Observability Analysis on HDL Descriptions for Effective Functional Validation

Tai-Ying Jiang, Chien-Nan Jimmy Liu, and Jing-Yang Jou, *Fellow, IEEE*

***Abstract*—Simulation-based *functional validation* is still one of the primary approaches for verifying designs described in hardware description languages. Traditional code coverage metrics do not address the observability issue and may overestimate the extent of *functional validation*. Observability-based code coverage metric (OCCOM) is the first code coverage metric considering the essential observability issue. However, *tags* can only be observed or unobserved, providing only two levels of measurement (i.e., 1 and 0). Errors with lower opportunities to be observed may still be judged as observable, thus misleading the verification results. Therefore, instead of extending tag coverage, we develop a probabilistic observability measure and its efficient computation algorithm. Besides being used as a new OCCOM, our new measure can point out hard-to-observe points for inserting *assertions* to prevent bugs from hiding behind these points. Experimental results show that the detection of the injected errors and the degree of our observability measure are strongly related. The results also show that our fine-grained observability measure is less likely to overestimate the extent of validation with reasonable computation time.**

***Index Terms*—Code coverage metric, hardware description language (HDL), observability analysis.**

I. Introduction

HARDWARE description languages (HDLs) are widely applied in modeling the behaviors of digital circuits. Researchers have extensively studied how to verify HDL descriptions for the last decade. Formal verification methods for *language containment* and *property checking* have made progress on this verification problem. However, there is still no indication that these formal techniques can verify all kinds of hardware designs completely. Simulation-based *functional validation* still remains the primary approach for verifying HDL descriptions.

In *functional validation*, the simulation values of some signals of interest must be compared with their expected values to determine the consistency with the specification. In this paper, the term observation points (OPs) is used to describe these signals because they act like observation windows to uncover bugs. Designers often select OPs according to their understanding of the specification and the availability of the expected values. However, erroneous effects caused by bugs are not always propagated to the assigned OPs. They may be *masked* while propagating to OPs. This situation prevents bug finding. Even worse, bugs may remain undiscovered through the manufacturing process if validation is not accurately gauged.

Most code coverage metrics, such as *statement coverage*, *branch coverage*, and *path coverage* metrics, only consider whether their concerned code structures are exercised [2]. They do not explicitly check whether erroneous effects caused by bugs propagate to OPs [3]. Bugs may still remain undetected even if they have been activated under these coverage metrics in the verification scenarios. Therefore, the aforementioned coverage metrics may overestimate the extent of the validation. The observability-based code coverage metric (OCCOM) is the first code coverage metric that considers the essential observability issue [3]–[5]. In their approach, the propagation of special *tags* that are attached to internal signals is simulated to predict the actual propagation of erroneous effects of bugs. The observed *tags* percentage is the OCCOM coverage. However, *tags* can only be observed or unobserved, providing only two levels of measurement, i.e., 1 and 0. Errors that have lower observation opportunities may still be judged as observable, thus giving misleading verification results. Moreover, if multiple errors exist in the design under validation (DUV), the *single tag model* in tag simulation may not precisely determine whether the tag can be observed.

Therefore, this paper attempts to develop a different observability measure for HDL descriptions that can provide intermediate values between 1 (observed) and 0 (unobserved) instead of merely extending tags. This could reduce the likelihood of misestimating real observability. In the software testing arena, Voas proposed a concept called *sensitivity analysis* [14], [15]. The *propagation probability* (PP) proposed in those papers may be a good observability measure for software programs written in C, C++, or even HDLs. Using a concept similar to the PP in [14], we define a probabilistic observability measure for HDL. However, the proposed statistics-based approach for calculating PP is quite time consuming. Thus, it may not be suitable for the HDL models of commercial products.

Since the computation time to obtain accurate PP in [14] is too long, we also develop a topology-based observability computation algorithm that can quickly produce results from the simulation dump file and provide a closed lower bound of observability measures. Although our algorithm uses some

Manuscript received December 13, 2005; revised May 5, 2006 and August 6, 2006. This work was supported by the National Science Council, Taiwan, R.O.C., under Grant NSC94-2220-E-009-041. This paper was recommended by Associate Editor R. F. Damianu.

T.-Y. Jiang is with the Department of Electronics Engineering, National Chiao Tung University, Hsinchu 300, Taiwan, R.O.C. (e-mail: giani@eda.ee.nctu.edu.tw).

C.-N. J. Liu is with the Department of Electrical Engineering, National Central University, Jhongli City 320, Taiwan, R.O.C. (e-mail: jimmy@ee.nctu.edu.tw).

J.-Y. Jou is with the Department of Electronics Engineering, National Chiao Tung University, Hsinchu 300, Taiwan, R.O.C., and also with the National Chip Implementation Center, National Applied Research Laboratories, Hsinchu 300, Taiwan, R.O.C. (e-mail: jyjou@faculty.nctu.edu.tw).

Digital Object Identifier 10.1109/TCAD.2007.891366

heuristics, its estimated observability is still very close to statistics-based estimations but with a much shorter computation time.

With the proposed observability measure, there are some possible applications.

1) A new OCCOM. In our new OCCOM, a statement is considered as covered if it is first exercised and the observability of the statement's output variable is high enough. This is similar to the well-known fault simulation that requires fault activation and propagation.
2) Indicating hard-to-observe points. If some signals are less likely to be observed, bugs may hide behind these points and become very difficult to reveal via limited OPs. By using our observability analysis, designers can designate candidates for assertion insertion to prevent potential bugs from hiding. This can increase the verification efficiency, too.

The remainder of this paper is organized as follows: Section II describes related works. Section III introduces the motivation and the definition of our observability measures. Section IV presents our observability computation algorithm and related theorems. Experimental results are given in Section V. Section VI discusses conclusion and potential directions for future research.

II. Related Works

A. Testability Analysis in Manufacturing Test

Manufacturing test is a process of checking that integrated circuits are manufactured correctly. The well-known *stuck-at-fault* model is often used to capture manufacturing defects [17]. Based on the fault models, test vectors are generated and applied to manufactured integrated circuits. *Fault coverage* analysis is then conducted to judge whether the integrated circuits are well tested or not. *Testability* here is used to guide *test pattern generation* or as a direct substitution of a fault coverage report. Observability is often defined as the difficulty of observing erroneous effects caused by some bit-level *stuck-at-faults* [18], [19]. This is quite different from our word-level observability measures for HDL descriptions without underlying design error models.

Recent research abstracted defects as higher level logical fault models [20]–[24]. However, the correspondence between logical fault models and HDL design errors is still weak in two aspects: 1) an erroneous statement may be synthesized into hundreds of erroneous gates and erroneous wires and 2) there are almost no credible design error models for HDL descriptions. Thus, logical fault models hardly link to HDL design errors. *Testability* for these logical fault models consequently differs from our observability measure.

Some RTL *testability* analysis research exploits the idea of *hierarchical testing* with a precomputed test vector set [25]–[27]. These studies define *testability* as the difficulty of generating input patterns for RTL circuits or instructions for processors to test internal RTL modules. They are different from our observability measures.

B. Sensitivity Analysis in Software Testing

Voas first proposed sensitivity analysis for software programs [14]–[16]. He estimates three probabilities for locating hard-to-detect bugs in a software program. The PP of a variable is the estimated probability that a variable's erroneous values caused by some bugs are observed in the program outputs. PP is a good observability measure for software programs, even for HDL programs. The PP of a variable v in the program is estimated by a statistics-based approach, repeatedly infecting the data state of v (injecting erroneous values on variables in memory) and simulating the program. The ratio of the number of program failures to the total number of experiments is the PP of v. This approach obviously requires lots of simulation and is very time consuming. A faster approach is desirable, especially for the HDL models of commercial products.

C. OCCOM

OCCOM is the pioneer that addresses the essential observability issue [3]. Dump-file-based OCCOM computation facilitates integration with commercial simulators and thus accelerates the analysis process [4], [5]. Tag-based observability measures also assess the extent of validation for C programs in recent works [6], [7]. Test pattern generation approaches for OCCOM make the entire work more practical [8]–[10]. In [10], the authors define *detectability* of a tag that consists of *controllability* and *observability* of a tag to guide test vector generation for tag coverage.

All of the works just mentioned adopt tag-based observability measures. Two special *tags*, i.e., Δ and $-\Delta$, are attached to each signal to simulate potential increasing and decreasing value changes caused by bugs on the signal, respectively. The propagation of *tags* is determined by *tag propagation rules* [3]–[5]. The OCCOM coverage is the percentage of *tags* observed at the OPs.

In the tag-based approaches, *tags* can either be observed (1) or unobserved (0), providing only two levels of measurement. Since some errors can only be observed under specific conditions, they may not always be observed at the OPs. However, because they can propagate to the OPs in some cases, they will be judged as observed in tag-based approaches. If the specific conditions in which those errors can be observed rarely happen, treating them as observable in all cases may give misleading verification results. Therefore, if the granularity of observability measures can be improved to provide intermediate values between 1 (observed) and 0 (unobserved), the possibility of having misleading verification results might be greatly reduced. Moreover, if multiple errors exist in the DUV, the *single tag model* in tag simulation may not precisely determine whether the tag can be observed.

III. Probabilistic Observability Measure

A. Motivation

A simple HDL example shown in Fig. 1(a) demonstrates what *error masking* is. We will also show how *tags* can help predict the propagation of bugs. Applying the input stimulus shown

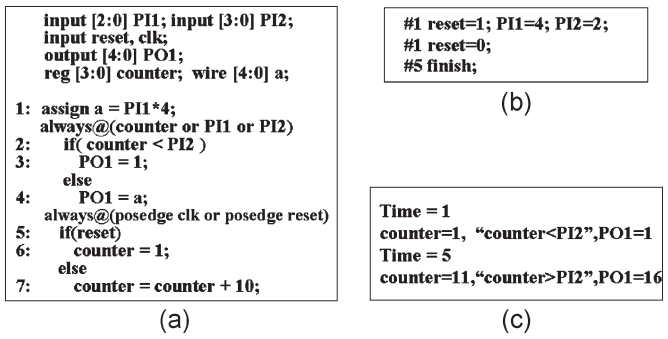

Fig. 1. HDL example. (a) HDL code. (b) Input stimulus. (c) Simulation result.

in Fig. 1(b) to simulate the HDL code fragment in Fig. 1(a), we can obtain the simulation results shown in Fig. 1(c). According to the results, both *statement coverage* and *branch coverage* metrics achieve 100%. The quality of the input stimulus is sufficient if the aforementioned code coverage metrics can be trusted.

