Chapter 1

Introduction

1.1 Role of Simulation-Based Functional Validation

While silicon capacity continues to increase and the size of devices and the minimal width of lines are decreasing rapidly, Integrated Circuit (IC) designers could integrate many functions, even a whole complex system, into a single chip. A typical single SoC may consist of millions of logic gates, raising the immense potential for design errors and thus significantly complicating the verification task. Verification is now perceived as the major bottleneck in integrated circuit design [1,2].

Formal verification techniques have partially alleviated this problem. These techniques use mathematical or formal techniques to exactly prove or disprove the properties about a hardware design. Equivalence checking [3,4,5] attempts to prove that the two compared designs (the specification and the implementation) are logically equivalent. A popular use of this kind of techniques is to verify that the gate-level netlist, which is often generated by a synthesis tool, correctly implements the original Hardware Description Language (HDL) codes. In this usage case, equivalence checking can be quite useful to ensure the conformance of the synthesis results. Nevertheless, to verify the correctness of the initial register-transfer level (RTL) HDL descriptions requires other approaches.

Model checking [6,7,8] is another application of formal verification techniques.

It often operates directly on Binary Decision Diagrams (BDDs) to formally prove or disprove some properties or assertions of a hardware design. Due to the rapid progress of SATisfiability solver (SAT solver), recently researches of model checking [9,10] tend to exploit SAT solvers instead of BDDs. These model checkers may be so powerful that they can determine whether some deadlock conditions may occur or they can formally verify relates to interfaces. Although the formal techniques for language containment, model checking, property checking, and assertion-based verification [11] are making progress on the problem of verifying the correctness of the initial HDL codes, there is no indication that these techniques will be able to offer comprehensive verification across a wide variety of designs. For this reason, and perhaps because of the intuitive appeal of simulation, it appears that simulation-based functional validation is still one of the popular means for design verification for some time to come.

Nevertheless, functional validation based on simulation can be only partially completed. To address this incompleteness, coverage-driven semiformal methods have been developed. These methods exert better control over simulation by using various schemes to generate input stimuli and assess the extent of verification completeness. The goal is to achieve comprehensive validation without wasted efforts. Coverage metrics drive simulation resources to right direction and help approximate the goal.

1.2 Classification of Coverage Metrics for Validation on HDL Descriptions

1.2.1 Code Coverage Metrics

The problem of verifying the correctness of Design Under Validation (DUV) described in a HDL using simulation is similar to the software testing problem because a HDL description is quite similar to a program written in a high-level programming language like C. As a result, code coverage metrics for HDL codes are largely derived from metrics used in software testing. They are mainly used to identify which code structure classes in the HDL code are exercised during simulation.

These code structures are defined on a Control Flow Graph (CFG), which is a graphical representation of a program's control structure [12, Ch. 3]. Given a set of program stimuli, one can determine the code structures of the HDL code activated by the stimuli. The simplest one should be *line* or *statement coverage* metric. The line coverage metric measures the number of times every statement is exercised by the program stimuli. More sophisticated code-based coverage metrics are *branches*, *expression*, and *path coverage*. These code coverage metrics involve the CFG corresponding to the HDL code. Control statements constitute the branching point in the CFG. We use the following Verilog code fragment shown in Figure 1-1 to illustrate each aforementioned code coverage metrics.

```
1: always@(sel or a or b or c or d1) begin
2:       case( sel )
3:            3'b001: z = a;
4:            3'b010: z = b;
5:            3'b100: z = c;
6:            default: z = 0;
7:       endcase
8:       d1 = z + 2;
9:       if( a | b )
10:           d = d1 + 5;
11:      else
12:           d = d1 – 1;
13: end
```

Figure 1-1: A HDL code example

The **if** statement on line 9 has **(a | b)** as the control expression. *Branch coverage* metric requires exercising each possible direction from a control statement. For the **if** statement, lines 10 and 12 must both be executed during a simulation run. Similarly, for the **case** statement on line 2, lines 3, 4, and 5 must be executed.

A more sophisticated metric, *expression coverage* metric, requires exercising all the possible ways that an expression can yield each value. For instance, for the control expression **(a | b)**, in addition to the case where **a = 0** and **b = 0**, we must exercise the two separate cases where the expression gives **1.** The first case is **a = 1** and **b = 0** and the other is **a = 0** and **b = 1.**

Path coverage metric refers to paths in the CFG. For instance, in the Verilog example, the branch of the **case** statement on line 4 followed by the **else** branch of the **if** statement defines one path through the CFG. The branch of the **case** statement on line 5 followed by the **true** branch of the **if** statement is another path through the CFG.

A potential goal of software testing is to have 100% *path coverage*, which implies 100% *branch* and *line coverage*. However, 100% *path coverage* is a very stringent requirement and the number of paths in a program may be exponentially related to program size. For this reason, exercising all paths may be impossible. Representative subsets are usually chosen by verification engineers or circuit designers with some heuristics.

Measuring the aforementioned code coverage metrics requires little overhead, and because of the ease of interpreting coverage results, these metrics are popularly used nowadays. Almost all design groups use some form of code coverage, and many commercial tools are available to measure them. Nevertheless, unlike the case with software, achieving certain extent of code coverage for hardware is a minimum requirement because that hardware designs are highly concurrent. More than one code fragment is active at a time, thus fundamentally distinguishing HDL code from sequential software. The aforementioned code coverage metrics do not address this essential hardware characteristic. Consequently, requiring complete code coverage for hardware, although necessary, is not enough.

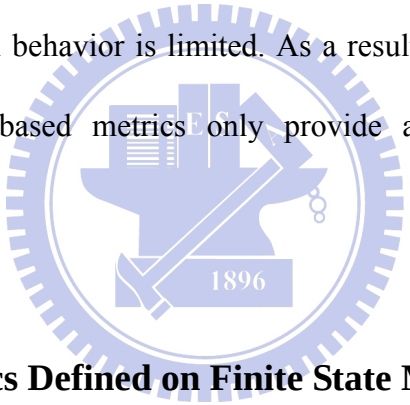
1.2.2 Coverage Metrics Based on Circuit Structure

Toggle coverage should be the simplest metric that is based on circuit structure. The metric requires that each wire in the circuit switches from 0 to 1 or 1 to 0 at some time instances during simulation. This metric can identify physical portions of the DUV that are not properly exercised. Many other sophisticated metrics in this class

are developed based upon toggle coverage.

Separating circuits into data path and control logics is reasonable for defining more useful circuit-structure based metrics. In the data path portion, registers deserve special attention during validation. Each register must be initialized, loaded, and read from, and each feasible register-to-register path must be exercised.

Coverage metrics of this class are defined on exercising the concerned circuit structures. Like code coverage metrics, they are easy to measure and intuitive to interpret and thus are popular. However, circuit-structure coverage metrics are defined on static, structural representations; hence their ability to quantify and pose requirements on sequential behavior is limited. As a result, similar to code coverage metrics, circuit-structure based metrics only provide a lower bound validation requirement as well.



1.2.3 Coverage Metrics Defined on Finite State Machine

In order to quantify and pose requirements on sequential behavior of the DUV, metrics defined on state transition graphs are developed and they are truly more powerful in this regard. These metrics require state, transition, or limited path coverage on a finite state machine (FSM) system representation;

Because FSM descriptions for complete systems are prohibitively large, these metrics must be defined on smaller, more abstract FSMs. We classify FSMs into two broad categories:

1. Hand-written FSMs that capture the behavior of the design at a high level.

2. FSMs automatically extracted from the design description. Typically, after a set of state variables is selected, the design is projected onto this set to obtain an abstract FSM.

Metrics in the first category are less dependent on implementation details and encapsulate the design intent more succinctly. However, constructing the abstract FSM and maintaining it as the design evolves takes considerable effort. Moreover, there is no guarantee that the implementation will conform to the high-level model. Despite these drawbacks, specifying the system from an alternative viewpoint is an effective method for exposing design errors. Experience shows that using test scenarios targeted at increasing this kind of coverage has detected many difficult-to-find bugs [14].

In the second category, the state variables of the abstract FSMs for metrics can be selected manually or with heuristics. Shen and Abraham present a heuristic technique for extracting the control state variable that changes most frequently, called the primary control state [15]. They compute an FSM reflecting the transitions of the primary control state variable and require coverage of all paths of a certain length in this FSM. Even small processors have a large number of such paths, but because each simulation run is short, the cost is tolerable. Kantrowitz and Noack use transition coverage on a hand-constructed abstract model of the system, as well as cache interface logic. Others select important, closely coupled control state variables based on the design's architecture [14,16].

Selecting abstract FSMs requires compromising between the amount of information that goes into the FSMs and the ease of using the coverage information.

The relative benefits of the choice of FSMs and the metrics defined on them are design dependent. Increasing the amount of detail in the FSMs increases the coverage metric's accuracy but makes interpreting the coverage data more difficult. If the abstract FSM is large, attaining high coverage with respect to the more sophisticated metrics is difficult.

The biggest challenge with state-space-based metrics is writing coverage-directed tests. Determining whether certain states, transitions, or paths can be covered may be difficult. The FSMs' state variables may be deep in the design, and achieving coverage may require satisfying several sequential constraints. Moreover, inspecting and evaluating the coverage data may be difficult, especially if the FSMs are automatically extracted. Some automated approaches involve sequential testing techniques [17]. Others establish a correspondence between coverage data and input stimuli using pattern matching on previous simulation runs. The capacity of automated methods is often insufficient for handling coverage-directed pattern generation on practical designs, whereas the user may need to understand the entire design to generate the necessary inputs. Nevertheless, state-space-based metrics are invaluable for identifying rare, error-prone execution fragments and FSM interactions that may be overlooked during simulation, thus justifying the high cost of test generation. Ultimately, carefully choosing abstract FSMs can alleviate many of the problems mentioned.

Coverage metrics of this class consider about exercising states, state transitions, and a particular sequence of transitions of the targeted FSM. As with code coverage, circuit-structure based ones, metrics of this class do not explicitly consider whether

erroneous effects caused by exercising some internal error portions of the DUV can be revealed during simulation.

1.3 Observability Issue in Simulation-Based Functional Validation

In a simulation-based validation framework, the simulation results or values should be compared against the correct values on some signals of interest to check the correctness of circuit behaviors. The correct values of these signals of interest may come from a reference model described at a different abstraction level or monitors and assertions. These signals of interest are called Observation Points (OPs) because they act like observation windows to uncover bugs in the DUV. During simulation, a discrepancy from the desired behavior is detected only if an OP takes on a simulation value that conflicts the correct value specified by the reference model.

Typically, OPs are Primary Outputs (POs) of the DUV and/or some other internal wires or register outputs that are selected by circuit designers. Designers usually follow their understanding to the specification and the behavior of the circuit to select these OPs, without explicit consideration of error propagation, i.e. whether erroneous effects of some internal signals caused by design errors are propagated to OPs. As a result, during simulation process, even if some erroneous values were generated by some activated internal bugs, the erroneous values may be masked during their propagation to Ops, causing that these erroneous values as well as the internal bugs remain undetected.

Traditional code coverage metrics from software testing, such as *statement coverage*, *branch coverage*, and *path coverage* metrics, only consider whether their concerned code structures are exercised. Circuit-based coverage metrics or FSM coverage metrics also do not explicitly check whether erroneous effects can be propagated to OPs for bug detection during simulation. They all do not explicitly consider whether erroneous effects caused by internal bugs are propagated to OPs. Design errors may be masked and still remain undetected even if they were said "covered" under these coverage metrics. The result is that verification completeness is overestimated by these coverage metrics.

