Accurate Rank Ordering of Error Candidates for Efficient HDL Design Debugging

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

***Abstract*—When hardware description languages (HDLs) are used in describing the behavior of a digital circuit, design errors (or bugs) almost inevitably appear in the HDL code of the circuit. Existing approaches attempt to reduce efforts involved in this debugging process by extracting a reduced set of error candidates. However, the derived set can still contain many error candidates, and finding true design errors among the candidates in the set may still consume much valuable time. A *debugging priority* method [21] was proposed to speed up the error-searching process in the derived error candidate set. The idea is to display error candidates in an order that corresponds to an individual's degree of suspicion. With this method, error candidates are placed in a rank order based on their probability of being an error. The more likely an error candidate is a design error (or a bug), the higher the rank order that it has. With the displayed rank order, circuit designers should find design errors quicker than with blind searching when searching for design errors among all the derived candidates. However, the currently used confidence score (CS) for deriving the *debugging priority* has some flaws in estimating the likelihood of correctness of error candidates due to the *masking error* situation. This reduces the degree of accuracy in establishing a *debugging priority*. Therefore, the objective of this work is to develop a new probabilistic confidence score (PCS) that takes the *masking error* situation into consideration in order to provide a more reliable and accurate *debugging priority*. The experimental results show that our proposed PCS achieves better results in estimating the likelihood of correctness and can indeed suggest a *debugging priority* with better accuracy, as compared to the CS.**

***Index Terms*—Error diagnosis, hardware description language (HDL), HDL code debugging, verification.**

I. Introduction

WITH THE increasing complexity of very large scale integration circuit designs, the design cycle of a digital circuit often involves several design stages. Verification is used to monitor the consistency in designs between subsequent stages. When the verification process finds that a design in the current stage (implementation) is not consistent with that in the previous stage (specification), design error diagnosis is needed.

Manuscript received March 18, 2008; revised September 26, 2008. Current version published January 21, 2009. This paper was recommended by Associate Editor V. Narayanan.

T.-Y. Jiang is with the Department of Electronics Engineering, National Chiao Tung University, Hsinchu 300, Taiwan (e-mail: taiyingjiang@realtek.com.tw).

C.-N. J. Liu is with the Department of Electrical Engineering, National Central University, Jhongli City 320, Taiwan.

J.-Y. Jou is with the Department of Electronics Engineering, National Chiao Tung University, Hsinchu 300, Taiwan and is Vice Chancellor (Academic Affairs) of the University System of Taiwan.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2008.2009163

Most of the previous studies on the topic of design error diagnosis target the diagnosis of gate-level or lower level implementations. These methods can be roughly divided into two categories: 1) symbolic approaches and 2) simulation-based approaches. Symbolic approaches [1]–[6] use a binary decision diagram (BDD) to represent functional manipulation with BDD to formulate the necessary and sufficient condition of fixing design errors. Some recent symbolic works exploit the progress of a Boolean satisfiability (SAT) solver and develops SAT-based approaches [6]. On the other hand, simulation-based approaches [7]–[14] rely on simulating erroneous test vectors to gradually reduce impossible error candidates. Some of these methods are error-model-based methods, some are structure-based methods, and some are a combination of the two. Such error models include gate errors (missing gate, extra gate, wrong logical connective, etc.), line errors (missing line, extra line, etc.), and unknown models (Xs).

In addition to the gate or other lower levels, design errors can also occur at the very first design stage, i.e., modeling the circuit behavior using hardware description languages (HDLs). Traditionally, debugging a faulty HDL design relies on manually tracing the faulty HDL code. However, a simple HDL design today can probably have thousands of code lines. Manually tracing the faulty HDL code to debug is not an effective debugging method.

In the literature, some researches have targeted techniques that assist HDL design debugging. Peischl and Wotawa [20] focused on the model-based diagnosis paradigm and employed structures and behaviors for source-code-level debugging with respect to their error models. There are also error-model-free methods, which should have better applicability to various kinds of design errors. Khalil *et al.* [15], [16] proposed an approach that can automatically derive four sets of error candidates in a sequence, i.e., from the smallest set to the biggest set. In doing so, they hope that designers can find bugs in the smaller error candidate sets to reduce the required debugging efforts. However, this is not always possible, and the efforts in reviewing the first three sets may be wasted. Shi and Jou [17] applied data dependence analysis, execution statistics, and the characteristics of HDL operations to filter out impossible error candidates. In this method, only one error space is derived for examination, and it is acceptable in size. Huang *et al.* [18] further exploited the extra observability of assertions in an attempt to derive a smaller error space. Instead of automatic methods, Hsu *et al.* developed two useful utilities to help designers explain the locality of bugs with manual interventions [19].

Deriving a reduced set of error candidates is certainly helpful for HDL debugging. However, the derived error candidate set (called "error space" in this paper) can still contain many error candidates, and identifying true design errors by examining candidates one by one still requires much effort and time. An interesting technique called *debugging priority* has been proposed for accelerating error searching in the derived candidate set [21]. A measurement called confidence score (CS) has been developed to assess the likelihood of correctness of each error candidate. Then, by sorting error candidates according to the CS, error candidates are displayed in a prioritized order: from the most likely to the least likely one. With this ranked order, true design errors would be placed in the first few lines and should be found by designers if they search errors according to the order. However, here, it is implicitly assumed that the *masking error* situation, in which the erroneous effects caused by design errors cannot be observed at the outputs, seldom occurs and is not considered within the CS [21]. Without considering the *masking error situation*, the CS may overestimate the likelihood of correctness of the true design error, as the latter may not be placed in the first few lines, as expected. Unfortunately, the benefits of *debugging priority* are limited.

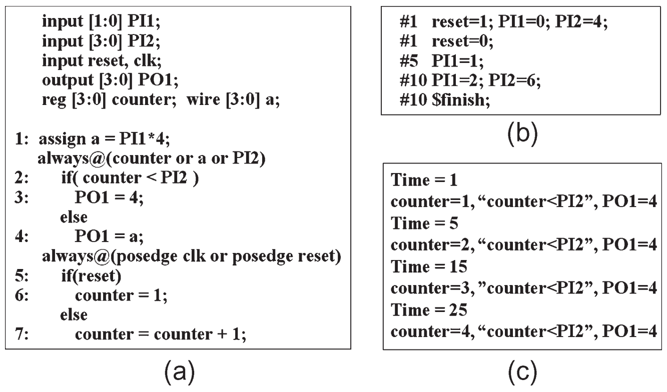
Therefore, this paper attempts to develop a new probabilistic measurement called probabilistic confidence score (PCS) that takes the *masking error situation* into consideration to more suitably estimate the likelihood of correctness of an error candidate. Instead of proposing a new approach for deriving a reduced set of error candidates, this paper focuses on obtaining a more accurate *debugging priority* to speed up the error-searching process among the candidates in the derived error space. Designers can apply any approaches to derive an error space and then use our method to obtain a reliable *debugging priority*. The inputs of our algorithm are given as follows: 1) an error space; 2) a faulty HDL design; and 3) a value change dump file obtained during the simulation. The output will be a candidate list, with the *debugging priority* sorted according to our proposed PCS.

The remainder of this paper is organized as follows: The motivation of this work is introduced in Section II. Section III describes the modeling of the likelihood that a *masking error situation* occurs and defines the PCS. An effective PCS calculation algorithm is introduced in Section IV. The experimental results are presented in Section V. Finally, we conclude this paper in Section VI.

II. Motivation

When estimating the likelihood of correctness of an error candidate to obtain the *debugging priority*, Jiang *et al.* assumed that the erroneous effects of activated errors are seldom masked and can often be propagated to the primary outputs (POs) [21]. With this assumption, if there were no incorrect simulation values at the POs $(\mathrm{PO}_1, \mathrm{PO}_2, \mathrm{PO}_3, \ldots, \mathrm{PO}_m)$ at time instance $t = t_i$, the sensitized statements (SSs) of the POs tended to be correct.[1] As a result, the SSs of the POs receive CS points according to the formula of the CS. In short, the CS did not model the *masking error situation* to estimate the likelihood

[1]Otherwise, erroneous values caused by the SSs should make the simulation values of the POs inconsistent with the expected values at $t = t_i$.



