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

Farzan Fallah
Fujitsu Labs. of America, Inc.
595 Lawrence Expwy
Sunnyvale, CA 94086
farzan@fla.fujitsu.com

Srinivas Devadas
Rm. NE43-258
545 Technology Square, MIT
Cambridge, MA 02139
devadas@caa.lcs.mit.edu

Kurt Keutzer
Department of EECS
University of California
Berkeley, CA 94720
keutzer@eecs.berkeley.edu

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. For most designs one of the hardest verification problem is verifying the correctness of the initial register-transfer level (RTL) description coded into a Hardware Description Language (HDL). 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. 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 is used in OCCOM is that of fault simulation used in manufacturing test. A fault simulator enables one to assess that the fault coverage is high enough. This is critical to ensuring that the design is adequately tested. By analogy, we are interested in providing HDL 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.

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 [3] 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 [20], and the impact analysis methods of Goradia [11].

Similar approaches can be taken in hardware; however, hardware offers much less observability of *meaningful* [1] data 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 HDL 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).

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

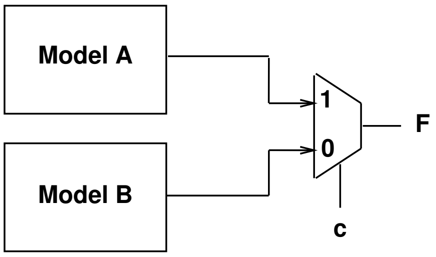

Fig. 1. Observability versus controllability coverage.

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.

A simple example of why observability information is important to the designer is shown in Figure 1. 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.

This paper describes 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.

The concurrent tag propagation algorithm proposed in this paper circumvents the drawbacks of previously-proposed coverage metrics with a new two-phase approach. The two phases are:

1. The given HDL model is modified automatically, in particular some new variables are added and some statements are moved out of conditionals, and *the given vectors are simulated using commercial* HDL *simulators.* Thus, current (future) simulation technology is (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. A flowgraph from the modified HDL model is created, and the results of simulation are used to determine coverage under OCCOM. Using concurrent evaluation techniques, this phase can be executed efficiently.

The remainder of this paper is organized as follows. Section II explains related work in manufacturing test, software testing, and HDL coverage analysis. Section III describes tag propagation which is the basic strategy for computing OCCOM coverage. Section IV explains HDL code modification. Section V describes the graph-based forward tag propagation method and the concurrent tag propagation method. Section VI presents experimental results. Section VII presents directions for future work.

II. Related Work

This section describes representative coverage analysis methods from the manufacture test and software test literature, as well as HDL coverage analysis work.

A. Manufacturing Test

The basic premise of manufacturing test is the modeling of manufacturing defects as logical faults. Since manufacturing is a physical process that can be analyzed, credible fault models can be derived. For example, defects are known to cause breaks and shorts in metal wires. These breaks or shorts can be modeled as logical faults since there is a direct correspondence between wires in silicon and connections in the logic circuit.

A.1 Fault Models

One of the most popular fault models in manufacturing test is the stuck-at fault model [1]. The stuck-at fault model is a logical fault model where any wire in the logic circuit can be stuck-at-1 or stuck-at-0. A test vector that produces the opposite value (0 for a stuck-at-1, and 1 for a stuck-at-0) will *excite* the fault. The effect of the fault has to be *propagated* to an observable circuit output in order for the fault to be detected by the vector.

A.2 Fault Coverage and Simulation

For any fault model, given a test vector set, the *fault coverage* of the test vector set can be computed using *fault simulation.* For every possible fault in the fault model, it is checked, for each vector in the vector set, whether the fault is excited and propagated to a primary output. Fault coverage for a vector set is defined as the number of detected faults divided by the total number of faults. Fault coverage measures the "goodness" of a vector set in detecting all the faults. A test set with higher fault coverage is more likely to detect bad integrated circuits and so fault coverage is used to drive the test generation process.

Fault simulation can conceivably be used to provide coverage metrics for HDL models. However, fault simulation would require a synthesis of the HDL model into gates, and is typically too time-consuming for a large model and a large set of vectors. Further, the faults targeted would be stuck-at faults on gate inputs/outputs, most of which are unrelated to HDL model errors.

A.3 Functional Testing and Functional Fault Models

As mentioned earlier, the direct correspondence between a metal wire in the silicon integrated circuit and a connection in the logic circuit motivates logical fault models. No such correspondence may exist for a behavioral description in an HDL or structural register-transfer level (RTL) description. Statements in the HDL description may correspond to hundreds of gates and wires in the final design. Some efforts have been made to model faults as perturbations of transitions in a State Transition Graph description of a circuit [6] and at the register-transfer level for microprocessors [4] [19]. Error models that reflect incorrect connections or gates in a gate-level circuit have been proposed along with error simulation methods in [16].

The quality of a functional fault model is determined by the number of single stuck-at faults detected by a functional test set that produces 100% coverage for the functional fault model. The proposed functional fault models attempt to obtain high stuck-at fault coverage, rather than attempting to discover bugs in the HDL description. Further, the effectiveness of test sequences cannot be evaluated directly at the functional level [1].

