course of implementation a designer makes specific choices which may not be reflected well in a functional coverage model developed entirely from a high-level specification. We present a method to automatically generate implementation-aware coverage models based on the static analysis of a HDL description of the DUV. Experimental results show that the functional coverage models generated using our technique correlate well with the detection of randomly injected errors into a design.

1. INTRODUCTION

Functional verification is known to be a difficult task accounting for 70% of the development time [22, 3]. Simulation-based verification and formal verification are two vehicles used for this task. Simulation-based verification is the principle means of verification well accepted in the industry [8, 9, 7, 23, 17] due to the tractability and usability of the simulation process. Simulation-based verification involves several steps, such as test generation and response evaluation, but coverage models [10] are central to all steps in the simulation process. A coverage model defines a set of criteria that are used to determine which design errors are detected by a test sequence. A coverage model provides an empirical measure of the completeness of a test sequence [11] and the error detection criteria can be used to direct the test generation process[5].

Most behavioral coverage metrics focus on the syntactic properties of a design [4], placing little emphasis on coverage from a functional standpoint. For example, statement coverage seeks maximum number of statements be executed, the number depending on the adequacy criteria being considered. Similarly, branch and path coverage criterion seek adequate coverage on branches and paths [14]. These metrics do not ensure completeness with respect to the implementation. In order to instill a measure of coverage confidence with respect to the functionality, a functional coverage model is required. The construction of such a coverage model is an entirely manual process [12, 21] even with the state of art in industry today. The development of a functional coverage model is done based on a thorough study of the high-level design specification documents written in English. A list of design features is extracted thereafter. Each of these features is interpreted in terms of a relation between signals in the executable design description. These relationships are expressed in terms of **coverage monitors** written in a Hardware Verification Languages (HVLs) such as *e*, SystemC, and Vera. These coverage monitors keep track of coverage of design functionality and provide a numerical measure for it.

A major downside to using such a model emanates from the inherent gap between specification and implementation. A hardware designer interprets an abstract specification and makes choices which culminate in an implementation [24]. In the course of this process the designer may introduce some restrictions on the behavior otherwise allowable from interpretation of only the specification. A functional coverage model derived only from specification is bound to miss such fine granular intricacies introduced in the behavior.

To the best of our knowledge there is no general approach to generate functional coverage models automatically based on the implementation. In this work we propose a method which allows generation of a functional coverage model based on a static analysis of the HDL description of the DUV, which allows us to obtain for every signal, a dependency chain of signals and the restrictions imposed on their values domains. The output of our technique is a set of *coverage groups* in Vera [13] which collectively describe the functional coverage model. Each coverage group specifies a function of a set of signals in the design. The function is evaluated at each clock cycle during simulation. Functional coverage is the fraction of coverage groups whose functions evaluate to TRUE at some point during simulation. The coverage groups are defined so that satisfying all coverage functions during simulation indicates that the test stimuli thoroughly explore the functionality as it related to the given signals. Functional coverage using our technique is well correlated to error detection.

The remainder of the paper is organized as follows. An example illustrating the need of an precise method for development of implementation-cognizant coverage models is shown in Section 3. The Overview of our approach is summarized in Section 4. The construction and algorithm for our static analysis approach is explained in Section 5. The experimental infrastructure required is discussed in Section 6. The

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

978-3-9810801-2-4/DATE07 © 2007 EDAA

```
01: always @(*)
02:   begin
03:   if(a)
04:     if(wr_en)
05:       b = 1;
06:   end
```

Figure 1: Functional Coverage Example

technique used to simulate real design errors is presented in Section 7. The results are presented in Section 8 and conclusions are described in Section 9.

2. PREVIOUS WORK

Previous research has explored the development and use of functional coverage models. Research presented in [6] proposes manual extraction of an FSM based coverage model from design specification. Work described in [12, 25] introduces a manually developed coverage model based on cross-product combinations of signal values. In [24] constraints specified as boolean expressions are used for generating coverage model for interface protocols. Research presented in [19] builds a hierarchical temporal event relation graph for constraints described in FLTL [22] and an event coverage model is generated by an eliminative traversal of the graph. In [16] authors propose a way to measure completeness of a set of properties expressed in CTL [20]. For each signal involved in the set of properties, they find the set of design states, where changing the value of that signal could invalidate a property from the given set of properties under verification. A union of all such design state sets is taken over all such signals. The number of states in the union expressed as a percentage of the total number of design states is considered an indicator of property coverage. However, the complexity of this process is same as that of model checking. The work described in [18] outlines a formal method to compare a specification to its corresponding implementation. They lay down four criteria for detecting a slack between specification and implementation. If any of these criteria is found to be a non-empty set then either the specification is under-specified or implementation is not in compliance with the intent of specification. But, the complexity of this process is worse than that of model checking.