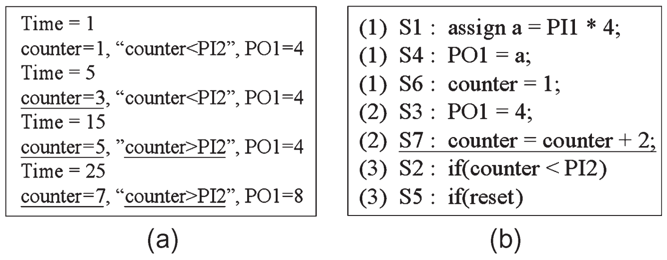
```
    input [1:0] PI1;
    input [3:0] PI2;
    input reset, clk;
    output [3:0] PO1;
    reg [3:0] counter;  wire [3:0] a;

1:  assign a = PI1*4;
    always@(counter or a or PI2)
2:      if( counter < PI2 )
3:          PO1 = 4;
        else
4:          PO1 = a;
    always@(posedge clk or posedge reset)
5:      if(reset)
6:          counter = 1;
        else
7:          counter = counter + 1;
```
(a)

```
#1  reset=1; PI1=0; PI2=4;
#1  reset=0;
#5  PI1=1;
#10 PI1=2; PI2=6;
#10 $finish;
```
(b)

```
Time = 1
counter=1, "counter<PI2", PO1=4
Time = 5
counter=2, "counter<PI2", PO1=4
Time = 15
counter=3, "counter<PI2", PO1=4
Time = 25
counter=4, "counter<PI2", PO1=4
```
(c)

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



```
Time = 1
counter=1, "counter<PI2", PO1=4
Time = 5
counter=3, "counter<PI2", PO1=4
Time = 15
counter=5, "counter>PI2", PO1=4
Time = 25
counter=7, "counter>PI2", PO1=8
```
(a)

```
(1) S1 : assign a = PI1 * 4;
(1) S4 : PO1 = a;
(1) S6 : counter = 1;
(2) S3 : PO1 = 4;
(2) S7 : counter = counter + 2;
(3) S2 : if(counter < PI2)
(3) S5 : if(reset)
```
(b)

Fig. 2. Erroneous simulation results and *debugging priority*. (a) Erroneous simulation result. (b) Debugging priority and the CS.

of correctness of the error candidates. However, many HDL operations can actually mask the erroneous effects and prevent them from appearing at the POs. If the erroneous effects were masked, preventing them from being observed at the POs, the SSs may get CS points, even if some of them have design errors hiding within. An example follows to illustrate this point.

Suppose that the HDL code that a designer intends to write is the Verilog HDL code in Fig. 1(a). This HDL code has only one PO, i.e., PO1, on which the simulation results are compared against the expected values. The clock period of the clock signal *clk* is assumed to be 10 ns. If the HDL code is simulated with the input stimulus shown in Fig. 1(b), the simulation result obtained is shown in Fig. 1(c). We consider the result in Fig. 1(c) as the golden result, because the code in Fig. 1(a) is what the designer intends to write.

However, if the statement in line 7 (which is denoted as S7) "$counter = counter + 1$" is carelessly written as "$counter = counter + 2$," the simulation result will become that shown in Fig. 2(a). It can be seen that the simulation value of PO1 at $t = 25$ in Fig. 2(a) is different from its expected value shown in Fig. 1(c). According to the definition in [21], PO1 is an erroneous PO, and the clock cycle ranging from $Time = 15$ to $Time = 25$ is the error-occurring clock cycle (EOC). By using the error space identification approach in [21], an error space, i.e.. $\{\mathrm{S1}, \mathrm{S2}, \mathrm{S3}, \mathrm{S4}, \mathrm{S5}, \mathrm{S6}, \mathrm{S7}\}$, can be obtained.

After obtaining the *error space*, the CS for each error candidate will be calculated. Each SS of a PO with correct simulation at a time instance before the EOC gets one CS point. Finding SSs requires backward tracing from the POs in the reverse direction of the data flow until the primary inputs (PIs), registers, or constants are reached. When reaching a conditional vertex, such as S2 and S5 in Fig. 1(a), the authors propose to traverse the taken branch(es) and the control signal, and ignore the untaken branch(es). For example, at $Time = 1$, since the evaluation result of "if(reset)" at $Time = 1$ is TRUE, the traversal reaches S5 and then backward traverses the TRUE branch and the control signal (i.e., reset). The obtained SSs for PO1 at $Time = 1$ are $\{\mathrm{S5}, \mathrm{S6}\}$, and both receive one CS point. As can be seen, all the traversals must commence with one PO. Each PO traversal is completed in a particular simulation instance. This process is repeated until all the PO traversals in a particular simulation instance have been completed. Finally, once all the PO traversals in all the simulation instances have been completed, the *debugging priority* shown in Fig. 2(b) is obtained. The numbers within parentheses are the CSs of the corresponding error candidates.

It can be seen that the design error in statement S7 "$counter = counter + 2$" is not placed in the first line but the fifth line in Fig. 2(b). If circuit designers examine error candidates according to this *debugging priority*, four trials would be wasted before the true error S7 can be found. Design error S7 is placed in the fifth line, because S7 receives two CS points; this is because the erroneous values caused by S7 are masked twice on its way propagating to PO1.

The first *masking error* situation occurs at $Time = 5$. It can be seen that the erroneous statement S7 causes an incorrect value (i.e., 3, which is different from the correct value shown in Fig. 1(c), i.e., 2, as highlighted using an underline) to be displayed on the signal *counter* at $Time = 5$. However, this incorrect value 3 is masked by the operation "$counter <$ PI2" in S2, because both the correct (i.e., 2) and incorrect values (i.e., 3) yield the same result at the output of the operation "$counter <$ PI2," i.e., they are both smaller than the value of PO2 (i.e., 4). A similar *masking error* situation occurs at $Time = 15$ although the incorrect value of *counter* is propagated through the output of "$counter <$ PI2," causing it to be FALSE. However, the incorrect result (i.e., FALSE) does not alter the value of PO1 (i.e., 4). This means that signal a is 4 at $Time = 15$. It is masked by the conditional operation "if $(\ldots)\ldots$else$\ldots$" and cannot cause incorrect values at PO1 at $Time = 15$. Because the CS does not consider the possible *masking error* situation that may be caused by the operation "$counter <$ PI2" and the conditional operation "if $(\ldots)\ldots$else$\ldots$," S7 is given a CS score of two points. This puts S7 in line 5 in the candidate list in Fig. 2(b). The accuracy of the *debugging priority* is reduced due to the lack of consideration for the *masking error* situation of the CS.

Observing the *masking error* situation, this work aims at estimating the likelihood of error masking (LOEM) for an SS to assess the score it can receive. If the LOEM of an arbitrary SS, i.e., SS_i, is quite low, it is almost impossible for the *masking error* situation to occur on SS_i's way to the POs. It should be comparatively safe to consider SS_i as a correct statement and give SS_i a high score. On the contrary, if the LOEM of SS_i is high, it should be given a low score.

III. PCS

In the following introduction, the input-faulty HDL design 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 the vertices. 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 to which v corresponds. Variable y_v is the output variable of f_v or the *left-hand variable* of the operation. The Verilog HDL code fragment in Fig. 1(a) is used as an example, and its CDFG is constructed, as shown in Fig. 3. Vertex "$1 : *$" corresponds to operation "$a = \mathrm{PI1} * 4$" in the statement in line 1 (S1). Function $f_{1:*}$ is multiplication "$(*)$," and $y_{1:*}$ is signal a. Vertex "$2 :$ if $(\ldots)\ldots$else$\ldots$." corresponds to the operation "if $(\ldots)\ldots$else$\ldots$" in lines 2–4. The functionality of vertex "$2 :$ if $(\ldots)\ldots$else$\ldots$" is quite similar to that of a multiplexer. Vertex PO1 is a special vertex representing the only PO of the circuit, i.e., PO1.

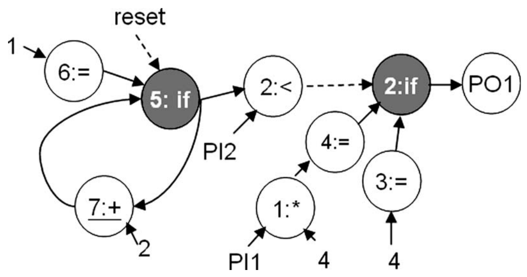


Fig. 3. CDFG of the HDL code in Fig. 1.

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. 3, 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. The *fan-in* of v is a set of vertices k, such that there is an edge from k to v. A path P from vertex u to vertex u' is a sequence of vertices $\langle v_0, v_1, v_2, \ldots, v_k\rangle$, such that $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in \boldsymbol{E}$.