We use the following example to explain the error masking situation. If we simulate the HDL code fragment shown in Figure 1-2(a) with the input stimulus given in Figure 1-2(b), the simulation result is as shown in Figure 1-2(c). As far as code coverage metrics are concerned, we will find that with respect to statement, branch, and path coverage metrics, 100% coverage is achieved. If the three coverage metrics are used to evaluate the extent of the simulation, the input stimulus in Figure 1-2(b) should be regard as a good test vector set for the design in Figure 1-2(a) since the test vector set exercises every target code structures during simulation.

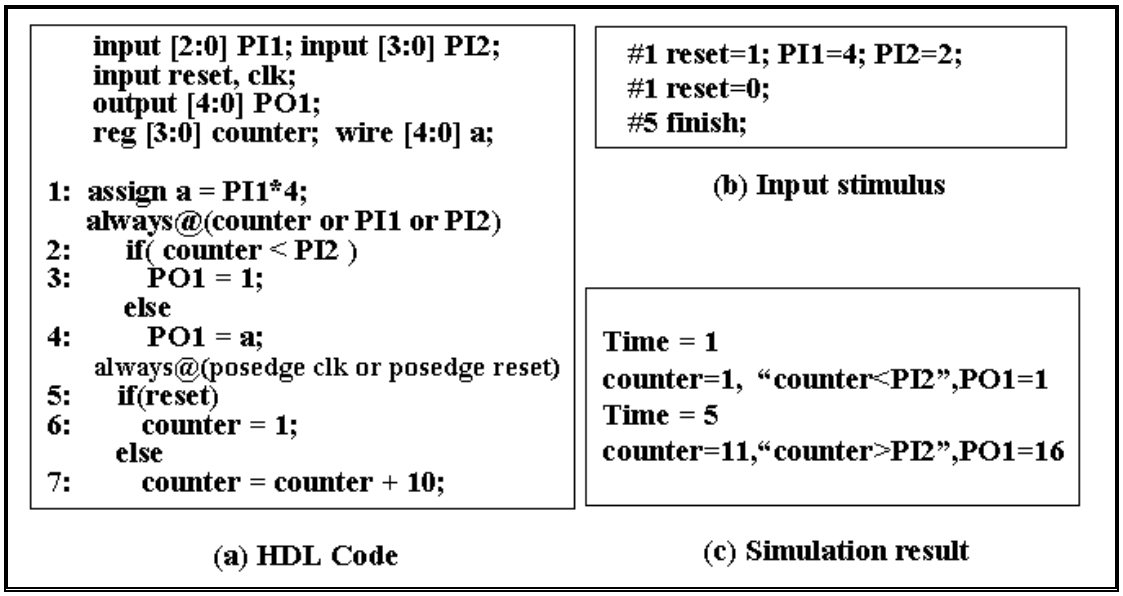


Figure 1-2: An example of functional validation on a HDL code fragment

However, assumed that statement 7 is carelessly written into "counter=counter+2", we surprisingly find that this careless design error can not be detected by this input stimulus of quality. Although the design error "counter=counter+2" changes the value of counter into 3 at Time = 5, different from the expected value 11, value 3 and value 11 cause the same evaluation result in the operation "counter<PI2". The erroneous value 3 is masked. The design error in statement 7 hides from being detected. In this case, the completeness of simulation is misjudged by the three coverage metrics. Therefore, observability consideration is important to suitably assess the comprehensiveness of validation.

The Observability-based Code COverage Metric (OCCOM) is the pioneer that addresses the essential observability issue [18]. Dump-file based OCCOM computation facilitates integration with commercial simulators and thus accelerates the analysis process [19]. Tag-based observability measures are extended to assess the

extent of validation for C programs in recent works [20]. Test pattern generation approaches for OCCOM make the entire work more practical [21].

In the above OCCOM approaches, two special *tags*, $_{\Delta}$ and - $_{\Delta}$, are injected on each signal to simulate potential increasing and decreasing value changes caused by some bugs. These tags on variables are not tied to particular design errors. The propagation of tags is used to simulate the propagation of potential erroneous effects. The percentage of tags that can be propagated to OPs is the coverage of OCCOM. However, *tags* can only be observed or unobserved, providing only two levels of measurement; 1 and 0. Erroneous effects that have lower observation opportunities may still be judged as observable. Thus, verification completeness may still be overestimated by OCCOM. If a new observability measure for HDL descriptions could provide intermediate values between 1 (observed) and 0 (unobserved), the likelihood of misestimating observability should be reduced.

1.4 Other Observability-Related Researches

1.4.1 Testability Analysis in Manufacturing Test

Manufacturing test is a process of checking that integrated circuits are manufactured correctly. The basic premise is the modeling of manufacturing defects as logical faults. Since manufacturing is a physical process that can be analyzed, credible fault models can be derived. For example, defects are known to cause breaks and shorts in metal wires. These breaks or shorts can be modeled as logical faults

since there is a direct correspondence between wires in silicon and connections in the logic circuit.

One of the most popular fault models in manufacturing test is the stuck-at fault model [22]. The stuck-at fault model is a logical fault model where any wire in the logic circuit can be stuck-at-1 or stuck-at-0. A test vector that produces the opposite value (zero for a stuck-at-1, and one for a stuck-at-0) will excite the fault. The effect of the fault has to be propagated to an observable circuit output in order for the fault to be detected by the vector.

The direct correspondence between a metal wire in the silicon integrated circuit and a connection in the logic circuit motivates logical fault models. No such correspondence may exist for a behavioral description in an HDL or structural RTL description. Statements in the HDL description may correspond to hundreds of gates and wires in the final design.

Based on the fault models, test vectors are generated and applied to test 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 [23]. Recent researches abstracted defects as higher-level logical fault models [24,25,26].

However, the correspondence between logical fault models and HDL design errors is still weak in two aspects. First, an erroneous statement may be synthesized into hundreds of erroneous gates and erroneous wires. Second, 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 the observability for HDL descriptions.

Some RTL testability analysis research exploits the idea of hierarchical testing with a pre-computed test vector set [27,28]. 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 the observability measures used to measure the likelihood of error propagation.

1.4.2. Sensitivity Analysis in Software Testing

In software testing arena, a sensitivity analysis, also called PIE analysis, for software programs to locate hard-to-detect bugs in a software program was proposed by J. Voas [29]. PIE analysis uses program instrumentation, syntax mutation, and changed values injected into data states to predict a location's ability to cause program failure if the location were to contain a fault. The program inputs are selected at random consistent with an assumed input distribution. This analysis does not require a testing oracle because PIE analysis uses the program itself as an oracle for examining the output of altered versions of the program.

The PIE analysis estimates the below three probabilities to predict a software program's dynamic computational behavior as well as where hard-to-detect bugs may hide. The three probabilities are 1) *Execution probability* - the probability that a location is executed, 2) *Infection probability* - the probability that a change to the

source program causes a change in the resulting internal computational state, and 3) *Propagation Probability* - the probability that a forced change in an internal computational state propagates and causes a change in the program's output.

Among the three probabilities, the *propagation probability* (PP) of a variable is the estimated probability that a variable's erroneous values caused by some bugs are observed in the program outputs. Propagation Probability 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 total number of experiments is the PP of *v*. This PP measures are quite accurate estimations for the likelihood of error propagation (observability for erroneous effects). However, the proposed statistical computation approach requires too much time and thus may be unsuitable for HDLs of commercial products because time to market is always important for commercial products.

1.5 Design Error Diagnosis on Faulty HDL Descriptions

1.5.1 Traditional Design Error Diagnosis Works

Due to the high complexity of modern Very Large Scaled Integrated (VLSI) circuit designs, verification process may often find that a design in the current stage (implementation) is not consistent with that in the previous stage (specification). Once

a functional mismatch is found, design error diagnosis is needed.

Most of the previous studies on this topic target the diagnosis of gate-level or lower-level implementations. These methods can be roughly divided into two categories: simulation-based approaches and symbolic approaches. Simulation-based approaches [30,31] first derive a set of input vectors that can differentiate the implementation and the specification. These binary or three-valued input vectors are called erroneous vectors. By simulating each erroneous vector, the possible error candidates can be trimmed down gradually. The heuristics to filter out impossible error candidates vary from one to another. Some of them rely on error models such as gate errors (missing gate, extra gate, wrong logical connective,…) and line errors (missing line, extra line,…) while other approaches are structure-based methods and require no error models.

On the other hand, symbolic approaches [32,33] do not enumerate erroneous vectors. They represent symbolic functional manipulation with Ordered Binary Decision Diagram (OBDD) to formulate the necessary and sufficient condition of fixing a single error. Based on these formulations, every potential error source can be precisely identified. An approach to combine the both symbolic and simulation-based techniques has also been proposed to reduce the run time of design error diagnosis. In comparison, the symbolic approaches are accurate and extendible to multiple design errors. However, constructing the required BDD representations may cause memory explosion when applied to large circuits. On the other hand, simulation-based approaches, although scale well with the size of circuits, are often not accurate enough. In order to avoid potential memory explosion of BDD-based symbolic

approaches, some recent symbolic works exploit the progress of Boolean satisfiability (SAT) solver and develops SAT-based approaches [34].

1.5.2 Software Debugging Techniques

In addition to gate-level or other lower-level implementations, design errors can also occur at the very first design stage – modeling the circuit behavior using HDLs. Traditionally, debugging a faulty HDL design relies on manually tracing the faulty HDL code. However, a simple HDL design today can have probably thousands of code lines and even more. Manually tracing the faulty HDL code to debug is not an effective debugging method. Approaches to assist HDL debugging are urgent.

For a Register-Transfer Level (RTL) HDL code, the distance between the HDL code and a software program is small: diagnosis may be seen as a software problem as well as a hardware one. In the software diagnosis domain, most of the methods are based on the slicing technique [35,36]. Program slicing, introduced by Weiser [36] is a technique for restricting the behavior of a program to some specified subsets of interest. The main idea behind this technique is to decompose the considered program into independent parts, called slices. Each slice contains all the statements that could have influenced the value of a variable at a given program point. It can be executed separately from the rest of the program. The difference between two slices is called a dice and is the basis of the fault location process.

For example, let us consider two slices **A** and **B** as illustrated in Figure 1-3.

Assume that one of them (**B**) gives a correct result; whereas the other (**A**) gives a faulty one. It is obvious that the faulty area will be in the dice **A** minus **B,** which is smaller than the area of the faulty slice **A**. Consequently, the effort of searching in the whole slice **A** is saved and the diagnosis duration time is reduced.

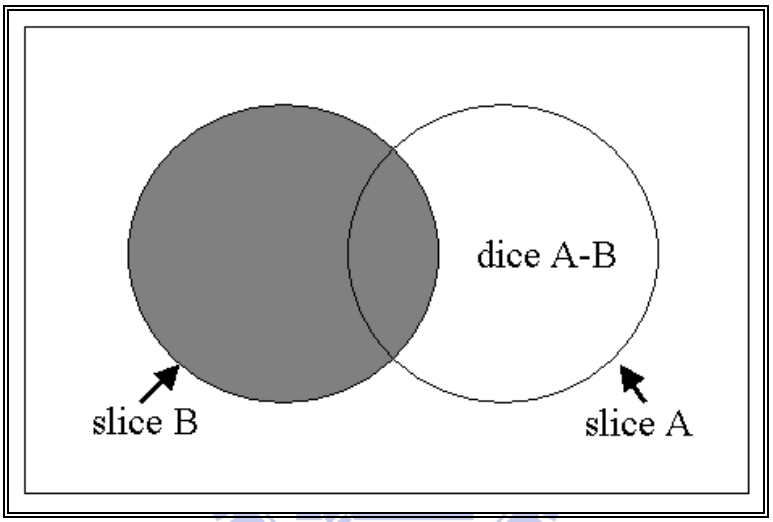


Figure 1-3: Slice and dice

The process of fault location implies to execute each slice and the correctness of each slice execution has to be determined: a human intervention is generally needed for establishing the oracle (the algorithmic debugger interacts with the user through queries about the intended program behavior). Moreover, in this manner, multiple times of simulation are required before real diagnosis can go on. This is too time-consuming.

