OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification

Farzan Fallah, Srinivas Devadas
Department of EECS
MIT, Cambridge

Kurt Keutzer
Department of EECS
University of California, Berkeley

Abstract—
Functional simulation is still the primary workhorse for verifying the functional correctness of hardware designs. Functional verification is necessarily incomplete because it is not computationally feasible to exhaustively simulate designs. It is important therefore to quantitatively measure the degree of verification coverage of the design. Coverage metrics proposed for measuring the extent of design verification provided by a set of functional simulation vectors should compute statement execution counts (controllability information), and check to see whether effects of possible errors activated by program stimuli can be observed at the circuit outputs (observability information). Unfortunately, the metrics proposed thus far, either do not compute both types of information, or are inefficient, i.e., the overhead of computing the metric is very large.

In this paper, we provide the details of an efficient method to compute an Observability-based Code COverage Metric (OCCOM) that can be used while simulating complex HDL designs. This method offers a more accurate assessment of design verification coverage than line coverage, and is significantly more computationally efficient than prior efforts to assess observability information because it breaks up the computation into two phases: Functional simulation of a modified HDL model, followed by analysis of a flowgraph extracted from the HDL model. Commercial HDL simulators can be directly used for the time-consuming first phase, and the second phase can be performed efficiently using concurrent evaluation techniques.

I. Introduction

Verification is increasingly perceived as the major bottleneck in integrated circuit design. On many design teams the staff involved in verification is larger than the staff involved in the implementation of the design. For most designs the hardest verification problem is verifying the correctness of the initial register-transfer level RTL description coded into a hardware description language HDL. Formal verification techniques promise to greatly ease the problem of verifying the equivalence of a design across many steps of implementation but this still leaves unresolved the problem of getting the initial HDL description correct. Formal techniques for language containment and property checking are making some progress on this problem; however, there is no indication that these techniques will be able to offer comprehensive verification across a wide variety of designs. Thus, it appears that simulation will continue to be the workhorse for design verification for some time to come.

Improving the speed of simulation and allowing for parallel and distributed approaches to simulation are two well-motivated approaches to improving verification. Unfortunately these do not help a designer assess how comprehensively the simulation vector set verifies the design. As a result there is a well-defined need for tools that enable designers to assess the comprehensiveness of verification coverage offered by a simulation vector set.

The analogy that we use to guide our work is that of fault simulation. A fault simulator first enables you to assess that your fault coverage is high enough. This is critical to ensuring that your design is adequately tested. A fault simulator can then enable you to perform vector-set compaction. By assessing the faults detected by each vector, a minimal-length vector set can be chosen which covers all faults. By analogy, we are interested in providing coverage metrics that allow designers to assess the comprehensiveness of their simulation vector set. Next we aim to help them diagnose what part of their design may be inadequately verified by the current vector set. This will help guide the writing of additional vectors. Finally, we aim to help them compact their vector sets.

Currently, the state-of-the-art in coverage metrics is found in commercial tools which principally rely on either line-coverage or path-coverage metrics inherited from software testing. In software testing, given a set of program stimuli, coverage metrics such as line coverage, branch coverage and path coverage are used for software quality assurance. Most coverage metrics in software testing [2] are based on the activation of statements, branches or sequences of statements and do not address observability requirements; the fact that a statement with a bug has been activated by input stimuli does not mean that the observed outputs of the program will be incorrect. Exceptions are the sensitivity analysis methods of Voas [16], and the impact analysis methods of Goradia [7] (cf. Section II).

Similar approaches can be taken in hardware; however, hardware offers much less observability of *meaningful* data [1] through primary outputs than software does through inspection of memory contents. Therefore, coverage metrics proposed for measuring the extent of hardware design verification provided by a set of functional simulation vectors should compute statement execution counts (controllability information), and check to see whether effects of possible errors activated by program stimuli can be observed at the circuit outputs (observability information). Unfortunately, the coverage metrics for hardware proposed thus far, either do not compute both types of information, or are inefficient, i.e., the overhead of computing the metric is very large. In Figure 1, we indicate the levels of performance currently achievable using various metrics, normalized to HDL simulation time. Computing simple controllability metrics adds some overhead [2]; somewhat higher overhead is incurred when computing observability information at the HDL level using a specialized tag simulator [5], or when using gate-level fault simulators.