Suppose that the verification finds an incorrect circuit behavior at the nth positive edge of the clock signal $t = c_n$.[2] This special positive edge of the clock is called the error-occurring edge (EOE). Assume that the faulty design under verification (DUV) has m POs $\{\mathrm{PO}_1, \mathrm{PO}_2, \ldots, \mathrm{PO}_m\}$ and that $n - 1$ clock cycles pass before the EOE $(t = c_n)$. To show how we model the *masking error* situation and estimate LOEM, we first consider that a design error hides within an arbitrary statement v. If the erroneous statement v caused an incorrect value w on its left-hand variable y_v at time instance $t = t_i$, this incorrect value w would not cause any incorrect behaviors at any POs at all the rising edges of the clock before $t = c_n$. Otherwise, the EOE is not $t = c_n$ but another earlier rising edge of the clock. More specifically, for an arbitrary PO_j at an arbitrary rising edge of clock $t = c_k$ before the EOE, the incorrect value w is *masked* by some vertices on the paths from statement v at $t = t_i$ (which is denoted as $v@t = t_i$) to PO_j at $t = c_k$ (which is denoted as $\mathrm{PO}_j@t = c_k$), causing the

[2]We assume that the simulation values of all the POs are compared with the correct values only on the rising edges of the clock signal. If the DUV is a falling-edge-triggered or double-edge-triggered design, the modeling and computation algorithm can easily be changed to fit to it.

simulation value of PO_j to be the same as the correct value at $t = c_k$, i.e.,

$$f_{v@t=t_i \rightarrow \mathrm{PO}_j@t=c_k}(w) = CV(\mathrm{PO}_j@t = c_k) \tag{1}$$

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

For all the other POs of the DUV, the incorrect value w would also be *masked* on the way to the POs at all the rising edges before the EOE, so that it could remain uncovered before EOE. That is, for each PO PO_j at each rising edge of clock $t = c_k$ before EOE, the function of the path(s) from vertex v at $t = t_i$ to PO_j at $t = c_k$ must generate the correct value of PO_j at $t = c_k$ with w, even if w is an incorrect value. The preceding description can be modeled as

$$\bigcap_{j=1}^{m} \bigcap_{k=0}^{n-1} f_{v@t=t_i \rightarrow \mathrm{PO}_j@t=c_k}(w) = CV(\mathrm{PO}_j@t = c_k). \tag{2}$$

We now consider the likelihood that the incorrect value w truly exists on y_v but is masked from causing any incorrect values on the POs at any time instances before the EOE. We first notice that all the possible values of y_v that can satisfy (2) form a special set of values. We call this the masked value set (MVS) of vertex v at time instance $t = t_i$, which is denoted as $\mathrm{MVS}(v@t = t_i)$ and is given by

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

Each element in $\mathrm{MVS}(v@t = t_i)$ retains the correct values of all the POs at all the rising edges of the clock before the EOE, regardless if it is a correct value or not. The correct value of the output of vertex v at $t = t_i$ is, of course, contained in $\mathrm{MVS}(v@t=t_i)$. This justifies the existence of $\mathrm{MVS}(v@t=t_i)$. If $\mathrm{MVS}(v@t= t_i)$ contains only one element, obviously, it will be the correct value of y_v at $t = t_i$. In this case, no incorrect values ever exist in $\mathrm{MVS}(v@t = t_i)$, and the *masking error* situation can never occur. Statement v at $t = t_i$ is given a high score. On the other hand, if the set contains many elements, an incorrect value is very likely to exist in the set and become an incorrect value that remains unrevealed at all the rising edges of the clock before the EOE. The correctness of statement v is less obvious. In other words, the more elements $\mathrm{MVS}(v@t = t_i)$ contains, the more likely the simulated value of v at $t = t_i$ is a masked incorrect value. Hence, we define the LOEM of statement v at time instance $t = t_i$ as follows:

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

Its complement is the likelihood that an erroneous value of v at $t = t_i$ is propagated to at least one PO before the EOE and observed, i.e., the likelihood of error propagating (LOEP) of v at $t = t_i$, which is given by

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

In the given input value change dump file, the output variable y_v of an arbitrary statement v can change its value many times, e.g., l times, at different time instances before the EOE $\{t = t_1, t = t_2, \ldots, t = t_l\}$. Each time the value of y_v changes at time instance $t = t_i$, there will be one particular value of $\mathrm{LOEP}(v@t = t_i)$. The PCS of v, i.e., $\mathrm{PCS}(v)$, is defined as the maximum among these LOEP values, as described in

$$\mathrm{PCS}(v) = \max \{\mathrm{LOEP}(v@t = t_i)\}, \quad \text{where } t_i \in \{t = t_1, t = t_2, \ldots, t = t_l\}. \tag{6}$$

A low LOEP (high LOEM) means that any erroneous effects caused by v at $t = t_i$ are very possible to be masked. The correctness of v at $t = t_i$ may become doubtful, even if the simulation values of all the POs are correct before the EOE. It is reasonable to give v less PCS due to its small LOEP value. On the other hand, if the LOEP value is high, it is equally reasonable to give it more PCS. Therefore, we define PCS as (6). It can be seen that PCS computation now turns into the problem of efficiently computing the MVSs of each error candidate at different time instances before an EOE.

IV. PCS Computation Algorithm

The proposed PCS computation algorithm is a topology-based analysis with *time frame expansion* for handling the sequential behavior of the DUV. While calculating the LOEP 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 $t = t_i$ to any PO in each time frame before the EOE. This path-oriented computation scheme is defined as follows:

$$\mathrm{MVS}(v@t = t_i) = \bigcap_{j=1}^{m} \bigcap_{k=0}^{n-1} \{x | f_{v@t=t_i \rightarrow \mathrm{PO}_j@t=c_k}(x) = CV(\mathrm{PO}_j@t = c_k)\} \tag{7}$$

which can be derived from (3).

The set $\{x | f_{v@t=t_i \rightarrow \mathrm{PO}_j@t=c_k}(x) = CV(\mathrm{PO}_j@t = c_k)\}$ is defined as the MVS of vertex v at time instance $t = t_i$ with respect to PO_j at $t = c_k$ (which is denoted as $\mathrm{MVS}(v@t = t_i)_{\mathrm{PO}_j@t=c_k}$). 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 path(s) from v at $t = t_i$ to PO_j at $t = c_k$, thus keeping the correct value of PO_j at $t = c_k$.

According to (7), if it is possible to derive $\mathrm{MVS}(v@t = t_i)_{\mathrm{PO}_j@t=c_k}$ for each PO_j at each time frame $t = c_k$, then intersecting these sets yields $\mathrm{MVS}(v@t = t_i)$. If there is exactly one path from v at $t = t_i$ to a PO PO_j at $t = c_k$, an induction-based computation approach is proposed to compute an exact $\mathrm{MVS}(v@t=t_i)_{\mathrm{PO}_j@t=c_k}$, which is introduced in Sections IV-A and IV-B. If there are multiple paths from v at $t = t_i$ to PO_j at $t = c_k$, i.e., *reconvergent paths*, a quick estimation approach that guarantees lower bound LOEP estimations will be applied,

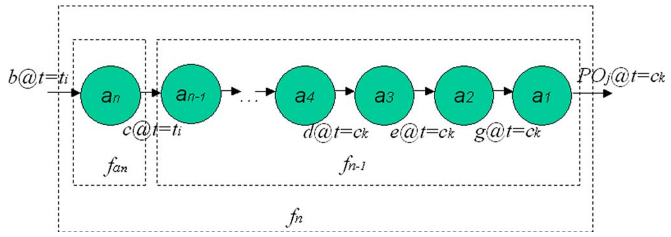


Fig. 4. Path from $b@t = t_i$ to $\mathrm{PO}_j@t = c_k$.

