An Observability-Based Code Coverage Metric for Functional Simulation

Srinivas Devadas
Department of EECS
MIT, Cambridge

Abhijit Ghosh
Sunnyvale R&D Laboratory
Mitsubishi ITA

Kurt Keutzer
Advanced Technology Group
Synopsys, Mountain View

Abstract— **Functional simulation is the most widely used method for design verification. At various levels of abstraction, e.g., behavioral, register-transfer level and gate level, the designer simulates the design using a large number of vectors attempting to debug and verify the design. A major problem with functional simulation is the lack of good metrics and tools to evaluate the quality of a set of functional vectors. Metrics used currently are based on instruction counts and are quite simplistic. Designers are forced to use ad-hoc methods to terminate functional simulation, e.g. CPU time limitations.**

We propose a new metric for measuring the extent of design verification provided by a set of functional simulation vectors. This metric is universal, and can be used uniformly for all designs. Our metric computes observability information to determine whether effects of errors that are activated by the program stimuli can be observed at the circuit outputs.

We provide preliminary experimental evidence that supports the validity of the proposed metric. We believe that using this metric in design verification will result in higher-quality functional tests and improved correctness checking.

I. Introduction

Design verification is the problem of verifying that a design, specified at whatever level, has certain properties required by a specification. The most common approach to design verification is to verify that a description of the design in some hardware description language has the proper behavior as elicited by a series of simulation vectors. The drawbacks of this approach are well understood; exhaustive simulation is required to guarantee correctness and is possible only for the smallest circuits. A degree of confidence can be obtained by simulating the design using a large number of vectors. Currently, there is a lack of good metrics to quantify this degree of confidence, or to gauge the quality of the vector set. The only metrics used correspond to instruction counts, i.e., how often, if at all, an instruction or statement in the code is exercised. More often than not, a design is simulated using a small set of designer-created vectors and then random vectors for as long as is feasible. In this paper we demonstrate the inadequacy of some existing metrics and propose an additional well-defined, easily-computable metric that provides a better measure of the extent of design verification obtained through functional simulation.

Formal verification methods are an alternative to simulation-based verification. Ideally, the architect would like to formally verify and guarantee that the design is correct relative to some specification, such that if it is implemented and fabricated without error, it will result in a correct circuit that performs all the tasks required by the specification. Unfortunately, writing such a specification is itself a hard problem, and further even if such a specification exists, the complexity of checking the equivalence of the specification and the implementation, in the worst case, degenerates to the complexity of exhaustive simulation. The use of formal verification methods for design verification is currently limited.

The manufacturing test and software test processes provide the inspiration for the metric proposed in this paper. In manufacturing test, defects that occur in fabrication are abstracted at the logic or gate level using the stuck-at fault or delay fault models. Given a set of test vectors, fault simulation (under the appropriate fault model) is used to measure the fault coverage of the test vector set. 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.

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. While fault coverage notions in manufacturing test address both controllability and observability requirements, they are, for the most part, limited to the gate level, although some functional fault models (e.g., [4], [16]) have been proposed.

In this paper, we propose a new metric that gives a better measure of the extent of design verification provided by a set of functional simulation vectors. This metric is universal, and can be used uniformly for all designs, unlike design-specific strategies such as those described in [12]. Our metric computes observability information to determine whether effects of errors that are activated by the program stimuli can be observed at the circuit outputs. In order to compute observability information we tag variables during simulation and use a simulation calculus, explained in this paper, to efficiently calculate the coverage provided by an arbitrary set of functional vectors.

We emphasize that 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. The designer of the circuit can use this coverage information to drive the functional verification process, generating vectors until a satisfactory level of coverage is achieved.

In Section II, we describe different validation mechanisms used in the verification, software test and manufacturing test disciplines and discuss the relative merits and demerits of these mechanisms. In Section III we present an observability metric based on tagging variables and an associated simulation calculus that can be used to quantify the degree of design verification provided by an arbitrary set of functional simulation vectors. We briefly describe a method to compute coverage in Section IV. We give a simple example illustrating the merits of the observability-based coverage metric in Section V. In Section VI we provide preliminary experimental evidence that demonstrates the effectiveness of the proposed metric.