B. Software Testing

The problem of verifying the correctness of an HDL description of circuit behavior is similar to the software testing problem because the description of circuit behavior is similar to a program written in a high-level programming language like C or C++. An important difference is that software programming languages are more expressive than HDL's, leading to more complicated descriptions and test procedures.

A *control flowgraph* is a graphical representation of a program's control structure [3, Chapter 3]. 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 instruction 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.

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.

C. HDL Coverage Analysis

Coverage analysis techniques proposed for general HDL models include – guaranteed coverage of every statement in an HDL model [7], evaluation of transition coverage of a test set [14], and abstraction of models and semantic control over transition coverage [10]. These metrics do not directly address observability requirements.

Observability requirements are addressed in the metric proposed in [8]. 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 [8].

1. The calculus was developed for only a subset of Verilog In particular, nested if statements cannot be handled, and neither can looping constructs used in popular HDL's.
2. 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.* [13] 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 [12]. 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., [2], [15], [17]).

III. Computing Coverage : Basic Strategy

An HDL model is viewed as a structural interconnection of modules. The modules can be comprised of combinational logic, and registers. The combinational logic can correspond to Boolean operators (e.g., AND, OR, NOT), or arithmetic operators (e.g., +, >).

A. Tag Coverage

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. It is, however, desired 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 [8]. 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. [2] 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 [5].)

B. Two-Phase Approach

Computing an observability metric for a collection of logic gates is similar to fault simulation using the D-calculus [18]. 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 arithmetic operations, conditionals, and looping constructs.

A calculus can be defined to handle HDL models, for example, augmenting the calculus of [8], 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, a two-phase approach can be used to compute OCCOM coverage metric. The two-phase approach is:

1. The given HDL model is modified. The modifications are necessary because of conditionals in the HDL model. In particular, some new variables and statements are added. Then the modified HDL model is simulated using a standard HDL simulator. Note that tags do not play a part in this phase.
   Simulating the modified HDL model will provide 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, tags are injected and propagated. A flowgraph is extracted from the HDL model. There are two different possibilities for doing tag propagation. In forward tag propagation, tag injection simply corresponds to introducing a tag on a vertex in the graph and tag propagation corresponds to selectively traversing paths from the vertex to the output nodes. In concurrent tag propagation, tag injection corresponds to introducing a tag on the observable vertex O in the graph corresponding to outputs and tag propagation corresponds to selectively traversing paths from the observable vertex O backward.

Phase 1 of algorithm is described in Section IV, and Phase 2 is described in Section V. The tag simulation calculus is described in Section III-C. Note that this calculus is not used in Phase 1, but only in Phase 2. Nevertheless, in order to understand why the HDL description is modified, the tag simulation calculus needs to be described first.

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. Both positive and negative tags are considered, $+\Delta$ written simply as Δ, and $-\Delta$. If the presence or sign of the tag is not known, an unknown tag is used. An unknown tag is shown by "?". Note that $? \equiv -?$, and also $0+? \equiv 1+?$. In the sequel, the tag simulation calculus is defined for multiple operators. Note that the following calculus is based on the likelihood of the propagation of the tag. If the propagation of the tag depends on its magnitude, it is assumed that the tag is propagated or blocked depending on which case is more likely. For example, in the Verilog statement $c = (a \ != b)$ with $a = 2$ and $b = 5$, if there is a positive tag on variable a, it is assumed that the tag is not propagated to the variable c. The reason is that the value of the variable c in the presence of the tag is TRUE unless the magnitude of the tag on a is exactly 3. As a result it is unlikely to have a tag on variable c.

The calculus can be changed easily, but the resulting analysis might substantially underestimate or overestimate the coverage of a set of test vectors.

C.1 Logic Gates

First, the calculus is defined for Boolean logic gates. This is similar to the D-calculus [18]. The calculus for an INVERTER, a two-input AND gate, and a two-input OR gate are shown in Table I. The five possible values at each input

[2] 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.

inverter	
0	1
1	0
0 + Δ	1 - Δ
1 - Δ	0 + Δ
0 + ?	0 + ?

and	0	1	0 + Δ	1 − Δ	0 + ?
0	0	0	0	0	0
1	0	1	0 + Δ	1 - Δ	0 + ?
0 + Δ	0	0 + Δ	0 + Δ	0	0 + ?
1 - Δ	0	1 - Δ	0	1 - Δ	0
0 + ?	0	0 + ?	0 + ?	0	0 + ?

or	0	1	0 + Δ	1 − Δ	0 + ?
0	0	1	0 + Δ	1 − Δ	0 + ?
1	1	1	1	1	1
0 + Δ	0 + Δ	1	0 + Δ	1	0 + Δ
1 - Δ	1 - Δ	1	1	1 - Δ	0 + ?
0 + ?	0 + ?	1	0 + Δ	0 + ?	0 + ?

TABLE I

Δ-CALCULUS FOR INVERTER, AND GATE, AND OR GATE.

adder	b	b - Δ	b + Δ	b + ?
a	a + b	a + b - Δ	a + b + Δ	a + b + ?
a - Δ	a + b - Δ	a + b - Δ	a + b + ?	a + b + ?
a + Δ	a + b + Δ	a + b + ?	a + b + Δ	a + b + ?
a + ?	a + b + ?	a + b + ?	a + b + ?	a + b + ?