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

MVS_for_Vertex(PreviousMVS, vertex $v$, time $t_i$)
1:    if MVS($v$@$t$=$t_i$) = =$\varnothing$
2:        MVS($v$@$t$=$t_i$) = PreviousMVS
3:    else
4:        MVS($v$@$t$=$t_i$) =
MVS($v$@$t$=$t_i$)$\cap$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:            $t_h$ = $t_i$ – clock_period
10:     if $t_h$ <0
11:         return
12:     Compute CurrentMVS, which is
        {$x$ | $f_v$($x$) $\in$ PreviousMVS}
13:     MVS_for_Vertex(CurrentMVS, $u$, $t_h$)
```

Fig. 5. Pseudocode of the MVS computation for a single path.

which is introduced in Section IV-C. Two time-saving strategies are introduced in Section IV-D. The entire algorithm is presented in Section IV-E and incorporates each part that was introduced before.

A. MVS Computation for a Single Path

Assume that there is a single path P from a vertex b at time instance $t = t_i$ to a PO PO_j at a rising edge of clock $t = c_k \langle b@t = t_i, a_n, a_{n-1}, \ldots, a_2, a_1, \mathrm{PO}_j@t = c_k\rangle$, as shown in Fig. 4. An algorithm written as a pseudocode, as shown in Fig. 5, is used to compute $\mathrm{MVS}(b@t = t_i)_{\mathrm{PO}_j@t=c_k}$.

For each PO at each rising clock edge, the algorithm in Fig. 5 will recursively call subroutine *MVS_for_Vertex* to perform an MVS computation and use a depth-first search strategy for backward traversals (from PO_j at $t = c_k$ to $b@t = t_i$). The input of the subroutine comprises 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 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 the computation of a set of all u's output values (*CurrentMVS*) that can make the function of v, i.e., f_v generate an output value that is in the set *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 the time information after the subroutine is called again. Section IV-B introduces the procedures for computing *CurrentMVS* based on *PreviousMVS* (line 12). The following is an explanation of how this algorithm can derive $\mathrm{MVS}(b@t = t_i)_{\mathrm{PO}_j@t=c_k}$, in the case of a *single path* from b at $t = t_i$ to PO_j at $t = c_k$.

Theorem 1: As shown in Fig. 4, function f_n is the composite function of the vertices from a_1 to a_n and comprises f_{a_n} and f_{n-1}. For an arbitrary value x on the output of vertex b at $t = t_i$, x is in $\mathrm{MVS}(b@t = t_i)_{\mathrm{PO}_j@t=c_k}$ if and only if $f_{a_n}(x)$ is in $\mathrm{MVS}(c@t = t_i)_{\mathrm{PO}_j@t=c_k}$, which can be represented as

$$\mathrm{MVS}(b@t = t_i)_{\mathrm{PO}_j@t=c_k} = \left\{x|f_{a_n}(x) \in \mathrm{MVS}(c@t = t_i)_{\mathrm{PO}_j@t=c_k}\right\}. \quad (8)$$

Proof: **Claim 1**

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

For each value x contained in $\{x \mid f_{a_n}(x) \in \mathrm{MVS}(c@t = t_i)_{\mathrm{PO}_j@t=c_k}\}$, x must satisfy $f_{n-1}(f_{a_n}(x)) = CV(\mathrm{PO}_j@t = c_k)$ and thus also satisfy $f_n(x) = CV(\mathrm{PO}_j@t = c_k)$. That is, x is contained in $\mathrm{MVS}(b@t = t_i)_{\mathrm{PO}_j@t=c_k}$. This proves Claim 1.

Claim 2

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

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

From Claims 1 and 2, it is proven that

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

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

From the preceding discussion, it shows that a *CurrentMVS* set is the MVS of a traversed vertex with respect to PO_j at $t = c_k$. According to (7), these MVSs will be intersected by other MVSs of the same vertex with respect to the other POs at different time instances. After all the POs at all the positive clock edge have been applied, the MVS of each traversed vertex in a time frame will be computed and recorded for the PCS calculation later in the process.

B. MVS Formula for Operations

Given a previously computed MVS set (*PreviousMVS*), vertex v, and one of the *fan-in* vertices u of v, *CurrentMVS* is the set of all the values at u's output y_u that make the function of vertex v, i.e., f_v, generate an output value that is in *PreviousMVS*. To compute *CurrentMVS*, we first consider a particular value p in *PreviousMVS* and try to 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 for each p in *PreviousMVS* yields *CurrentMVS*. We denote this special set for value p as $Sub\_CurrentMVS_p$.

For most *unary* and *binary* operations, $Sub\_CurrentMVS_p$ can easily be derived by inversing f_v. 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. Integer b_1 is the simulated value of an operand other than the output of u y_u and is recorded in the dump file. The formula for computing $Sub\_CurrentMVS_p$ is summarized in the third column of Table I. The second column shows the necessary conditions for the result of $Sub\_CurrentMVS_p$ to exist. If the conditions are not met, in most of cases, $Sub\_CurrentMVS_p = \{\varnothing\}$, except in *comparisons*. The derivation of $Sub\_CurrentMVS_p$ formulas for some representative operations is explained here.

1) Operations that choose a bit range ("$[i]$" and "$[i:j]$"): For operation "$[i:j]$," the only constraint on the input values is that the binary values of the bits selected by "$[i:j]$" must be the same as the those of value p. The binary values of the unselected bits are arbitrary and can be of any possible value. Thus, the value of the unselected bits from 0 to $j-1$ can be any integer ranging from 0 to $2^j - 1$. The value of the unselected bits from $i+1$ to $w-1$ can be any integer ranging from 0 to $2^{w-i-1} - 1$. Hence, the formula for operation "$[i:j]$" appears in the third column of Table I. $Sub\_CurrentMVS_p$ for "$[i]$" can be derived by treating i the same as j in the "$[i:j]$" formula.
2) Control vertices ("if $(\ldots)\ldots$else$\ldots$" and "$case(\ldots)$"): If y_u is the control signal, y_u can only have values that select suitable branches to keep the output of vertex v y_v at p. This computation 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 have 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 a *right shift* or a *left shift* by b_1 incurs *information loss*. The "$[i:j]$" formula can tackle the *information loss*. As shown in Fig. 6, 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 a *right shift* (*left shift*) operation, the *divide* (*multiply*) formula is first applied; then, the "$[i:j]$" formula is applied. For formula derivation of other operations, such as "+," "$-$," and "$*$," the "$[i:j]$" MVS formula can also be applied to model the *information loss* if encountered.

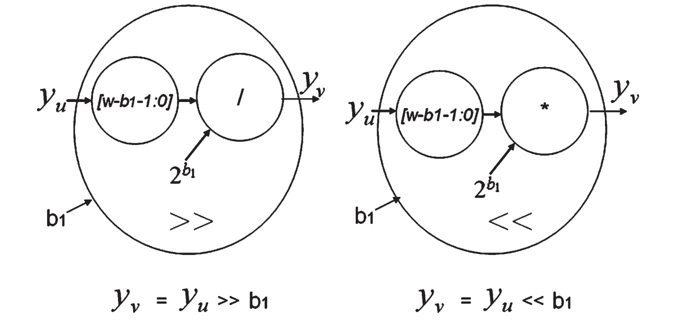


Fig. 6. Modeling information loss in the right and left shifts.

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 before deriving the union of all the $Sub\_CurrentMVS_p$ to produce *CurrentMVS*. Take the operation "$b = a[1:0]$" as an example. Assume that a is four bit wide, b is two bit wide, and $PreviousMVS = \{0, 1, 2\}$. To compute *CurrentMVS*, first, the "$[i:j]$" formula with $i = 1$, $j = 0$, $w = 4$, and $p = 0$ is applied, which yields