A simple example of why observability information is important to the designer is shown in Figure 2. Assume that model A and model B are both exercised thoroughly using a functional vector set. Controllability metrics will report 100% statement coverage for both models. However, it may be that statements in model A are only exercised with vectors for which $c = 0$, implying that the variables assigned in these statements *never* affect the observed output F for any simulated vector.

In this paper, we provide the details of an efficient method to compute an Observability-based Code COverage Metric (OCCOM) that can be used while simulating complex HDL designs. The computed coverage information serves as a diagnostic aid to the designer; it helps to debug and design, and/or create better functional tests.

We circumvent the drawbacks of previously-proposed coverage metrics in our new approach of two-phase OCCOM computation. The two phases are:

1. We automatically modify the given HDL model, in particular adding new variables and moving statements out of conditionals, and *simulate the given vectors using commercial* HDL *simulators*. Thus, current (future) simulation technology is

[1] By meaningful, we mean that the designer should be able to easily differentiate incorrect values from correct values for a given input stimulus.

[2] This data on controllability metrics is consistent with our information on commercial tools such as Simulation Technology/Summit's Vericov or Advanced Technology Corporation's Covermeter.

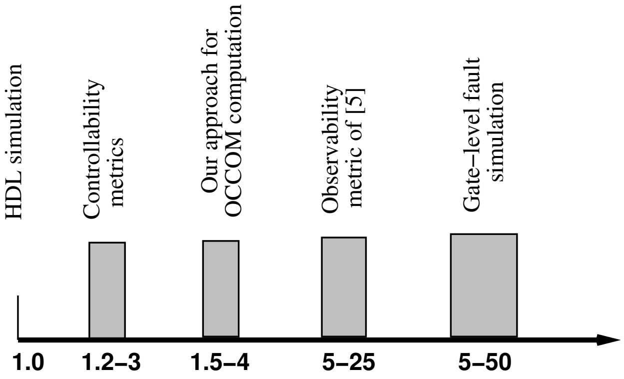

Fig. 1. Coverage Metric Computation Overheads

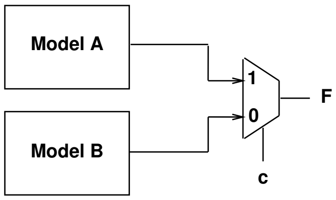

Fig. 2. Observability Versus Controllability Coverage

(can be) directly exploited. There is a loss of simulation efficiency due to the addition of new variables, but this is not large (cf. Section VI).

2. We create a flowgraph from the modified HDL model, and use the results of simulation to determine coverage under OCCOM. Using concurrent evaluation techniques, this phase can be executed efficiently.

We describe related work in Section II. In Section III, we describe our overall strategy, and present the simulation calculus. The phase of modifying the given HDL model to handle conditionals is described in Section IV. In Section V, we describe the method for OCCOM computation given the results of simulation. Experiments on a set of benchmarks are described in Section VI, that show the new method has significantly less overhead than previous methods. Ongoing work is briefly described in Section VII.

II. Related Work

We describe representative coverage analysis methods from the software test literature, and representative HDL coverage analysis work.

A. Software Testing

A *control flowgraph* is a graphical representation of a program's control structure [2, Chapter 3]. A control flowgraph is comprised of *processes*, *decisions*, and *junctions*. A process is a sequence of program statements uninterrupted by either decisions or junctions. A process has one entry and one exit. A decision is a program point at which the control flow can diverge. A junction is a point in the program where the control flow can merge.

A path in the control flowgraph is a sequence of processes that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision or exit.

Given a set of program stimuli, one can determine the statements activated by the stimuli by applying the stimuli to the control flowgraph. The *line coverage* metric measures the number of times every process (and therefore constituent statements) is exercised by the program stimuli. In the case of *branch coverage*, we measure the number of times each branch is taken, or not taken, under the set of program stimuli. *Path coverage* measures the number of times every path in the control flowgraph is exercised by the set of program stimuli. A potential goal of software testing is to have 100% path coverage, which implies branch and line coverage. However, 100% path coverage is a very stringent requirement and the number of paths in a program may be exponentially related to program size.

There are other coverage metrics as well, for example, multi-condition coverage, loop coverage and relational operator coverage that are variants of branch coverage. The Generic Coverage Tool GCT [13] can automatically determine the coverage of an arbitrary set of program stimuli.

These coverage metrics require activation but say nothing about the observability conditions required to see the effect of possible errors in the activated statements. The path coverage metric will satisfy observability requirements if paths from program inputs to program outputs are exercised and the values of variables are such that the erroneous value is not masked (this is analogous to side inputs having non-controlling value in fault propagation). However, the path coverage metric does not explicitly evaluate whether the effect of an error is observable at an output.