II. Design Validation Models and Mechanisms

We describe different validation methods that are relevant to the design verification problem.

A. Formal Design Verification

Approaches to design verification include the use of temporal-logic-based model checking (e.g., [5]), automata-oriented techniques (e.g., [7], [15]), and the use of higher-order logic and theorem proving techniques (e.g., [8] [10]).

Formal verification approaches have the advantage of guaranteeing partial or complete correctness but can be computationally expensive. Further, they often require the designer to specify correctness properties and/or abstract models of the design.

B. *Manufacturing Test*

From a commercial and popular usage standpoint, manufacturing testing is easily one of the most successful of the validation mechanisms. 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 (as opposed to program writing), 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.

B.1 Fault Models

One of the most popular fault models in manufacturing test is the stuck-at fault model [1]. The stuck-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.

B.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, we check for each vector in the vector set, if 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.

B.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 some Hardware Description language (HDL) or structural 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 [4] and at the register-transfer level for microprocessors [3] [16]. Error models that reflect incorrect connections or gates in a gate-level circuit have been proposed along with error simulation methods in [11].

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].

C. *Software Testing*

The problem of verifying the correctness of an HDL description of circuit behavior is similar to the software testing problem as the description of circuit behavior is similar to a program written in some high-level programming language like C or C++. Two main differences are:

- Software programming languages are more expressive than HDL's, leading to more complicated descriptions and test procedures. Examples are pointers, complex types, recursion, inheritance, etc.
- Hardware descriptions are usually written by a process of successive refinement, i.e., abstract descriptions of circuit behavior are converted to structural descriptions by progressively adding detail. It is not unusual to begin with a software programming language description like C and move toward behavioral HDL and finally to structural HDL. In the software engineering world, program specifications rarely exist, and the refinement strategy is not used.

C.1 Control Flowgraphs and Path 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.

An example of a control flowgraph for a simple program is given in Fig. 1. A process is a rectangle, a decision is a diamond, and a junction is a circle.

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 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. The goal of software testing is to have 100% path coverage, which implies branch and line coverage. However, 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, multicondition 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. For example, in our control flowgraph of Fig. 1 an error in the computation of V never propagates to the LOOP if $Z >= 0$. 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.

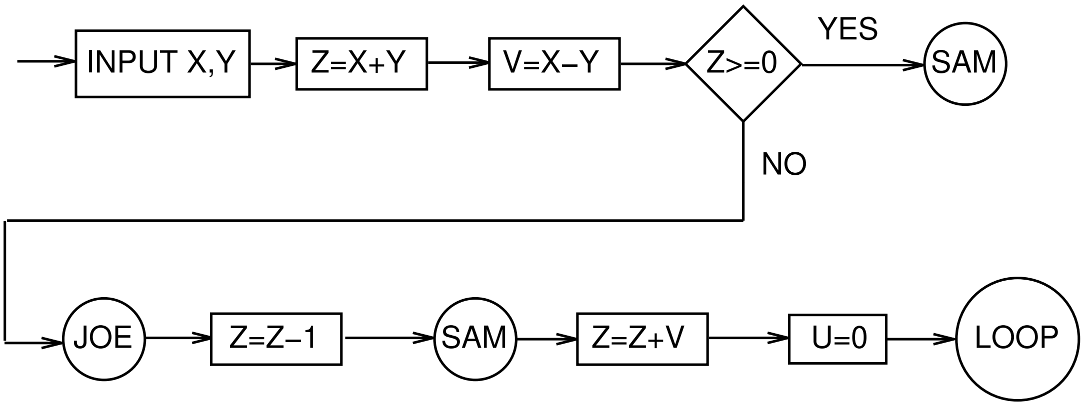

Fig. 1. A Control Flow Graph of a Program

The problem of observing the effect of an activated (possibly erroneous) statement is deferred to the test implementation step in software testing [13]. In test implementation, additional testing code is added to points in the program which are hard to force to particular values (added controllability), and at those points where information is lost (added observability). One problem with these internal test drivers is that variable values may not be easily translatable into useful data that the programmer can understand and check against the specification.