$$\bigcup_{k=0}^{2^{4-1-1}-1} \{[0 \cdot 2^0 + k \cdot 2^{1+1} \sim 0 \cdot 2^0 + k \cdot 2^{1+1} + 2^0 - 1]\} = \{0, 4, 8, 12\}. \quad (9)$$

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 much time if there are many elements in *PreviousMVS*. The following observations are used to refine the *Sub_CurrentMVS* formulas for deriving the MVS formulas used in the subroutine *MVS_for_Vertex*.

Taking a closer look at the results obtained with $p=0$, $p=1$, and $p=2$, it can be observed 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, as well as $1 * 2^0 + k * 2^{1+1} + 2^0 - 1 = k * 2^{1+1} + 1$ and $2 * 2^0 + k * 2^{1+1} = k * 2^{1+1} + 2$. Therefore, the union of the preceding three sets can more concisely be represented as

$$\bigcup_{k=0}^{2^{4-1-1}-1} \{[0 \cdot 2^0 + k \cdot 2^{1+1} \sim 2 \cdot 2^0 + k \cdot 2^{1+1} + 2^0 - 1]\} = \bigcup_{k=0}^{3} \{[k \cdot 4 \sim 2 + k \cdot 4]\}. \quad (10)$$

TABLE I
$Sub\_CurrentMVS_p$ FORMULAS

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

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 $[i \sim j]$ means a set of continuous integers from integer i to integer j.

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

$$\bigcup_{k=0}^{2^{w-i-1}-1} \left\{[p \times 2^j + k \times 2^{i+1} \sim q \times 2^j + k \times 2^{i+1} + 2^j - 1]\right\}. \tag{11}$$

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

The "$\ll$" operation is another example of the derivation of the "$\ll$" formula listed in the third column of Table II. First, the smallest integer p' in the set $\{[p \sim q]\}$ is found, which satisfies $p' \% 2^{b_1}[b_1] = 0$. If there is no such p' in the set $\{[p \sim q]\}$, *CurrentMVS* will be ϕ. If p' exists in $\{[p \sim q]\}$, $p' + 2^{b_1}$ needs to be is in the range of $p{-}q$. If it is, the union of the two result sets obtained by p' and $p' + 2^{b_1}$ can be expressed as

$$\bigcup_{k=0}^{2^{b_1}-1} \left\{[k \cdot 2^{b_1} + p'/2^{b_1} \sim k \cdot 2^{b_1} + (p' + 2^{b_1})/2^{b_1}]\right\}. \tag{12}$$

Repeating the preceding derivations produces the "$\ll$" formula in Table II.

For a subset of integers $\{[p \sim q]\}$ in *PreviousMVS*, applying the MVS formulas listed in the third column of Table II can much more quickly produce results than the application of 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 and alleviates the problem of memory explosion.

C. *MVS Estimation for Reconvergent Paths*

The algorithm shown in Fig. 5 can compute the exact MVS of vertex b in time frame $t = t_i$ with respect to PO$_j$ in time frame $t = c_k$ only if there is a *single path* from b at $t = t_i$ to PO$_j$ at $t = c_k$. If there are multiple *reconvergent paths*, another approach is needed.

A quick estimation strategy for handling *reconvergent paths*, instead of some other complex methods that may need more computation time, is adopted in this paper. If there are multiple *reconvergent paths* from v at $t = t_i$ to PO$_j$ at $t = c_k$, the universe $\boldsymbol{U}$ is used, instead of real MVS$(b@t = t_i)_{\mathrm{PO}_j@t=c_k}$ in the intersection operation. The estimation result obtained using the universe must include the exact result obtained by intersecting the real MVS$(b@t = t_i)_{\mathrm{PO}_j@t=c_k}$, as the latter is included in the universe $\boldsymbol{U}$. Consequently, this estimation result has a larger MVS set. That causes the estimated LOEP to be less than the real one. Therefore, this estimation approach guarantees lower bound estimations of LOEP for an arbitrary error candidate v on *reconvergent paths*. This reduces the possibility of the correctness of any design errors to be overestimated and the design errors to be placed in the first few lines of a *debugging priority*.

TABLE II
MVS FORMULAS FOR HDL OPERATIONS

Operation	Condition	*Sub_CurrentMVS[p~q]*
$y_v = y_u$	-	$\{[p \sim q]\}$
$y_v = \sim y_u$	-	$\{[2^w - 1 - q \sim 2^w - 1 - p]\}$
$y_v = - y_u$	-	$\{[2^w - q \sim 2^w - p]\}$
$y_v = y_u\,[i{:}j]$	-	$\bigcup_{k=0}^{2^{w-i-1}-1} \{[p \cdot 2^j + k \cdot 2^{i+1} \sim q \cdot 2^j + k \cdot 2^{i+1} + 2^j - 1]\}$
$y_v = y_u\,[i]$	-	$\bigcup_{k=0}^{2^{w-i-1}-1} \{[p \cdot 2^i + k \cdot 2^{i+1} \sim q \cdot 2^i + k \cdot 2^{i+1} + 2^i - 1]\}$
$y_v = y_u + b_1$	-	$\{[p\text{-}b1 \sim q\text{-}b1]\}$
$y_v = y_u - b_1$	-	$\{[p+b1 \sim q+b1]\}$
$y_v = b_1 - y_u$	-	$\{[b1\text{-}q \sim b1\text{-}p]\}$
$y_v = y_u * 0$	$p == 0$	$\{[0 \sim 2^w - 1]\}$
$y_v = y_u * b_1\ (b_1 > 0)$	$\lfloor p/b_1 \rfloor < \lfloor q/b_1 \rfloor$	$\{[\lfloor p/b_1 \rfloor \sim \lfloor q/b_1 \rfloor]\}$
$y_v = y_u \% b_1$	$q < b_1$	$\bigcup_{k=0}^{\lfloor (2^w - 1)/b_1 \rfloor} \{[k \cdot b_1 + p \sim k \cdot b_1 + q]\}$
$y_v = y_u >> b_1$	-	$\{[p \cdot 2^{b_1} \sim q \cdot 2^{b_1} + 2^{b_1} - 1]\}$
$y_v = b_1 >> y_u$	$\lfloor b_1/q \rfloor < \lfloor b_1/p \rfloor$	$\{[\log_2 \lfloor b_1/q \rfloor \sim \log_2 \lfloor b_1/p \rfloor]\}$
$y_v = y_u << b_1$	$\lvert p/2^{b_1} \rvert < \lfloor q/2^{b_1} \rfloor$	$\bigcup_{k=0}^{2^{b_1}-1} \{[k \cdot 2^{b_1} + \lceil p/2^{b_1} \rceil \sim k \cdot 2^{b_1} + \lfloor q/2^{b_1} \rfloor]\}$
$y_v = b_1 << y_u$	$\lfloor p/b_1 \rfloor < \lfloor q/b_1 \rfloor$	$\{[\log_2 \lfloor p/b_1 \rfloor \sim \log_2 \lfloor q/b_1 \rfloor]\}$
$y_v = y_u > b_1$	$p == 0$ 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*.

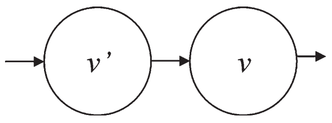


Fig. 7. Vertex v and one of its *fan-in* vertex v'.

D. Time-Saving Strategies

To reduce the computation time, we develop the *bounded traversal* strategy and the limited traversed frame (LTF) strategy. The *bounded traversal* strategy can avoid unnecessary traversals during MVS computation without causing any accuracy loss. The LTF strategy saves additional time at the expense of accuracy loss. It provides additional flexibility for tool users to determine how they like to trade off between accuracy and time.

1) Bounded Traversal Strategy: In the proposed MVS computation, after some backward traversals, there are MVS sets recorded on vertices that have been traversed. As shown in Fig. 7, let vertex v' at $t = t_n$ be one of the *fan-in* vertices of vertex v at $t = t_n$, and both were also traversed. Assume that MVS$(v@t = t_n)$ and MVS$(v'@t = t_n)$ are already recorded on v and v'. MVS$(v'@t = t_n)$ should be $\{x \mid f_v(x) \in$ 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 a PO arrives at vertex v in time frame $t = t_n$, *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 to be MVS$(v@t = t_n)$, i.e., MVS$(v@t = t_n) \subseteq PreviousMVS$, then, when the computation reaches v', the result of the intersection will also be MVS$(v'@t = t_n)$. More specifically, 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 statement and proof.

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

Proof: The MVS$(v'@t{=}t_n)$ is computed based on 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 is thus also in *PreviousMVS* since MVS$(v@t{=}t_n) \subseteq PreviousMVS$. Therefore, 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.

If v' has at least one *fan-in* vertex v'', by *mathematical deduction*, MVS$(v''@t = t_n)$ should also remain unchanged