Sensitivity and impact analysis take into account observability requirements. Voas [16] estimates probabilities that a fault on a location will result in an output error under a specific input distribution. The program is executed for a large number of inputs to estimate the propagation probability for a *single* location. Impact analysis [7], estimates "impact strengths" of all entity instances in an execution, in a time proportional to the execution time. If x is an input to y, then the impact strength of x depends on whether a change in the value of x from $V(x)$ to $V(x')$ propagates to y or not. However, the impact strength does not appear to depend on the values of other inputs, say z, that affect y.

B. HDL Coverage Analysis

Coverage analysis techniques proposed for general HDL models include – guaranteed coverage of every statement in an HDL model [4], evaluation of transition coverage of a test set [10], and abstraction of models and semantic control over transition coverage [6]. These metrics do not directly address observability requirements.

Observability requirements are addressed in the metric proposed in [5]. In order to compute observability information, variables are tagged during simulation, and a simulation calculus is used to calculate the coverage provided by an arbitrary set of functional vectors. There are several drawbacks with the tag simulation algorithm and calculus presented in [5]. Firstly, the calculus for only a subset of VERILOG [15] was developed. In particular, nested if statements cannot be handled, neither can looping constructs used in popular HDL's. Secondly, the efficiency of the simulation algorithm leaves much to be desired because the calculus dictated the use of an augmented simulator whose speed is much slower than the compiled-code speed of commercial simulators. The speed can be improved by incorporating all the optimizations currently present in commercial HDL simulators, but this is a huge undertaking.

Ho *et al.* [9] use transition tours on the implementation control FSM to automate test generation of corner cases, for validation of an embedded dual-issue pipelined processor. The issue of error coverage is not addressed. A formal metric for coverage computation is presented in [8]. This method has several attractive features, including a guarantee of design error coverage, however, it has not been automated and has only been applied to a processor example.

Design-specific coverage analysis and test generation approaches have been proposed for processors (e.g., [1], [11], [12]).

III. Computing Coverage: Basic Strategy

A. Context

Given an HDL model and a set of functional vectors, computing controllability metrics during functional simulation is relatively easy. Assuming an event-driven simulator, counters can keep track of how many times, if any, each statement is executed, or action taken. We, however, wish to compute observability metrics as well.

A **tag** at a location represents the possibility that an incorrect value was computed at that location. These tags on variables are not tied to particular design errors; they serve as a mechanism for extending standard coverage metrics to include observability requirements [5]. Each location corresponds to an assigned variable in some statement in the HDL model. Our goal, given a set of functional vectors and an HDL model, is to determine if a tag **injected** in any particular location is **propagated** to the circuit output. That is, we want to see if incorrectly computed values are propagated to circuit outputs, or not. [3] This is dependent on the data values at other locations in the circuit; other data values may *block* the tag from reaching any circuit output. The quality of a vector set is determined by how many injected tags are propagated to the output. The percentage of propagated tags is what we call **code coverage** under the metric.

Note that we will use the **single tag** model, where the effects of exactly one injected tag are computed, for many different injected tags. Each injected tag can be thought of creating a distinct "faulty" HDL model, and several such faulty models are processed in parallel by the simulation algorithm. (This is akin to concurrent fault simulation [3].)

B. Our Two-Phase Approach

Computing an observability metric for a collection of logic gates is similar to fault simulation using the D-calculus [14]. An error at the input of an AND gate is propagated to the output only if the other inputs are at a 1, or have errors of the same polarity. (Similarly for an OR gate.) Generalizing this notion to HDL's requires us to handle three main issues:

- Arithmetic operations,
- Conditional and looping constructs, and
- Concurrency constructs.

We can define a calculus to handle HDL models, for example, augmenting the calculus of [5], but the calculus should be easily computable. Designers wish to simulate many thousands of vectors on complex HDL models. Commercial HDL simulators incorporate many optimizations to improve logic simulation speed. A new calculus implies that the simulation engine has to be modified, and this can result in a dramatic loss in efficiency, unless a major effort to incorporate optimizations is made. In order to avoid this problem, we propose a two-phase approach to computing our coverage metric OCCOM.