1.5.3 Techniques for Debugging HDL Descriptions

In the literature, some researches have targeted on techniques that assist HDL design debugging. Peischl et al [37] focus on Model-Based Diagnosis (MBD) paradigm. They employ structure and behavior with respect to their error models for software debugging. Besides, error-model based approaches, there are also error-model free methods that should have better applicability to various kinds of design errors.

Maisaa Khalil et al [38] proposed an automatic diagnosis approach based on the cross check on the result of each test case. By using four strategies based on different four hypotheses, four error candidate sets are sequentially obtained, from the smallest one to the biggest one. It is expected that tool users or debugging engineers can locate design errors in the first few error candidate sets, whose size are relatively smaller, to save the debugging efforts. However, the first three hypotheses are not always satisfied since design errors in a fault HDL description can be multiple and the oracle can be unsure. True design errors may be absent in the first three error candidate sets, resulting in that the efforts of searching design errors in these sets are wasted. Even worse, we still have to search design errors in the fourth set of error candidates, the largest one.

Shi et al [39] applied data dependency analysis, execution statistics, and the characteristics of HDL operations to filter out impossible error candidates. In this method, only one reduced set of error candidates is derived for examination with a single time of simulation. And, the size of error candidate set (error space) is acceptable in size. Huang et al [40] further exploited the extra observability of

assertions in an attempt to derive smaller error space.

Instead automatic methods to derive error candidates, Y.C. Hsu et al have developed two useful utilities to help designers reason the locality of bugs with manual interventions [41]. However, the number of derived error candidates can still be plenty. Searching design errors among these candidates by examining them one by one blindly may still takes much valuable time.

1.6. Organization

This thesis is organized into five chapters. Chapter 1 gives the introduction to the thesis. Chapter 2 introduces some related works and gives preliminaries. In Chapter 3, we introduce our proposed probabilistic observability measures on HDL designs for efficient functional validation. Besides being used for efficient functional validation, Chapter 4 introduces another application of the probabilistic observability measures on HDL designs - design error diagnosis on faulty HDL descriptions. Finally, we conclude the thesis in Chapter 5 and discuss some future research directions.

Chapter 2

Preliminaries

2.1 Observability-Based Code Coverage Metrics

During simulation-based functional validation, simulation values of Observation Points (OPs) should be compared against the expected values to check the correctness of certain circuit behavior. A discrepancy from the desired behavior can be detected only if some of OPs have simulation values that are different from the expected values. Coverage metrics should explicitly consider the observability requirement (requirement of error propagation) to detect internal design errors such that comprehensiveness of validation can be suitably gauged.

Observability-based Code COverage Metric (OCCOM) is the pioneer that addresses the essential observability issue [18]. A dump-file based OCCOM computation approach is later proposed to facilitate the integration with commercial simulators and thus accelerate the analysis process [19]. In these OCCOM works, two special *tags* Δ and $-\Delta$ are injected on each signal to simulate potential increasing and decreasing value changes caused by activated errors. Tag propagation rules are defined for using tag propagation to predict the potential propagation behaviors of these erroneous value changes. The percentage of the *tags* observed at the OPs is the coverage with respect to OCCOM

The dump-file based OCCOM approach proposed in [19] is a two-phase

approach. The two phases are abstracted as below. We will introduce the two phases individually in detail later in this section.

1) **Conditional Statement Transformation**: The transformation involves moving some statements and creating new variables to contain extra information during simulation for the next phase calculation. After the transformation, the modified HDL model is then simulated using a standard HDL simulator to obtain a dump-file for the later tag simulation calculus.
2) **Tag Simulation Calculus**: In this phase of computation, tags are first injected and then the propagation of the injected tags is computed based on the tag simulation calculus developed by the authors [19]. The tag simulation calculus is composed of tag propagation rules for various HDL operations, in which the propagation through the HDL operations is based on likelihoods. For one injected tag, the tag propagation result can be that it is observed at some OPs or it is not observed at any OPs. The third possible result is the presence of special *unknown* tags *?* when the tag propagation is not so sure in the computation.

The conditional statement modifications on the HDL code proposed in [19] are for obtaining sufficient information during HDL simulation for the later tag simulation calculus. Consider a HDL code fragment with a simple conditional statement "**if… else …**" shown in Figure2-1. The original code fragment is shown on left-hand side and the transformed code is shown to the right in Figure 2-1. Consider the case of a tag on **cexp**. During the simulation of the modified code, the values of both **expr1** and **expr2** are computed and stored in the new variables **y1** and **y2**. The new values of **y** corresponding to the execution of both the **then** and **else** clauses are

known, regardless of the value of **cexp** during simulation. This will help to correctly propagate positive or negative tags on **cexp** in their tag simulation calculus.

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

Figure 2-1: A simple conditional statement modification

The case of nested conditionals is more complicated. Further, the situation where variables such as **y** are assigned values that depend on the old values (e.g., increment operation) have to be considered. As an example, consider the Verilog statements shown in Figure 2-2.

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

Figure 2-2: A nested conditional statement example

Transformation starts with transforming the statement "**if(cexp1)**". The result after the transformation on "**if(cexp1)**" is shown in Figure 2-3. In the next step, the "**if(cexp2)**" and "**if(cexp3)**" the statements are transformed. **y** is the only variable in the original Verilog code whose value is changed inside the if statement and, as a

result, in order to transform the code, a new variable **y3** is introduced.

```
if (cexp2)
  y = expr1 ;

if (cexp3)
  y = y + expr2 ;

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

Figure 2-3: Code after first phase of conditional statement transformation

```
y3 = y ;
y1 = expr1 ;

if (cexp2)
  y3 = y1 ;

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

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

Figure 2-4: Code fragment after the entire conditional statement transformation

Note that if the value of **cexp2** is false, variable **y3** is read before assigning any value to it. As a result, it is necessary to initialize its value to the value of **y**. The transformed code after the entire conditional statement modification is shown in Figure2-4. The transformed code will compute the necessary information to perform propagation of tags on **cexp1**, **cexp2**, or **cexp3**. It can easily be verified that the two pieces of code result in the same values for variable **y**.

The second phase of OCCOM computation is tag simulation calculus, which is used to predict error propagation and is similar to the D calculus [42]. A tag is represented by the symbol $_{\Delta}$, which signifies a possible change in the value of the variable due to an error. Both positive and negative tags are considered, $+_{\Delta}$ written simply as $_{\Delta}$, and $-_{\Delta}$. Both tags are injected onto each internal signal of the DUV first. If the presence or sign of the tag is not known, an unknown tag "?" is used. Note that ? = -? and also 0+? = 1+?.

Tag simulation calculus in [18.19] is based on the likelihood of the propagation of the tag. It is assumed that the tag is propagated or blocked depending on which case is more likely. For example, in the Verilog statement **c= (a!=b)** with **a=2** and **b=5,** if there is a positive tag on variable **a**, it is assumed that the tag is not propagated to the variable **c** [19]. The reason is that the value of the variable **c** in the presence of the tag is TRUE unless the magnitude of the tag on **a** is exactly three. As a result, the authors think that it is unlikely to have a tag on variable **c.** The tag propagation rule for "!=" created in [19] will block the tag from being propagated through this operation.

The tag simulation calculus for some other common representative operators

proposed in [18,19] is briefly introduced in the sequel. For each operator *op*, after the simulator computes *v(f)=v(a)(op)v(b)*, *v(f)* might be tagged with a positive $_{\Delta}$ or negative $-_{\Delta}$ or ? and it is written as *v(f)*+ $_{\Delta}$, *v(f)*- $_{\Delta}$, *v(f)*+?.

1) The calculus for an INVERTER, a two-input AND gate, and a two-input OR gate are shown in Table 2-1, Table 2-2, and Table 2-3, repectively. The five possible values at each input are {0, 1, $0+_{\Delta}$, $1-_{\Delta}$, 0+?}. (Note that $0-_{\Delta}$= 0 and $1+_{\Delta}$=1.) As an example, if the input of an inverter gate is zero and it has positive tag on it, the value of the output of the inverter will be one and it will have a negative tag on it. The case that the input of the inverter is one and the input has a negative tag is similar. As another example, if one of the inputs of an AND gate is zero and the input has a positive tag and the value of the other input is one and it has a negative tag on it, the value of the output of the AND gate will be zero because the erroneous value of one of the inputs is zero. Using the above calculus, any collection of Boolean gates comprising a combinational logic module can be tag simulated.

Table 2-1: Tag calculus for INVERTER gate in [18,19]

INVERTER	
0	1
1	0
0 + Δ	1- Δ
1- Δ	0+ Δ
0 + ?	0 + ?

Table 2-2: Tag calculus for AND gate in [18,19]

AND	0	1	0+Δ	1-Δ	0+?
0	0	0	0	0	0
1	0	1	0+Δ	1-Δ	0+?
0+Δ	0	0+Δ	0+Δ	0	0+?
1-Δ	0	1-Δ	0	1-Δ	0
0+?	0	0+?	0+?	0	0+?

Table 2-3: Tag calculus for OR gate in [18,19]

OR	0	1	0+Δ	1-Δ	0+?
0	0	1	0+Δ	1-Δ	0+?
1	1	1	1	1	1
0+Δ	0+Δ	1	0+Δ	1	0+Δ
1-Δ	1-Δ	1	1	1-Δ	0+?
0+?	0+?	1	0+Δ	0+?	0+?

2) Adder: If all tags on the adder inputs are positive and if the value $v(f) <$ MAX*INT*, the adder output is assigned to $v(f) + \Delta$. MAX*INT* is the maximum value possible for *f*. This is similar if all tags are negative. If both positive and negative tags exist at adder inputs, the output is assumed to be unknown tag. Table 2-4 shows calculus for tag propagation through an adder.

Table 2-4: Tag calculus for ADD (+) Operation in [18,19]

ADDER	b	b - Δ	b + Δ	b + ?
a	a + b	a + b - Δ	a + b + Δ	a + b +?
a - Δ	a + b - Δ	a + b - Δ	a + b +?	a + b +?
a + Δ	a + b + Δ	a + b +?	a + b + Δ	a + b +?
a + ?	a + b +?	a + b +?	a + b +?	a + b +?

3) Multiplier: All tags have to be of the same sign for propagation. A positive $_{\Delta}$ on input *a* is propagated to the output *f* provided *v(b)!*=0 or if *b* has a positive $_{\Delta}$. The output of multiplier is assigned to *v(f)* + $_{\Delta}$. This is similar for negative -$_{\Delta}$.

4) Comparators: If tags exist on inputs *a* and *b*, they have to be of opposite sign, else the output will have an unknown tag. Assume a positive tag on *a* alone or a positive tag on *a* and a negative tag on *b*. If *v(a)* is smaller than or equal to *v(b)*, then the tag(s) is (are) propagated to the output, else the tag(s) is (are) not. The output of comparator is assigned to 0+$_{\Delta}$. This is similar for other tags and other kinds of comparators. Table 2-5 and Figure 2-6 show the calculus for tag propagation through operator ">" when the result of operation is TRUE and FALSE, respectively.

Other tag propagation rules can be found in [18,19,43].