However, if statement 7 is changed to "counter $=$ counter $+$ 2," the input stimulus cannot reveal this bug. The design bug causes an incorrect value 3 (different from the correct value 11) on the signal *counter* at $t = 5$. However, this incorrect value is *masked* by statement 2 "if(counter $<$ PI2)" since the evaluation results of the correct and incorrect values are the same; i.e., 11 and 3 are both bigger than 2. Therefore, the observability, or the *error masking*, issue is essential to be considered while gauging the extent of the validation.

Applying OCCOM [3]–[5] to gauge the extent of the validation can also obtain 100% OCCOM coverage in this case. The *tag propagation rule* for "$<$" in [4] indicates that tag Δ and tag $-\Delta$ injected on the signal *counter* can pass through statement 2 "if(counter $<$ PI2)" and appear at PO1 at $t = 1$ and $t = 5$, respectively. Both tags injected on the signal a also pass through the operation "$=$" at $t = 5$. However, in this example, the incorrect value 3 (which can be regard as $11 - \Delta$) does not propagate to PO1 as the *tags* predict.

Tag propagation rules assume that a decreasing value change can "often" be so dramatic that the evaluation result of "counter $<$ PI2" changes from FALSE to TRUE [4]. However, the actual value change caused by design errors is not always dramatic enough to change the result of "counter $<$ PI2," as demonstrated in this example. If the likelihood that erroneous values change the result of "counter $<$ PI2," i.e., the likelihood that erroneous effects can propagate through the operation "counter $<$ PI2" can be estimated, we can obtain closer estimations for the real observability of those signals.

B. Probabilistic Observability Measure

Despite completion of a successful simulation in which the simulated values of all the OPs are consistent with the correct values, it is still possible that some incorrect values existed at some time instances but remain hidden due to the *error masking*. Assuming that the simulation values of all the OPs are consistent with the expected values, the goal of our work is to analyze which signals will most likely have incorrect values hiding at which time instances. This prevents overestimating validation completeness and points out hard-to-observe points. In the following descriptions, the *error masking* model and our observability will be introduced.

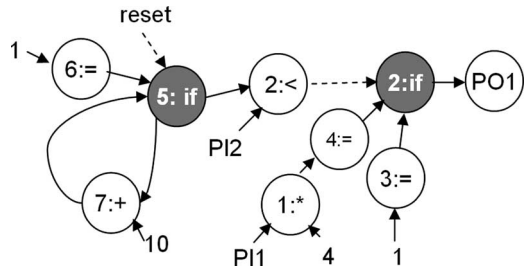

Fig. 2. CDFG of the HDL code in Fig. 1.

The DUV is modeled as a modified control/data flow graph (CDFG) $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, where $\boldsymbol{V}$ is the set of vertices and $\boldsymbol{E}$ is the set of edges connecting vertices. In order to explain the CDFG more clearly, the CDFG appearing in Fig. 2 is used as an example of the HDL code shown in Fig. 1. Let v be a vertex in $\boldsymbol{V}$. Each vertex v corresponds to an operation in the HDL code. Function f_v and variable y_v are also associated with vertex v. Function f_v is the function of the operation that v corresponds to. Variable y_v is the output variable of f_v or the *left-hand variable* of the operation. For example, vertex "$1:^*$" in Fig. 2 corresponds to the operation "$a = \text{PI1} * 4$" at line 1 in the HDL code. Function $f_{1:^*}$ is multiplication "*," and $y_{1:^*}$ is signal a. Vertex "$2:$ if" corresponds to the operation "if$(\ldots)\ldots$else$\ldots$" in lines 2–4 of the HDL code, and its functionality is quite similar to a multiplexer. Vertex PO1 is a special vertex representing the primary output PO1 in the circuit. An edge $(v, u) \in \boldsymbol{E}$ indicates that the input of vertex u is data dependent on the output of v. As shown in Fig. 2, an edge $(1:^*, 4:=)$ exists since the operation "$4:=$" takes the output of vertex "$1:^*$" as its input. The *fan-out* of v is a set of vertices u such that there is an edge from v to u. Similarly, the *fan-in* of v is a set of vertices k such that there is an edge from k to v. A path from vertex u to vertex u' is a sequence $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ of vertices such that $u = v_0$, $u' = v_k$ and $(v_{i-1}, v_i) \in \boldsymbol{E}$.

If a single incorrect value w ever existed on the output variable of vertex vy_v at time instance $t = t_i$ in the DUV during simulation, this incorrect value w should cause no incorrect behaviors at any OPs at all positive edges of clock.[1] If not, the simulation phase is not successful. More specifically, the simulated value of an OP_j at an arbitrary positive edge of clock $t = c_k$ must be the same as the correct value. The incorrect value w must be *masked* by some vertices on the paths from vertex v at $t = t_i$ (denoted as $v@t = t_i$) to OP_j at $t = c_k$ (denoted as $\text{OP}_j@t = c_k$). In the following descriptions, "v at $t = t_i$" and "v in time frame $t = t_i$" will be used in turn. A formal description of *error masking* is given as

$$f_{v@t=t_i \rightarrow \text{OP}_j@t=c_k}(w) = CV(\text{OP}_j@t = c_k) \qquad (1)$$

[1]We assume that the simulation values of all the observation points are compared with the correct values only on the positive edges of clock signal. If the design under validation is a falling-edge-triggered or double-edge-triggered design, the assumption along with the modeling and the computation can easily be changed to fit to it.

where $f_{v@t=t_i \to \mathrm{OP}_j@t=c_k}$ is the function of the paths from v in time frame $t=t_i$ to OP_j in time frame $t=c_k$, and $CV(\mathrm{OP}_j@t=c_k)$ is the correct value of OP_j at $t=c_k$.

If there are m total OPs $\{\mathrm{OP}_1, \mathrm{OP}_2, \ldots, \mathrm{OP}_m\}$ and n clock cycles in the simulation phase, the incorrect value w must be *masked* on its way to all the OPs in all time frames such that it is not uncovered during the entire simulation process. More formally, for each OP_j in each time frame $t=c_k$, the function of the paths from vertex v in time frame $t=t_i$ that go to OP_j at $t=c_k$ must generate the correct value of OP_j at $t=c_k$ with this incorrect value w, i.e.,

$$\bigcap_{j=1}^{m}\bigcap_{k=0}^{n} f_{v@t=t_i \to \mathrm{OP}_j@t=c_k}(w) = CV(\mathrm{OP}_j@t=c_k). \quad (2)$$

The set of all possible values of vertex v's output that can satisfy (2) is defined as the *masked value set* (MVS) of vertex v at time instance $t=t_i$ ($\mathrm{MVS}(v@t=t_i)$). A more formal definition is given in (3). Each element in $\mathrm{MVS}(v@t=t_i)$ retains the correct values of all the OPs at all positive edges during simulation, i.e.,

$$\mathrm{MVS}(v@t=t_i) = \left\{ x \middle| \bigcap_{j=1}^{m}\bigcap_{k=0}^{n} f_{v@t=t_i \to \mathrm{OP}_j@t=c_k}(x) = CV(\mathrm{OP}_j@t=c_k) \right\}. \quad (3)$$

The correct value of the output of vertex v at $t=t_i$ is in $\mathrm{MVS}(v@t=t_i)$, and this can justify the existence of $\mathrm{MVS}(v@t=t_i)$. If $\mathrm{MVS}(v@t=t_i)$ has only one element, this element must be the correct value, and no *error masking* can occur. On the other hand, if the set contains many elements, there will be many elements other than the correct values[2] in the set. An incorrect value caused by some bugs may very possibly be one of these elements and thus be *masked*. (The incorrect value can also be outside the set such that it is revealed.) The more elements in $\mathrm{MVS}(v@t=t_i)$, the more likely the simulation value of v is one of these *masked* incorrect values. Hence, the likelihood of error masking (LOEM) of v at $t=t_i$ is defined as (4). Its complement is the observability measure of v at $t=t_i$, as described in (5), i.e.,

$$\mathrm{LOEM}(v@t=t_i) = \frac{|\mathrm{MVS}(v@t=t_i)|-1}{2^n-1} \quad (4)$$

$$\mathrm{Observability}(v@t=t_i) = 1 - \frac{|\mathrm{MVS}(v@t=t_i)|-1}{2^n-1}. \quad (5)$$

IV. Observability Computation Algorithm

Our observability computation algorithm is a topology-based analysis with *time frame expansion* to handle the sequential behavior of the DUV. While calculating the observability of the output variable of vertex v in time frame $t=t_i$, the algorithm will consider each sensitized path from v in time frame $t=t_i$ that goes to any OP in each time frame. The path-oriented computation scheme is defined in (6), which can be transformed from (3), i.e.,

$$\mathrm{MVS}(v@t=t_i) = \bigcap_{j=1}^{m}\bigcap_{k=0}^{n} \left\{ x \middle| f_{v@t=t_i \to \mathrm{OP}_j@t=c_k}(x) = CV(\mathrm{OP}_j@t=c_k) \right\}. \quad (6)$$

The set $\{x | f_{v@t=t_i \to \mathrm{OP}_j@t=c_k}(x) = CV(\mathrm{OP}_j@t=c_k)\}$ is defined as the MVS of vertex v at time instance $t=t_i$ with respect to OP_j at $t=c_k$ (denoted as $\mathrm{MVS}(v@t=t_i)_{\mathrm{OP}_j@t=c^k}$). An element of the set other than the correct value can be regarded as an incorrect value that is *masked* by some vertices on the paths from v at $t=t_i$ to OP_j at $t=c_k$, thus keeping the correct value of OP_j at $t=c_k$.