The two-phase approach is:

1. We modify the given HDL model. The modifications are necessary because of conditionals in the HDL model. In particular, we will add new variables and extract statements out of conditional clauses. We then simulate the modified HDL model using a standard HDL simulator. Note that tags do not play a part in this phase.
   Simulating the modified HDL model will give us more information than simulating the original model. This extra information is used in the second phase to perform tag propagation.
2. In the second phase, we will actually perform tag injection and propagation. We extract a flowgraph from the HDL model. Tag injection simply corresponds to introducing a tag on an edge in the graph and tag propagation corresponds to selectively traversing paths from the edge to the output nodes.

We will describe Phase 1 in Section IV, and Phase 2 is described in Section V. We now describe the tag simulation calculus. Note that this calculus is not used in Phase 1, but only in Phase 2. Nevertheless, in order to understand why we are modifying the HDL description, we need to first describe the tag simulation calculus.

C. Tag Simulation Calculus

A tag is represented by the symbol Δ which signifies a possible change in the value of the variable due to an error. We will consider both positive and negative tags, $+\Delta$ written simply as Δ, and $-\Delta$. Later we will also consider tags of unknown sign (cf. Section V).

C.1 Logic Gates

We will first define the calculus for Boolean logic gates. This is identical to the D-calculus [14]. The calculus for a two-input AND gate and an INVERTER are shown in Table I. The four possible values at each input are $\{0, 1, 0+\Delta, 1-\Delta\}$. (Note that $0-\Delta \equiv 0$ and $1+\Delta \equiv 1$.) The entries are self-explanatory.

Using the above calculus any collection of Boolean gates comprising a combinational logic module can be tag simulated.

[3]While there is full observability of internal locations in the HDL model during simulation, the particular data values at internal locations may be incomprehensible to the designer. For example, the designer may be able to verify that the output of a Wallace tree multiplier is an incorrect 6, for inputs of 4 and 4, but will not be able to determine if an arbitrary internal wire is at a faulty 1 or 0.

AND	0	1	$0+\Delta$	$1-\Delta$
0	0	0	0	0
1	0	1	$0+\Delta$	$1-\Delta$
$0+\Delta$	0	$0+\Delta$	$0+\Delta$	0
$1-\Delta$	0	$1-\Delta$	0	$1-\Delta$

INVERT
1
0
$1-\Delta$
$0+\Delta$

TABLE I
Δ-CALCULUS FOR AND GATE AND INVERTER

C.2 Arithmetic Operators

We describe the tag simulation calculus procedurally for some common arithmetic operators. All modules are assumed to be n bits wide. For each operator op, after the simulator computes $v(f) = v(a)\ \langle op \rangle\ v(b)$, we tag $v(f)$ with a positive or negative Δ and we write it as $v(f) + \Delta$ or $v(f) - \Delta$.

Adder : If all tags on the adder inputs are positive, and if the value $v(f) < MAXINT$, we will assign the adder output $v(f) + \Delta$. $MAXINT$ is the maximum value possible for f. Similarly, if all tags are negative. If both positive and negative tags exist at adder inputs, the output is assumed to be unknown sign tag.

Multiplier : All tags have to be of the same sign for propagation. A positive Δ on input a is propagated to the output f provided $v(b) \neq 0$ or if b has a positive Δ. We will assign the multiplier output $v(f) + \Delta$. Similarly for negative Δ.

$>$ *Comparator* : If tags exist on inputs a and b, they have to be of opposite sign, else the tag is not propagated to the output. Assume a positive tag on a alone, or a positive tag on a and a negative tag on b. If $v(a) \leq v(b)$ then the tag(s) is (are) propagated to the output, else the tag(s) is (are) not. We will assign the comparator output $0 + \Delta$. Similarly for other tags and other kinds of comparators.

C.3 Conditionals

When a conditional is encountered by the tag simulator, there are two cases:

- There is no tag in the control condition. In this case simulation proceeds normally. In the appropriate processes, statements are executed and tags (if any) are propagated/injected.
- If there is a tag on the control condition $expr$ it means that the tag will result in the incorrect branch being taken. Under the tag condition, some assignments will be missed, and others incorrectly made. Unfortunately, we do not have information regarding what happens when the incorrect branch is taken; we only have information regarding the simulation of the correct branch. [4]

As an example of the second case, consider,

```
if ( y )
      x = exp1 ;
else z = exp2 ;
```

If `y = 1` and there is a negative tag on `y`, we have to consider the case where `x` is not assigned, and `z` is assigned. While we know the values of `x` prior to and after the assignment, we only know the old value of `z`, since the simulator does not compute `exp2`.