TABLE II

Δ-CALCULUS FOR ADDER.

are $\{0, 1, 0+\Delta, 1-\Delta, 0+?\}$. (Note that $0 - \Delta \equiv 0$ and $1+\Delta \equiv 1$.) As an example, if the input of an inverter gate is 0 and it has positive tag on it, the value of the output of the inverter will be 1, and it will have a negative tag on it. The case that the input of the inverter is 1 and the input has a negative tag is similar. As another example, if one of the inputs of an AND gate is 0 and the input has a positive tag and the value of the other input is 1 and it has a negative tag on it, the value of the output of the AND gate will be 0, because the erroneous value of one of the inputs is 0.

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

C.2 Arithmetic Operators

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

Adder : If all tags on the adder inputs are positive, and if the value $v(f) < MAXINT$, the adder output is assigned to $v(f)+\Delta$. $MAXINT$ 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 tag. Table II shows calculus for tag propagation through an adder.

>	b	b + Δ	b - Δ	b + ?
a	1	1 - Δ	1	1 + ?
a + Δ	1	1+ ?	1	1 + ?
a - Δ	1 - Δ	1 - Δ	1 + ?	1 + ?
a + ?	1 + ?	1 + ?	1 + ?	1 + ?

TABLE III

Δ-CALCULUS FOR $>$ WHEN RESULT OF $a > b$ IS TRUE.

>	b	b + Δ	b - Δ	b + ?
a	0	0	0 + Δ	0 + ?
a + Δ	0 + Δ	0 + ?	0 + Δ	0 + ?
a - Δ	0	0	0 + ?	0 + ?
a + ?	0 + ?	0 + ?	0 + ?	0 + ?

TABLE IV

Δ-CALCULUS FOR $>$ WHEN RESULT OF $a > b$ IS FALSE.

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. The output of multiplier is assigned to $v(f) + \Delta$. Similarly for negative Δ.

$>$ *Comparator* : If tags exist on inputs a and b, they have to be of opposite sign, else the output will have an unknown tag. 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. The output of comparator is assigned to $0 + \Delta$. Similarly for other tags and other kinds of comparators. Tables III and IV show the calculus for tag propagation through a $>$ operator when the result of operation is TRUE and FALSE, respectively.

Bit extraction : If tags exist on A, where A is an n-bit variable, it is assumed that the tag cannot propagate to a single bit of A. For example, $A[2]$ is assumed to be tag free even if there is a tag on A. The reason is that, in order to have a tag on $A[2]$, it is necessary to have $A'[2] \neq A[2]$, where $A' = A + \Delta$. This requires to have either the third bit of (Δ) equal to 1 or a carry from the sum of the lower bits of A and Δ. In other words the values of A and Δ have to satisfy the following formula, $2^3 > A[1:0]+\Delta[2:0] \geq 2^2$. This restricts the magnitude of tags which can be propagated and makes the tag propagation analysis very complex (Note that in order to be pessimistic or conservative, it can be assumed that there is an unknown tag on $A[2]$. In our system, we assume that there is no tag on A).

Concatenation : If a tag exists on A, it is assumed that it is propagated to $\{A, B\}$.

Bitwise AND : If one of the operands is 0 and tag-free, the tag is not propagated through operator. If all bits of one of the operands is 1 and tag-free, the tag on the other operand is propagated through operator. For other cases, refer to [9].

Bitwise NOT : $v(f_i) = \overline{v(a_i)},\ 0 \leq i < n$. For a positive tag on a, the output is assigned $v(f) - \Delta$.

[9] gives the calculus for all operators used in the Ver-

ilog hardware description language.

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, it is not known what happens when the incorrect branch is taken; the only information is regarding the simulation of the correct branch. [3]

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, the case where x is not assigned and z is assigned has to be considered. While the values of x prior to and after the assignment are known, only the old value of z is known, since the simulator does not compute *exp2*.

Nested conditionals and loops exacerbate the above situation. In the next section the way various cases are handled by modifying the HDL model prior to simulation is described.

IV. Phase 1: Modifying the HDL Model

In this section various commonly occurring cases of conditionals and loops in Verilog are considered, and it is shown how to 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.

A. Simple Conditional

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

```
                            y1 = expr1 ;
                            y2 = expr2 ;
if (cexp)                   if (cexp)
      y = expr1 ;                 y = expr1 ;
else y = expr2 ;            else y = expr2 ;
```

[3] The approach of [8], somewhat arbitrarily, tags all the variables in the assignments that would be missed under the tag condition.

Consider the case of a tag on *cexp*. During the simulation of the modified code, the values of both *expr1* and *expr2* are computed, and stored in the new variables *y1* and *y2*.

The new values of y corresponding to the execution of both the *then* and *else* clauses are known, regardless of the value of *cexp* during simulation. This will help to correctly propagate positive or negative tags on *cexp* in Phase 2 (cf. Section V).

Note that it has been assumed that computing expressions has no side effect, otherwise it is necessary to save the values of the variables and restore them to make the behavior of the modified model and the original model equivalent.