```
PCS_computation (DUV, Dumpfile, EOE, Error space, frame_limit)
1: 3-address Code Generation and Conditional statement odification  // Preparation Phase 1
2: CDFG Construction                                                 // Preparation Phase 2
3: Initialize each vertex as "untraversed"
4: for each positive edge of clock $t=c_k$ before EOE
5:   for each primary output $PO_j$
6:     InitialMVS = $\{CV(PO_j@t=c_k)\}$; Find the fanin vertex $a_1$ of $PO_j$ at $t=c_k$
7:     MVS_Com_for_Vertex (InitialMVS, $a_1$, $PO_j$, $c_k$, $c_k$, frame_limit)
8: Calculate PCS with the computed MVSs and derive debugging priority

MVS_Com_for_Vertex(PreviousMVS, vertex v, StartPO, StartTime, time $t_j$, frame_limit )
//*** Modification for incorporating MVS computation for reconvergent paths ***
1: if traversed for first time in traversal starting from StartPO at StartTime
2:   if MVS($v@t=t_j$) = = $\varnothing$
3:     MVS($v@t=t_j$) = PreviousMVS
4:   else
5:     if MVS($v@t=t_j$) $\subseteq$ PreviousMVS  //**Condition of Bounded traversal
6:       return
7:     MVSforRecovery($v@t=t_j$) = MVS($v@t=t_j$)
8:     MVS($v@t=t_j$) = MVS($v@t=t_j$) $\bigcap$ 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-Traversed-Frame
17:          return
18:        frame_limit - -
18:      Compute CurrentMVS, which is $\{ x \mid f_v(x) \in$ PreviousCVS $\}$
19:      MVS_Com_for_Vertex(CurrentMVS, u, StartPO, StartTime, $t_h$, frame_limit)
```

Fig. 8. Pseudocode of the PCS computation algorithm.

after the intersection. So do the vertices that are in the transitive *fan-in* of vertex v. Therefore, when *PreviousMVS* includes the recorded MVS of a vertex v, an immediate return from subroutine *MVS_for_Vertex* can avoid unnecessary traversals and computations since further computations will not change any recorded MVSs.

2) LTF Strategy: The *bounded 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 the LTF strategy, which provides an optional and flexible tradeoff between accuracy and speed.

The idea of the 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 (which is denoted as the $frame\_limit$). $frame\_limit$ is a configurable parameter that can be adjusted by the users. It can be set as a small number for a quick estimation or as infinite for the highest accuracy. Unlike the *bounded traversal* strategy, 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 here.

For a vertex u in time frame $t = c_k$, if the expanded frames are not limited, each MVS of u at $t = c_k$ will be intersected with respect to an observation point at a positive clock edge in the set of MVS sets $\{\mathrm{MVS}_1, \mathrm{MVS}_2, \ldots, \mathrm{MVS}_m\}$. With the $frame\_limit$ restriction, some MVSs 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 $\{\mathrm{MVS}_1, \mathrm{MVS}_2, \ldots, \mathrm{MVS}_n\}$, where $n < m$. The intersection of all the MVSs in the set $\{\mathrm{MVS}_1, \mathrm{MVS}_2, \ldots, \mathrm{MVS}_n\}$ includes the intersection of all the MVSs in the set $\{\mathrm{MVS}_1, \mathrm{MVS}_2, \ldots, \mathrm{MVS}_m\}$. 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. PCS Computation Algorithm

To incorporate MVS estimation for *reconvergent paths* and the time-saving strategies, we modify the algorithm shown in Fig. 4 and derive the one abstracted as pseudocode in Fig. 8. Hence, the algorithm that we use to calculate PCS incorporates the following: 1) MVS computation for a *single path*; 2) MVS estimation for *reconvergent paths*; 3) the bounded-traversal strategy; and 4) the LTF strategy. The input of this algorithm are given as follows: 1) the DUV described in an HDL; 2) the value change dump file during simulation; 3) the EOE; 4) an *error space* obtained by any *error space* identification approach; and 5) the $frame\_limit$ value selected by the tool user.

During traversal(s) that starts from a PO (*StartPO*) at a time instance (*StartTime*), if vertex v is visited for the first time, the *single-path* case is temporarily assumed. The *PreviousMVS* will be intersected with $\mathrm{MVS}(v@t = t_i)$, which is already the intersection of many *PreviousMVS*s. However, if this vertex v is found to be traversed for two or more times in the traversal starting from *StartPO* at *StartTime*, there are *reconvergent paths* from v at $t = t_i$ to *StartPO* at *StartTime*. Then, the previously

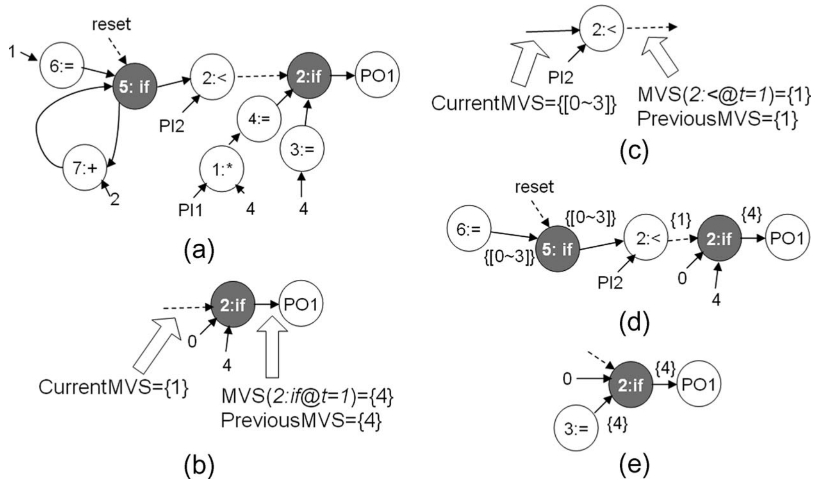


Fig. 9. Computation processes starting from PO1 at $t = 1$.

recorded $MVSforRecovery(v@t = t_i)$ is used to cancel the intersections made in this traversal before.

Two conditions are added for incorporating the two time-saving strategies. The condition in line 5 of the *MVS_Com_for_Vertex* subroutine is added to incorporate the *bounding traversal* strategy. The last condition in line 16 is added because of the LTF strategy. Once one of the conditions is met, the succeeding computation processes can be skipped, and the program can directly return from the subroutine to save computation time. Aside from 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.

The preparation phases of this algorithm are shown in lines 1 and 2 of *CS_computation*. The *three-address code generations* and the *conditional statement modification* developed in [22] must be conducted first for the information required in the MVS computation for control vertices (conditional statements). The detailed *conditional statement modification* algorithm can be found in [22]. Next, a CDFG based on the input DUV described in HDL is constructed.

The example in Fig. 1 is used to demonstrate the processes of our PCS computation and its performance in the derivation of a *debugging priority*. After some initializations, the CDFG of the DUV based on the HDL code in Fig. 1 is constructed, as shown in Fig. 9(a). Then, a backward traversal from PO1 at $t = 1$ commences by calling subroutine *MVS_Com_for_Vertex*, with the inputs $PreviousMVS = \{4\}$, vertex $v =$ "2 : if", $StartPO =$ PO1, and $StartTime = 1$.

When subroutine *MVS_Com_for_Vertex* is called for the first time, the traversal also reaches vertex "2:if" in time frame $t = 1$ for the first time. As shown in Fig. 9(b), the recorded MVS$(2 : \text{if}@t = 1) = \{4\}$, 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. We decide to traverse "2 :<" before traversing "3 :=" and compute *CurrentMVS*. Because *PreviousMVS* is $\{4\}$, the MVS computation for the conditional statements will be used, and we obtain *CurrentMVS* $\{1\}$. The computation process is shown in Fig. 9(b).

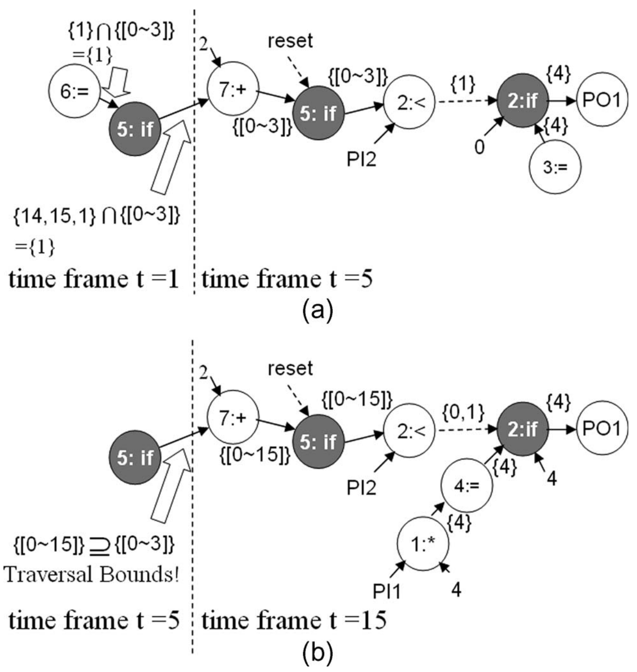


Fig. 10. Computation starting from PO1 at $t = 5$ and $t = 15$.

Subroutine *MVS_Com_for_Vertex* is then called for the second time, and "2 :<" in time frame $t = 1$ is reached. The computation status is shown in Fig. 9(c). Similar computations are recursively repeated vertex by vertex until vertex "6 :=" in time frame $t = 1$ is reached. The computation results along the traversal from "2:if" to "6 :=" are shown in Fig. 9(d), where each set of integers next to 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." The repetitious calling of subroutine *MVS_Com_for_Vertex* can produce the results shown in Fig. 9(e). After completing the traversals and MVS computations starting from PO1 in time frame $t = 1$, continuing the backward-traversal-based MVS computation from PO1 in time frame $t = 5$ can produce the results shown in Fig. 10(a) and (b). When the computation reaches vertex "5:if" in time frame

(0.8)	S7 :	counter = counter + 2;
(1.0)	S1 :	assign a = PI1 * 4;
(1.0)	S4 :	PO1 = a;
(1.0)	S6 :	counter = 1;
(1.0)	S3 :	PO1 = 4;
(1.0)	S2 :	if(counter < PI2)
(1.0)	S5 :	if(reset)

Fig. 11. *Debugging priority* and the PCS.

$t = 1$, *PreviousMVS* $\{[0 \sim 15]\}$ will include $\text{MVS}(5 : \text{if@}t = 1) = \{[0 \sim 3]\}$. The bounded traversal condition is satisfied, and the traversal is bounded here.