Table 2-5: Tag calculus for ">" when result of a > b is true

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

Table 2-6: Tag calculus for ">" when result of a > b is false

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

In summary, OCCOM is indeed more stringent than statement coverage metric because it considers only the execution requirement but also the observability requirement to detect internal design errors. The proposed dump-file based OCCOM computation is also an effective approach to derive the OCCOM coverage. In the works, experimental data are also available to demonstrate that the conditional statement modification conducted in the first phase of OCCOM computation introduce very little overhead to HDL simulation. The OCCOM works based on tags indeed provide observability information for effectively and suitably assess the extent of validation. However, tags can only provide two levels of measurement 1 (observed) or 0 (unobserved). A more accurate observability measure is always desirable for the observability issue in simulation-based validation. As the future works in [19] had discussed, the possible future direction they proposed for accuracy improvement of tags are 1) relative magnitude of tag or 2) absolute magnitude of tag.

2.2 PIE Analysis in Software Testing

J.Voas et al [29,44,45] present a dynamic technique to statistically estimate three software program characteristics that affect a software program's computational behavior: 1) Execution Probability (EP) - the probability that a particular section of a program is executed, 2) Infection Probability (IP) - the probability that the particular section affects the data state, and 3) Propagation Probability (PP) - the probability that a data state produced by that section has an effect on program output. These three characteristics can be used to predict whether faults are likely to be uncovered by software testing.

Among the three probabilities, the third probability PP can be regard as an observability measure for software programs and probably for HDL models as well. PP of a variable ***a*** (denoted as PP(***a***)) is the probability that variable ***a***'s erroneous values caused by some bugs are observed in the program's outputs and cause program failures. The algorithm to obtain estimated values of PP(*a*) proposed by J. Voas is abstracted as bellows.

Step1. Set variable *count* to 0.

Step2. Randomly select an input *x* according to the input distribution.

Step3. Alter the sampled value of variable *a* to create a mutant of this program.

Step4. For each different output result in program output after a is changed, increment *count*. If a time limit for termination related to the altered state has been exceeded, increment *count*. This precaution is necessary because of the effects that altered variables can cause to Boolean conditions that terminate indefinite loops.

Step5. Repeat steps 2-4 *n* times.

Step6. Divide *count* by *n* to derive *PP* of variable *a.*

This calculation algorithm to obtain PP(*a*) is a statistics-based estimation approach. The accuracy of estimated result highly depends on the iteration numbers, *n*. If *n* is big enough, the result of this algorithm can be a quite accurate estimation for PP(***a***). However, it is obvious that to obtain accurate estimations for PP(***a***) with a big *n* requires lots of iteration of simulation as well as computation time. If we intend to analyze observability of each point in a HDL model using this statistics-based approach, the entire procedure may take too much time. Other approaches or other observability measures are required for the observability analysis for HDL models.

2.3 Error Space Identification Approaches for HDL Debugging

When verification finds some discrepancy between the specification and the implementation written in a HDL, the debugging process traditionally relies on designers' manually tracing HDL codes. However, this manual debugging scheme could be tough and time-consuming because a relatively simple HDL design today can have more than thousands code lines. If a reduced set of error candidates can be obtained automatically by some approaches, these approaches should be helpful to this HDL debugging problem.

Maisaa Khalil et al [38] proposed an automatic diagnosis algorithm that contains four hypotheses to diagnose design errors using the HDL information. For systematic analysis, the algorithm classified all possible situations into four hypotheses that are defined from looseness to strictness. The first two hypotheses assume that there is only one erroneous statement in the HDL design. The first ant the third hypotheses assume that the executed statements of correct test cases are impossible to be the error sources. By using four strategies based on different four hypotheses, four error candidate sets are sequentially obtained, from the smallest one to the biggest one. It is expected that tool users or debugging engineers can locate design errors in the first few error candidate sets, whose size are relatively smaller, meaning that searching design errors in the three sets requires less efforts. However, the first three hypotheses do not always stand since design errors in a faulty HDL description can be multiple and the *oracle* can be unsure. As a result, true design errors may be absent in the first three error candidate sets, resulting in that the efforts of searching design errors in

these sets are wasted. Even worse, it is still required to search design errors in the fourth and the largest set of error candidates. It is also assumed that many test cases can trigger design errors. However, in practice, it is not easy to generate lots of test cases that can trigger the same design errors, especially when designers do not actually know where the design errors are and what they are.

Jiang and et al [46] proposed another error-model free automatic error space identification approach that exploits both data dependency analysis and execution trace to obtain an error space (a reduced error candidate set). The error space is the intersection of the execution trace of EOC (the clock cycle, in which discrepancy between the simulation values of all the primary outputs and the associated expected values is detected) and the result of data dependency analysis on Erroneous Primary Outputs (primary outputs that have simulation values are not consistent with the expected values in EOC). The size of the obtained error space in this approach should be relatively smaller than the one derived by the approach in [38] because additional data-dependency analysis is used to trim down the size. Take the HDL code shown in Figure2-5 as an example to demonstrate the error space identification in [46].

```
            module exm(PO1,PO2,PI1,PI2,PI3,PI4,clk);
             input PI1,PI2,PI3,PI4,clk;
             output PO1,PO2;
             reg PO1, w2;
             wire sel1, sel2, w1;
s1:           assign sel1 = PI1 & PI2;
s2:           assign sel2 = PI3 |    PI4;
s3:           assign w1 = PI2 ^ PI3;
s4:           assign PO2 = w2 | PI1;
event1:    always@( posedge clk ) begin
dec.1:        case( sel1 )
s5:             1'b0 : PO1 = w1;
s6:             1'b1 : PO1 = w2;
s7:             default : PO1 = w1;
              endcase
            end
event2:    always@( sel2 or PI1 or PI3 or PI4 ) begin
dec.2:        if( sel2 )
s8:             w2 = PI3;
              else
s9:             w2 = PI4 | PI1;
            end
            endmodule
```

Figure 2-5: An HDL example

Assume that the code in Figure 2-5 is the correct design that designers expect. The applied input vectors for each time instance and the corresponding values of POs are shown in Figure 2-6 (a) and (b).

```
reg clk, PI1, PI2, PI3, PI4;
always #5 clk = ~clk;
initial begin
        clk=0;
#2      PI1=1; PI2=1; PI3=1; PI4=0;
#8      PI1=0; PI2=1; PI3=0; PI4=1;
#10     PI1=1; PI2=1; PI3=0; PI4=0;
end
```

(a) Input stimuli

```
Time = 0:    PO1=1'bx; PO2=1'bx;
Time = 2:    PO1=1'bx; PO2=1'b1;
Time = 5:    PO1=1'b1; PO2=1'b1;
Time = 10:   PO1=1'b1; PO2=1'b0;
Time = 15:   PO1=1'b1; PO2=1'b0;
Time = 20:   PO1=1'b1; PO2=1'b1;
Time = 25:   PO1=1'b1; PO2=1'b1;
```

(b) Simulation results on POs

Figure 2-6: Input stimuli and expected simulation results

However, for some reasons, the statement **s9** is written incorrectly to be "*w2 = PI4*". Because of this design error, the simulation value of PO1 at 25ns 1'b0 and a discrepancy from the correct value will be observed. The clock cycle from 15ns to 25ns is called EOC, Error-Occurring Cycle, according to the definition in [46].

Then, we apply the error space identification approach in [46] to narrow down the set of error candidates. First, we find executed statements. At time=20ns, **s1, s2, s4**, and **event2** are triggered because of the value changes of **PI1** and **PI4**. Since **sel2**=1'b0, the execution statistics of statements under the event control of **event2** is that **dec.2** (decision or conditional statement) and **s9** are executed. **Event1** is triggered

due to the rising edge of **CLK** at 25ns. Because the **event1** is triggered and **sel1**=1'b1, **dec.1** and **s6** are executed. Therefore, the executed statements in EOC are {**s1, s2, s4, s6, s9**, **event1**, **dec.1**, **event2**, **dec.2}**.

Then, the relation space will be extracted. The extraction of the relation space relies on data dependency analysis based on control data flow graph (CDFG). The CDFG of the HDL example in Figure2-5 is shown in Figure 2-7, where **s** denotes a statement and **dec.** represents a conditional statement or a decision.

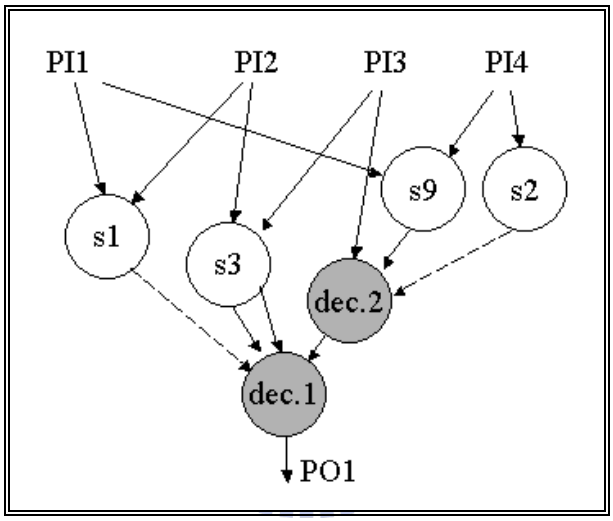


Figure 2-7: Control Data Flow Graph (CDFG) of PO1

To obtain relation space of PO1 relies on a back trace from PO1 to the PIs according to the relationship in the data flow. In the back-tracing starting from PO1, the first traversed node is **dec.1**. **dec.1** is added into the relation space. Then, because **s6** is the statement on the taken branch of **dec.1**, **s6** is the next traversed node. **s6** is also added in the relation space. The driving statements of **s6** are **dec.2** and **s9** and the driving statements of **dec.1** is **s1**. They are all added in the relation space, too. Similarly, the driving statements of dec.2 and s9 are found and added. Finally, the

relation space of PO1 are {**dec.1, s6, s1, event1, dec.2, s9, s2, event2**}. Therefore, the error space is {**s1, s2, s6, s9, event1, dec.1, event2, dec.2**}.

In additional to simple data dependency analysis of EPOs, Shi and et al further exploits the structure analysis and the nature of HDL operations to filter out more impossible error candidates [39]. A simple Verilog HDL code fragment shown in Figure 2-8 is used to illustrate how to apply the Rule I in [39] for error space reduction.

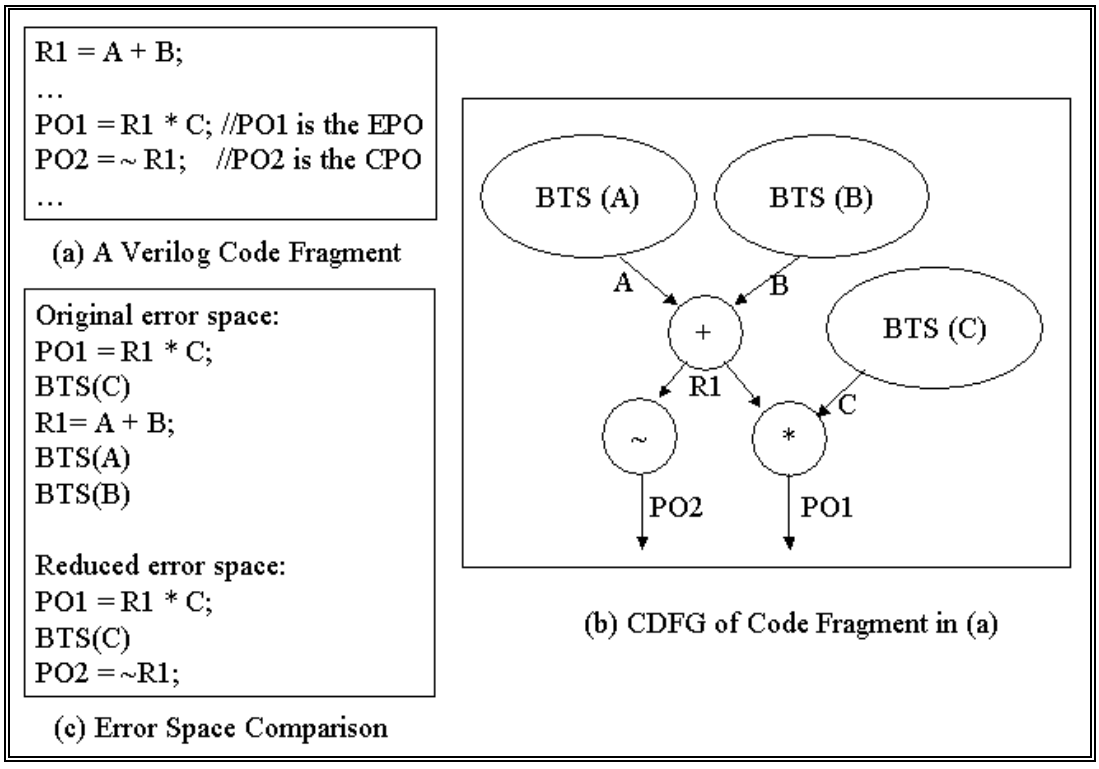


Figure 2-8: An example of applying Rule I in [39]

When an incorrect behavior is observed at PO1, if the error space identification approach in [46] is used, the back-tracing operation from PO1 on the CDFG will be the same as shown in Figure2-8(b). The resulted error space will be the same as the one shown in the upper part of Figure2-8(c).