B. Nested Conditionals

The case of nested conditionals is more complicated. Further, the situation where variables such as y are assigned values that depend on the old values (e.g., increment operation) has to be considered.

As an example, consider the following Verilog statements.

```
if (cexp1)
  begin
    if (cexp2)
      y = expr1 ;
    if (cexp3)
      y = y + expr2 ;
  end
```

Transformation starts with transforming the *if(cexp1)* statement,

```
if (cexp2)
   y = expr1 ;

if (cexp3)
   y = y + expr2 ;

if (cexp1)
  begin
    if (cexp2)
      y = expr1 ;
    if (cexp3)
      y = y + expr2 ;
  end
```

In the next step, the *if(cexp2)* and the *if(cexp3)* statements are transformed. y is the only variable in the original Verilog code whose value is changed inside the *if* statement, and as a result, in order to transform the code, a new variable, $y3$, is introduced.

```
y3 = y ;
y1 = expr1 ;
```

```
if (cexp2)
  y3 = y1 ;

y2 = y3 + expr2 ;
if (cexp3)
  y3 = y2 ;

if (cexp1)
  begin
    if (cexp2)
      y = expr1 ;
    if (cexp3)
      y = y + expr2 ;
  end
```

Note that if the value of *cexp2* is false, variable $y3$ is read before assigning any value to it. As a result, it is necessary to initialize its value to the value of y.

The original code and the transformed code are shown below. The transformed code will compute the necessary information to perform propagation of tags on *cexp1*, *cexp2*, or *cexp3*.

```
if (cexp1)
  begin
    if (cexp2)
      y = expr1 ;

    if (cexp3)
      y = y + expr2 ;
  end
```

```
y 3 = y ;
y1 = expr1 ;

if (cexp2)
  y3 = y1 ;

y2 = y3 + expr2 ;
if (cexp3)
  y3 = y2 ;

if (cexp1)
  begin
    if (cexp2)
      y = expr1 ;
    if (cexp3)
      y = y + expr2 ;
  end
```

It can easily be verified that the two pieces of code result in the same values for y.

C. Loops

Consider the *for* loop shown on 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.

```
for (i=0; i<N; i++)
    y = y + expr ;
```

```
for (i=0; i<N; i++)
    y = y + expr ;
y1 = y + expr ;
```

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.

It is assumed that the values of the variables inside the body of loop are monotonic functions of N. As a result, in order to do the tag propagation, instead of using the values of the variables when the loop iterates $N + \Delta$ times, their values after $N + 1$ iterations can be used. Similarly, instead of using the values of the variables when the loop iterates N - Δ times, their values after N - 1 times can be used. Using this assumption helps to determine the propagation of tag independent of its magnitude and simplifies the coverage analysis problem. Otherwise, the propagation of the tag will depend on the magnitude of the tag which is unknown.

Clearly, the monotonicity assumption is not always true. To be pessimistic, it can be assumed that the tag on the variables present in the left-hand-side of statements in the body of the loop is "?".

The modified loop given above results in the same value for y as the unmodified code. However, *y1* will contain the value for the case where the loop is iterated N + *1* times.

Note that tags are not injected on the loop counter variable i. It is assumed that errors on i are reflected by changes in the number of loop iterations, i.e., N. Propagating tags on variables such as y does not require additional information. Similar transformations can be made for *repeat* loops. Readers can find the transformation algorithms in [9].

After modifying the HDL model, the modified version is simulated using commercial HDL simulators. Information about the values of the variables and simulation trace are stored in a file. This information is used in the Phase 2 of the algorithm to perform tag propagation; we describe this propagation of tags in the next section.

V. Phase 2: Graph Analysis

In this section, it is described how forward and concurrent tag propagation are performed, given the information regarding variable values for the simulated functional vectors. A flowgraph data structure is used to enable efficient tag propagation.

A. Graph Structure

A graph $G(V, E, L(E))$ from the given HDL model is created. Figure 2 shows a Verilog code fragment and its corresponding graph.

Each vertex $v \in V$ corresponds to a variable in the HDL model, like variable A in Figure 2. Also, there is a constant vertex *CONST* in the graph. 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. As an example, the edge between nodes A and C in Figure 2 shows a data dependence between variables A and C.

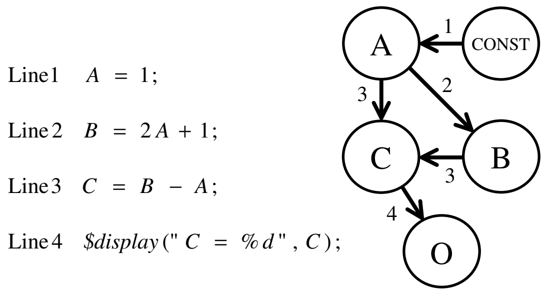

Fig. 2. A Verilog model and its corresponding graph.

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

- Line numbers: 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, in Figure 2 the edges from A to C have Line 3 as a label. Also, a particular line of the *unmodified* HDL file might be associated with the edge.
- Conditional expression: There might be 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 $x > y$.
- 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] ;
```
there will be a single node for the array b and the array a and an edge from b to a. Information corresponding to the array indices are associated with the edge from b to a. In general, two indices, one for b and one for a are required. For each index, one expression is used. These indices have to be computed 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 the 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: There is a multiplier associated with each edge that will be used to multiply tags propagating through 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 the code fragment below.
```
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. Note that only the sign of *a1 - a2* is used, not its magnitude.