Nested conditionals and concurrency constructs exacerbate the above situation. We describe how we handle various cases in the next section by modifying the HDL model prior to simulation.

IV. Phase 1: Modifying the HDL Model

We consider various commonly occurring cases of conditionals and events in VERILOG [15], and show how to automatically modify the HDL model such that the HDL simulation produces enough information to compute coverage in Phase 2. The modification is illustrated for several commonly-occurring cases in the following sections. The modifications all require the addition of new variables.

[4]The approach of [5], somewhat arbitrarily, tags all the variables in the assignments that would be missed under the tag condition.

A. Simple Conditional

Consider the code on the left-hand-side below. The transformed code is shown to the right.

```
                            x1 = expr1 ;
if (cexp)                   x2 = expr2 ;
      x = expr1 ;           if (cexp)
else  x = expr2 ;                 x = x1 ;
                            else  x = x2 ;
```

Consider the case of a tag on `cexp`. During the simulation of the modified code, we will compute the values of both `expr1` and `expr2`, and store them in the new variables `x1` and `x2`. We know the old value of `x`, and we will know the new values of `x` corresponding to the execution of both the *then* and *else* clauses, regardless of the value of `cexp`. This will allow us to correctly propagate positive or negative tags on `cexp` in Phase 2 (cf. Section V).

B. Nested Conditionals

The case of nested conditionals is more complicated. Further, we have to consider the situation where variables such as `x` are assigned values that depend on the old values (e.g., increment operation). As before, we show the original code to the left, and the transformed code to the right. The transformed code will compute the necessary information to perform propagation of tags on `cexp1` or `cexp2`.

```
                            x2 = x ;
if (cexp1) begin            x1 = expr1 ;
  if (cexp2)                if (cexp2)
    x = expr1 ;               x2 = x1 ;
  if (cexp3)                x3 = x2 + expr2 ;
    x = x + expr2 ;         if (cexp3)
end                           x2 = x3 ;
                            if (cexp1)
                              x = x2 ;
```

C. Simple Loop

Consider the simple loop shown to the left-hand-side below. The interesting case is where there is a tag on the variable `N`. The transformed code is shown to the right.

```
                         x1 = x ;
for(i=0; i<N; i++)       for(i=0; i<N+1; i++)
   x = x + expr ;          begin
                            x1 = x1 + expr ;
                            if (i < N)
                              x = x1 ;
                           end
```

If there is a negative tag on `N`, then that corresponds to the situation that the loop is iterated fewer than `N` times. If there is a positive tag on `N`, then that corresponds to the situation where the loop is iterated more than `N` times. To minimize simulation overhead, we have chosen to perform exactly one more iteration of the loop. Given the extra conditional in the modified loop, the two pieces of code result in the same value for `x`. However, `x1` will contain the value for the case where the loop is iterated `N + 1` times.

Similar transformations can be made for `while` and `repeat` loops.

We do not inject tags on the loop counter variable `i`. We assume that errors on `i` are reflected by changes in the number of loop iterations, i.e., `N`. Propagating tags on variables such as `x` does not require additional information.

D. Conditional with Event

Events in VERILOG correspond to setting variables that affect other processes. Consider the code on the left-hand-side below.

```
if (cexp) begin             x1 = expr1 ;
  x = expr1 ;               y1 = expr2 ;
  --> ec1 ;                 x2 = expr3 ;
  y = expr2 ;               if (cexp) begin
end                           x = x1 ;
else  x = expr3 ;             --> ec1 ;
                                y = y1 ;
                              end
                              else  x = x2 ;
```

This is similar to the simple conditional case, where we are concerned with a tag on `cexp`. The additional complication is the event `ec1`. We should not move the event outside the *then* clause, because we have no way of undoing the effect of the event in the simulator, if the clause is not going to be executed.

E. Loop and Conditional

A loop inside a conditional is handled like any other statement.

A conditional inside a loop requires a more complicated transformation.

```
                              x3 = x ;
                              for(i=0; i<N+1; i++)
for(i=0; i<N; i++)              begin
   if (cexp)                     x1 = x3 + expr1 ;
      x = x + expr1 ;            x2 = expr2 ;
   else  x = expr2 ;             if (cexp)
                                       x3 = x1 ;
                                 else  x3 = x2 ;
                                 if (i < N)
                                   x = x3 ;
                                end
```

We wish to propagate tags on both iteration count variable `N`, and the conditional expression `cexp` in Phase 2. We need to transform the code such that there is enough information to propagate either tag. The right-hand-side code corresponds to the modified version.