According to (6), if it is possible to derive $\mathrm{MVS}(v@t=t_i)_{\mathrm{OP}_j@t=c^k}$ for each OP_j at each time frame $t=c_k$, then intersecting these sets produces $\mathrm{MVS}(v@t=t_i)$. If there is exactly one path from v at $t=t_i$ to an OP_j at $t=c_k$, an induction-based computation approach is proposed to compute exact $\mathrm{MVS}(v@t=t_i)_{\mathrm{OP}_j@t=c^k}$, which is introduced in Sections IV-A and B. If there are multiple paths from v at $t=t_i$ to OP_j at $t=c_k$, i.e., *reconvergent paths*, a quick estimation approach that guarantees lower bound observability estimations will be applied, which is introduced in Section IV-C. Section IV-E introduces the entire algorithm incorporating both of them, and Section IV-D discusses time-saving strategies.

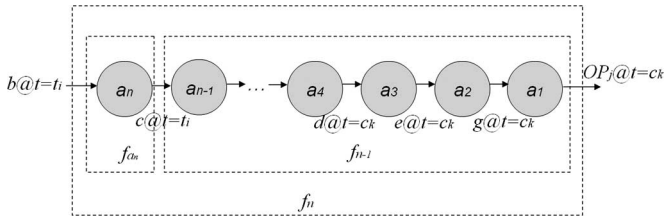

Fig. 3. Path from $b@t=t_i$ to $\mathrm{OP}_j@t=c_k$.

A. MVS Computation for Single Path

Assume that there is a sensitized path P from a vertex b at time instance $t=t_i$ to an OP_j at a positive edge of clock $t=c_k$. As an example, one such path $\langle b@t=t_i, a_n, a_{n-1}, \ldots, a_2, a_1, \mathrm{OP}_j@t=c_k\rangle$ is shown in Fig. 3 and will be used in the following explanations. For the case of a *single path*, we develop an algorithm to compute $\mathrm{MVS}(b@t=t_i)_{\mathrm{OP}_j@t=c^k}$, as shown in the pseudocode in Fig. 4.

For each OP at each positive clock edge, the algorithm will recursively call subroutine *MVS_for_vertex* to perform MVS computation and use a depth first search strategy for backward traversals. The input of the subroutine is a previously computed set of integers (PreviousMVS), the currently traversed vertex v, and the current time frame t_i. If the currently traversed vertex v is a normal vertex, all the *fan-in* vertices of vertex v will be traversed (line 7). However, if vertex v is a *control*

[2]Although the elements other than incorrect value in the MVS of v at $t=t_i$ are not all masked incorrect values, some of them may be don't care values of v at $t=t_i$. However, the identification of *don't care* values requires formal proofs or probably many more simulations. Thus, for safety, we here consider these values other than the correct ones as masked incorrect values.

```
MVS_Computation_for_Single_Path
1:  for each positive edge of clock t=c_k
2:    for each observation point OP_j
3:      InitialMVS={CV(OP_j@t=c_k)}
4:      Find the fanin vertex a_1 of OP_j at t=c_k
5:      MVS_for_vertex(InitialMVS, a_1, c_k)

MVS_for_Vertex(PreviousMVS, vertex v, time t_j)
1:  if MVS(v@t=t_j) = =∅
2:    MVS(v@t=t_j) = PreviousMVS
3:  else
4:    MVS(v@t=t_j) = MVS(v@t=t_j)⋂PreviousMVS
5:  if v is a control vertex
6:    Mark each fanin vertex on the untaken branch as
      "inactive"
7:  for each "active" fanin vertex u of v
8:    if edge (u,v) across time frame
9:      t_h = t_j – clock_period
10:   if t_h <0
11:     return
12:   Compute CurrentMVS, which is
        {x | f_v(x) ∈ PreviousMVS}
13:   MVS_for_Vertex(CurrentMVS, u, t_h)
```

Fig. 4. Pseudocode of MVS computation for a single path.

vertex, the *fan-in* vertices on the untaken branch(es) will be marked as "*inactive*" and will not be traversed (line 5). The key step of this algorithm (line 12) is computing the set of all the u's output values (CurrentMVS) that can make the function of vf_v generate an output value that is in PreviousMVS. Then, the newly computed set CurrentMVS will become the input PreviousMVS of subroutine *MVS_for_vertex* and will be recorded on vertex u along with time information after the subroutine is called again. Section IV-B will introduce how to compute CurrentMVS based on PreviousMVS (line 12). The following explains how this algorithm can derive $\text{MVS}(b@t = t_i)_{\text{OP}_j@t=c^k}$ in the case of a *single path* from b at $t = t_i$ to OP_j at $t = c_k$.

Theorem 1: As shown in Fig. 3, function f_n is the composite function of the vertices from a_1 to a_n and comprises $f_{a_n}(x)$ and f_{n-1}. For an arbitrary value x on the output of vertex b at $t = t_i$, x is in $\text{MVS}(b@t = t_i)_{\text{OP}_j@t=c^k}$ if and only if $f_{a^n}(x)$ is in $\text{MVS}(c@t = t_i)_{\text{OP}_j@t=c^k}$, which can be represented as

$$\text{MVS}(b@t = t_i)_{\text{OP}_j@t=c_k} = \left\{x | f_{a_n}(x) \in \text{MVS}(c@t = t_i)_{\text{OP}_j@t=c_k}\right\}. \quad (7)$$

Proof:

Claim 1:

$$\text{MVS}(b@t = t_i)_{\text{OP}_j@t=c_k} \supseteq \left\{x | f_{a_n}(x) \in \text{MVS}(c@t = t_i)_{\text{OP}_j@t=c_k}\right\}.$$

For each value x in $\{x \mid f_{a_n}(x) \in \text{MVS}(c@t = t_i)_{\text{OP}_j@t=c^k}\}$, x must satisfy $f_{n-1}(f_{a_n}(x)) = CV(\text{OP}_j@t = c_k)$ and, thus, also satisfy $f_n(x) = CV(\text{OP}_j@t = c_k)$. That is, x is in $\text{MVS}(b@t = t_i)_{\text{OP}_j@t=c^k}$. This proves that $\text{MVS}(b@t = t_i)_{\text{OP}_j@t=c_k} \supseteq \{x | f_{a_n}(x) \in \text{MVS}(c@t = t_i)_{\text{OP}_j@t=c_k}\}$.

Claim 2:

$$\text{MVS}(b@t = t_i)_{\text{OP}_j@t=c_k} \subseteq \left\{x | f_{a_n}(x) \in \text{MVS}(c@t = t_i)_{\text{OP}_j@t=c_k}\right\}.$$

By way of contradiction, first, assume that there is a value x that is in $\text{MVS}(b@t = t_i)_{\text{OP}_j@t=c^k}$, but $f_{a_n}(x)$ is not in $\text{MVS}(c@t = t_i)_{\text{OP}_j@t=c^k}$. Since x is in $\text{MVS}(b@t = t_i)_{\text{OP}_j@t=c^k}$, then $f_n(x) = CV(\text{OP}_j@t = c_k)$ implies that $f_{n-1}(f_{a_n}(x)) = CV(\text{OP}_j@t = c_k)$. This means that $f_{a_n}(x)$ is in $\text{MVS}(c@t = t_i)_{\text{OP}_j@t=c^k}$. This is a contradiction!

From Claims 1 and 2, it is proven that

$$\text{MVS}(b@t = t_i)_{\text{OP}_j@t=c_k} = \left\{x | f_{a_n}(x) \in \text{MVS}(c@t = t_i)_{\text{OP}_j@t=c_k}\right\}.$$

When subroutine *MVS_for_vertex* is called for the first time, the computed CurrentCVS $\{x | f_{a1}(x) \in \{CV(\text{OP}_j@t = c_k)\}\}$ is actually $\text{MVS}(g@t = c_k)_{\text{OP}_j@t=c^k}$ according to the definition. When the subroutine is called for the second time, the computed CurrentMVS $\{x | f_{a2}(x) \in \text{MVS}(g@t = c_k)_{\text{OP}_j@t=c^k}\}$ should be $\text{MVS}(e@t = c_k)_{\text{OP}_j@t=c^k}$ according to Theorem 1. Similarly, the computed CurrentMVS $\{x | f_{a3}(x) \in \text{MVS}(e@t = c_k)_{\text{OP}_j@t=c^k}\}$ is $\text{MVS}(d@t = c_k)_{\text{OP}_j@t=c^k}$ when the subroutine is called for the third time. Therefore, when the computation reaches vertex a_n, the computed CurrentMVS $\{x | f_{a_n}(x) \in \text{MVS}(c@t = t_i)_{\text{OP}_j@t=c^k}\}$ is the MVS of b at $t = t_i$ with respect to OP_j at $t = c_k$.

From the aforementioned discussion, it shows that a CurrentMVS set is an MVS of a traversed vertex with respect to OP_j at $t = c_k$. These MVSs will be intersected with the other MVSs of the same vertex with respect to other OPs at different time instances, according to (6). After all the OPs at all the positive clock edges have been applied, the MVS of each traversed vertex in a time frame will be computed and recorded for the later observability calculation.

B. MVS Formula for Operations

Given a previously computed MVS set (PreviousMVS), a vertex v, and one of v's *fan-in* vertex u, computing CurrentMVS is to find the set of all the values at u's output y_u that make the function of vf_v generate an output value that is in PreviousMVS. First, consider a particular value p in PreviousMVS and find the set of all the values that make f_v generate p at v's output y_v. If such a set can be derived for each particular value p in PreviousMVS, then the union of these sets derives CurrentMVS. The set of all such values for a particular p is denoted as Sub_CurrentMVS_p.