A single observable vertex O in the graph that does not correspond to any variable is created. 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*. 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.

A single vertex *CONST* in the graph is created. There is an edge from the vertex *CONST* to every variable in the left-hand-side of each assignment for whose right-hand-side is constant.

Note that the data dependence graph can be constructed irrespective of the number of processes in the HDL description.

B. Forward Tag Propagation

Forward tag propagation is performed on the graph structure, and uses information obtained from the simulation trace of the modified HDL model. For each vector, and each tag injected in the graph, it is determined if the tag is propagated to the observable vertex O. 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. All inactive edges are deleted from the graph.
3. A positive tag and a negative tag are injected in an edge which corresponds to an assignment in the unmodified HDL model.
4. The edges are traversed in the graph, in the order of simulation trace. For each edge (v, w), if it is the first edge corresponding to a line in HDL code, make the node w tag-free before propagating the tag through the edge. If there is a tag on node v, and it can be propagated through the edge, it is propagated to node w and its sign is defined. After that, it is combined with the previous tag of node w. If a tag has been injected on edge (v, w), it is propagated to node w, its sign is defined and it is combined with the previous tag of node w.
5. If during the tag propagation, the tag propagates to the observable vertex O, it is possible to detect the injected tag under the current test vector.

Figure V shows the progress of the forward tag propagation algorithm for a simple Verilog model. B is input of the model and its value is 0 for the selected test vector. First, inactive edges in the graph are found and deleted from the graph. Because $B = 0$, the edge between A and C is inactive and can be deleted from the graph, as shown in Figure V (b). Tag propagation starts with injecting a tag in the first assignment. After processing the first line of the Verilog code, there will be a tag on variable A. Note that this tag appears on all instances of variable A (cf. Figure V (c)). In the next step, the second line of the Verilog code is processed. There is a tag in the right-hand-side of the assignment, but because the value of the variable B is 0, it prevents the tag from being propagated to the left-hand-side. As a result, there will be no tag on variable C after processing the second assignment (cf. Figure V (d)). In the next step, the *display* statement is processed. Because there is no tag on variable C, there will be no tag in the output (cf. Figure V (e)). This means that using the current test vector, the injected tag in the first line cannot be detected in the output. Note that in order to find out if the tag injected on the second line is propagated to the output using the current test vector, it is necessary to do the analysis again. In this case, it is possible to detect the tag injected on the second line.

C. Concurrent Tag Propagation

For each vector, all the tags that can be propagated to the observable vertex O can be defined, using concurrent tag propagation. The steps are:

1. Each edge in the graph is first determined to be active or inactive; see Section V-B.
2. All inactive edges are deleted from the graph.
3. A positive (negative) tag is injected on the observable vertex O.
4. Starting from the observable vertex O, and assuming a positive (negative) tag on the vertex, the edges are traversed 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.) There will be a positive or negative tag on reachable vertices. Note that paths with multipliers of different signs, might result in tag cancellation, and it is conservatively assumed that the tag is not propagated. Note that after taking the last edge in a set of edges in the fanin of a vertex corresponding to a single line in HDL code, the tag is removed from that vertex. The reason is that taking that edge corresponds to simulating a line of HDL description with the variable of that vertex in the left-hand-side.
5. Line numbers (in the unmodified HDL model) of edges taken whose head vertices are reachable from observable vertex before taking those edges, determine the tags which are observable in O.

Note that if a variable appears in two different *display* statements, one with positive sign and the other with negative sign, the concurrent tag propagation algorithm as described will find that variable unreachable, hence it is pessimistic. In order to fix this, we would need different observability vertices for different *display* statements.

In practice, the concurrent tag propagation algorithm uses a graph which is slightly different than the graph used in the forward tag propagation algorithm. Note that if we process the HDL statements backward, there might be cases that there are several possibilities for propagating a tag through an operator. As an example, consider the following Verilog statement $c = (a \mathrel{!}= b)$, when values of variables a and b are equal to 2 and there is a positive tag on variable c. A positive tag on variable a and a negative tag on variable b will result in a positive tag on variable c. Similarly, a negative tag on variable a and