Consider a tag on `N`, and no tag on `cexp`. This is similar to the simple loop case, except that we are computing values in both clauses of the conditional. The case of no tag on `N` and a tag on `cexp` is the same as the simple conditional case.

While we are considering the single tag model, it may be that a tag injected on a variable `y` is propagated to both `N` and `cexp`. The propagation of tags to variable `x` (Phase 2, Section V) is dependent on the signs of the tags on `N` and `cexp`, and also on the values (computed during simulation) of the new `xi` variables.

The propagation of tags is the subject of the next section.

V. Phase 2: Graph Analysis

We describe, given the information regarding variable values for the simulated functional vectors, how to perform concurrent tag propagation. We work with a flowgraph data structure that enables efficient tag propagation.

A. Graph Structure

We create a graph $G(V, E, L(E))$ from the given HDL model. Each vertex $v \in V$ corresponds to a variable in the HDL model. Each edge $e(v, w) \in E$ is a directed edge from source node v to destination node w. The edge implies a data dependence between the nodes. Node w depends on v.

The label of an edge $l(e) \in L$ contains the information below:

- Line number: The edge exists because of a data dependence. A particular line of the *modified* HDL file containing the description of the model is associated with the edge. For example, given `a = b + c` in Line 154, the edges from `b` to `a` and `c` to `a` will both have Line 154 as a label.
- Conditional expression: We have a conditional expression which has to be true, in order for the data dependency to exist. For example, given

  ```
  if (x > y)
        a = b + c ;
  ```

  the conditional expression for the edge from `b` to `a` would be `x > y`. Since we have removed nested conditionals in the modified HDL model, we do not need to conjunct multiple expressions. The expression will correspond to the conditional clause of a single `if` statement in the original HDL model.
- Indices for array variables: Arrays are common in HDL models. There will be a separate tag associated with each array element, resulting in an array of tags. Given

  ```
  a[i] = b[i+4] ;
  ```

we will have a single node for the array `b` and the array `a` and an edge from `b` to `a`. We will associate information corresponding to the array indices with the edge from `b` to `a`. In general, we require two indices, one for `b` and one for `a`. We will need to compute these indices dynamically during tag propagation.

- Type of dependence: Dependence can be, normal, conditional, through task or function call, and through module instantiation. For example, given

```
if (x > y)
    a = b + c ;
```

  there is an edge from `x` to `a` and `y` to `a`. These edges are conditional edges. The edges from `b` and `c` to `a` are normal edges.

  A task or function call dependency is a dependency between inputs and outputs of a task enable or function call. If there is a tag on one of inputs or the tag-injected line is inside the task or function, there may be tags propagated to the output of the task or function.
- Propagation multiplier: We will have a multiplier associated with each edge that will be used to multiply tags propagating down the edge. These multipliers can be expressions. For example, given `a = b - c`, the edge between `b` and `a` has a $+1$ multiplier, but the edge from `c` to `a` has a -1 multiplier. (This is because a positive tag on `c` should result in a negative tag on `a`.)

  There is an additional complication for edges whose dependence type is conditional. Consider

```
if (x > y)
      a = a1 ;
else  a = a2 ;
```

  In this case, the conditional dependence edge from `x` to `a` will have the expression `a1 - a2` as the propagation multiplier. We are only interested in the sign of `a1 - a2`, not its magnitude.

We will create a single observable vertex O in the graph that does not correspond to any variable. The variables in each "display" or "monitor" statement in the HDL model will be connected to O via edges. Each edge will have a Line number corresponding to the display statement. The conditional expression will be always true, and the propagation multiplier will be $+1$ or -1. If array variables are displayed, the index of the variable will be attached as a label for the edge.

B. Concurrent Tag Propagation

Tag propagation is performed on the graph structure, and uses information obtained from the simulation trace of the modified HDL model. For each vector, we determine all the tags that can be propagated to the observable vertex O using concurrent tag propagation.