C.2 Error-Sensitive Test Case Analysis

It has long been clear that even complete path coverage does not detect all bugs [9]. The technique of error-sensitive test case analysis [6] inspired by hardware fault simulation, provides three rules for generating test cases that are sensitive to code errors. These rules are heuristics to guide a test case generation process and are necessary in software testing since comprehensive observability analysis is not performed.

D. Summary

Formal design verification methods are currently limited in their applicability. Fault models and fault simulation methods have been very successful in manufacturing test; however, they have only been applied at the logic circuit level. Software test methodologies work for high-level languages, however their coverage metrics ignore observability issues for the most part.

Coverage analysis in functional simulation is relatively well-developed for specific classes of designs such as microprocessors. Many different coverage metrics are used [12], including line coverage in HDL models, toggle coverage which determines if signals are switching or not, and transition coverage for finite state machines. Rarely are observability requirements factored into computing coverage metrics.

III. A Coverage Metric Incorporating Observability Requirements

The notions of fault coverage in manufacturing test and coverage metrics of software test lead us to our metric for evaluating the extent of design verification.

In order to model observability requirements, we have to check the sensitizability of paths from inputs to outputs, i.e., whether effects of errors are propagated through paths. Checking the sensitizability of paths requires us to "tag" variables.

Given input stimuli to a HDL model, we can compute the output response using a simulator. We will enhance the simulator to propagate tags from module inputs to module outputs. In order to deterministically propagate tags through a module, we will need to make certain assumptions. However, these assumptions will not require us to assign particular binary values to the tag, except in the case of tags on single binary variables. (In the D-calculus used in stuck-at fault testing the tag is the Boolean variable D, which is tied to a fault-free value of 1 and a faulty value of 0.)

We develop a simulation calculus for tags which defines the propagation of tags through HDL constructs. Coverage computed by tag simulation is our measure of the extent of design verification provided by a set of functional vectors.

A. Tags

We view a circuit as computing a function and we view the computation as a series of assignments to variables. Errors in computation are therefore modeled as errors in the assignment, i.e., any value assigned to any variable in the behavioral or RTL description may possibly be in error.[1]

The possibility of an error is represented by tagging the variable (on the left hand side of the assignment) 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 $-\Delta$.

We introduce positive and negative tags on every assignment statement in the HDL description of a circuit. For each functional vector, or functional vector sequence, we simulate the circuit to determine which of the tags are first activated (by the activation of the corresponding assignment statement) and then propagated to the circuit's outputs. When a tag is propagated to an output, it means that there is a path from the point where the tag was introduced to the output, and that the path was activated by the functional vectors. We then compute the coverage for the functional vector set. Note that during simulation, the effect of each tag is considered separately (analogous to the single stuck-at fault paradigm). However, the simulator can simulate effects of several tags in parallel as in parallel fault simulation.

We do not claim that bugs in the code will always result in an incorrect value of some HDL variable. Bugs that are errors of omission, or global misassumptions regarding program/algorithm behavior may not cause this effect. Tags reflect the two basic requirements

[1] This includes variables in control statements.

in verifying a model, namely activating statements in the code, and observing the effect of activation.

B. Tag Propagation Issues

In order for an input's value to be propagated through a module, the other inputs to the module must be at non-controlling values. For example, if an AND gate has a 0 input, no value on the other input will propagate to the output since 0 is a controlling value for the AND gate. In a multiplier, a 0 at one of the inputs will block propagation of any value on the other input.

If there is an error in some assignment, the magnitude of the error determines whether the error will be propagated or not. As an example consider the expression $f = a + b > c$. Assume 4-bit integers for a, b and c, and assume that the functional simulation vector given is $S = \langle a = 3,\ b = 4,\ c = 5\rangle$. A positive error on a is not propagated to f, since the comparison provides the same value of 1 for f. A negative error may be propagated to f resulting in an erroneous value of 0. This will occur if the magnitude of the error is greater than or equal to 3.