3. FUNCTIONAL COVERAGE PROBLEM

A complete functional coverage model must consider details of the implementation because the implementation can include relationships between signals which are not described in the specification. Consider a small Verilog design example shown in Figure 1 consisting of three boolean signals a, b and wr_en. Assume that the specification requires the CTL property in Equation 1 must always hold in order for the design to be functionally correct.

$$AG(a \rightarrow b) \tag{1}$$

The property in Equation 1 requires that if the value of a is 1 then the value of b must be 1. Any deviation from this will be considered an erroneous behavior. This property involves only the signals a and b. Each of the two signals has a domain of $\{0,1\}$. Hence, a total of following four *valuations* exist.

$$(\mathbf{a},\mathbf{b}) = \{(\mathbf{0},\mathbf{0}),(\mathbf{0},\mathbf{1}),(\mathbf{1},\mathbf{0}),(\mathbf{1},\mathbf{1})\}$$

Of the four possible valuations, three of them are legal $\{(0,0),(0,1),(1,1)\}$, and one is illegal $\{(1,0)\}$. However, the analysis of implementation in Figure 1 reveals an additional dependency on the signal wr_en. In this case, a total of following eight *valuations* exist.

$$(\mathbf{a},\mathbf{b},\mathbf{wr\_en}) = \{(\mathbf{0},\mathbf{0},\mathbf{0}),(\mathbf{0},\mathbf{0},\mathbf{1}),(\mathbf{0},\mathbf{1},\mathbf{0}),(\mathbf{0},\mathbf{1},\mathbf{1}),$$

$$(\mathbf{1},\mathbf{0},\mathbf{0}),(\mathbf{1},\mathbf{0},\mathbf{1}),(\mathbf{1},\mathbf{1},\mathbf{0}),(\mathbf{1},\mathbf{1},\mathbf{1})\}$$

Of the eight possible valuations, seven of them are legal $\{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,1,0),(1,1,1)\}$, and one is illegal $\{(1,0,1)\}$. It can be noticed that the now legal valuation, $(a,b,wr\_en) = (1,0,0)$ violates the property depicted in Equation 1 since $(a,b) = (1,0)$. In this case if the design is not analyzed and coverage model is developed only from the high level property, the simulation results will indicate a false design error.

Here, each valuation of the signals will become a coverage monitor. The set of coverage monitors will be divided into two groups.

- **Good Points -** These are valuations which legal and are allowed to occur during simulation. A coverage monitor will be created for each good point to detect whether or not it occurs during simulation. The fraction of good points which occur during simulation is the reported functional coverage value.

- **Bad Points -** These are valuations which are illegal and must never occur. A coverage monitor will also be created for each bad point, but if a bad point is found to occur then simulation is halted because an error has been detected. Bad points are equivalent to assertions.

Once the good points and bad points have been identified, the VERA language is used to implement coverage monitors using its *coverage group* [13] construct. The coverage groups are added to the testbench and their conditions are checked during simulation. The functional coverage value is the fraction of good points whose coverage groups have detected at least one occurrence during simulation.

4. SYSTEM OVERVIEW

The input to our coverage model generation engine is a description of the DUV in Verilog HDL. The output is a functional coverage model expressed as a set of VERA coverage groups [13].

The coverage points allow us to measure the degree of exploration of design behavior pertaining to associated signals and in the functional vicinity of aggregate of those signals. In order to keep this discussion independent of any particular verification language, we will refer to a VERA coverage group as a *coverage monitor* for the rest of this paper. A functional coverage model is a set of coverage monitors which correspond to the HDL code in question.

Figure 2 illustrates the structure of our functional coverage model generation engine.