However, when Rule I in [39] is applied, the authors state that the back-tracing from PO1 can stop at R1 by means of including the reversible path which contains a single reversible statement “PO2=~PO1” to the error space. This is because that R1 is on a reversible path to PO2. If R1 is incorrect, there must be some errors in the path from R1 to PO2 such that PO2 can have correct simulation value. As a result, the authors state that they can remove the statements “R1=A+B”, BTS(A), and BTS(B) and then add the reversible statement “PO2=~R1”. A reduced error space can be obtained by applying the Rule I. The resulted error space will be the one shown in the lower part of Figure2-8(c).

Besides Rule I, the authors also developed Rule II and III. Rule II states that given a HDL operation whose erroneous simulation value of left-hand variable and the correct value of left-hand variable are known, if there does not exist any values of other right-hand variables to produce the correct value of the left-hand variable while fixing the value of the target right-hand variable, the statement is incorrect or the simulation value of the target right-hand variable is incorrect. On the other hand, Rule III states that when the simulation value of one right-hand variable is a controlling value of the statement, the back-tracing from the other right-hand variables can be stopped. At least one erroneous statement requires to be retained in the error space.

The above works that focus on reducing number of error candidates are of course helpful for debugging faulty HDL designs. However, the size of the obtained error space can still vary from case to case. The number of error candidates may be plenty and searching true design errors in the obtained error space still requires much time.

Chapter 3

Observability Analysis on HDL Descriptions for Effective Functional Validation

3.1 Motivation

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. 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.

The Observability-based Code COverage Metric (OCCOM) is the first code coverage metric which considers the essential observability issue [18,19]. In their approach, the propagation of special *tags* that are attached to internal signals is simulated to predict the actual propagation of erroneous effects caused by design errors. Base on the likelihood that erroneous effects are propagated through each HDL

operation, the authors create *tag simulation calculus* and *tag propagation rules* to judge whether a tag can be propagated through an HDL operation or not.

However, the status of the tag propagation can only be **propagated** or **un-propagated**, providing only two levels of measurement; 1 and 0. However, the error propagation is obviously not so certain and can be modeled by just **propagated** and **un-propagated**. Inevitably, erroneous effects with low observation opportunities may still be judged as **propagated** in some cases, thus giving overestimate the verification completeness. Even worse, mislead the verification resources to other portions of the DUV and let a design error remain undetected. We use the following example to illustrate this.

Consider a simple HDL example shown in Figure 3-1(a). Applying the input stimulus shown in Figure 3-1(b) to simulate the HDL code fragment in Figure 3-1(a), we can obtain the simulation results shown in Figure 3-1(c).

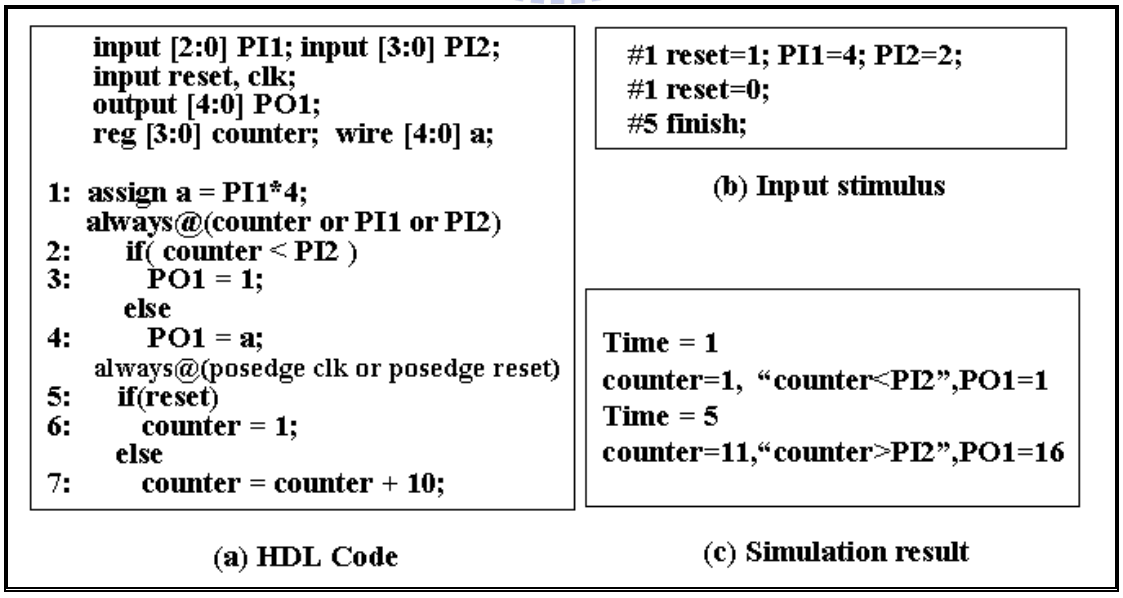


Figure 3-1: A HDL example

If we apply OCCOM to gauge the extent of the validation in the case shown in Figure 2-1, the tag propagation rule for "<" in [19] says 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.

Consider a case that statement 7 carelessly written to be "counter=counter+2" by the circuit designer. The design error "counter=counter+2" in statement 7 causes an incorrect value 3 on **counter** at t=5, i.e. 3 is different from the correct value of **counter** 11. Because 3 is smaller than the correct value 11, the propagation of this incorrect value 3 should be simulated by the $-\Delta$ injected on signal **counter**. We can regard the incorrect value 3 as $11-\Delta$.

According to the tag propagation rules [18,19,43] for operation "<", $-\Delta$ on **counter** can be propagated through the operation "if(counter<PI2)" and makes the output become $0+\Delta$. This implies that it is assumed that a decreasing value change is very likely to change the evaluation result of "counter<PI2", from FALSE to TRUE. However, we can see that the incorrect value 3 does not alter the evaluation result of "counter<PI2" as tag simulation calculus predicts. Tag simulation calculus fail to predict the error propagation in this example. In fact, in this example, the likelihood that a decreasing value change on **counter** is very unlikely to alter the evaluation result and to cause any erroneous effects on the output of "<". We explain this fact by the below analysis.

Although in the above example we assumed that the design error "counter=counter+2" caused an incorrect value 3 at t=5, in practice, an incorrect value of **counter** can be any possible value that is different from the correct value 11. It can

be 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, or 15. Among the 15 possible candidatess, the values {0~10} are smaller than the correct value 11 and each one of them can be can be regard as an individual 11-Δ. Because PI2 at t=5 is 2, only 1 and 0 can make the evaluation result change from FALSE to TRUE. The other values {2~10} cannot, even if each of them is an erroneous value smaller than 11. Nine values ({2~10}) out of all possible fifteen incorrect values ({0~10},{12~15}) can not alter the output of "counter<PI2" to make the output result as 1+Δ. The likelihood that 11-Δ is propagated through the operation "counter<PI2" should be quite low. Nevertheless, tag propagation rules did not actually take this likelihood into consideration and still assume that decreasing erroneous value change can be quite dramatic that it always change the evaluation result of "<". Similar situations may also happen if tag propagation rules are used on some operations that may mask erroneous effects, such as ">", "==", "!=", and ">>".

In addition to inaccuracy, it is also unreasonable that tag propagation rules assume that erroneous values can never propagate through bit-select operations "[]" and "[:]". In practice, erroneous effects of course can propagate through bit-select operations. Moreover, the assumed *single tag model* can only model the propagation behavior of exact one erroneous effect of a design error. If multiple design errors exist in the DUV, tag simulation calculus may not precisely determine whether the erroneous effects can be observed or not.

On the other hand, although Propagation Probability (PP) in PIE analysis is an accurate observability measure, its computation approach requires too much time as we've introduced in section 2.2. Therefore, we intend to develop a new Probabilistic

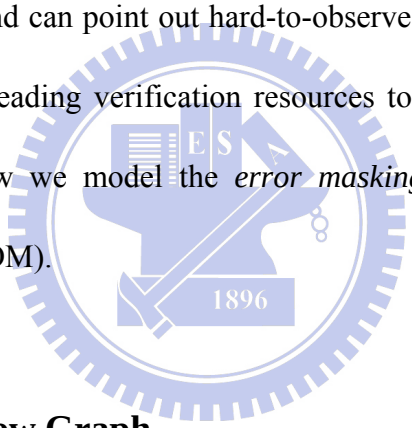
Observability Measures (POM), which are more accurate than tag-based observability measures and require much less computation time than PP. There are some possible applications for our proposed POM.

A new observability-based code coverage metric – In our new observability-based code coverage metric, 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.

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 observation points. It does not mean that behind these signals there must be some design errors, but it provides where the input stimuli does not suitably verify with both exercitation and observability considerations. 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.

3.2 Probabilistic Observability Measure for HDL Descriptions

Despite completion of a successful simulation in which the simulated values of all the Observation Points (OPs) 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 can point out hard-to-observe points during the previous phases of simulation for leading verification resources to those weak points. In the section, we introduce how we model the *error masking* and define Probabilistic Observability Measure (POM).



3.2.1 Control Data Flow Graph

The Design Under Validation (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 Figure 3-2 is used as an example of the HDL code shown in Figure 3-1. 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 Figure 3-2

corresponds to the operation "a=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(…) … else ..." in lines 2 to 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 Figure 3-2, an edge (*1:**, *4:=*) exists since the operation "4:=" takes the output of vertex "1:*" as its input. The *fanout* of *v* is a set of vertices *u* such that there is an edge from *v* to *u*. Similarly, the *fanin* 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 $<v_0, v_1, v_2, \ldots, v_k>$ of vertices such that $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in \boldsymbol{E}$.