For most *unary* and *binary* operations, inversing f_v can easily derive Sub_CurrentMVS_p. Take the operation "$y_v = -y_u$" as an example. If $p = -2$, inversing the minus operation "$-$" produces $y_u = 2$. Take the operation "$y_v = y_u + b_1$" as another example. If $p = 8$ and $b_1 = 3$, inversing "+," i.e., $y_u = 8 - 3$, shows that y_u is equal to 5. The integer b_1 is the simulated value of the operand other than the output of uy_u, and it is recorded in the dump file. The formula to compute Sub_CurrentMVS_p is summarized in the third column of Table I. The column "condition" shows the necessary conditions for the result of Sub_CurrentMVS_p. If the conditions are not met, in most of the

TABLE I
FORMULAS OF Sub_CurrentMVS$_p$

Operation	Condition	*Sub_CurrentMVS$_p$*
$y_v = y_u$	-	$\{p\}$
$y_v = \sim y_u$	-	$\{2^w - 1 - p\}$
$y_v = -y_u$	-	$\{2^w - p\}$
$y_v = y_u\,[i:j]$	-	$\bigcup_{k=0}^{2^{w-i-1}-1}\{[p\cdot 2^j + k\cdot 2^{i+1} \sim p\cdot 2^j + k\cdot 2^{i+1} + 2^j - 1]\}$
$y_v = y_u\,[i]$	-	$\bigcup_{k=0}^{2^{w-i-1}-1}\{[p\cdot 2^i + k\cdot 2^{i+1} \sim p\cdot 2^i + k\cdot 2^{i+1} + 2^i - 1]\}$
$y_v = y_u + b_1$	-	$\{p-b_1\}$
$y_v = y_u - b_1$	-	$\{p+b_1\}$
$y_v = b_1 - y_u$	-	$\{b_1-p\}$
$y_v = y_u * 0$	p==0	$\{[0 \sim 2^w - 1]\}$
$y_v = y_u * b_1\ (b_1>0)$	$p\%b_1==0$	$\{p/b_1\}$
$y_v = y_u \% b_1$	$p < b_1$	$\bigcup_{k=0}^{\lfloor (2^w-1)/b_1 \rfloor}\{k\cdot b_1 + p\}$
$y_v = y_u >> b_1$	-	$\{[p\cdot 2^{b_1} \sim p\cdot 2^{b_1} + 2^{b_1} - 1]\}$
$y_v = b_1 >> y_u$	$b_1\%p$	$\{\log_2 b_1 / p\}$
$y_v = y_u << b_1$	$(p\%2^{b_1}) == 0$	$\bigcup_{k=0}^{2^{b_1}-1}\{k\cdot 2^{b_1} + p/2^{b_1}\}$
$y_v = b_1 << y_u$	$(p\%b_1) == 0$	$\{\log_2 p / b_1\}$
$y_v = y_u > b_1$	$p==1$	$\{[b_1+1 \sim 2^w - 1]\}$
$y_v = y_u >= b_1$	$p==1$	$\{[b_1 \sim 2^w - 1]\}$
$y_v = y_u < b_1$	$p==1$	$\{[0 \sim b_1-1]\}$
$y_v = y_u <= b_1$	$p==1$	$\{[0 \sim b_1]\}$
$y_v = y_u == b_1$	$p==1$	$\{b_1\}$
$y_v = y_u != b_1$	$p==1$	$\{[0 \sim b_1-1],[b_1+1 \sim 2^w - 1]\}$

1. w is the bit width of y_u and b_1 is the simulated value of the operand other than y_u.
2. The notation [i~j] means a set of continuous integers from integer i to integer j.

cases, Sub_CurrentMVS$_p = \{\emptyset\}$ except for *comparisons*. The following explains how to derive Sub_CurrentMVS$_p$ for some representative operations.

1) Operations That Choose a Bit Range "$[i]$" and "$[i:j]$": For an operation "$[i:j]$," the only constraint on the input values is that the bit assignment of the bits selected by "$[i:j]$" must be the same as the output value p. The bit assignment of the unselected bits can be any combination. Thus, the value of the unselected bits from 0 to $j-1$ can be any integer in the range from 0 to $2^j - 1$. The value of the unselected bits from $i+1$ to $w-1$ can be any integer in the range from 0 to $2^{w-i-1} - 1$. Hence, the formula for operation "$[i:j]$" appears in the third column of Table I. Deriving Sub_CurrentMVS$_p$ for "$[i]$" can be achieved by treating i the same as j in the "$[i:j]$" formula.

2) Control Vertexes: If y_u is the control signal, y_u can only be the values that select suitable branches to keep the output of vertex vy_v at p. This can be done by comparing the value of each variable on each branch with p. If y_u is the signal on the taken branch, y_u can only be p such that y_v is p.

3) Comparison Operations ">," "<," etc.: Take "$<$" as an example. If p is equal to 1, y_u can only be values smaller than b_1. These values are $\{[0 \sim b_1 - 1]\}$. The derivations for other comparisons are quite similar.

4) Right Shift "$\gg$" and Left Shift "$\ll$": Either *right shift* or *left shift* by the amount b_1 incurs *information loss*. The "$[i:j]$" formula can tackle this. As illustrated in Fig. 5(a) and (b), the entire *right shift* (*left shift*) is the cascade of an operation that selects the bit range from i to j "$[i:j]$" and a *divide* (*multiply*) operation. Therefore, to derive the formula of *right shift* (*left shift*), first, apply the *divide* (*multiply*) formula and, then, the "$[i:j]$" formula. If *information loss* is encountered in other operations, e.g., "+," "−," and "*," the "$[i:j]$" MVS formula can also model it.

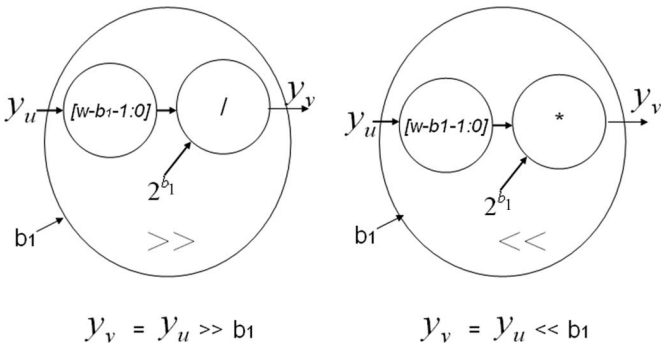

Fig. 5. Modeling *information loss* in *right shift* and *left shift*.

If the formulas listed in the third column of Table I are directly applied to compute CurrentMVS, for a PreviousMVS with n integers, the formula should be applied n times, and then, the union of all the Sub_CurrentMVS$_p$ produces CurrentMVS. Take the operation "$b = a[1:0]$" as an example. Assume

that a is 4-bit wide, b is 2-bit wide, and PreviousMVS $= \{0, 1, 2\}$. To compute CurrentMVS, first, apply the "$[i : j]$" formula with $i = 1$, $j = 0$, $w = 4$, and $p = 0$. The result is

$$\bigcup_{k=0}^{2^{4-1-1}-1} \left\{[0 \cdot 2^0 + k \cdot 2^{1+1} \sim 0 \cdot 2^0 + k \cdot 2^{1+1} + 2^0 - 1]\right\} = \{0, 4, 8, 12\}. \quad (8)$$

The same formula can be used with $p = 1$ and $p = 2$ in sequence to obtain $\{1, 5, 9, 13\}$ and $\{2, 6, 10, 14\}$, respectively. The union $\{0, 1, 2, 4, 5, 6, 8, 9, 10, 12, 13, 14\}$ is CurrentMVS. The computation may take lots of time if there are many elements in PreviousMVS. The following observations can be used to improve this MVS computation.

Taking a closer look at the results obtained with $p = 0$, $p = 1$, and $p = 2$, we observe that $0 * 2^0 + k * 2^{1+1} + 2^0 - 1 = k * 2^{1+1}$ and $1 * 2^0 + k * 2^{1+1} = k * 2^{1+1} + 1$ are two continuous integers. Also, $1 * 2^0 + k * 2^{1+1} + 2^0 - 1 = k * 2^{1+1} + 1$ and $2 * 2^0 + k * 2^{1+1} = k * 2^{1+1} + 2$ are two continuous integers. Therefore, the union of the aforementioned three sets can be represented more concisely as

$$\bigcup_{k=0}^{2^{4-1-1}-1} \left\{[0 \cdot 2^0 + k \cdot 2^{1+1} \sim 2 \cdot 2^0 + k \cdot 2^{1+1} + 2^0 - 1]\right\} = \bigcup_{k=0}^{3} \left\{[k \cdot 4 \sim 2 + k \cdot 4]\right\}. \quad (9)$$

More generally, for a set of continuous integers from p to q in PreviousMVS, the computed CurrentMVS is

$$\bigcup_{k=0}^{2^{w-i-1}-1} \left\{[p \times 2^j + k \times 2^{i+1} \sim q \times 2^j + k \times 2^{i+1} + 2^j - 1]\right\}. \quad (10)$$

The "$[i : j]$" MVS formula is derived now and listed in the third column of Table II.

The operation "$\ll$" is another example of how to derive the "$\ll$" formula listed in the third column of Table II. First, try to find the smallest integer p' in the set $\{[p \sim q]\}$, which satisfies $p'\%2^{b_1} == 0$. If there is no such p' in the set $\{[p \sim q]\}$, CurrentMVS will be ϕ. If p' exists in $\{[p \sim q]\}$, check if $p' + 2^{b_1}$ is in the range from p to q. If so, the union of the two result sets obtained by p' and $p' + 2^{b_1}$ can be represented as

$$\bigcup_{k=0}^{2^{b_1}-1} \left\{\left[k \cdot 2^{b_1} + p'/2^{b_1} \sim k \cdot 2^{b_1} + (p' + 2^{b1})/2^{b_1}\right]\right\}. \quad (11)$$

Repeating the aforementioned derivations produces the formula in the third column in Table II.

For a subset of integers $\{[p \sim q]\}$ in PreviousMVS, applying the MVS formulas listed in the third column in Table II can derive results much more quickly than applying the formulas in Table I. In addition, all the integers in the subset $\{[p \sim q]\}$ can be memorized by recording only p, q, and the special tag "$\sim$." This storage format enhances memory usage efficiency and alleviates the *memory explosion* problem.