It is obvious that the detection of an error in an assignment depends on the magnitude and sign of the error. Positive and negative tags are used to model the sign of the errors. However, the magnitude issue results in a dilemma – if we add magnitude as a parameter to the tags, not only are we vastly increasing the number of possible tags, we are also tying ourselves to particular design errors which may or may not occur.

Our approach is to make the assumption that the error will be of the right magnitude(s) to propagate through modules; an optimistic assumption. As mentioned above, positive and negative tags model the sign of error in relation to variable values. (Note that in the binary variable case, the sign determines the value, since Δ is equivalent to $\overline{D}$ which corresponds to a 1 in the faulty circuit and 0 in the fault-free circuit. Similarly for $-\Delta$.)

In our Δ-model for tags, we will assume that the tag introduced in an assignment (i.e., the *change* in the value of the variable on the left hand side of the assignment) corresponds to an error of appropriate magnitude such that it will propagate through any module provided (a) it is of the appropriate sign and (b) the other inputs to the module are non-controlling (i.e., do not block propagation). Coverage is computed based on whether a tag is propagated through modules to a circuit output under the above rules.

C. Functional Simulation and Statement Coverage

Tag simulation is carried out on a given circuit description in exactly the same manner as functional simulation. A functional simulator such as the VERILOG simulator executes initialization statements, and propagates values from the inputs of a circuit to the outputs. It keep tracks of time, causing the changed values to appear at specified times in the future. Future changes are typically stored in a time-ordered event queue. When the simulator has no further statements to execute at a given time instant, it finds the next time-ordered event from the event queue, updates time to that of the event, and executes the event. This simulation loop continues until there are no more events to be simulated or the user terminates the simulation [17].

Some functional simulators have features that allow the user to determine the execution counts of every module/statement in the HDL description, when the description is simulated with a functional vector set. This allows the user to detect possibly unreachable statements and redundant conditional clauses. Our tag simulation method augments this coverage as described in the sequel.

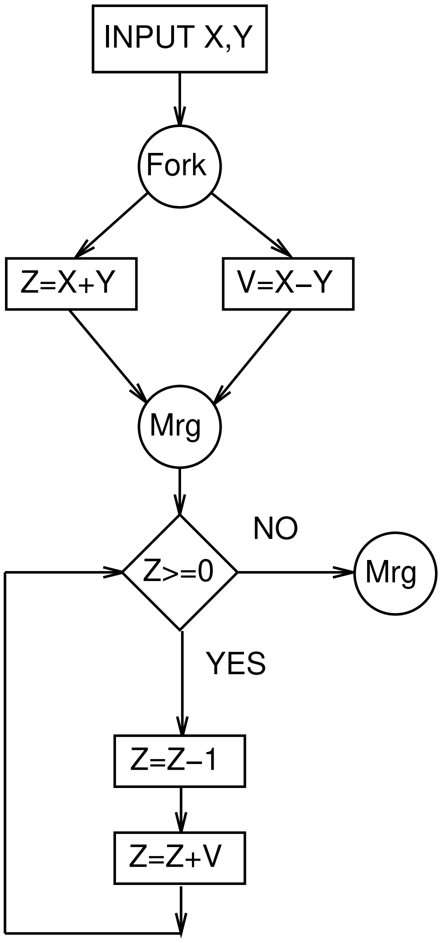

Fig. 2. Control Flow Graph with Fork Nodes

D. Control Flow Graph

The circuit description is preprocessed to extract a control flowgraph of the circuit. The control flowgraph of Fig. 1 does not model concurrency; the HDL model may have interacting concurrent processes. We will add a **fork** node to the control flowgraph that will allow us to model parallelism. An example of a control flowgraph with fork nodes is shown in Fig. 2.

Note that we do not use the control flowgraph to perform simulation. Rather, we compute reachability information for use in tag propagation (cf. Section III-F.3).

E. Tag Injection

We will consider *assignment* statements consisting of arithmetic operations, Boolean word operations and Boolean operations. Positive and negative tags are injected at every assignment statement and attached to the variables or the left hand side of the assignment.