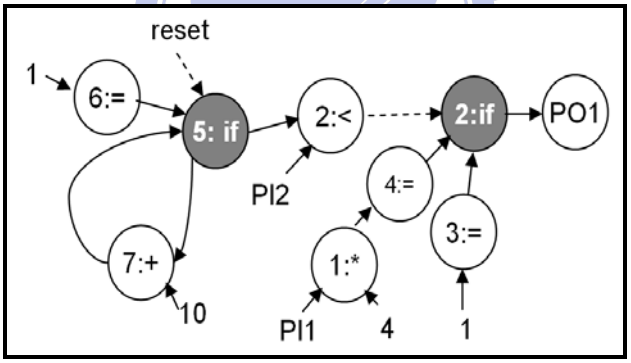


Figure 3-2: The CDFG of the HDL code in Figure3-1

3.2.2 Masked Value Set and Probabilistic Observability Measure

If a single incorrect value *w* ever existed on the output variable of vertex *v* y_v at time instance $t=t_i$ in the design under validation during simulation, this incorrect value *w* should cause no incorrect behaviors at any observation points at all positive edges

of clock[1]. If not, the simulation phase is not successful. More specifically, the simulated value of an observation point 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 observation point OP_j at $t=c_k$ (denoted as $OP_j@t=c_k$). 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 in (3.1).

$$f_{v@t=t_i \to OP_j @t=c_k}(w) = CV(OP_j @t=c_k) \tag{3.1}$$

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

If there are m total observation points $\{OP_1, OP_2, \ldots, OP_m\}$ and o clock cycles in the simulation phase, w must be *masked* on its way to all the observation points in all time frames such that it is not uncovered during the entire simulation process. For each observation point 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 as described in (3.2).

$$\bigcap_{j=1}^{m}\bigcap_{k=1}^{o} f_{v@t=t_i \to OP_j @t=c_k}(w) = CV(OP_j @t=c_k) \tag{3.2}$$

[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.

The set of all possible values of vertex v's output that can satisfy (3.2) is defined as the *Masked Value Set* (MVS) of vertex v at time instance $t=t_i$ (MVS($v@t=t_i$)). A more formal definition is given in (3.3). Each element in MVS($v@t=t_i$) retains the correct values of all the observation points at all positive edges during simulation.

$$MVS(v@t=t_i)=\{x \mid \bigcap_{j=1}^{m}\bigcap_{k=1}^{o} f_{v@t=t_i \rightarrow OP_j@t=c_k}(x)=CV(OP_j@t=c_k)\} \tag{3.3}$$

The correct value of the output of vertex v at $t=t_i$ is in MVS($v@t=t_i$) and this can justify the existence of MVS($v@t=t_i$). If 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 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 (3.4).

$$LOEM(v@t=t_i)=\frac{|MVS(v@t=t_i)|-1}{2^{BW}-1} \tag{3.4}$$

, where BW is the bit width of y_v. Its complement is the observability measure of v at $t=t_i$, as described in (3.5).

$$Observability(v@t=t_i)=1-\frac{|MVS(v@t=t_i)|-1}{2^{BW}-1} \tag{3.5}$$

[2] Although the elements other than incorrect value in the Masked Value Set 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 one as masked incorrect values.

3.3 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 observation point in each time frame. The path-oriented computation scheme is defined in (3.6), which can be transformed from (3.3).

$$MVS(v@t=t_i)=\bigcap_{j=1}^{m}\bigcap_{k=1}^{o}\{x \mid f_{v@t=t_i\rightarrow OP_j@t=c_k}(x)=CV(OP_j@t=c_k\} \qquad (3.6)$$

The set $\{x \mid f_{v@t=t_i\rightarrow OP_j@t=c_k}(x)=CV(\mathrm{OP}j@t=c_k)\}$ is defined as the *Masked Value Set* of vertex v at time instance $t=t_i$ with respect to OPj at $t=c_k$ (denoted as $\mathrm{MVS}(v@t=t_i)_{\mathrm{OP}j@t=ck}$). 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 (3.6), if it is possible to derive $\mathrm{MVS}(v@t=t_i)_{\mathrm{OP}j@t=ck}$ for each observation point 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 observation point OP_j at $t=c_k$, an induction-based computation approach is proposed to compute exact $\mathrm{MVS}(v@t=t_i)_{\mathrm{OP}j@t=ck}$, which is introduced in section 3.3.1 and 3.3.2. If there are multiple paths from v at $t=t_i$ to OP_j at $t=c_k$, a quick estimation approach that guarantees lower-bound observability estimations will be applied, which is introduced

in section 3.3.3. Section 3.3.5 introduces the entire algorithm incorporating both of them and section 3.3.4 discusses time-saving strategies.

3.3.1 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 observation point OP_j at a positive edge of clock $t=c_k$. As an example, one such path $<b@t=t_i, a_n, a_{n-1},\ldots, a_2, a_1, OP_j@t=c_k>$ is shown in Figure3-3 and will be used in the following explanations. For the case of a *single path*, we develop an algorithm to compute $MVS(b@t=t_i)_{OP_j@t=c_k}$ as shown in the pseudo code in Figure3-4.

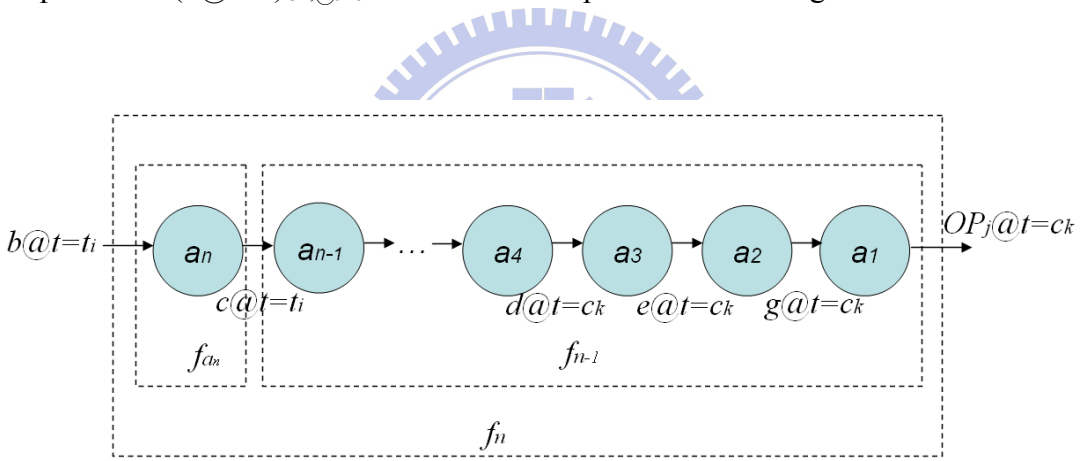


Figure 3-3: A Path from $b@t=t_i$ to $OP_j@t=c_k$

For each observation point at each positive clock edge, the algorithm will recursively call subroutine *MVS_for_vertex* to perform MVS computation and use a Depth First Search (DFS) 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 *fanin* vertices of vertex v will be traversed (line 7). However, if

vertex *v* is a *control vertex*, the *fanin* vertices on the untaken branch(es) will be marked as "*inactive*" and will not be traversed (line 5).