C. MVS Estimations for Reconvergent Paths

The algorithm shown in Fig. 4 can compute the exact MVS of vertex b in time frame $t = t_i$ with respect to an OP$_j$ in time frame $t = c_k$ only if there is just one *single path* from b at $t = t_i$ to OP$_j$ at $t = c_k$. If there are multiple *reconvergent paths*, another approach is necessary because possible propagation methods become more complex.

In tag-based approaches [3]–[5], the authors simply put *unknown tags* "?" on the *reconvergent paths* instead of computing exact solutions because precisely handling the *reconvergent paths* is too complex. If *unknown tags* are propagated to OPs, they seem to be considered as *not covered* with respect to OCCOM. In other words, the contributions of *reconvergent paths* are handled in a conservative way.

In order to reduce the complexity, we adopt a strategy to handle *reconvergent paths* similar to tag-based approaches. If there are multiple *reconvergent paths* from v at $t = t_i$ to an OP$_j$ at $t = c_k$, the universe $(\boldsymbol{U})$ is used instead of real MVS$(b@t = t_i)_{\mathrm{OP}_j@t=c^k}$ in the intersection operation. This estimation result obtained using the universe must include the exact result obtained by intersecting with the real MVS$(b@t = t_i)_{\mathrm{OP}_j@t=c^k}$ because the universe includes MVS$(b@t = t_i)_{\mathrm{OP}_j@t=c^k}$. Consequently, this estimation result has a larger MVS set, which turns out to be less observable according to the definition of observability in (5). Therefore, this estimation approach guarantees lower bound estimations of observability.

This estimation approach may incur some accuracy loss. Because the estimated observability may be lower than the actual value, it is possible to underestimate the coverage or insert assertions on some points that are actually safe. While conducting verifications, this conservative strategy that checks more points is often acceptable and will not cause too many problems.

D. Time-Saving Strategies

To reduce computation time, we develop: 1) the *bounding traversal* strategy and 2) the limited-traversed-frame (LTF) strategy. *Bounding traversal* strategy can avoid unnecessary traversals during MVS computation without causing any accuracy loss. LTF strategy saves additional time at the expense of accuracy loss. However, it can always have a lower bound of observability (a pessimistic estimation).

1) Bounding Traversal Strategy: In our observability computation, after some backward traversals, there are MVS sets recorded on vertices that have been traversed. As shown in Fig. 6, let a vertex v in time frame $t = t_n$ be a vertex that was traversed, and let v' be one of v's *fan-in* vertices that was also traversed. Hence, MVS$(v@t = t_n)$ and MVS$(v'@t = t_n)$ are already recorded on v and v'. In addition, MVS$(v'@t = t_n)$ should be $\{x | f_v(x)\text{MVS}(v@t = t_n)\}$ according to the CurrentMVS computation shown in line 12 of the *MVS_for_vertex* pseudocode in Fig. 4.

If another backward traversal from an OP arrives at vertex v in time frame $t = t_n$ again, PreviousMVS and MVS$(v@t = t_n)$ are intersected, as described in line 4 of the *MVS_for_vertex* pseudocode. If the result of the intersection remains MVS$(v@t = t_n)$, i.e., MVS$(v@t = t_n) \subseteq$ PreviousMVS, then when the computation arrives at v', the

TABLE II
MVS FORMULAS FOR HDL OPERATIONS

Operation	Condition	$Sub\_CurrentMVS_{[p\sim q]}$
$y_v = y_u$	-	$\{[p\sim q]\}$
$y_v = \sim y_u$	-	$\{[2^w-1-q \sim 2^w-1-p]\}$
$y_v = - y_u$	-	$\{[2^w-q \sim 2^w-p]\}$
$y_v = y_u\ [i:j]$	-	$\bigcup_{k=0}^{2^{w-i-1}-1}\{[p\cdot 2^j + k\cdot 2^{i+1} \sim q\cdot 2^j + k\cdot 2^{i+1} + 2^j - 1]\}$
$y_v = y_u\ [i]$	-	$\bigcup_{k=0}^{2^{w-i-1}-1}\{[p\cdot 2^i + k\cdot 2^{i+1} \sim q\cdot 2^i + k\cdot 2^{i+1} + 2^i - 1]\}$
$y_v = y_u + b_1$	-	$\{[p\text{-}b_1\sim q\text{-}b_1]\}$
$y_v = y_u - b_1$	-	$\{[p+b_1\sim q+b_1]\}$
$y_v = b_1 - y_u$	-	$\{[b_1\text{-}q\sim b_1\text{-}p]\}$
$y_v = y_u * 0$	$p == 0$	$\{[0 \sim 2^w-1]\}$
$y_v = y_u * b_1\ (b_1>0)$	$\lfloor p/b_1 \rfloor < \lfloor q/b_1 \rfloor$	$\{[\lfloor p/b_1 \rfloor \sim \lfloor q/b_1 \rfloor]\}$
$y_v = y_u \% b_1$	$q < b_1$	$\bigcup_{k=0}^{\lfloor (2^w-1)/b_1 \rfloor}\{[k\cdot b_1 + p \sim k\cdot b_1 + q]\}$
$y_v = y_u >> b_1$	-	$\{[p\cdot 2^{b_1} \sim q\cdot 2^{b_1} + 2^{b_1} - 1]\}$
$y_v = b_1 >> y_u$	$\lfloor b_1/q \rfloor < \lfloor b_1/p \rfloor$	$\{[\log_2\lfloor b_1/q \rfloor \sim \log_2\lfloor b_1/p \rfloor]\}$
$y_v = y_u << b_1$	$\lfloor p/2^{b_1} \rfloor < \lfloor q/2^{b_1} \rfloor$	$\bigcup_{k=0}^{2^{b_1}-1}\{[k\cdot 2^{b_1} + \lceil p/2^{b_1} \rceil \sim k\cdot 2^{b_1} + \lfloor q/2^{b_1} \rfloor]\}$
$y_v = b_1 << y_u$	$\lfloor p/b_1 \rfloor < \lfloor q/b_1 \rfloor$	$\{[\log_2\lfloor p/b_1 \rfloor \sim \log_2\lfloor q/b_1 \rfloor]\}$
$y_v = y_u > b_1$	$p == 0$ and $q == 1$	$\{[0 \sim 2^w-1]\}$
$y_v = y_u >= b_1$	$p == 0$ and $q == 1$	$\{[0 \sim 2^w-1]\}$
$y_v = y_u < b_1$	$p == 0$ and $q == 1$	$\{[0 \sim 2^w-1]\}$
$y_v = y_u <= b_1$	$p == 0$ and $q == 1$	$\{[0 \sim 2^w-1]\}$
$y_v = y_u == b_1$	$p == 0$ and $q == 1$	$\{[0 \sim 2^w-1]\}$
$y_v = y_u != b_1$	$p == 0$ and $q == 1$	$\{[0 \sim 2^w-1]\}$

1. w is the bit width of y_u and b_1 is the simulated value of the operand other than y_u.
2. The notation [*i*~*j*] means a set of continuous integers from integer i to integer j.

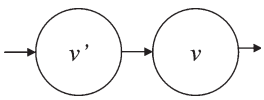

Fig. 6. Vertex v and one of its *fan-in* vertex v'.

result of the intersection will also be MVS$(v'@t = t_n)$. More formally, if MVS$(v@t = t_n) \subseteq$ PreviousMVS, then MVS$(v'@t = t_n) \subseteq \{x | f_v(x) \in$ PreviousMVS$\}$. Theorem 2 provides a formal description and proof.

Theorem 2: If MVS$(v@t = t_n) \subseteq$ PreviousMVS, then MVS$(v'@t = t_n) \subseteq \{x | f_v(x) \in$ PreviousMVS$\}$. The originally recorded MVS$(v'@t = t_n)$ remains unchanged after the intersection.

Proof: The MVS$(v'@t = t_n)$ is computed based on the MVS$(v@t = t_n)$. That is, MVS$(v'@t = t_n)$ is the set $\{x | f_v(x) \in$ MVS$(v@t = t_n)\}$. For an arbitrary element x in MVS$(v'@t = t_n)$, $f_v(x)$ is in MVS$(v@t = t_n)$ and, thus, is also in PreviousMVS since MVS$(v@t = t_n) \subseteq$ PreviousMVS. Therefore, if MVS$(v@t = t_n) \subseteq$ PreviousMVS, MVS$(v'@t = t_n) \subseteq \{x | f_v(x) \in$ PreviousMVS$\}$. The originally recorded MVS$(v'@t = t_n)$ remains unchanged after the intersection.

If v' has at least one *fan-in* vertex v'', by *mathematical deduction*, MVS$(v''@t = t_n)$ should also remain unchanged after the intersection. So do the vertices that are in transitive *fan-in* of vertex v. Therefore, when PreviousMVS includes the recorded MVS of a vertex v, return from subroutine *MVS_for_vertex* can avoid unnecessary traversals and computations since further computations will not change the recorded MVSs.

2) LTF Strategy: The *bounding traversal* strategy can avoid unnecessary traversals. However, in some cases, necessary backward traversals can still expand many frames. Although accurate results are produced, the required computation time may become unaffordable. Therefore, we propose an LTF strategy, which provides an optional and flexible tradeoff between accuracy and speed.