If a variable is a collection of bits, we will treat it as a single entity. However, if the variable is defined as a collection of bits but individual bits are sometimes manipulated, we will treat the individual bits as Boolean variables throughout. For example, if a (an n bit entity) is used only as inputs to arithmetic and Boolean word operators, then we will treat a as a single entity. If a is used as an input to a combinational logic module that operates on any $a_i,\ 0 \leq i \leq n-1$, we will treat each of the a_i's as a Boolean variable and introduce tags on each of the a_i's separately.

An **if** statement has a control condition which is a Boolean vari-

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**

able. If this Boolean variable is computed, i.e., it is not a primary input, it appears as the left hand side of some assignment. The tag that is injected for that assignment models errors in the control flow.[2] No tags are injected for control constructs, though we have to propagate tags based on the control flow.

F. Tag Calculus

Every assignment statement executed during simulation of functional vectors is also "tag simulated" to determine which tags are propagated. Tags attached to variables on the right hand side of the assignment are propagated and attached to the left hand side according to the calculus presented in this section.

In the case of strictly Boolean variables, this calculus is equivalent to the D-calculus [14]. For ease of exposition, we demonstrate the propagation of a single tag during simulation. In actual implementation, several tags can be injected and propagated in parallel.

F.1 Tag Propagation through Logic Gates

For ease of exposition we will first define the calculus for Boolean logic gates. The calculus for a two-input AND gate and an inverter are shown in Table I. The four possible values at each input are 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.

F.2 Tag Propagation through Arithmetic Operators

We describe the tag simulation calculus procedurally for arithmetic and Boolean word 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. We write it as $v(f)+\Delta$ or $v(f)-\Delta$.

We now describe the tag propagation rules for different modules.

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 tag-free.

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.

Bitwise AND : All tags have to be of the same sign for propagation. Given a positive tag on a and an tag-free b, if at least one of the bits in b is not 0, and $v(a) \neq 2^n - 1$, then we will compute $v(f) = v(a)\ \&\ v(b)$ and assign the output $v(f)+\Delta$. For positive tags on both a and b, if $v(a) \neq 2^n - 1$ and $v(b) \neq 2^n - 1$, then we will compute $v(f) = v(a)\ \&\ v(b)$ and assign the output $v(f)+\Delta$. Similarly for negative tags.

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

F.3 Tag Propagation through If Statements

Propagation through **if** statements requires preprocessing the program. We determine the reachability of processes from decisions (**if** statements) in the control flowgraph. Since each process has a single entry and a single exit, all the statements within each process will have the same reachability properties.

Assume without loss of generality that each decision is binary. We determine for each decision the set of all processes that can be reached if the control condition c is true, namely P_c, and the set that can be reached if the control condition is false, namely $P_{\overline{c}}$.

When an if statement is encountered by the simulator, there are two cases:

1. 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.
2. If a tag is attached to the control condition c (which is a Boolean variable), it means that the tag will result in the incorrect branch being taken. Assume a positive tag on c. If the value of c on the applied vector is 1, the tag is not propagated. However, if the value of c is 0, then we tag all the assigned variables in the processes $P_{\overline{c}} - P_c$. Under the tagged condition these assignments will be *missed*, and hence the output variables are tagged. A positive tag is applied if the new value (after assignment) is less than the old value, a negative tag if the new value is greater than the old value, and no tag if both values are equal or if old value is undefined. This assumes that equivalent statements in both clauses have been extracted out of the **if** (placed before the decision).[3]

IV. COMPUTING COVERAGE USING TAG SIMULATION

A. Tag Simulation

We describe a procedure to compute coverage based on tag simulation. Our focus in this paper is on establishing coverage metrics rather than computational efficiency; many tag simulation methods are possible based on different fault simulation algorithms [1].

For each functional vector, or functional vector sequence, for every positive or negative tag injected on each assigned variable, we use the calculus described in the previous section to determine which of the tags are propagated to the model's outputs. We then compute the coverage for the functional vector set as the percentage of tags propagated to the outputs divided by the total number of tags.

Simulation proceeds on the HDL simulation model in an event-driven manner as described in Section III-C. For each functional vector sequence, though the effect of each tag is considered separately, a parallel tag simulation can be performed where the propagation of multiple independent tags is determined.[4] After each assignment

[2] Errors will result in the then branch being taken rather than the else branch, or vice versa.