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)
```

Figure 3-4: The pseudo code of MVS computation for a single path

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 *v* f_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 3.3.2 will introduce how to compute CurrentMVS based on PreviousMVS (line 12). The belows explains how this algorithm can derive $MVS(b@t=t_i)_{OPj@t=ck}$ in the case of a *single path* from *b*

at $t=t_i$ to OPj at $t=c_k$.

Theorem.1 *As shown in Figure3-3, function f_n is the composite function of the vertices from a_1 to a_n and comprises f_{an} and f_{n-1}. For an arbitrary value x on the output of vertex b at $t=t_i$, x is in* MVS($b@t=t_i$)$_{OPj@t=ck}$ *if and only if $f_{an}(x)$ is in* MVS($c@t=t_i$)$_{OPj@t=ck}$, *which can be represented as* (3.7).

$$MVS(b@t=t_i)_{OP_j@t=c_k} = \{x \mid f_{a_n}(x) \in MVS(c@t=t_i)_{OP_j@t=c_k}\} \qquad (3.7)$$

Proof :

Claim 1: $MVS(b@t=t_i)_{OP_j@t=c_k} \supseteq \{x \mid f_{a_n}(x) \in MVS(c@t=t_i)_{OP_j@t=c_k}\}$

For each value x in $\{x \mid f_{an}(x) \in MVS(c@t=t_i)_{OPj@t=ck}\}$, x must satisfy $f_{n-1}(f_{an}(x)) = CV(OP_j@t=c_k)$ and thus also satisfy $f_n(x) = CV(OP_j@t=c_k)$. That means that x is in MVS($b@t=t_i$)$_{OPj@t=ck}$. This proves that

$$MVS(b@t=t_i)_{OP_j@t=c_k} \supseteq \{x \mid f_{a_n}(x) \in MVS(c@t=t_i)_{OP_j@t=c_k}\}$$

Claim 2: $MVS(b@t=t_i)_{OP_j@t=c_k} \subseteq \{x \mid f_{a_n}(x) \in MVS(c@t=t_i)_{OP_j@t=c_k}\}$

By way of contradiction, first assume that there is a value x that is in MVS($b@t=t_i$)$_{OPj@t=ck}$ but $f_{an}(x)$ is not in MVS($c@t=t_i$)$_{OPj@t=ck}$. Since x is in MVS($b@t=t_i$)$_{OPj@t=ck}$, then $f_n(x) = CV(OP_j@t=c_k)$ that implies $f_{n-1}(f_{an}(x)) = CV(OP_j@t=c_k)$. This means $f_{an}(x)$ is in MVS($c@t=t_i$)$_{OPj@t=ck}$. This is a contradiction!

From **Claim 1** and **2**, it is proven that

$$MVS(b@t=t_i)_{OP_j@t=c_k} = \{x \mid f_{a_n}(x) \in MVS(c@t=t_i)_{OP_j@t=c_k}\}.$$

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

From the above discussion, it shows that a current MVS set (CurrentMVS) is a *Masked Value Set* of a traversed vertex with respect to OP_j at $t=c_k$. These *Masked Value Sets* will be intersected with the other *Masked Value Sets* of the same vertex with respect to other observation points at different time instances according to (3.6) in the algorithm for MVS computation for a single path. After all the observation points at all the positive clock edges have been applied, the *Masked Value Set* of each traversed vertex in a time frame will be computed and recorded for the later observability calculation.

3.3.2 MVS Formula for Operations

Given a previously computed MVS set (PreivousMVS), a vertex v, and one of

v's *fanin* vertex u, computing CurrentMVS is to find the set of all the values at u's output y_u that make the function of v f_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 u y_u and it is recorded in the dump-file. The formula to compute Sub_CurrentMVS$_p$ is summarized in the third column of Table.1. The column "condition" shows the necessary conditions for the result of Sub_CurrentMVS$_p$. If the conditions are not met, in most of cases, Sub_CurrentMVS$_p$ = $\varnothing$, except for *comparisons*. The following explains how to derive Sub_CurrentMVS$_p$ for some representative operations.

Table 3-1 The 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.

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 of 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.1. 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 *v* y_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* ">","<", and 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~b_1-1]}. The derivations for other comparisons are quite similar.

4) *Right shift* ">>" and *left shift* "<<":

Either *right shift* or *left shift* by the amount b_1 incurs *information loss*. The "*[i:j]*" formula can tackle this. As illustrated in Figure 3-5 (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.

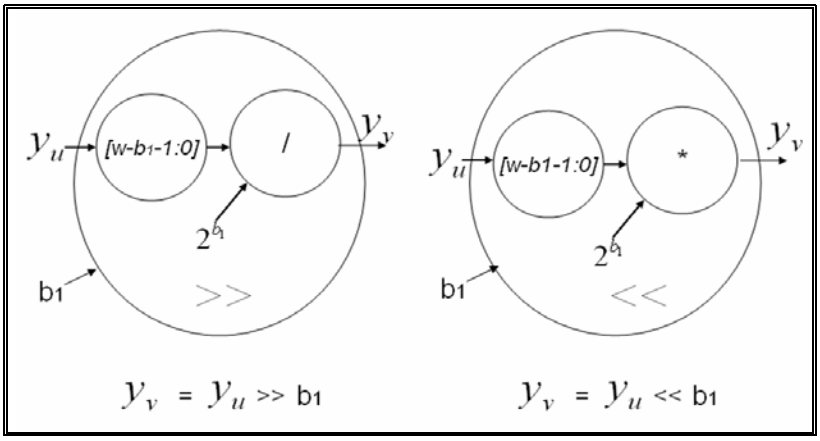


Figure 3-5: Modeling *information loss* in *right shift* and *left shift*

If the formulas listed in the third column of Table 3-1 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

$$\bigcup_{k=0}^{2^{4-1-1}-1}\{[0\cdot 2^0 + k\cdot 2^{1+1} \sim 0\cdot 2^0 + k\cdot 2^{1+1} + 2^0 - 1]\} = \{0, 4, 8, 12\} \qquad (3.8)$$

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.

It is obvious that the computation using the formulas in Table 3-1 may take lots of time if there are many elements in PreviousMVS. In fact, the formulas in Table 3-1 are not what we really used in MVS computation algorithm. We have the following observations used to transform the formulas in Table 3-1 to improve the efficiency of the formulas.

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 above three sets can be represented more concisely as

$$\bigcup_{k=0}^{2^{4-1-1}-1}\{[0\cdot 2^0+k\cdot 2^{1+1}\sim 2\cdot 2^0+k\cdot 2^{1+1}+2^0-1]\}=\bigcup_{k=0}^{3}\{[k\cdot 4\sim 2+k\cdot 4]\} \qquad (3.9)$$

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

$$\bigcup_{k=0}^{2^{w-i-1}-1}\{[p\times 2^j+k\times 2^{i+1}\sim q\times 2^j+k\times 2^{i+1}+2^j-1]\} \qquad (3.10)$$

The "*[i:j]*" MVS formula is derived now and listed in the third column of Table3-2. The operation "<<" is another example of how to derive the "<<" formula listed in the third column of Table 3-2. First try to find the smallest integer p' in the set $\{[p\sim q]\}$, which satisfies $p'\%2^{b_i}=0$. 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 $p'+2^{b_i}$ is in the range of p to

q. If so, the union of the two result sets obtained by *p'* and *p'*+2^{b1} can be represented as

$$\bigcup_{k=0}^{2^{b_1}-1} \{[k \cdot 2^{b_1} + p'/2^{b_1} \sim k \cdot 2^{b_1} + (p'+2^{b1})/2^{b_1}]\} \quad (3.11)$$

Repeating the derivations above produces the formula in the third column in Table3-2 at page 60. For a subset of integers {[*p*~*q*]} in PreviousMVS, applying the MVS formulas listed in the third column in Table 3-2 can derive results much more quickly than applying the formulas in Table 3-1. In addition, all the integers in the subset {[*p*~*q*]} can be memorized by recording only *p*, *q*, and the special tag "~". This storage format enhances memory usage efficiency and alleviates the *memory explosion* problem.

Table 3-2: The MVS formulas for HDL operations

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

3.3.3 MVS Estimations for Multiple Paths

The algorithm shown in Figure 3-4 can compute the exact MVS of vertex b in time frame $t=t_i$ with respect to an observation point 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 *paths* from b at $t=t_i$ to OP_j at $t=c_k$, another approach is necessary because possible propagation methods become more complex.

In tag-based approaches [18,19], the authors simply put *unknown tags* "?" on the *reconvergent paths* instead of computing exact solutions. If *unknown tags* are propagated to observation points, they seem to be considered as *not covered* with respect to OCCOM in a conservative way.

In order to reduce the complexity, we adopt an estimation that is similar to tag-based approaches. If there are multiple paths from v at $t=t_i$ to an observation point OP_j at $t=c_k$, the universe ($\boldsymbol{U}$) is used instead of real $MVS(b@t=t_i)_{OPj@t=ck}$ in the intersection operation. This estimation result obtained using the universe must include the exact result obtained by intersecting with the real $MVS(b@t=t_i)_{OPj@t=ck}$ because the universe includes $MVS(b@t=t_i)_{OPj@t=ck}$. Consequently, this estimation result has a larger MVS set, which turns out to be less observable according to the definition of observability in (3.5). 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.

3.3.4 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. Limited-Traversed-Frame (LTF) strategy saves additional time at the expense of accuracy loss. However, it can always have a lower bound of observability (a pessimistic estimation).

3.3.4.1 Bounded Traversal Strategy

In our observability computation, after some backward traversals, there are MVS sets recorded on vertices that have been traversed. As shown in Figure 3-6, let a vertex v in time frame $t=t_n$ be a vertex that was traversed and v' be one of v's *fanin* vertices that was also traversed. Hence, MVS($v@t=t_n$) and MVS($v'@t=t_n$) are already recorded on v and v'. And, MVS($v'@t=t_n$) should be $\{ x \mid f_v(x)$ MVS($v@t=t_n$) $\}$ according to the CurrentMVS computation shown in line 12 of the *MVS_for_Vertex* pseudo code in Figure 3-4.

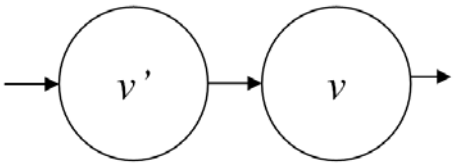


Figure 3-6: Vertex v and one of it *fanin* vertex v'

If another backward traversal from an observation point 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* pseudo code. 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 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($v'@t=t_n$) $\subseteq$ $\{ x \mid f_v(x) \in$* 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 the MVS($v@t=t_n$). That is, MVS($v'@t=t_n$) is the set $\{x \mid f_v(x) \in$ 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 *fanin* 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 *fanin* 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.

3.3.4.2 Limited-Traversed-Frame (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 a Limited- Traversed-Frame (LTF) strategy, which provides an optional and flexible trade-off 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 *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 design under validation. The reason is given below.

For a vertex u in time frame $t=c_k$, if expanded frames are not limited, each *Masked Value Set* 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 $\{MVS_1, MVS_2, \ldots, MVS_m\}$. 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 the obtained MVSs are $\{MVS_1,MVS_2,\ldots,MVS_n\}$, where $n<m$. The intersection of all the MVSs in the set $\{MVS_1,MVS_2,\ldots,MVS_n\}$ includes the intersection of all the MVSs in the set $\{MVS_1,MVS_2,\ldots,MVS_m\}$. Larger MVS set intersections turn out to be less observable according to the definition of observability in (3.5). Therefore, our LTF strategy also guarantees lower-bound estimations of observability.

3.3.5 Algorithm of Observability Computation