The steps are:

1. Each edge in the graph is first determined to be active or inactive for the given vector. An active edge is an edge that can propagate a tag from a predecessor node to the successor node. Whether an edge is active or not depends on the tag simulation calculus (cf. Section III-C). For example, given the statement $C = A \times B$, if B is 0, then the edge from A to C will be inactive. A complication is that edges corresponding to the control statements may be active or inactive depending on the sign of the tag which is propagated, as well as the value of the control predicate under the given vector. For example, in

```
if ( y )
      x = exp1 ;
else  x = exp2 ;
```

   there is an edge with conditional dependence from `y` to `x`. This edge is inactive if `exp1 - exp2 = 0` for the given vector, and is marked active otherwise. The propagation multiplier of the edge is $+1$ if `expr1 - expr2 > 0`, and -1 if `expr1 - expr2 < 0`. If `y = 1`, then a positive tag on `y` cannot be propagated to `x` from `y`, but a negative tag can.
2. We delete all inactive edges in the graph.
3. We inject a positive and negative tag on the observable vertex O.
4. Starting from the observable vertex O, and assuming positive (negative) tags on the vertex, we traverse edges in the graph backwards, in the reverse order of simulation trace, determining all nodes $n \in V$ that can reach the observable vertex. The vertex O is reachable from node n, if before traversing any of n's fanin edges, all traversed paths from n to O have propagation multipliers of the same sign. (The propagation multiplier of a path is simply the product of the propagation multipliers of the constituent edges.) Note that paths with multipliers of different signs, might result in tag cancellation, and we conservatively assume that the tag is not propagated.
5. Line numbers of edges encountered during the backward traverse, determine the tags which are observable in O.

VI. Experiments

A. Metric Comparison

The examples used in Table II correspond to various algorithms and processors implemented in VERILOG. Example **dcnew** is a train system relay, **eight queens** is an algorithm to solve the eight queens problem in chess, **crd** is a traffic controller, **amp** is a general-purpose processor, and **coherence** is a cache coherence protocol.

The results in Table II serve to illustrate the usefulness of OCCOM in relation to vanilla controllability measures. We gave a simple example of how controllability information can be misleading in Figure 2. In Table II, we give statistics comparing line coverage and OCCOM coverage for designer-generated functional tests (labeled Directed), and randomly-generated tests (labeled Random).

It is clear that the extra observability information in OCCOM is required to distinguish between the "good" design-generated test and the random tests. In general, the designer generates tests by exercising chosen paths through the HDL model. If a path from inputs to outputs is exercised by an input vector sequence, the tags injected on the assignment statements on the path will almost always[5] be propagated to the output. (Using a path coverage metric has the disadvantage that an exponential number of paths exist in the design, and one cannot possibly exercise all paths. Further, many of these paths may be false, i.e., may not be exercisable.)

B. Performance Comparison

We compare the performance of our OCCOM computation methodology against functional simulation, and the method of [5]. The results are summarized in Table III. Example **arbiter** is a bus arbiter, **counter** is a 3-bit counter, **crd** is a traffic controller, **ctlp3** is the logic for a Dining philosophers problem, and **8251A** is an USART. The time required to simulate the original HDL model (prior to modification) for 10,000 vectors using a commercial VERILOG simulator is given as a baseline for comparison purposes. The same simulator and the same vectors were used for OCCOM computation. OCCOM computation is broken up into two parts; time required to simulate the modified HDL model, and the time required for tag propagation. All the times correspond to seconds on an UltraSparc-2.

We could not run the simulator of [5] for some of the examples, since the simulator does not handle certain VERILOG constructs such as nested if statements. Our method is faster than the specialized tag simulation because it allows for the use of highly-optimized commercial HDL simulator and concurrent tag propagation.

Our method is significantly faster than fault simulation. Note that in order to use the fault simulation strategy, the HDL model has to be converted to gate-level form, and this in itself is a time-consuming operation.

C. Diagnosis: Feedback to Designer

OCCOM coverage obtained for a given HDL model can be used to debug the model, or create better functional tests.

Assume that a certain coverage has been obtained for an HDL model. The tags that are not propagated to the outputs are examined, one by one. For each such tag, we have the information regarding whether the line on which the tag was initially injected was executed (controllability information). We determine all the vectors (if any)

[5] One exception is when tag propagation is blocked because of arithmetic overflow or underflow.