[3] We could have also assigned tags to variables in $P_c - P_{\overline{c}}$, because under the tagged condition these assignments are wrongly made, but doing so would require multiple trajectory simulation which we deem too expensive.

[4] This is similar to the parallel fault simulation methods used in logic testing. See [1].

statement is evaluated, we determine if the tags on the variables on the right hand side of the statement are propagated to the variable on the left hand side. Tags are also injected for the assignment on the variable on the left hand side. Some tags may not be activated by the functional test sequence if the statement in which the tag occurs is not reached.

After each vector in the sequence is simulated, we inspect the observable outputs of the HDL model to determine the tags propagated to the outputs. In the case of sequential machines, tags may propagate to the output only after multiple vectors have been applied.

B. Coverage on Specific Paths or Subpaths

In many cases a designer may be interested in exercising particular sequences of statements in the HDL model, and observing the effect at the output. Alternately, a designer may be interested in exercising particular modules in a given order and checking the output response. The tag simulation algorithm can be extended to keep track of the subpath(s) that are activated in propagating the tag to the output. Information regarding the activated paths can be passed along with the coverage numbers.

C. Using Coverage Based on Tags

The designer typically learns from the tags that are *not* propagated to the output. For example, a tag on a frequently exercised assignment statement that goes undetected tells the designer that the assigned variable is never used to compute useful output under the applied vector set. If a tag on the condition of an **if** statement is not detected, it means that the statements in the **then** and **else** clauses are interchangeable under the vector set.

All of the above assumes a comprehensive functional vector set. Else, the first step of the designer should be to devise vector sets that detect a larger fraction of errors. This may be difficult to do for models with limited observability. In this case, tag simulation can point out intermediate variables or statements that are blocking error propagation. The designer can place a trace on the variable(s) to improve coverage.

V. ILLUSTRATIVE EXAMPLE

We give a simple example of a design error that may not be detected if functional simulation is terminated using the line or branch coverage metrics, but which will be detected if a vector set with 100% tag coverage is applied. The design error would also be detected using path coverage metrics, but applying vector sets with 100% path coverage is not viable.

The VERILOG example in Figure 3 is a stripped-down version of an an address generation unit that is used for generating addresses to store queue entries. The `ext_mode` signal when 1 indicates that memory external to the chip is available for storing queue entries. When 0, it means only internal memory has to be used. The `queue_full` signal when 1 indicates that the internal queue memory is full. When the internal queue memory is full, future entries are stored in the external memory. It is assumed that if there is external memory, there is enough so that the queue never gets full. If there is no external memory, a particular entry in the internal queue (as given by `entry`) is overwritten. The address generation formula is given in the code. The only error is in the generation of `extern_base`, where the shift should be by 2 and not by 4.

This module is embedded inside a complicated finite state machine and initially `ext_base_reg` and `int_base_reg` were set to the same value. It was tested with the entire system. Upon analysis of the results of testing, it was clear that all lines in the module had been covered and all branches had been covered. However, only paths $\{1, 3, 5\}$, $\{1, 4, 5\}$ and $\{2, 3, 5\}$ were covered. Path $\{2, 4, 5\}$ was not covered and therefore the design error was not detected, though we had 100% line and branch coverage.

However, if we add observability as a measure, our tag simulation algorithm will immediately indicate that the Δ at the line marked `ERROR` is never propagated to the output under all vector sequences and therefore more testing is needed. When a vector that propagates the Δ to the output is applied, the error will be detected.

VI. EXPERIMENTS

In this section we provide evidence of the effectiveness of our proposed coverage metric model through experimental data on several different examples.

Our current simulator is a rudimentary implementation that works on a subset of VERILOG. For each vector, tags are injected one at a time and the vector simulated in an event-driven manner to determine propagation of tags and circuit output values.

Tag simulation speed is slower than commercial VERILOG simulators; however, this speed can be significantly improved by incorporating the tag simulation calculus directly into compiled-code simulation.