The idea of LTF strategy is to restrict the number of backward-traversed frames in time frame expansion. It only requires a simple check on whether the number of expanded frames reaches the maximum allowable number of frames (denoted as *frame_limit*). *frame_limit* is a configurable parameter that can be adjusted by users. It can be set as a small number for a quick estimation or as infinite to disable LTF strategy for the highest accuracy. Unlike the *bounding traversal* strategy,

```
Preparation Phases:
1: 3-address Code Generation and Conditional statement modification
2: Simulation with commercial HDL simulator to obtain the dumpfile

Observability_computation (DUV, Dumpfile, OPs, frame_limit)
1: CDFG construction
2: Initialize each vertex as "untraversed"
3: for each positive edge of clock t=c_k
4:   for each observation point OP_j
5:     InitialMVS = {CV(OP_j@t=c_k)}; Find the fanin vertex a_1 of OP_j at t=c_k
6:     MVS_Com_for_Vertex(InitialMVS, a_1, OP_j, c_k, c_k, frame_limit)
7: Calculate observability with the computed MVSs

MVS_Com_for_Vertex(PreviousMVS, vertex v, StartOP, StartTime, time t_j, frame_limit)
//*** Modification for incorporating MVS computation for reconvergent paths ***
1: if traversed for first time in traversal starting from StartOP at StartTime
2:   if MVS(v@t=t_j) = = ∅
3:     MVS(v@t=t_j) = PreviousMVS
4:   else
5:     if MVS(v@t=t_j) ⊆ PreviousMVS  //**Condition of Bounding traversal
6:       return
7:     MVSforRecovery(v@t=t_j) = MVS(v@t=t_j)
8:     MVS(v@t=t_j) = MVS(v@t=t_j) ⋂ PreviousMVS
9: else //Reconvergent paths. Recovering to the previous status before intersection
10:    MVS(v@t=t_j) = MVSforRecovery(v@t=t_j)
//*** Modification for incorporating MVS computation for reconvergent paths ***
11:  if v is a control vertex
12:    Mark the fanin vertex(es) on the untaken branch(es) as "inactive"
13:  for each active fanin vertex u of v
14:    if edge (u, v) across time frame
15:      t_h = t_j – clock_period
16:      if t_h <0 or frame_limit = = 0  //** Condition of Limited Traversing Frame
17:        return
18:      Frame_limit - -
19:    Compute CurrentMVS, which is { x | f_v(x) ∈ PreviousCVS }
20:    MVS_Com_for_Vertex(CurrentMVS, u, StartOP, StartTime, t_h, frame_limit)
```

Fig. 7. Pseudocode of observability computation algorithm.

this strategy may experience some accuracy loss. However, a lower bound estimation of observability is always guaranteed such that our observability measures seldom overestimate the correctness of the DUV. The reason is given as follows.

For a vertex u in time frame $t = c_k$, if expanded frames are not limited, each MVS of u at $t = c_k$ will be intersected with respect to an OP at a positive clock edge in the set of MVS sets $\{\text{MVS}_1, \text{MVS}_2, \ldots, \text{MVS}_m\}$. With the *frame_limit* restriction, some MVS of u at $t = c_k$ with respect to some OPs are not obtained since the backward traversals are bounded and do not reach u in time frame $t = t_k$. Assume that the obtained MVSs are $\{\text{MVS}_1, \text{MVS}_2, \ldots, \text{MVS}_n\}$, where $n < m$. The intersection of all the MVSs in the set $\{\text{MVS}_1, \text{MVS}_2, \ldots, \text{MVS}_n\}$ includes the intersection of all the MVSs in the set $\{\text{MVS}_1, \text{MVS}_2, \ldots, \text{MVS}_m\}$. Larger MVS set intersections turn out to be less observable according to the definition of observability in (5). Therefore, our LTF strategy also guarantees lower bound estimations of observability.

E. Algorithm of Observability and MVS Computation

The entire algorithm of our observability computation is abstracted as the pseudocode in Fig. 7. This incorporates: 1) MVS estimation for *single path*; 2) MVS computation for *reconvergent paths*; 3) bounding traversal strategy; and 4) LTF strategy. The entire algorithm is quite similar to the one in Fig. 4. The modifications are indicated with comments.

The modification on the steps in subroutine *MVS_Com_for_vertex* from line 1 to line 10 incorporates MVS estimation for *reconvergent paths*. During traversal(s) starting from an OP (StartOP) at a time instance (StartTime), if vertex v is visited for the first time, it is treated as the *single-path* case. This PreviousMVS is intersected with $\text{MVS}(v@t = t_i)$, which is already the result of intersecting many PreviousMVSs. Then, if this vertex v is traversed for two or more times in the traversal(s) starting from StartOP at StartTime, there are *reconvergent paths* from v at $t = t_i$ to StartOP at StartTime. Then, the $\text{MVSforRecovery}(v@t = t_i)$ subroutine is used to resume the status of $\text{MVS}(v@t = t_i)$ to the status without intersection in this traversal.

Two conditions are added for incorporating the two time-saving strategies into the algorithm. The condition in line 5 of the *MVS_Com_for_vertex* subroutine is for *bounding traversal* strategy. The last condition in line 16 is for the LTF strategy. Once one of the conditions is met, succeeding computation processes can be skipped, and the program can directly return from the subroutine to save computation time. Besides being bounded by time-saving strategies, traversals are also bounded if there is no frame to expand $(t_h < 0)$ or there is no *fan-in* vertex to traverse.

Some preparations are required before observability computation can begin. The *three-address code generations* and the *conditional statement modification* developed in [4] and [5] must be conducted first for the information required in

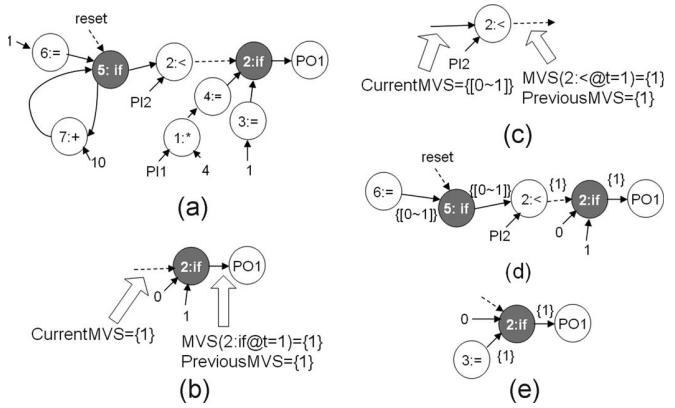

Fig. 8. Computation processes starting from PO1 at $t = 1$.

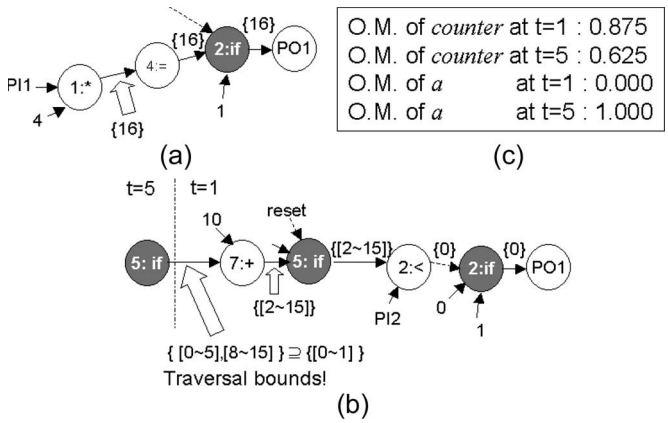

Fig. 9. Observability computation results.

computing MVSs for control vertices (conditional statements). The detailed *conditional statement modification* algorithm can be found in [4] and [5]. Next, run a simulation and obtain the value change dump file (Dumpfile), which is one of the essential inputs of our algorithm. The other inputs are DUV described in an HDL, the list of OPs, and an LTF strategy parameter *frame_limit* as introduced in Section IV-D.

The example in Fig. 1 can also be used to demonstrate the processes of our observability computation. We first construct the CDFG of the DUV. The CDFG of the HDL code in Fig. 1 is shown in Fig. 8(a). After some initializations, we start backward traversal from PO1 at $t = 1$ by calling subroutine *MVS_Com_for_vertex* with the inputs PreviousMVS $= \{1\}$, vertex $v =$ "2 : if", StartOP = PO1, StartTime $= 1$, and $frame\_limit = \infty$.

When subroutine *MVS_Com_for_vertex* is called for the first time, the traversal reaches vertex "2 : if" in time frame $t = 1$ for the first time. As shown in Fig. 8(b), the recorded MVS$(2 : if@t = 1) = \{1\}$, and no *MVSforRecovery* is recorded. Vertex "2 : if" in time frame $t = 1$ is a control vertex. Therefore, there are two fan-in vertices "2 :<" and "3 :=" for further backward traversals. Here, assume that "2 :<" is traversed first. Based on PreviousMVS $= \{1\}$, the MVS computation for conditional statements will be used to compute CurrentMVS and obtain the result $\{1\}$.

Subroutine *MVS_Com_for_vertex* is then called for the second time to traverse to "2 :<" in time frame $t = 1$. When the traversal arrives at vertex "2 :<" in time frame $t = 1$, the computation status is shown in Fig. 8(c). Repeat the similar computations until reaching vertex "6 :=" in time frame $t = 1$. Computation results along the traversal from "2 : if" to "6 :=" are shown in Fig. 8(d), where each set of integers aside an edge is the recorded MVS. Since vertex "6 :=" in time frame $t = 1$ has no *fan-in* vertex, the computation will traverse another *fan-in* vertex "3 :=" of vertex "2 : if." Repeatedly calling subroutine *MVS_Com_for_vertex* can produce the results shown in Fig. 8(e).

After completing the traversals and MVS computations starting from PO1 in time frame $t = 1$, starting backward traversals from PO1 in time frame $t = 5$ can produce the results shown in Fig. 9(a) and (c). When the computation reaches vertex "5 : if" in time frame $t = 1$, PreviousMVS $\{[0 \sim 5], [8 \sim 15]\}$ will include MVS$(5 : \text{if}@t = 1) = \{[0 \sim 1]\}$. The bounding traversal condition is satisfied, and the traversal is bounded here.

After all the OPs at all the positive clock edges are applied in MVS computation, calculating the observability of each internal signal with (5) can produce the result, as shown in Fig. 9(c). The observability of the signal *counter* at $t = 5$ is not high enough to be considered as an observed tag, i.e., 0.625 is not close to 1. However, as we discussed in Section III-A, *tag propagation rules* cannot represent intermediate values between 1 and 0. The rules thus determine that tags injected on *counter* can propagate to PO1 at $t = 5$. This induces some inaccuracy and, even worse, overestimates the actual likelihood that an erroneous effect propagates through "counter $<$ PI2." Experimental results in Section VI also show the same situation of overestimation as we have discussed previously.