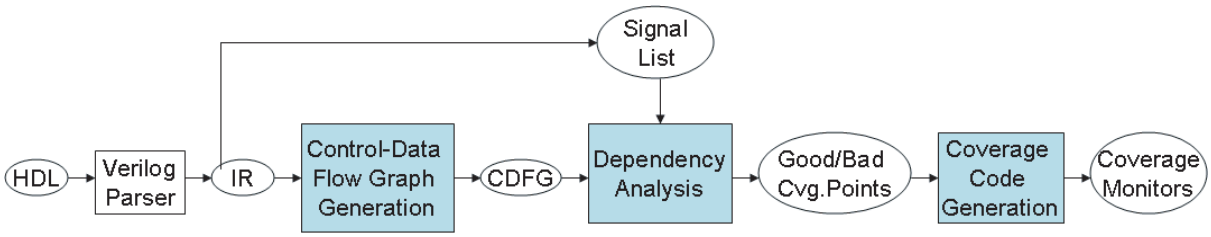

Figure 2: The Engine Organization

Verilog Parser Synopsys VCS simulator's parser was used to parse the input Verilog description. The VCS simulator allows access to its parsed intermediate representation through an interface to its internal data structures.

Control-Data Flow Graph Generator This component takes as input the intermediate representation from the Verilog parser. This was implemented in as a VPI application in *C* language with the aid of standard functions available from VPI (PLI 2.0).

Dependency Analyzer The input to this unit is a set of CDFGs and a list of signals in the Verilog description. For every signal, the assignments made to it along every control flow path closest to the leaf node are analyzed for its dependencies on the other signals. The other signals are then analyzed in their own turn. This dependency analysis spans across the set of CDFGs and continues until a terminal dependency is hit.

Coverage Code Generator The coverage code generator takes as input the above partition of coverage points. It generates VERA code to implement coverage monitors based on this partition.

5. FUNCTIONAL COVERAGE MODEL GENERATION

Each of the important components highlighted in Figure 2 will be described in the following sections.

5.1 Control-Data Flow Graph Generation

A Verilog design D can be specified as a *tuple* $\langle V, B \rangle$, where:

- V is the set of signals in the design, where $v \in V$ has an associated finite domain of values.
- B is a set of processes each of which contain procedural statements blocks.

Let us assume G is the set of Control-Data Flow Graphs (CDFGs) g_b corresponding to each of the processes $b, b \in B$. Let us also assume that $C(b)$ is the set of all conditional constructs in the process b. A CDFG g_b for a process b can be defined as a *tuple* $\langle N_C, N_D, E \rangle$, where:

- N_C is a set of nodes that represent control flow in the graph such that for every $n, n \in N_C$ there is exactly one $c, c \in C(b)$.
- N_D is a set of nodes that represent only the data flow in the graph.

```
01: always @(*)
02:    begin
03:    b = 1;
04:    if(a)
05:       if(wr_en)
06:          b = 2;
07:    b = 3;
08:    end
```

Figure 3: A Verilog Process Block

- E is the set of edges where $E \subseteq (N_C \cup N_D) \times (N_C \cup N_D)$. For an edge $(u, v) \in E$, u is a direct predecessor of v and v is the direct successor of u.

Let N be a set of all the nodes such that $N = N_C \cup N_D$. Let us assume P is the set of all control flow paths in the graph. A control flow path $p \in P$ is a sequence of nodes $n, n \in N$ connected by edges $e, e \in E$.

Let A be the set of all assignments in the process corresponding to the graph. Let $A(n)$ be the set of assignment operations performed in a node $n, n \in N_D$.

$$A = \bigcup_{\forall n \in N_D} A(n)$$

We assume that assignments to a signal are performed only in a single process. Let $Graph(v)$ be the graph for the process in which the assignments to signal $v, v \in V$ are made such that $|Graph(v)| \leq 1$. Let A_v be set of all the assignments made to signal v. Hence, $A(m) \cap A_v$ is the set of assignments made to a signal v in a node $m, m \in N_D$. We assume that $|A(m) \cap A_v| \leq 1$ such that there is at most one assignment made to signal v in node m.

Let $A_p(v)$ be an assignment to a signal $v, v \in V$ such that it is made in a node $N_p(v), N_p(v) \in N_D$ which is closest to the terminal node on a control flow path $p, p \in P$.

Let us define a set $Dom(n, p)$ which contains all the nodes preceding the node n on a path p in the graph, where $Dom(n, p) \subset N$. Let $COND(n)$ be the conditional construct c corresponding to the node n, such that $c \in C(b)$ and $COND(n) = \varphi, \forall n \in N_D$.