A. Line Coverage Versus Tag Coverage

We ran several different examples to compare the line coverage metric against our observability-based tag coverage metric. For each example we computed the tag coverage for a directed functional test set, which is either created by a designer or automatically generated from a gate-level description (e.g., ATPG patterns). We also tag simulated 5 random vector sets of different lengths on each example and computed tag coverage for each random set. For each random vector set, we continued generating vectors from a random seed until 90% line coverage was achieved.

In Table II we summarize the results obtained. The number of tags for each example are indicated in the first column. In the column marked Directed we provide coverage information for the directed functional test set. The number of vectors, tag coverage and line coverage is indicated. In the column marked Random we provide the same information averaged over the 5 random vector sets.

The line coverage numbers are virtually the same for the directed and random vector sets. However, the tag coverage metric is always higher for the better directed vector set. In terms of finding bugs, we find the directed vectors to be superior to the random vectors. The directed vectors test the designs more thoroughly than random vectors (for these examples) and our metric indicates likewise. This demonstrates the usefulness of adding this metric to line and branch coverage metrics.

B. Details of Examples

B.1 Wallace Tree Multiplier

Our first example is a register-transfer level description of a 16-bit Wallace tree multiplier written in VERILOG. It consists of four modules written in about 130 lines of VERILOG code. The model was written in synthesizable VERILOG with a one-to-one correspondence between the model constructs and hardware.

Upon inspecting the unobserved tags for the functional test set, we found the unobserved tags were as follows:

- Some tags were not propagated to the outputs because there were initialization statements for variables followed immediately by assignments to the variables. Thus, the initialization statements are unnecessary, though in general it is good practice to include them.

Ex	#Tags	Directed				Random			
		#Vec	#Observed Tags	Tag Coverage%	Line Coverage%	#Vec	#Observed Tags	Tag Coverage%	Line Coverage%
mult	44	109	38	86%	100%	25	34	77%	100%
pport	33	18	29	87%	92%	100	22	67%	90%
arbiter	60	88	56	93%	100%	5000	34	57%	90%
count	100	3000	68	68%	92%	3000	68	68%	92%
schsm	52	70	44	85%	94%	4300	28	54%	90%

TABLE II
COMPARING COVERAGE METRICS

- Some tags were not propagated because they appeared on a carry out signal of a module that was instantiated in the multiplier array, but was not declared as a primary output.
- Some tags were not propagated because the carry in signal was always zero and meant that certain signals could have arbitrary values without affecting the outputs of the multiplier.

In the case of the multiplier, the stuck-at fault test set was clearly designed to propagate faults to the outputs, and hence it gives a good coverage based on tags also. The random tests also give reasonably good coverage because multipliers are inherently random pattern testable.

B.2 Parallel Port for Embedded Processor

Next, we experimented with a parallel port for an embedded processor. This circuit is connected to the bus of the processor on one side and to external inputs/output wires on the other. The processor can write the direction register in the port to determine which external wires are used as inputs and which are used as outputs. The data written by the processor into the data latch of the port is transmitted to the external output lines and whenever the processor reads the data latch, external inputs are sampled and their values stored in the data latch. It consists of about 170 lines of VERILOG code.

The parallel port is a more interesting example since its description is at a higher level than the register-transfer level. The designer of the parallel port provided the directed vector set which thoroughly exercised the port, except for the reset logic. The tags unobserved by the functional vector set are the tags injected in the reset logic. We were able to design a functional vector set that also exercised the reset logic and this set resulted in 100% tag coverage.

B.3 Other examples

The other examples in the table are from an Asynchronous Transfer Mode segmentation and reassembly circuit. Example **arbiter** is a bus arbiter that arbitrates access to memory from several on-chip modules. The example **count** counts upto 87 and then to 2784 clock pulses, producing a pulse at the output whenever the count reaches 87 or 2784. Example **schsm** is a small finite-state machine that controls requests to the arbiter to access shared memory. All circuit descriptions are in the synthesizable RTL subset of Verilog.

For each example, directed test sets exercised each module thoroughly, especially for the finite-state machines where almost all branches of the machine were exercised. Several bugs were found during the design stage by the directed tests. However, the random tests, terminated after 90% line coverage was obtained did not exercise the designs thoroughly, and found few bugs. This was reflected in the poorer tag coverage for the the random vector set.