F. Observability Analysis for Multiple Design Errors

Incorrect values caused by bugs may be masked and, thus, escape detection. Thus, the simulation values recorded in the dump file may not be completely correct. Therefore, in our observability computation, we do not assume the correctness of the simulation values. We also do not assume the correctness of the DUV. Observability is computed based only on the values of involved signals recorded in the dump file regardless of the correctness of these values. Even if the values used in the computation are incorrect, we can still provide some meaningful values for users' reference based on these incorrect values. When multiple errors occur, this method can reduce the risk of misleading the verification results more than using binary decisions only.

For example, let signals a and b be two 3-bit signals in the DUV. As shown in Fig. 10(a), if the values of a and b are both correct, the observability of a and b are both 0.625. However, as shown in Fig. 10(b), if the value of b recorded in the dump file is 5 instead of the correct value 4, the observability of a can still be determined to be 0.500. The observability of a becomes smaller as the value of b becomes larger. The computed observability of a reasonably corresponds to the value change.

The situation can become even worse. As shown in Fig. 10(c), the values of a and b are 4 and 5, which are both different from their correct values. However, our approach can

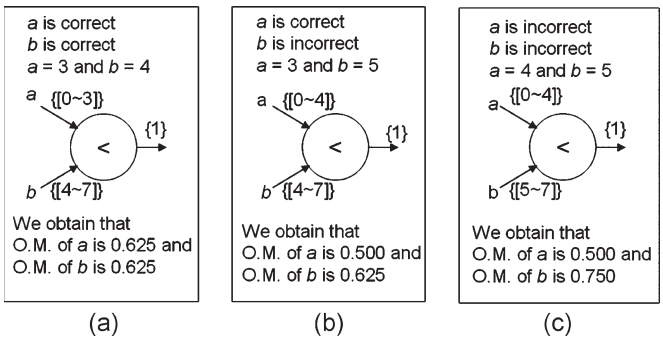

Fig. 10. Observability analysis with correct and incorrect values.

still derive that the observabilities of a and b are 0.500 and 0.750 respectively. The computed observability still adequately corresponds to the value changes of a and b. Therefore, our observability seems to have some degree of immunity to multiple errors.

On the other hand, if we use tags in the example in Fig. 10(c), tag Δ on a and tag $-\Delta$ on b can propagate through the operation "$a < b$." *Tag propagation rules* determine that those *tags* are observable, although, in fact, the real incorrect values of a and b are masked. The resulting tags do not correspond to the value change of a or b. Therefore, if multiple errors exist in the DUV, *tags* may provide incorrect predications on error propagation.

Besides the cases shown in Fig. 10(a)–(c), there is still one case where incorrect values are not *masked* and can cause discrepancies in observable outputs. For example, if a is changed to 4 and b is changed to 3, the output of the comparator will become FALSE. In such a case, internal design errors are considered as detected during simulation. Although the observability of a and b may be underestimated in this case due to multiple errors, it will not mislead the verification results because users know that an error occurs and causes output discrepancy.

V. Experimental Results

We conducted experiments on a subset of the 1999 International Test Conference benchmark written in VHDL [1] and four designs written in Verilog HDL. The information for these design cases is presented in Table III, including the total number of lines (#Line), the number of variables (#Var), the number of test vectors (#Vec), and the simulation time (Sim. Time). The test vectors applied in our experiments were randomly generated with very little manual guidance (e.g., reset handling) targeted on high statement coverage ($\sim$90%). The number of test vectors increased in increments of 1000 until statement coverage reaches our target.

The coverage reports of the statement coverage metric and our observability-enhanced statement coverage metric (OSCOM) are recorded in the columns "Stmt" and "OSCOM," respectively. For each design case, OSCOM coverage is often less than the statement coverage. This means that some statements are exercised, but their observability is not high enough to reach our threshold of 0.9.[3] Without sufficient observability, we are not confident about the accuracy of the simulation values if we only observe from the OPs. Consequently, OSCOM filters out these exercitations of statements, acting as a more stringent code coverage metric than statement coverage metric.

We also conducted experiments to compare the *propagation probabilities* [14]–[16], *tag simulation calculus* [3]–[5], and our observability measures. We designed an experiment to compare their capabilities in predicting the propagation of potential design bugs. For each design case, we randomly selected one expression and changed it into a different expression to inject a design bug. The change we made was randomly selected from typical bugs that designers usually induce according to research in the arena of *mutant analysis* [28]. By simulating faulty HDL design and comparing OP simulation values with the values of the original HDL design, we can determine whether or not the injected bugs are detected in this experiment. For each injected bug, the bug injection and identification process is repeated 300 times. The overall results are reported in Table III.

We then calculate the three aforementioned observability measures for the detected or undetected bug. PPs were calculated according to the approach proposed in [14]. This required repeating the following steps for 5000 iterations. The steps include infecting the data state of a variable using the *perturbation function*, simulating the program under test, and monitoring the results at the OPs and recording the number of program failures.

Tag-based observability (Tag) is calculated according to the *tag simulation calculus proposed* in [3]–[5]. If a *tag* injected on our injected bug was observed, we considered the computed observability to be 1. Otherwise, the observability was set to 0. Our observability measures were calculated using the proposed approach with $frame\_limit = 20$ ($\text{Ours}_{\text{FL}=20}$) and $frame\_limit = \infty$ ($\text{Ours}_{\text{FL}=\infty}$). For the 300 iterations we ran, the average values of these observability measures for both detected and undetected design bugs are listed in Table III.

Experimental results reveal that the detection of a design error is strongly related to the values of all three observability measures. Errors with low observability are indeed difficult to detect at the OPs. In addition, the values of tag-based observability measures for undetected bugs tend to be higher than the other measures. For undetected bugs, if the observability is overestimated, the completeness of the validation and the correctness of the DUV can be misjudged. For example, in the test case div16, the average tag-based observability is 0.344. This implies that 34.4% of undetected bugs will be judged as observable, which may lead to wrong conclusions.

On the other hand, our observability measures exhibit quite similar results as the PP for both detected and undetected errors. These similar values mean that our approach should have capabilities similar to the statistics-based approach. For a hard-to-observe point that PPs can identify, our measures may very well do the same. Even if some heuristics, such as the LTF strategy, are used in our approach to reduce computational complexity, we can see that observability results are still very close to the results without any heuristics ($\text{FL} = \infty$).

Since the accuracy of our approach is very similar to the statistics-based approach, we conducted another experiment to compare the computation time of both approaches. For each

[3]The threshold of observability measures can be adjusted by tool users of our coverage analysis. It represents the observability requirement that tool users want every signal in the design to reach.

TABLE III
COMPARING OUR OBSERVABILITY WITH PROPAGATION PROBABILITIES, TAG-BASED OBSERVABILITY, AND STATEMENT COVERAGE METRIC

Design Name	#Line	#Var	#Vec	Sim. Time (s)	Stmt (%)	OSCOM (%)	Detected Bugs				Undetected Bugs			
							PP	Tag	Ours (FL=∞)	Ours (FL=20)	PP	Tag	Ours (FL=∞)	Ours (FL=20)
B01	110	7	1000	0.2	100.0	92.1	0.988	1.000	0.992	0.992	0.117	0.125	0.122	0.122
B02	70	5	1000	0.1	100.0	100.0	1.000	1.000	1.000	1.000	0.005	0.150	0.008	0.008
B03	141	21	1000	0.2	95.2	67.2	0.939	0.954	0.937	0.937	0.078	0.133	0.081	0.081
B04	102	19	1000	0.3	93.1	64.1	0.957	0.980	0.967	0.964	0.108	0.122	0.115	0.114
B05	332	25	1000	0.3	94.2	70.1	0.966	0.988	0.977	0.972	0.034	0.190	0.036	0.034
B06	128	9	1000	0.2	100.0	91.3	0.991	1.000	0.988	0.988	0.074	0.107	0.074	0.074
B07	92	11	2000	0.3	96.5	70.6	0.907	0.954	0.910	0.897	0.140	0.238	0.136	0.127
B08	89	23	2000	0.3	94.2	81.1	0.947	1.000	0.978	0.964	0.103	0.250	0.095	0.088
B11	118	21	2000	0.3	90.3	66.7	0.821	0.868	0.811	0.801	0.044	0.268	0.053	0.051
B14	509	27	5000	1.8	89.1	50.2	0.738	0.852	0.721	0.701	0.132	0.306	0.131	0.123
B21	1089	65	5000	3.8	90.2	53.1	0.778	0.915	0.770	0.766	0.113	0.298	0.110	0.103
div16	235	11	1000	0.3	100.0	77.2	0.934	0.964	0.942	0.939	0.063	0.344	0.065	0.065
pcpu	952	54	5000	1.6	87.3	59.3	0.738	0.862	0.813	0.793	0.168	0.285	0.171	0.171
blkJ	156	20	1000	0.2	97.3	80.1	0.938	0.957	0.958	0.950	0.079	0.118	0.081	0.074
Mtrx	80	18	1000	0.2	100.0	100.0	0.984	1.000	0.985	0.983	0.030	0.000	0.031	0.031