Let $Cdom(n, p)$ be a set of nodes, such that

$$Cdom(n, p) = \bigcup_{l \in Dom(n,p) \cup \{n\}} COND(l)$$

Let $Cdom_{var}(n, p)$ be set of variables representing the value of conditional predicates such that there is exactly one $c_{var}, c_{var} \in Cdom_{var}(n, p)$ for every $c, c \in Cdom(n, p)$.

Let $Cdom_{val}(n, p)$ be set of values of the variables of conditional predicates such that there is exactly one $c_{val}, c_{val} \in Cdom_{val}(n, p)$ for every $c, c \in Cdom_{var}(n, p)$.

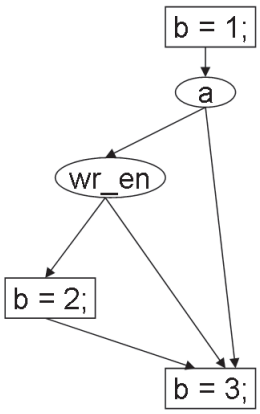

Figure 4: The Control-Data Flow Graph for a Verilog Process

5.2 Dependency Analysis

We need to determine data and control dependencies between signals across all Verilog processes. This dependency information is required to build a coverage model based on cross product of signals. Let us define a function $Rhs(a)$ which returns the expression on the RHS of an assignment a. In case, the expression is a constant, the constant value is returned. Let $Expr(c)$ be a function which returns the expression corresponding to the predicate of the conditional construct c. Let $Var(E)$ be the set of signals involved in an expression E.

Let $q_{v,p}$ be a variable such that for every $(A_p(v))$ $\exists q_{v,p} \in \{0,1\}$ such that

$$q_{v,p} = \begin{cases} 1 & : \quad v = Rhs(A_p(v)) \\ 0 & : \quad v \neq Rhs(A_p(v)) \end{cases}$$

Let L_{id} and L'_{id} be a set of signals/variables and L_{val} and L'_{val} be the set of values for these signals/variables such that there is a one-to-one correspondence between these two sets. This set pair represents a coverage point.

Let M be a set consisting of pairs (X, Y) such that X is set of signals $v, v \in V$ (or variables) and Y is a set containing their corresponding valuations. Hence, every $m, m \in M$ corresponds to a coverage point. Let M correspond to the set of good coverage points and assume a set M' which represents bad coverage points.

Figure 5 shows the algorithm for our coverage model generation approach. We use a Verilog process shown in the Figure 3 and its corresponding CDFG in the Figure 4 in order to illustrate our algorithm. The rectangular nodes are data nodes whereas oval nodes represent control nodes. We initialize the sets $L_{id}, L_{val}, L'_{id}, L'_{var}, M, M'$. Lets say we pick the signal b, and find $Graph(v)$, which gives the graph corresponding to the process in which b is being assigned, as shown in line 01 of routine *ProcessSig*.

There are 3 control flow paths in the $Graph(v)$, let us call them $p1, p2, p3$. We start with path $p1$. $A_{p1}(v)$ corresponds to the assignment $b = 3$. So $Rhs(A_{p1}(b)) = 3$. Here, $N_{p1}(v)$ is the terminal node of the graph. We have $Cdom_{var}(N_{p1}(v), p1) = \{a, wr\_en\}$ and the corresponding

```
ProcessSig(Signal v, Set G)
01: foreach (Graph(v) ∈ G)
02:   foreach (p ∈ P)
03:     if (Rhs(A_p(v)) == constant)
04:       L_id = L_id ∪ Cdom_var(N_p(v), p) ∪ {q_v};
05:       L'_id = L'_id ∪ Cdom_var(N_p(v), p) ∪ {q_v};
06:       L_val = L_val ∪ Cdom_val(N_p(v), p) ∪ {1};
07:       L'_val = L'_val ∪ Cdom_val(N_p(v), p) ∪ {0};
08:     else
09:       L_id = L_id ∪ Cdom_var(N_p(v), p);
10:       L_val = L_val ∪ Cdom_val(N_p(v), p);
11:     M = M ∪ {(L_id, L_val)};
12:     M' = M' ∪ {(L'_id, L'_val)};
13:     foreach (c ∈ Cdom(N_p(v), p)
14:       foreach (q ∈ Var(Expr(c)))
15:       ProcessSig(q, G);
16:     foreach (r ∈ Var(Rhs(A_p(v))))
17:       ProcessSig(r, G);
18: return 0;

BuildModel()
01: M ← φ; M' ← φ;
02: foreach v ∈ V
03:   L_id ← φ; L_val ← φ; L'_id ← φ; L'_val ← φ;
04:   ProcessSig(v, G);
05: return (M, M');
```