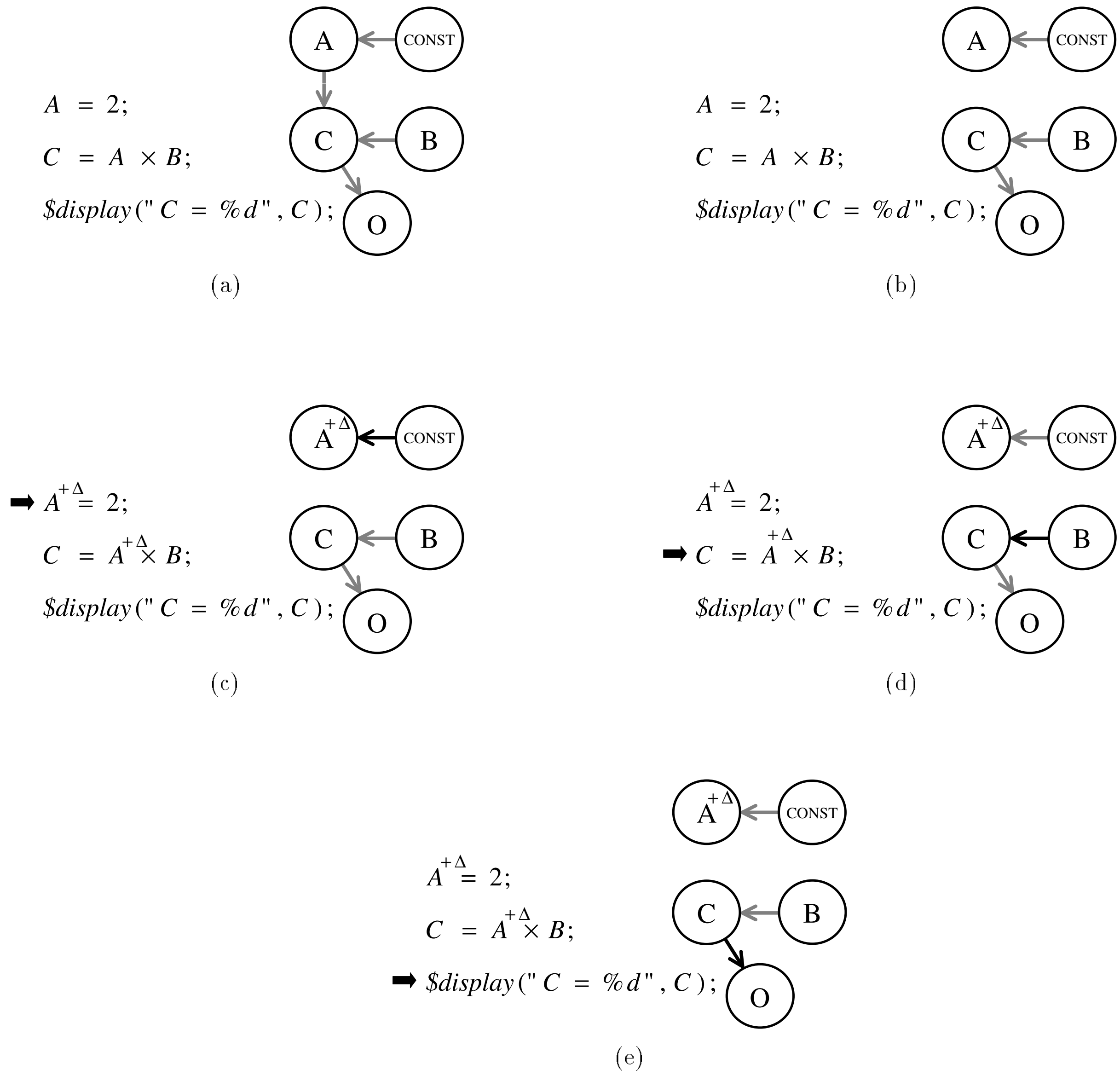

TABLE V
FORWARD TAG PROPAGATION EXAMPLE.

a positive tag on variable b will result in a positive tag on variable c. When building the graph, only *one* of the possibilities is considered and the corresponding edges are added to the graph. Because of this simplification, the result of the concurrent tag propagation algorithm can be different than the result of the forward tag propagation algorithm. In particular, the concurrent tag propagation algorithm might report that some detectable tags are undetectable. In order to make the result consistent with the result of the forward tag propagation algorithm it is necessary to try both possibilities in the above example. This makes the worst-case running time of the algorithm exponential in the number of operators. A better solution is to use the forward tag propagation algorithm for the tags which have been reported undetectable by the concurrent tag propagation and to check to see if they can be detected. This needs to be done only in the case where certain possibilities have been discarded during graph generation.
Figure VI shows the steps involved in running the concurrent algorithm on an example. Figure VI (a) shows the Verilog description and the corresponding graph. Concurrent tag propagation starts with injecting a tag on the