For the example **count**, the random set and functional set were identical because the only input to the module is a clock signal.

It is noteworthy that for many examples high line coverage does not necessarily imply high tag coverage. This is especially true for the example **schsm**. More directed tests were designed to improve the tag coverage to 100% for this example. Though no additional bugs were discovered, the new vector set covered all the paths of the circuit.

VII. ONGOING WORK

Several extensions need to be made to the simulation calculus. Concurrency and looping constructs used in popular HDL's need to be handled. Tristate signals are another important extension; we need to precisely define what it means for a tag to be observed at a tristate output.

Once the simulation calculus is augmented, efficient compiled-code simulation methods popularly used in HDL simulators can be extended to propagate tags in parallel. Efficient simulation will allow coverage-directed test generation by the designer under the new metric, thereby enabling a greater degree of design verification.

ACKNOWLEDGMENTS

S. Devadas was supported in part by a grant from Siemens Corporation, Munich, and and in part by a grant from Schlumberger Foundation. We thank Curt Widdoes and Steve Tjiang for criticism on a earlier draft of this paper.

REFERENCES

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

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

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

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

[5] E. M. Clarke and O. Grumberg. Research on Automatic Verification of Finite-State Concurrent Systems. *Annual Reviews of Computer Science*, 2:269–290, 1987.

[6] K. A. Foster. Error-Sensitive Test Case Analysis. *IEEE Transactions on Software Engineering*, SE-8(3):258–264, May 1980.

[7] I. Gertner and R. P. Kurshan. Logical Analysis of Digital Circuits. In M. Barbacci and C. J. Koomen, editors, *Computer Hardware Descriptions Languages and Their Applications*, pages 47–67, New York, 1987. Elsevier.

[8] M. Gordon. Hardware Verification by Formal Proof. In *Technical Report No. 74, Computer Laboratory, University of Cambridge, Cambridge, England*, August 1985.

```
module get_address (ext_mode, queue_full,
 int_base_reg, ext_base_reg, ext_index_reg,
 entry, phy_address);

 input ext_mode; // 1 if ext. mem. available
 input queue_full; // internal queue full
                   // new messages to ext. mem.

 input [15:0] int_base_reg; // int. base addr.
 input [15:0] ext_base_reg; // ext. base addr.
 input [15:0] ext_index_reg; // ext. base index
 input [15:0] entry;  // number of entry

 output [15:0] phy_address; // phys. address
 reg [15:0] phy_address;

 reg [15:0] extern_base; // ext. phys. addr.
 reg [15:0] internal_base; // int. phys. addr.
 reg [15:0] queue_ptr; // Ptr to entry queue

 initial
   begin
     internal_base = int_base_reg;
   end

 always @ (ext_mode or queue_full)
   begin

     extern_base = ext_base_reg;

     if (ext_mode == 1'b0) begin
       // Call this branch 1
       internal_base = int_base_reg;
     end
     else begin
       // Call this branch 2
       extern_base = ext_base_reg +
         ext_index_reg << 4;    // ERROR !!
     end

     if (queue_full == 1'b0) begin
       // Call this branch 3
       queue_ptr = internal_base + entry << 8;
     end
     else begin
       // Call this branch 4
       queue_ptr = extern_base + entry << 8;
     end

     // Call this edge 5
     phy_address = queue_ptr + 4;
   end

endmodule //of get_address
```

Fig. 3. Example illustrating difference between line and tag coverage

[9] W. E. Howden. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering*, SE-2(3):208–215, September 1976.

[10] W. Hunt. The Mechanical Verification of a Microprocessor Design. In D. Borrione, editor, *From HDL descriptions to guaranteed correct circuit designs*, pages 89–129, Amsterdam, 1986. North Holland.

[11] 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.

[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] K. K. Sabnani, A. M. Lapone, and M. U. Uyar. An Algorithmic Procedure for Checking Safety Properties of Protocols. *IEEE Transactions on Communications*, 37(9):940–948, September 1989.

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

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