After all the POs at all the positive clock edges before the EOE are applied in the MVS computation, the PCS is calculated with formulas based on the computed MVSs. With the PCS, a *debugging priority* with the PCS (in round brackets) is obtained, as shown in Fig. 11. It can be seen that the design error S7 is displayed in the first line. A search for design errors according to this *debugging priority* will immediately succeed. In the experimental results in Section V, it is also proven that the proposed PCS can efficiently deliver the *debugging priority* with high accuracy, which greatly reduces both the time and effort required for design error searches in the input *error space*.

V. Experimental Result

The experiments were conducted on a subset of the ITC'99 benchmark [23] and four other designs written in Verilog HDL. The four designs are given as follows: pcpu is a simple 32-bit pipelined DLX central processing unit; div16 is a 16-bit divider; blkJ is a controller of a black jack card game, and Mtrx implements a two-by-two matrix multiplication. The number of lines (#line) of all the designs and the number of variables (#var) are presented in Table III.

For every design case, one statement is randomly chosen and injected with an artificial design error based on typical bugs that designers usually induce [24]. With the injected error, a simulation is run until some incorrect values occur on the POs. Then, the *error space* identification approach proposed in [21][3] is applied to obtain an *error space*. After that, the CS calculation in [21] and our PCS calculation are both applied to derive two respective debugging priorities for the same *error space*.

With a *debugging priority*, the error candidates that a digital circuit designer has to examine before he/she can find a design error are often less than those with blind searching. In a sense, we can think that the size of the *error space* is thus reduced, as a result of the *debugging priority*. With respect to the two debugging priorities, since the injected error may have two different ranked orders, the effectiveness of the two debugging priorities on the size reduction of the same *error space* is also different. To compare the effectiveness of the two debugging priorities, a quantitative metric called the effective size ratio (ESR) is formulated as "the rank of the injected error divided by the number of error candidates in the *error space*." The two debugging priorities sorted with the CS and PCS have their own different ESRs. A smaller ESR means that the error has a better rank with respect to the size of the *error space*. This also implies that the effective size reduction contributed by the corresponding *debugging priority* is larger and that the efforts required for design error searching in the *error space* are less.

With each design case, the aforementioned experimental processes are repeated 50 times. In each repetition, the ESRs of the CS and PCS are calculated and recorded. The average ESR values with respect to the CS and PCS are also presented in the "Avg_ESR_CS" and "Avg_ESR_PCS" columns, respectively. The number of times in which the ESR values of the CS and PCS appear in three ESR value ranges is also recorded to show the distribution of the ESR values. The three ESR ranges are "$\text{ESR} < 0.2 (0 \sim 0.2)$," "$0.2 < \text{ESR} < 0.5\ (0.2 \sim 0.5)$, " and "$0.5 < \text{ESR} (0.5 \sim 1.0)$." The number of times is presented in the "#case_CS" and "#case_PCS" columns.

In Table III, it can be observed that, when the PCS is used to obtain a *debugging priority*, in all the design cases, the average ESR values are all less than 0.2 and the average ESR values of the CS. For example, in design "B02," if the CS is used to derive the *debugging priority*, the ESR value is less than 0.2 in 38 of the 50 experiments. In other words, our created errors are placed within the first 20% of the displayed list of error space in 38 of the 50 experiments. However, if the PCS is applied instead, in each of the 50 repetitions, the injected error always appears in the first 20%. If a designer conducts error searching on design "B02," with the *debugging priority* sorted with the PCS, he/she can locate the error by checking less than 20% of the derived error candidates. At least 80% of the searching effort is saved. Moreover, it can be seen that the ESR values of the PCS is never greater than 0.5 in the 50 repetitions. This means that, even in the worst case of the 50 repetitions, a *debugging priority* sorted with the PCS can still save more than half the amount of effort needed for design error searching in the error space. In contrast, the CS method cannot offer this benefit.

From the values of Avg_ESR_PCS and Avg_ESR_CS, it can be observed that the effective size reduction with respect to the PCS is much greater than that with respect to the CS. The ratio of Avg_ESR_PCS to Avg_ESR_CS shown in the "ESR Ratio" column is about 0.49 in average and 0.38 in the best case; this means that, with the PCS, further size reduction of 50%, on average, is possible, and a size reduction of 62% is also achieved in the best case, compared to the CS.

In design cases "B01" and "B02," it can be observed that the ESR ratio is not as small as that in other cases. In these two cases, there are few operations that can mask erroneous effects. In other words, the erroneous effects are seldom masked in the two cases, making the advantage of calculating the PCS not obvious. However, in most designs, the HDL operations that can mask erroneous effects, such as "$==$," "$>$," "[:],"and "$\gg$," are widely applied. Using the PCS in those cases should be beneficial. As demonstrated in other design cases, the proposed PCS method saves more time in the error-searching process than the CS method, showing that the PCS is more suitable to derive the debugging priority in most cases.

Another interesting observation is also found in our experiments. In the design case "pcpu," we observe that the design

[3]We apply the *error space* identification method proposed in [22]. However, the proposed PCS is theoretically applicable to any other error candidate identification methods.

TABLE III
COMPARISON OF THE CS AND PCS PERFORMANCES

design name	#line	#var	Confidence Score (CS)					Probabilistic Confidence Score (PCS)					ESR Ratio
			#cases_CS			Avg_ ESR_CS	t(s)	#case_PCS			Avg_ ESR_PCS	t(s)	
			0~0.2	0.2~0.5	0.5~1.0			0~0.2	0.2~0.5	0.5~1.0			
B01	110	7	40	10	0	0.11	0.3	49	1	0	0.07	0.5	0.64
B02	70	5	38	12	0	0.16	0.3	50	0	0	0.11	0.5	0.69
B03	141	21	35	15	0	0.18	0.4	45	5	0	0.09	0.5	0.50
B04	102	19	32	17	1	0.23	0.3	45	5	0	0.11	0.4	0.48
B05	332	25	24	23	3	0.26	1.3	43	7	0	0.10	1.7	0.38
B07	92	11	37	13	0	0.21	0.4	46	4	0	0.09	0.6	0.43
B08	89	23	32	17	1	0.24	0.6	44	6	0	0.10	0.9	0.42
B14	509	27	17	26	7	0.36	3.8	37	13	0	0.15	5.2	0.42
B21	1089	65	14	28	8	0.42	6.7	31	19	0	0.17	9.7	0.40
pcpu	952	54	15	30	5	0.37	4.1	33	17	0	0.16	6.1	0.43
div16	235	11	23	24	3	0.25	0.7	42	8	0	0.12	1.0	0.48
mtrx	80	11	37	13	0	0.19	0.4	50	0	0	0.11	0.6	0.58
rankf	656	48	18	27	5	0.29	3.1	33	17	0	0.17	4.6	0.59

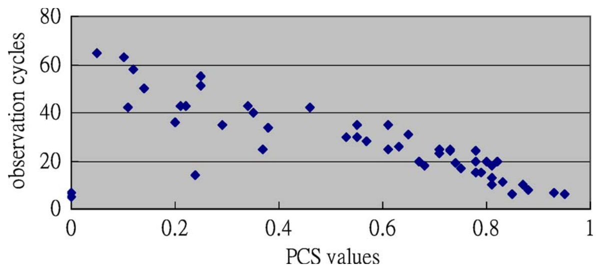


Fig. 12. PCS values of the design errors and *observation cycles*.

errors with smaller PCS values often take more simulation cycles to observe their erroneous effect on the POs, as shown in Fig. 12, after the errors are exercised. Similar correlations also exist in design cases with many operations that can mask erroneous effects in our experiments. However, for small design cases with few operations that can mask erroneous effects, such as design cases "B02" and "B01," such relationship may not be obvious.

The cost of this performance improvement is little computation time, compared to the CS. The computation time needed for the two measurements PCS and CS is presented in the "t(s)" columns. It can be seen that, in the worst case, it takes additional 2 s to obtain the PCS, compared to the time required to obtain the CS (4.1 s). The extra computation time is acceptable if we notice that it should usually take more than 2 s for a designer to examine one error candidate to justify its correctness, but the number of examinations saved as a result of applying the PCS is numerous.

VI. CONCLUSION