TABLE IV
PERFORMANCE COMPARISON WITH PP

Design Name	Propagation Prob.			Our approach (FL=20)			*Spdup*
	Avg. time for one var. (s)	Avg. time for all vars (s)	Normalized Sim. time	Avg. time for all vars (s)	Normalized Sim. time	Mem. (MB)	
B01	1620	11340	$5.7*10^4$	0.4	2.0	1.1	$2.9*10^4$
B02	1728	8640	$8.6*10^4$	0.5	5.0	0.8	$1.7*10^4$
B03	3373	70833	$3.5*10^5$	0.4	2.0	1.5	$1.8*10^4$
B04	3419	64961	$2.2*10^5$	1.3	4.4	3.3	$4.9*10^4$
B05	3229	80725	$2.7*10^5$	1.1	3.7	6.3	$7.3*10^4$
B06	1562	14058	$7.0*10^4$	0.3	1.5	0.9	$4.8*10^4$
B07	3597	39567	$1.3*10^5$	0.8	2.7	2.5	$4.9*10^4$
B08	3410	78430	$2.6*10^5$	5.2	17.3	4.8	$1.5*10^4$
B11	3735	78435	$2.6*10^5$	3.8	12.6	5.1	$2.1*10^4$
B14	19781	534090	$3.0*10^5$	201.2	111.8	18.3	$2.7*10^3$
B21	37301	2424600	$6.4*10^5$	452.2	119.0	39.5	$5.4*10^3$
div8	1640	18040	$6.0*10^4$	0.9	3.0	1.6	$2.0*10^4$
pcpu	21981	1187000	$7.4*10^5$	145.1	90.7	12.9	$8.2*10^3$
blkJ	1981	39620	$2.0*10^5$	1.8	9.0	1.2	$2.2*10^4$
Mtrx	1781	32166	$1.6*10^5$	0.6	3.0	1.0	$5.4*10^4$

design, the computation time required to obtain observability measures for all signals is presented in the column "Avg. time for all vars" under "Our approach" in Table IV. Since the approach in [14] can only derive PPs for one signal at a time, the computation time for one signal is shown in the column "Avg. time for one var." For a design case with n variables, the total computation time of the statistics-based approach to obtain PPs for all signals is "n $*$ the computation time in the column PP for one var." This is recorded in the column "Avg. time for all vars." It is obvious that our approach is much faster than the statistics-based approach [14]. The speedup ratio (recorded in the column "Spdup") is defined as the ratio of "PP for all vars" to "OM for all vars." Normalized simulation time, which is defined as the ratio of the computation time for observability to the plain HDL simulation time, is also provided in Table IV for both approaches to show the efficiency of observability calculation. The results show that our approach can greatly reduce the required computation time to a reasonable region.

VI. CONCLUSION AND FUTURE WORKS

In this paper, we present a new probabilistic observability measure for HDL descriptions along with its efficient computation algorithm. Unlike tag-based approaches, which can provide only two levels of measurement, our fine-grained observability measures have less risk of overestimating the extent of validation with reasonable computation time. Even when multiple errors occur, we can still provide some meaningful values for users' reference to reduce the risk of misleading the verification results. This is better than using binary decisions only.

Experimental results show that the observability measures computed by our dump-file-based approach have almost the same capability to identify hard-to-observe locations as the

statistics-based approach [14]. However, our method is much faster and is more suitable to be applied in the HDL codes of commercial products.

Since hard-to-observe points can be identified using our observability measure, designers can insert assertions in those locations to find hidden bugs more easily. This observability-driven assertion insertion is simple but should be very effective. Of course, it is also possible to generate a test vector set that creates some highly transparent sensitized paths to propagate potential incorrect values of the exercised statements to OPs, such as the extension works of OCCOM [8], [9]. We will try to study this direction in the future based on our observability measures to provide a comprehensive solution for the observability issue during simulation.

REFERENCES

[1] ITC99 Benchmark. [Online]. Available: http://www.cad.polito.it/tools/itc99.html

[2] B. Beizer, *Software Testing Techniques*, 2nd ed. New York: Van Nostrand, 1990.

[3] S. Devadas, A. Ghosh, and K. Keutzer, "An observability-based code coverage metric for functional simulation," in *Proc. Int. Conf. Comput.-Aided Des.*, Nov. 1996, pp. 418–425.

[4] F. Fallah, S. Devadas, and K. Keutzer, "OCCOM: Efficient computation of observability-based code coverage metrics for functional simulation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 8, pp. 1003–1015, Aug. 2001.

[5] F. Fallah, "Coverage directed validation of hardware models," Ph.D. dissertation, M.I.T., Cambridge, MA, 1999.

[6] J. C. Costa, S. Devadas, and J. C. Monteiro, "Observability analysis of software for coverage-directed validation," in *Proc. Int. Conf. Comput.-Aided Des.*, Nov. 2000, pp. 27–32.

[7] F. Fallah, I. Ghosh, and M. Fujita, "Event-driven observability enhanced coverage analysis of C programs for functional validation," in *Proc. Asian and South Pacific Des. Autom. Conf.*, Jan. 2003, pp. 123–128.

[8] F. Fallah, S. Devadas, and K. Keutzer, "Functional vector generation for HDL models using linear programming and Boolean satisfiability," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 8, pp. 994–1002, Aug. 2001.

[9] F. Fallah, P. Ashar, and S. Devadas, "Functional vector generation for sequential HDL models under observability-based code coverage metric," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 10, no. 6, pp. 919–923, Dec. 2002.

[10] S. Tasiran, F. Fallah, D. Chinnery, S. Weber, and K. Keutzer, "A functional validation technique: Biased random simulation guided by observability-based coverage," in *Proc. Int. Conf. Comput. Des.*, Sep. 2001, pp. 82–88.

[11] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Des. Test. Comput.*, vol. 18, no. 4, pp. 36–45, Jul./Aug. 2001.

[12] Q. Zhang and I. Harris, "A data flow fault coverage metric for validation of behavior HDL descriptions," in *Proc. Int. Conf. Comput.-Aided Des.*, Nov. 2000, pp. 369–372.

[13] J. Fernandes, M. Santos, A. Oliveira, and J. Teixeira, "A probabilistic method for the computation of testability of RTL constructs," in *Proc. Des. Autom. and Test Eur. Conf.*, Feb. 2004, pp. 176–181.

[14] J. Voas, "PIE: A dynamic failure-based technique," *IEEE Trans. Softw. Eng.*, vol. 18, no. 8, pp. 717–727, Aug. 1992.

[15] J. Voas and K. Miller, "Software testability: The new verification," *IEEE Softw.*, vol. 12, no. 3, pp. 17–28, May 1995.

[16] J. Voas, G. McGraw, L. Kassab, and L. Voas, "A 'crystal ball' for software reliability," *Computers*, vol. 30, no. 6, pp. 29–36, Jun. 1997.

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

[18] L. H. Goldstein, "Controllability/observability analysis," *IEEE Trans. Circuits Syst.*, vol. CAS-26, no. 9, pp. 685–693, Sep. 1979.

[19] F. Brglez, "On testability of combinational networks," in *Proc. Int. Symp. Circuits and Syst.*, 1984, pp. 221–225.

[20] C. H. Chen and P. R. Menon, "An approach to functional level testability analysis," in *Proc. Int. Test Conf.*, Oct. 1989, pp. 373–379.

[21] K. Thearling and J. Abraham, "An easily computed functional level testability measure," in *Proc. Int. Test Conf.*, 1989, pp. 381–390.

[22] C. H. Chen and D. G. Saab, "A novel behavioral testability measure," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 12, no. 12, pp. 1960–1970, Dec. 1993.

[23] S. Bhattacharya, S. Dey, and F. Brglez, "RT-level transformations for gate-level testability," in *Proc. Eur. Conf. Des. Autom.*, Feb. 1993, pp. 162–169.

[24] P. A. Thaker, V. D. Agrawal, and M. E. Zaghloul, "Validation vector grade (VVG): A new coverage metric for validation and test," in *Proc. VLSI Test Symp.*, Apr. 1999, pp. 182–188.

[25] S. Ravi, G. Lakshminarayana, and N. K. Jha, "TAO: Regular expression-based register-transfer level testability analysis and optimization," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 9, no. 6, pp. 824–832, Dec. 2001.

[26] I. Ghosh, A. Raghunathan, and N. K. Jha, "Hierarchical test generation and design for testability methods for ASPP's and ASIP's," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 18, no. 3, pp. 357–370, Mar. 1999.

[27] B. Murray and J. P. Hayes, "Hierarchical test generation using precomputed test for modules," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 9, no. 6, pp. 594–603, Jun. 1990.

[28] A. Offutt and G. Rothermel, "An experimental evaluation of selective mutation," in *Proc. Int Conf. Softw. Eng.*, May 1993, pp. 100–107.

Tai-Ying Jiang received the B.S. degree in electrical engineering from the National Tsing Hua University, Hsinchu, Taiwan, R.O.C., in 1999, and the M.S. degree in electronics engineering from the National Chiao Tung University, Hsinchu, in 2001. He is currently working toward the Ph.D. degree in electronics engineering at the Department of Electronics Engineering, National Chiao Tung University.

His research interests include functional validation and semiformal verification for HDL designs and error diagnosis.

Chien-Nan Jimmy Liu received the B.S. and Ph.D. degrees in electronics engineering from the National Chiao Tung University, Hsinchu, Taiwan, R.O.C.

He is currently an Assistant Professor at the Department of Electrical Engineering, National Central University, Jhongli City, Taiwan. His research interests include functional verification for HDL designs, high-level power modeling, and analog behavioral models for system verification.

Prof. Liu is a member of Phi Tau Phi.

Jing-Yang Jou (S'82–M'83–SM'02–F'05) received the B.S. degree in electrical engineering from the National Taiwan University, Taipei, Taiwan, R.O.C., in 1979 and the M.S. and Ph.D. degrees in computer science from the University of Illinois, Urbana-Champaign, in 1983 and 1985, respectively.

He is currently the Director of the National Chip Implementation Center, National Applied Research Laboratories, Hsinchu, Taiwan, R.O.C. He is a Full Professor and was the Chairman of the Department of Electronics Engineering, National Chiao Tung University, Hsinchu, from 2000 to 2003. Before joining the National Chiao Tung University, he was with GTE Laboratories and AT&T Bell Laboratories. He has published more than 100 technical papers. His research interests include behavioral, logic, and physical synthesis, design verification, and CAD for low power.

Dr. Jou is a member of Tau Beta Pi. He served as the Technical Program Chair of the Asia–Pacific Conference on Hardware Description Languages (APCHDL'97). He was the recipient of the Distinguished Paper Award of the 1990 IEEE International Conference on Computer-Aided Design.