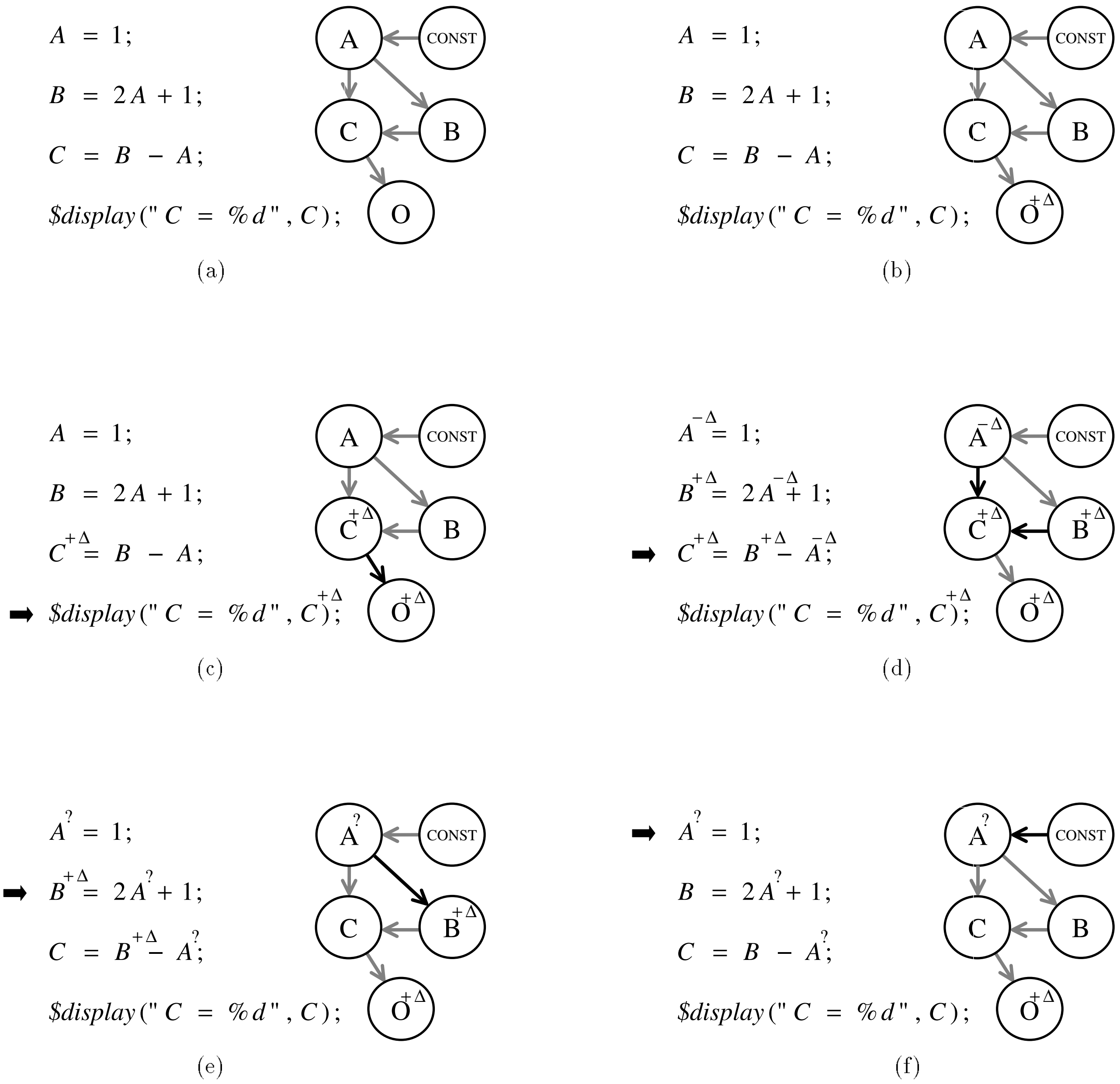

TABLE VI
CONCURRENT TAG PROPAGATION EXAMPLE.

observable vertex O (cf. Figure VI (b)). After that algorithm proceeds with processing the last statement in the description, *display* statement. This results in backward propagation of the tag through bold edge in the graph to node C (cf. Figure VI (c)). In the next step, the third statement is processed and tag on variable C is propagated to variables B and A. Note that tag on variable A has negative sign (cf. Figure VI (d)). After processing the third statement, variable C becomes tag free (cf. Figure VI (e)). The algorithm proceeds with processing the second statement. A positive tag on variable B is propagated to variable A. There is already a negative tag on variable A, so after propagating the tag, they cancel each other. This situation is shown by putting a "?" on variable A which is an unknown tag (cf. Figure VI (e)). After processing the second statement, variable B becomes tag free (cf. Figure VI (f)). Finally, the first statement is processed (cf. Figure VI (f)). In the next step, lines of Verilog code corresponding to edges taken when their head nodes where reachable at the time those edges were taken are found. For this example, there are the second and the third lines. Tags injected in the second and the third lines can be detected in the output and other tags cannot be detected.

VI. Results

A. Metric Comparison

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

The results in Table VII serve to illustrate the usefulness of OCCOM in relation to vanilla controllability measures. A simple example of how controllability information can be misleading in Figure 1 was given. In Table VII, statistics comparing line coverage and OCCOM coverage for designer-generated functional tests (labeled Directed), and randomly-generated tests (labeled Random) have been given.

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[4] 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

Comparison of the performance of concurrent tag propagation algorithm against functional simulation, and the method of [8] has been given in the Table VIII. 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 a 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 a Sun Ultra-Sparc 30/300 with 256 MB of RAM running at 300 MHz.

The simulator of [8] could not be run for some of the examples, since the simulator does not handle certain Verilog constructs such as nested if statements. The described method is faster than the specialized tag simulation because it allows for the use of highly-optimized commercial HDL simulator and because it uses concurrent tag propagation.

The concurrent tag propagation algorithm 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.

The CPU time for modifying HDL description and simulating it grows linearly in terms of size of the description. Also, the tag propagation analysis can be done in the worst case in quadratic time, hence it is possible to use this method for large designs. In practice, the results show a linear increase in CPU time in comparison to simple HDL simulation.