Figure 5: Coverage Model Generation Algorithm

$Cdom_{val}(N_{p1}(v), p1) = \{1, 1\}$. Since $Rhs(A_{p1}(b))$ is a constant, we will need to add $q_{b,p1}$ to our coverage points. So, $L_{id} = \{a, wr\_en, q_{b,p1}\}$, $L_{val} = \{1, 1, 1\}$ and $L'_{id} = \{a, wr\_en, q_{b,p1}\}$, $L'_{val} = \{1, 1, 0\}$. We add these pairs to the lists M and M' of good and bad coverage points respectively. This is computed in lines 02 through 12 of routine *ProcessSig*. Although, there are no expressions in the conditional predicates and in the assignments in the given example, the above steps will have to be repeated for each of the signals involved in any such expressions in case they are encountered, as shown in lines 13 through 17 of routine *ProcessSig*.

5.3 Coverage Code Generation

The process discussed in Section 5.2is repeated for the paths $p2$ and $p3$ and we obtain the following set of coverage points. $\mathbf{M} = \{(a = 1, wr\_en = 1, q_{b,p1} = 1), (a = 1, wr\_en = 0, q_{b,p2} = 1), (a = 0, wr\_en = 0, q_{b,p2} = 1)\}$ and $\mathbf{M'} = \{(a = 1, wr\_en = 1, q_{b,p1} = 0), (a = 1, wr\_en = 0, q_{b,p2} = 0), (a = 0, wr\_en = 0, q_{b,p2} = 0)\}$. The coverage monitor tracks the occurrences of these points. Occurrence of a point from the set M signifies coverage of that point from functional standpoint while occurrence of a point from the set M' indicates a functional error. Figure 6 shows the VERA coverage group for the above obtained functional coverage model.

6. EXPERIMENTAL SETUP

In order to illustrate our approach we built the following experimental infrastructure.

Benchmark Design The most challenging issue for any functional verification technique is its scalability. With this caveat in consideration we decided to use design of an industrial scale microprocessor for the purpose of demonstrating our approach. We chose a Verilog design description of a DLX processor available from

```
01: coverage_group example {
02:   sample_event = @ (posedge CLOCK);
03:   sample a, wr_en, q_b,p1, q_b,p2;
04:   cross func_cov (a, wr_en, q_b,p1, q_b,p2){
05:     state cvg_1 ((a == 1)&&(wr_en == 1)&&(q_b,p1 == 1));
06:     state cvg_2 ((a == 1)&&(wr_en == 0)&&(q_b,p2 == 1));
07:     state cvg_3 ((a == 0)&&(wr_en == 0)&&(q_b,p2 == 1));
08:     bad_state fault_1 ((a == 1)&&(wr_en == 1)&&(q_b,p1 == 0));
09:     bad_state fault_2 ((a == 1)&&(wr_en == 0)&&(q_b,p2 == 0));
10:     bad_state fault_3 ((a == 0)&&(wr_en == 0)&&(q_b,p2 == 0));
11:     }
12:   }
```

Figure 6: Coverage Code

ASPIDA (Asynchronous Open-Source IP of the DLX Architecture) project [1]. The DLX is a 32-bit 5-stage pipelined RISC CPU architecture [15].

Simulator We used the commercial VCS Verilog simulator available from Synopsys. It also supports the Verilog Procedural Interface (VPI) library of Verilog Programming Language Interface (PLI 2.0).

Test Bench The test bench was written in VERA [2]. The ASPIDA DLX implementation does not include Verilog implementations for peripherals like instruction memory, data memory, bus etc. These were modeled as high-level transaction based [13] components in VERA. The test bench takes the DLX program provided and stores it in the modeled instruction memory.

Stimuli Generation Stimulus to the DUV are valid DLX programs with their semantics and instruction defined in [15]. A basic DLX program consisting of about 23 instructions was constructed from code fragments from [15]. This program showed a fairly good functional coverage with respect to the set of properties used. A DLX test program generator was implemented using VERA Stream Generator (VSG) [13]. It generates legal DLX programs using VERA's randomization and constraint solving capabilities. This test program generator was used to generate different DLX programs by randomly appending valid DLX instructions to the core DLX program. 20 DLX programs were generated using the program generator.