In this paper, a probabilistic measurement PCS for deriving an accurate and reliable *debugging priority* for quick error searching among error candidates in an *error space* has been presented. Instead of assuming that the erroneous effects caused by some activated errors are seldom masked, the proposed PCS takes the *masking error* situation into consideration and estimates the LOEP of an error candidate. The idea is, if the LOEP is high, a *masking error* situation is unlikely to occur, and the error candidate is a false candidate with high possibility, i.e., the candidate tends to be a correct statement. On the other hand, if the LOEP is low, occurrence of a *masking error* situation becomes quite possible. Suspicion of the error candidate still remains, and this candidate should thus receive a low PCS score.

The experimental results confirm that the proposed PCS measurement is indeed accurate in estimating the likelihood of correctness of error candidates. In most experimental cases, the created design errors can be located in the first few lines of the candidate list of the input *error space*. As a result, the *debugging priority* sorted with the proposed PCS can effectively speed up the error-searching process in the input *error space*. As compared to the CS, the proposed PCS-based *debugging priority* can save more than half of the effort (or time) needed for the error-searching process in an *error space* in our experiments, at the cost of a little extra computation time. The savings in time contributed by the proposed PCS method is usually much larger than the extra computation time that the PCS calculation needs. Therefore, the gain from the proposed PCS can often outweigh the cost of the extra computation time that the PCS needs.

REFERENCES

[1] C. C. Lin, K. C. Chen, S. C. Chang, and M. M. Sadowska, "Logic synthesis for engineering change," in *Proc. IEEE/ACM Des. Autom. Conf.*, Jun. 1995, pp. 647–652.
[2] H. T. Liaw, J. H. Taih, and C. S. Lin, "Efficient automatic diagnosis of digital circuits," in *Proc. Int. Conf. Comput. Aided Des.*, Nov. 1990, pp. 464–467.
[3] D. Brand, "Incremental synthesis," in *Proc. Int. Conf. Comput. Aided Des.*, Nov. 1994, pp. 14–18.
[4] M. Tomita and H. H. Jiang, "An algorithm for locating logic design errors," in *Proc. IEEE/ACM Des. Autom. Conf.*, Jun. 1990, pp. 468–471.
[5] P. Y. Chung, Y. M. Wang, and I. N. Hajj, "Diagnosis and correction of logic design errors in digital circuits," in *Proc. IEEE/ACM Des. Autom. Conf.*, Jun. 1993, pp. 503–508.
[6] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 10, pp. 1606–1621, Oct. 2005.

[7] S. Y. Huang, K. T. Cheng, K. C. Chen, and D. I. Cheng, "ErrorTracer: A fault simulation-based approach to design error diagnosis," in *Proc. IEEE Int. Test Conf.*, 1997, pp. 974–981.

[8] S. Y. Huang and K. T. Cheng, "ErrorTracer: Design error diagnosis based on fault simulation techniques," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 18, no. 9, pp. 1341–1352, Sep. 1999.

[9] D. W. Hoffmann and T. Kropf, "Efficient design error correction of digital circuits," in *Proc. Int. Conf. Comput. Des.*, Sep. 2000, pp. 465–472.

[10] A. D'Souza and M. Hsiao, "Error diagnosis of sequential circuits using region-based model," in *Proc. 14th Int. Conf. VLSI Des.*, Jan. 2001, pp. 103–108.

[11] N. Sridhar and M. S. Hsiao, "On efficient error diagnosis of digital circuits," in *Proc. Int. Test Conf.*, 2001, pp. 678–687.

[12] A. Veneris, S. Venkataraman, I. N. Hajj, and W. K. Fuchs, "Multiple design error diagnosis and correction," in *Proc. VLSI Test Symp.*, Apr. 1999, pp. 58–63.

[13] V. Boppana, I. Ghosh, R. Mukherjee, J. Jain, and M. Fujita, "Hierarchical error diagnosis targeting RTL circuit," in *Proc. Int. Conf. VLSI Des.*, Jan. 2000, pp. 436–441.

[14] V. Boppana and M. Fujita, "Modeling the unknown! Towards model-independent fault and error diagnosis," in *Proc. Int. Test Conf.*, 1998, pp. 1094–1101.

[15] M. Khalil, Y. Le Traon, and C. Robach, "Towards an automatic diagnosis for high-level validation," in *Proc. Int. Test Conf.*, 1998, pp. 1010–1018.

[16] M. Khalil, O. Benkahla, and C. Robach, "An experimental comparison of software diagnosis methods," in *Proc. 25th EUROMICRO Conf.*, Sep. 1999, pp. 146–149.

[17] C. H. Shi and J. Y. Jou, "An efficient approach for error diagnosis in HDL design," in *Proc. Int. Symp. Circuits Syst.*, May 2003, pp. IV-732–IV-735.

[18] B. R. Huang, T. J. Tsai, and C. N. Liu, "On debugging assistance in assertion-based verification," in *Proc. 12th Workshop Synthesis Syst. Integr. Mixed Inform. Technol.*, Oct. 2004, pp. 290–295.

[19] Y. C. Hsu, B. Tabbara, Y. A. Chen, and F. Tsai, "Advanced technique for RTL debugging," in *Proc. IEEE/ACM Des. Autom. Conf.*, Jun. 2003, pp. 362–367.

[20] B. Peischl and F. Wotawa, "Automated source-level error localization in hardware designs," *IEEE Des. Test Comput.*, vol. 23, no. 1, pp. 8–19, Jan./Feb. 2006.

[21] T. Y. Jiang, C. N. Liu, and J. Y. Jou, "Effective error diagnosis for RTL designs in HDLs," in *Proc. 11th Asia Test Symp.*, 2002, pp. 362–367.

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

[23] F. Corno, M. S. Reorda, and G. Squillero, "RT-level ITC'99 benchmarks and first ATPG results," *IEEE Des. Test Comput.*, vol. 17, no. 3, pp. 44–53, Jul.–Sep. 2000. [Online]. Available: http://www.cad.polito.it/tools/itc99.html

[24] 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 National Tsing Hua University, Hsinchu, Taiwan, in 1999 and the M.S. degree in electronics engineering from National Chiao Tung University, Hsinchu, in 2001. He is currently working toward the Ph.D. degree in electronics engineering in the Department of Electronics Engineering, National Chiao Tung University.

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

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

He is currently an Assistant Professor with 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 National Taiwan University, Taipei, Taiwan, 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 was with GTE Laboratories from 1985 to 1986 and AT&T Bell Laboratories, Murray Hill, KY, from 1986 to 1994. From 2000 to 2003, he was a Full Professor and the Chairman of the Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan. From February 2004 to March 2007, he was the Director General of the National Chip Implementation Center, National Applied Research Laboratories, Hsinchu. Since April 2007, he has also been the Executive Director of the National SoC Program. He is with the Department of Electronics Engineering, National Chiao Tung University, Hsinchu 300, Taiwan and is Vice Chancellor (Academic Affairs) of the University System of Taiwan (consisting of four research universities: National Central University, Jhongli City, Taiwan; National Chiao Tung University; National Tsing Hua University, Hsinchu; and National Yang Ming University, Taipei). He has authored more than 160 technical papers. His research interests include logic and physical synthesis, design verification, computer-aided design for low power, and network on chips.

Dr. Jou currently serves as Associate Editor for the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS. He was the Technical Program Chair of the 2007 International Symposium on VLSI Design, Automation, and Test (VLSI-DAT), the 12th VLSI Design/CAD Symposium (2001), and the Asia-Pacific Conference on Hardware Description Languages (APCHDL'97). He was also the Honorary Chair of the 2006 International Workshop on Multi-Project Chip and the Executive Chair of the 2nd Taiwan–Japan Microelectronics International Symposium (2002). He was also elected President of the Taiwan Integrated Circuit Design Society during 2007–2008. He was the recipient of the Distinguished Paper Award of the IEEE International Conference on Computer-Aided Design in 1990; the Outstanding Academy–Industry Cooperation Achievement Award from the Ministry of Education, Taiwan, in 2002; and the Outstanding Electrical Engineering Professor Award from the Chinese Institute of Electrical Engineering in 2006.