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, the information regarding whether the line on which the tag was initially injected was executed (controllability information) is known. All the vectors (if any) for which the line was executed are determined. 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. Future Work

It is possible to improve tag analysis method described in the previous sections to achieve more accurate results at the expense of the speed of the algorithm. In this section some possible improvements are described.

A. Relative Magnitude of a Tag

In the tag analysis described in the previous sections, it was assumed that a variable is tag-free, has a positive tag, a negative tag, or an unknown tag. During tag propagation,

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

Examples	#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 VII
Comparing OCCOM and a line coverage metric.

Examples	HDL Simulation	OCCOM			Specialized Simulator [8]
		Phase 1	Phase 2	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 VIII
Comparing times required to compute metrics.

the algorithm only kept track of the sign of the tag, not its magnitude relative to the original injected tag. As an example, during the tag analysis for $A = 2 \times B$, when B has positive tag (i.e., Δ), the tag on A was assumed to be Δ, not $2 \times \Delta$.

Using the sign of the tag helped to detect the possibility of tag cancellation or asymmetric tag propagation (e.g., the propagation of tag through a control expression). This achieves more accurate results than a method that ignores the sign. On the other hand it requires performing tag analysis twice, once for each sign of the tag.

It is possible to keep track of the relative magnitude of the tag during tag analysis. This will help to obtain more accurate results for tag cancellation, but it means considering more cases for a tag, which in turn will expend more CPU time.

Note that, even in this case, it is necessary to make some assumptions on the magnitude of the tag in order to perform tag analysis. Also, in some cases, it is necessary to approximate the relative magnitude of the tag when it is propagated through different operators. For example, if $A = B \times C$, and both B and C are zero and have positive tags on them (i.e., Δ), the tag on A will be Δ^2. In this case, it is possible to keep track of all powers of Δ, or to simply approximate it with $K \times \Delta$, where K is an integer number.

B. Absolute Magnitude of a Tag

A possible improvement on simple tag analysis which ignores the magnitude of the tag, is assuming a specific magnitude for the injected tag and performing more detailed analysis to see if the values of any of the outputs are changed. This requires tag analysis assuming the absolute magnitude of the injected tag is 1, 2, 3, etc, all the way up to the maximum value of the variable. This achieves the most accurate results, on the other hand it is too time consuming to do this for all possible magnitudes of the tag.

C. Injecting Tag on Expressions

In the described OCCOM analysis, the design errors were modeled with injection of tags on the assignments. Injecting tags on the assignments can capture some design errors, but not all of them. A better result can be achieved using injection of tag on every expression. For example consider the following Verilog description,

```
A = 0 ;
B = 0 ;
if ( C > 4 )
     A = 1 ;
else B = 1 ;
```

Assume that there is an error in the control expression of the *if* statement and that the expression should be $C <= 4$ instead of $C > 4$. Every time the *if* statement is simulated, the values of both variables A and B will be erroneous, and there will be a tag on them. Those tags can interact and possibly cancel each other.

If a tag is injected only on the assignments, it is injected on only one of the assignments in *then* or *else* at a time. As a result, there will be no interaction or cancellation effect and the error in the control expression cannot be modeled.

By injecting tags on the control expression this case can be handled easily.

D. *Multiple Tag Model*

During OCCOM analysis, a single tag assumption has been used. In other words it was assumed that there was only one error in the design and effect of that error was modeled by injecting a tag in one of the assignments. It is possible to use a multiple tag model and inject a tag on several assignments at the same time and then perform tag propagation. Different tags can interact during tag propagation and cancel each other, however, the described algorithm will work for this case as well. To achieve better results, some information regarding relative magnitudes of the tags can be used.

The main issue with the multiple tag model is that there are an exponential number of possible multiple tag injections, and methods to choose a small number of them are an absolute necessity. These methods may have to rely on designer intuition, and may be difficult to automate.

References

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design.* IEEE Press, 1990.

[2] 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*, pages 279–285, June 1995.

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

[4] D. Brahme and J. A. Abraham. Functional Testing of Microprocessors. *IEEE Transactions on Computers*, C-33(6):475–485, June 1984.

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

[6] K-T. Cheng. Transition Fault Testing in Sequential Circuits. *IEEE Transactions on Computer-Aided Design*, 12(12):1971–1983, December 1993.

[7] 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.

[8] 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.

[9] F. Fallah. *Coverage Directed Validation of Hardware Models.* PhD thesis, MIT, 1999.

[10] 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*, pages 143–158, November 1996.

[11] T. Goradia. Dynamic Impact Analysis: A Cost Effective Technique to Enforce Error Propagation. In *Proceedings of Int'l Symposium on Software Testing and Applications*, pages 171–181, March 1993.

[12] 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.

[13] 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*, pages 404–413, June 1995.

[14] 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.

[15] 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.

[16] S. Kang and S. A. Szygenda. Modeling and Simulation of Design Errors. In *Proceedings of the Int'l Conference on Computer Design: VLSI in Computers and Processors*, pages 443–446, October 1992.

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

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

[19] S. M. Thatte and J. A. Abraham. Test Generation for Microprocessors. *IEEE Transactions on Computers*, C-29(6):429–441, June 1980.

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