Coverage Monitoring The coverage monitors were implemented as VERA coverage groups and generated as described in Section 4. If a good coverage point is encountered during simulation, it contributes towards the coverage number while if a bad coverage point is encountered, a verification error is flagged and the simulation exits.

Error Simulation In order to compute the number of errors detected we needed an error simulator which could excite equivalents of realistic design errors. The error injection mechanism is discussed in more detail in Section 7.

Each of the generated DLX programs was used to simulate the design. The number of good coverage points exhibited for each program was recorded. This number computed as percentage of the total number of good points provided functional coverage for the corresponding program. Errors were then injected at random for each program. An injected error was considered detected if it caused a bad coverage point to be exhibited and hence verification error being flagged. The number of errors detected was recorded. This number computed as percentage of injected errors provided percentage error detection. After each detected error a fresh simulation with the same DLX program was started. This processes was repeated until an adequate number of errors were simulated.

7. ERROR INJECTION

Error simulation is required to demonstrate the efficacy of our approach for generating functional coverage model. An efficient error injection mechanism is needed which is capable of injecting errors which successfully mimics real design errors. We chose to randomly change the values of signals during the simulation. Even in case of non-boolean signal values, our coverage function required either the signal has a specified value or it has any other value. So the random value injection is essentially boolean in nature.

We use the following mechanism for error injection. It employs three levels of randomization. We exploit VERA's randomization and constraint solving capabilities for this purpose.

Signals A list of signals satisfying the criteria mentioned later in this section is prepared. A signal is chosen at random for injecting error for each simulation run.

Values A value domain for each signal is specified. It is effectively reduced to a boolean domain for all signals as discussed in this section earlier. A value is chosen at random for the signal chosen in the previous step for each simulation run.

Time For every simulation run, the random signal and value selected in the previous steps is injected during simulation at a random simulation time.

During any single simulation run only one randomly error generated is injected. Errors are only injected on signals which are involved in the generated coverage model. Error injection is restricted in this way in order to focus on errors which impact the developed coverage model. The coverage monitors generated by our technique are meant to depict the developed coverage model. If an error does not impact the coverage model then there is no guarantee that is will be captured by our coverage monitors. For this reason we need to inject errors which have a high likelihood of impacting the coverage model.

8. EXPERIMENTAL RESULTS

We evaluate the effectiveness of our technique by computing functional coverage for the DLX benchmark and comparing functional coverage to *error coverage*, the fraction of injected errors actually detected. The measure of the effectiveness of our functional coverage is to see how closely functional coverage predicts the error detection rate. An optimal functional coverage model would exactly match the error coverage.

We generated our coverage model in 37 seconds on a 1503 MHz Sun UltraSPARC cpu with SunOS version 5.9. Our coverage model consisted of a total of 7887 coverage points,

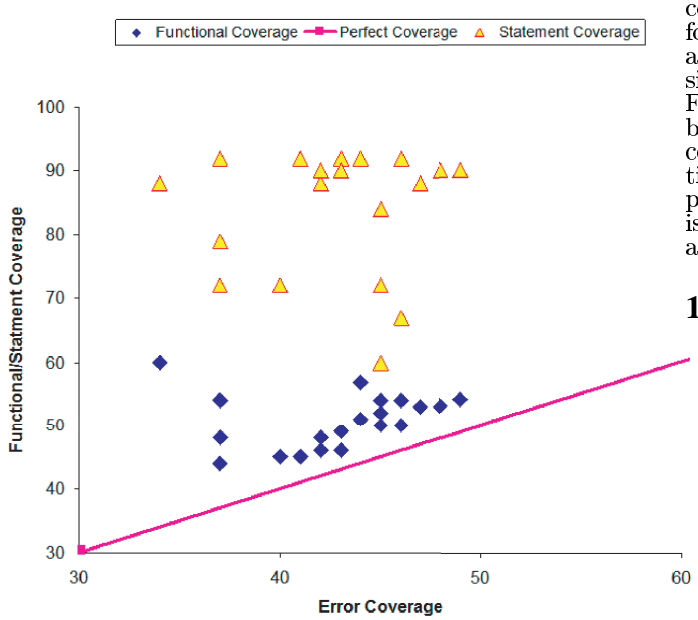

Figure 7: Coverage vs Error Detection

out of which 5349 were good points and 2538 were bad points. These constitute a small percentage (<< 1%) of the total number of points in the coverage space.