Ex	#Lines	#Tags	Directed			Random		
			#Vec	#OCCOM Coverage%	Line Coverage%	#Vec	OCCOM Coverage%	Line Coverage%
eight queens	220	146	179	87%	92%	2000	67%	90%
dcnew	353	256	338	86%	100%	3000	77%	100%
crd	120	88	156	94%	100%	2000	67%	90%
amp	288	198	168	68%	92%	3000	68%	92%
coherence	530	346	442	85%	94%	5000	54%	90%

TABLE II
COMPARING OCCOM AND A LINE COVERAGE METRIC

Ex	HDL Simulation	OCCOM			Specialized Simulator [5]
		HDL	Graph	Total	
arbiter	9 s	13 s	20 s	33 s	150 s
counter	7.5 s	10 s	1.5 s	11.5 s	25 s
crd	19 s	21 s	19 s	40 s	100 s
ctlp3	11.5 s	13 s	10 s	23 s	NA
8251A	15 s	22 s	20 s	42 s	NA

TABLE III
COMPARING TIMES REQUIRED TO COMPUTE METRICS

for which the line was executed. For each vector, the tag propagation path is examined to determine the **blocking** statement(s). The values of the variables in the blocking statements need to be changed in order to propagate the tag further. The designer determines a legal change in variable values and creates a new functional test, which is tag-simulated, and the process repeated. If the designer is unable to come up with such a test, then it may be that there is a redundancy in the HDL model, or that there is a design error.

VII. ONGOING WORK

There are three general directions of work to be explored in improving verification-coverage metrics for hardware designs: exploring new error models, computationally improving coverage analysis and providing diagnostics.

Associating an error with every possible assignment captures most of the errors that can at all be discovered by structural analysis, but probably produces more errors than are necessary to be analyzed. It would be useful to find an error model that produces fewer errors to be analyzed. Any changes in the error model will require a rethinking of the calculus used for propagating error values. We are exploring other error models and associated simulation calculi.

ACKNOWLEDGEMENTS

F. Fallah and S. Devadas were supported in part by a grant from Schlumberger Foundation. We thank Abhijit Ghosh for his contributions to this project.

REFERENCES

[1] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek. Test Program Generation for Functional Verification of PowerPC Processors in IBM. In *Proceedings of the* 32^{nd} *Design Automation Conference*, June 1995.

[2] B. Beizer. *Software Testing Techniques*. Van Nostrand Rheinhold, New York, second edition, 1990.

[3] M. A. Breuer and A. D. Friedman. *Diagnosis and Reliable Design of Digital Systems*. Computer Science Press, 1976.

[4] K-T. Cheng and A. S. Krishnakumar. Automatic Functional Test Generation Using the Extended Finite State Machine Model. In *Proceedings of the* 30^{th} *Design Automation Conference*, pages 86–91, June 1993.

[5] S. Devadas, A. Ghosh, and K. Keutzer. An Observability-Based Code Coverage Metric for Functional Simulation. In *Proceedings of the International Conference on Computer-Aided Design*, pages 418–425, November 1996.

[6] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolsfthal. Coverage-Directed Test Generation Using Symbolic Techniques. In *Proceedings of the Int'l Conference on Formal Methods in CAD*, November 1996.

[7] T. Goradia. Dynamic Impact Analysis: A Cost Effective Technique to Enforce Error Propagation. In *Proceedings of Int'l Symposium on Software Testing and Applications*, March 1993.

[8] A. Gupta, S. Malik, and P. Ashar. Toward Formalizing a Validation Methodology Using Simulation Coverage. In *Proceedings of the* 34^{th} *Design Automation Conference*, pages 740–745, June 1997.

[9] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill. Architecture Validation for Processors. In *Proceedings of the* 22^{nd} *Annual Symposium on Computer Architecture*, June 1995.

[10] Y. Hoskote, D. Moundanos, and J. A. Abraham. Automatic Extraction of the control flow machine and application to evaluating coverage of verification vectors. In *Proceedings of the Int'l Conference on Computer Design*, pages 532–537, October 1995.

[11] K. D. Jones and J. P. Privitera. The Automatic Generation of Functional Test Vectors for Rambus Designs. In *Proceedings of the* 33^{rd} *Design Automation Conference*, pages 415–420, June 1996.

[12] 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 *Proceedings of the* 33^{rd} *Design Automation Conference*, pages 325–330, June 1996.

[13] B. Marick. *The Craft of Software Testing*. Prentice-Hall, Englewood Cliffs, N. J., 1995.

[14] J. P. Roth. Diagnosis of Automata Failures: a Calculus and a Method. *IBM journal of Research and Development*, 10:278–291, July 1966.

[15] D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Boston, MA, second edition, 1994.

[16] J. M. Voas. PIE: A Dynamic Failure-Based Technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, August 1992.