Figure 7 shows a graphical representation of the evaluation results. It shows two sets of points, 1. **Functional Coverage** based on our functional coverage values, and 2. **Statement Coverage**. A total of 20 DLX programs generated as described in Section 6 were used. Each of the 20 DLX programs were executed before any error injection. The coverage monitors generated as described in Section 4 provided the functional coverage and the percentage functional coverage for each of these correct executions was computed. A total of 100 errors were then injected as explicated in Section 7 for each program. The number of injected errors detected was recorded for each DLX program and error detection percentage was computed. The line labeled **Perfect Coverage** shows where all points would lie for a perfect coverage metric which is always exactly equal to error coverage.

It is clear at a glance that the points for our functional coverage are closer to the perfect line than those for statement coverage. This improvement can be seen formally by evaluating the average **coverage difference**, the difference between functional/statement coverage and error coverage. A small difference indicates a more accurate metric. The average coverage difference for our functional coverage is 7.9% as compared to 41.4% for statement coverage. We also compute the standard deviation for **coverage difference** for both functional and statement coverage. It is 5.5% for our functional coverage as compared to 10.7% for statement coverage.

9. CONCLUSIONS

We have successfully performed automatic generation of coverage monitors representing a functional coverage model for the design of industrial scale DLX processor from a static analysis of the HDL description of its design. The design was simulated with randomly generated valid DLX programs. Functional coverage was computed from the data collected by coverage monitors. Errors were then injected and percentage error detection was computed. The computed functional coverage was found to closely track error detection percentage than the statement coverage tracked it, which is a testimony to the quality of the coverage model generated.

10. REFERENCES

[1] ASPIDA. http://www.ics.forth.gr/carv/aspida.

[2] Open vera. http://www.open-vera.com.

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

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

[5] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets. A study in coverage-driven test generation. In *Design Automation Conference*. ACM Press, 1999.

[6] D. G. et.al. Coverage-directed test generation using symbolic techniques. In *FMCAD*. IBM Science and Technology, Haifa Research Lab, Haifa Israel, 1996.

[7] F. C. et.al. Functional verification methodology of chameleon processor. In *Design Automation Conference*. ACM Press, 1996.

[8] A. Evans, A. Silburt, G. Vrckovnik, and T. Brown. Functional verification of large asics. In *Design Automation Conference*. ACM Press, 1998.

[9] L. Fournier, Y. Arbetman, and M. Levinger. Functional verification methodology for microprocessors using the genesys test program generator. In *Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 1999.

[10] E. Gaudette, M. Moussa, and I. G. Harris. A method for the evaluation of behavioral fault models. In *High-Level Design Validation and Test Workshop*, 2003.

[11] A. Gluska. Coverage-oriented verification of banias. In *Design Automation Conference*. ACM Press, 2003.

[12] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv. User defined coverage - a tool supported methodology for design verification. In *Design Automation Conference*. ACM Press, 1998.

[13] F. Haque, J. Michelson, and K. Khan. *The Art of Verification with Vera*. Verification Central, 2001.

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

[15] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002.

[16] Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Design Automation Conference*. ACM Press, 1999.

[17] M. Kantrowitz and L. M. Noack. I'm done simulating; now what? verification coverage analysis and correctness checking of the decchip 21164 alpha microprocessor. In *Design Automation Conference*. ACM Press, 1996.

[18] S. Katz, O. Grumberg, and D. Geist. "have i written enough properties?" - a method of comparison between specification and implementation. IBM Haifa Research Lab, Haifa Israel, 1997.

[19] Y.-S. Kwon, Y.-I. Kim, and C.-M. Kyung. Systematic functional coverage metric synthesis from hierarchical temporal event relation graph. In *Design Automation Conference*. ACM Press, 2004.

[20] K. L. McMillan. *Symbolic Model Checking*. Springer, 1993.

[21] J. Monaco, D. Holloway, and R. Raina. Functional verification methodology for the powerpc 604tm microprocessor. In *Design Automation Conference*. ACM Press, 1996.

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

[23] J. Shen, J. Abraham, D. Baker, T. Hurson, and M. Kinkade. Functional verification of equator map1000 microprocessor. In *Design Automation Conference*. ACM Press, 1999.

[24] K. Shimizu. Deriving a simulation input generator and a coverage metric from a formal specification. In *Design Automation Conference*. ACM Press, 2002.

[25] A. Ziv. Cross-product functional coverage measurement with temporal properties-based assertions. In *Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 2003.