The entire algorithm of our observability computation is abstracted as the pseudo code in Figure 3-7. The entire algorithm incorporates 1) MVS estimation for single path, 2) MVS computation for multiple paths, 3) Bounding-traversal strategy, and 4) Limited-Traversed-Frame (LTF) strategy. This algorithm is modified from the algorithm shown in Figure 3-4 and thus it is quite similar to it. 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 multiple paths. During traversal(s) starting from an observation point (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 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 multiple paths from v at $t=t_i$ to StartOP at StartTime. Then the MVSforRecovery($v@t=t_i$) subroutine is used to resume the status of 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 *fanin* vertex to traverse.

Some preparations are required before observability computation can begin. The *3-address code generations* and the *conditional statement modification* developed in [19] must be conducted first for the information required in computing MVSs for control vertices (conditional statements). The detailed *conditional statement modification* algorithm can be found in [19] and in section 2.1.

Preparation Phases:
1: *3-address Code Generation* and *Conditional statement odification*
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 multiple paths ***
1: **if** traversed for first time in traversal starting from StartOP 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 *Bounding traversal*
6: **return**
7: MVSforRecovery($v@t=t_j$) = MVS($v@t=t_j$)
8: MVS($v@t=t_j$) = MVS($v@t=t_j$) $\cap$ PreviousMVS
9: **else** //Multiple 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 multiple 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 $\{ x \mid f_v(x) \in \text{PreviousCVS} \}$
20: MVS_Com_for_Vertex(CurrentMVS, u, StartOP, StartTime, t_h, *frame_limit*)

Figure 3-7: The pseudo code of observability computation algorithm

3.3.6 An Illustration Example

The example in Figure3-1 can also be used to demonstrate the processes of our observability computation. We first construct the control/data flow graph of the DUV. The CDFG of the HDL code in Figure 3-1 is shown in Figure 3-8(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=∞.

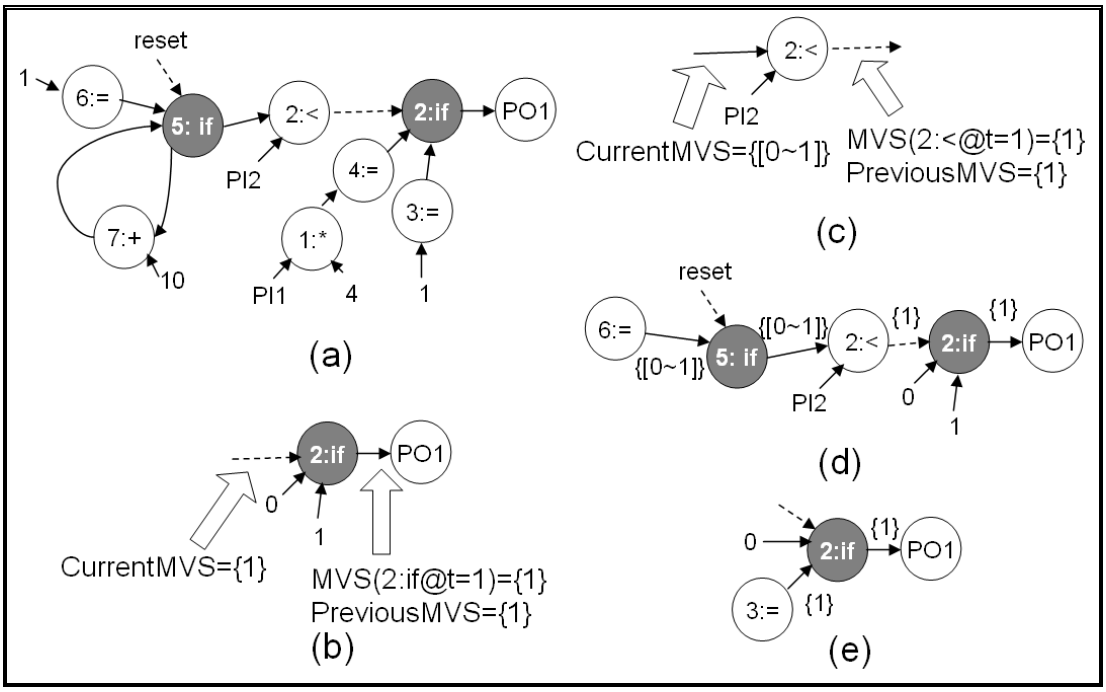


Figure 3-8: Computation processes starting from PO1 at t=1

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 Figure 3-8(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. Therfore, there are two fanin 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 Figure 3-8(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 Figure 3-8(d), where each set of integers aside an edge is the recorded MVS. Since vertex “6:=” in time frame t=1 has no *fanin* vertex, the computation will traverse another *fanin* vertex “3:=” of vertex “2:if.” Repeatedly calling subroutine *MVS_Com_for_Vertex* can produce the results shown in Figure 3-8(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 Figure 3-9(a) and (c). When the computation reaches vertex “5:if” in time frame t=1, PreviousMVS {[0~5], [8~15]} will include MVS(5:if@t=1)={[0~1]}. The bounding traversal condition is satisfied and the traversal is bounded here.

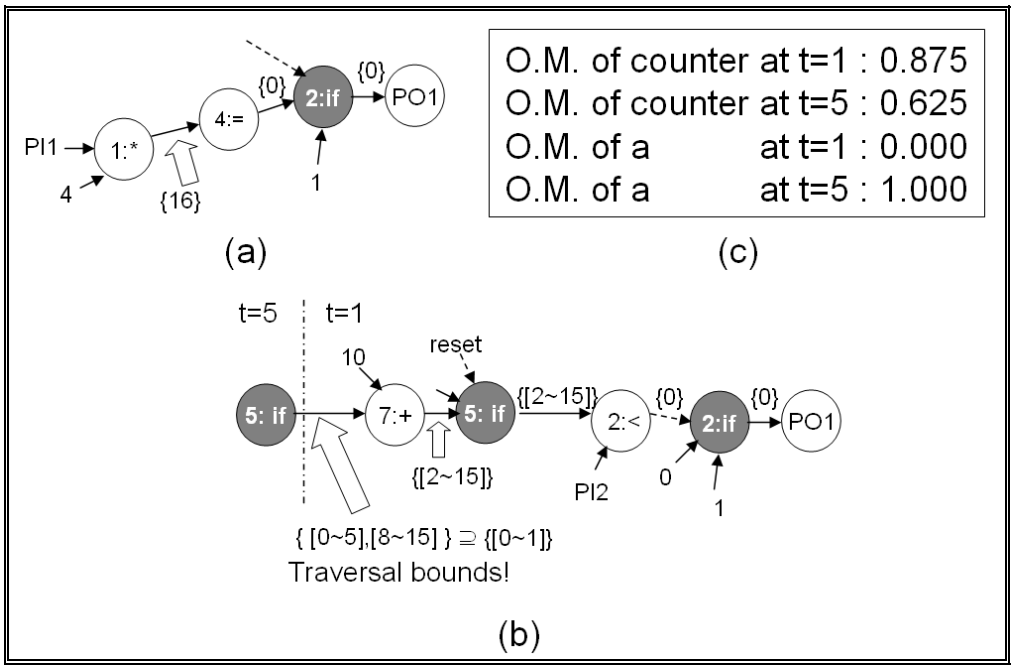


Figure 3-9: Observability computation results

After all the observation points 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 Figure 3-9(b). 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* can not represent intermediates values between 1 and 0. The rules thus determine that tags injected on *counter* can propagate to PO1 at t=5. This induce some inaccuracy and even worse overestimates the actual likelihood that an erroneous effect propagates through "counter<PI2". Experimental results in section VI also shows the same situation of overestimation as we discussed above.

3.4 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 design under validation. 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 design under validation. As shown in Figure 3-10(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 Figure 3-10(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.

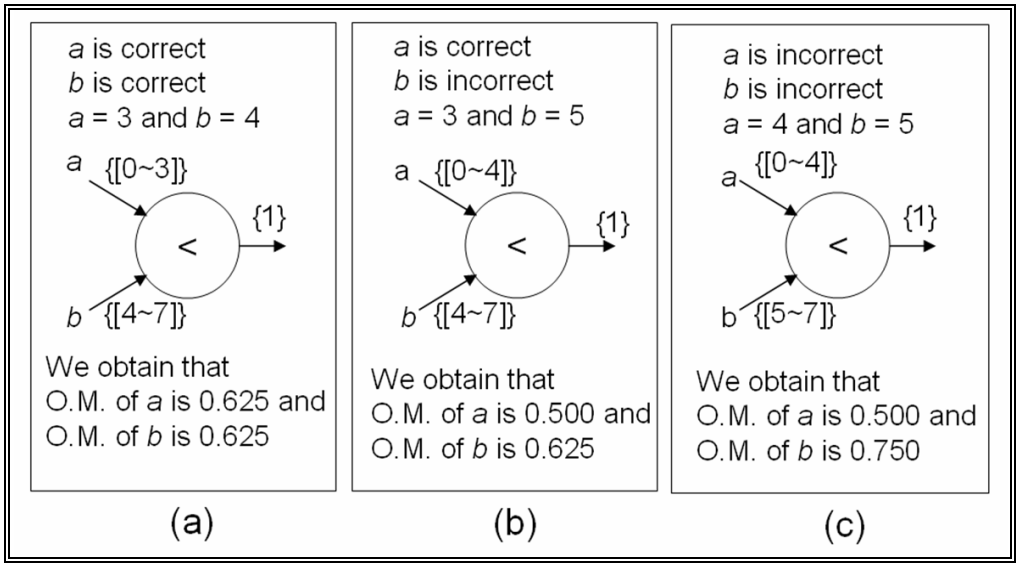


Figure 3-10: Observability analysis with correct and incorrect values

The situation can become even worse. As shown in Figure 3-10(c), the value of *a* and *b* are 4 and 5, which are both different from their correct values. However, our approach can still derive that the observability 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 Figure 3-10(c), tag $_{\Delta}$ 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 design under validation, *tags* may provide incorrect predications on error propagation.

Besides the cases shown in Figure 3-10 (a), (b), and (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.

3.5 Experimental Results

We conducted experiments on a subset of ITC'99 benchmark in VHDL and four designs written in Verilog HDL. The four designs are as follows: pcpu is a simple 32-bit pipelined DLX CPU; div16 is a 16-bit divider; blkJ is a controller of black jack card game, and Mtrx implements a two by two matrix multiplication. The information for these design cases is presented in Table 3-3, 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 (~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 Statements 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 observation points. Consequently, OSCOM filters out these exercitations of statements, acting as a more stringent code coverage metric than statement coverage metric.

Table 3-3 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

We also conducted experiments to compare the *propagation probabilities* [29,44,45], *tag simulation calculus* [18,19] and our observability measures. We

[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.

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* [47]. 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 for 300 times. The overall results are reported in Table 3-3.

We then calculate the three observability measures above for the detected or undetected bug. *Propagation probabilities* (PP) were calculated according to the approach proposed in [29], which was introduced in chapter 2.2. 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.

We calculated tag-based observability (Tag) is calculated according to the *tag simulation calculus proposed* in [19], which was introduced in chapter 2.1. 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* = ∞ ($\text{Ours}_{\text{FL=}\infty}$). For the 300 iterations we ran, the average values of these observability measures for both detected and undetected design bugs are listed in Table 3-3.

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 observation points. In addition, the values of tag-based observability measures for undetected bugs tended 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 design under validation 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 propagation probability 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 limited-traversed-frame 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 (FL=∞).

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 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 3-4. 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 ration of the computation time for observability to the plain HDL simulation time, is also provided in Table 3-4 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.

Table 3-4: Performance comparison with propagation probability

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	$5.4*10^4$

3.6 Summary

In this chapter, 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 [29]. 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 observation points, such as the extension works of OCCOM [18,19]. 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.

Chapter 4

Accurate Error Candidate Rank Ordering for Efficient HDL Debugging

4.1 Debugging Priority for Quick Error Localization in Error Space

Deriving a reduced set of error candidates is helpful for HDL debugging. However, the derived error candidate set (called error space in this work) can still contain many error candidates and identifying true design errors by examining candidates one by one still requires much efforts and time. An interesting technique, called *debugging priority*, has been proposed for accelerating error searching in the derived candidate set [46]. A measurement, called Confidence Score (CS), has been developed to assess the likelihood of correctness for each error candidate.

The formula of CS is simple. Each sensitized statement of a CPO (a primary output with correct simulation value) can get one point of CS. We use the following example shown in Figure 4-1 to illustrate what sensitized statement is. The evaluation result of the decision "*if(sel1)…else…*" is "*TRUE*" and the evaluation result of the decision "*if(sel2)…else…*" is "*FALSE*". Therefore, only statements *f2* and *f4* are possible to affect the value of PO1 and are also observed by PO1. These two statements *f2* and *f4* are defined as the *sensitized statements* of PO1 (SS(PO1)). Each

sensitized statement tends to be a correct statement because if it were a statement with design errors, the erroneous effects caused by this erroneous statement should cause the value of PO1 to be inconsistent with the expected value. However, there are two situations mentioned in [46], in which sensitized statements with design errors can not cause any observable incorrect behavior at PO1. One is that the design error is "non-activated" and the other is error masking. That is the erroneous effects are masked so that the value of PO1 can still remain correct.

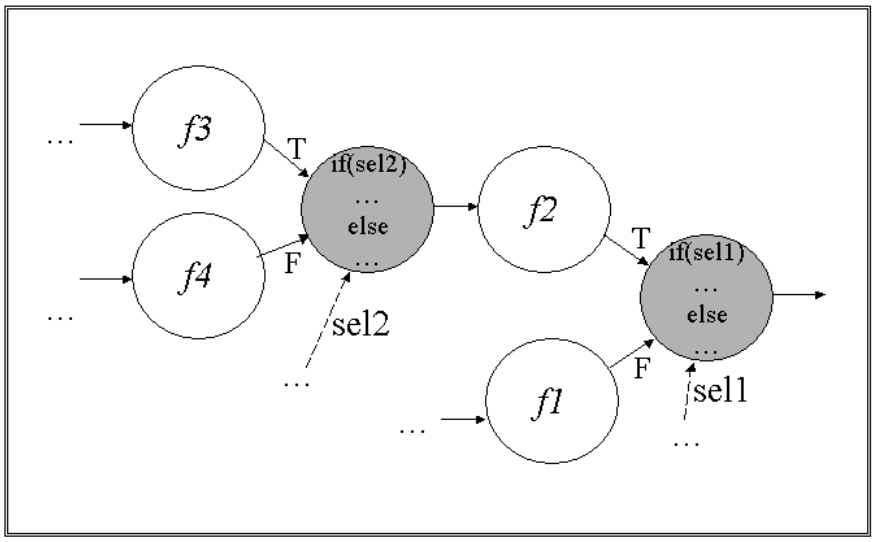


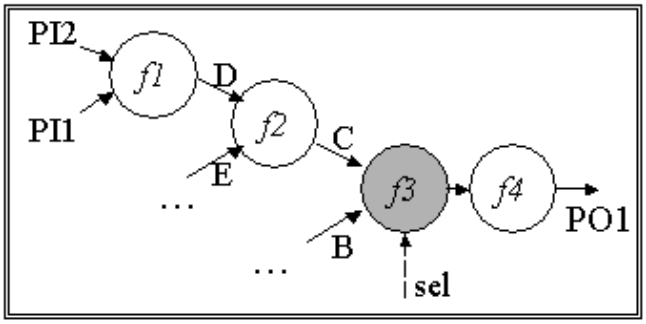
Figure 4-1: An example of sensitized path

What error masking is ? The authors used a simple example shown in Figure 4-2 to explain. The applied input vector is "*PI1=2'b10; PI2=2'b01;"* and the values of all variables are *"E=2'b10; sel=1'b1; B=2'b11; D=2'b11; C=2'b10; A=2'b10; PO1=2'b01;"*. If the statement *f1* becomes an erroneous statement "*D=PI1*;", the value of *D* will become *2'b10* instead of *2'b11*. However, the output of the statement

f2 is still *C=2'b10*. There is no syndrome shown at *PO1* because the activated error is masked by the statement *f2*.

```
D = PI1 + PI2;              //f1
C = D & E ;                 //f2
if (sel ) A=C; else A=B;    //f3
PO1 = A;                    //f4
```

(a) A HDL code fragment



(b) CDFG of the HDL code of (a)

Figure 4-2: An example of error masking

The authors think the probability that errors are not activated, *P*(*non-activated*), is generally very small. For example, if the correct statement is "*assign c = a + b;*" and the erroneous one is "*assign c= a * b*", only applying the input patterns "*a=2; b=2;*" and "*a=0; b=0;*" may generate the same outputs for both statements. Otherwise, errors are activated. In addition, the probability *P(mask|activated)* is not high in general as well. Given *P(non-activated)* and *P(mask | activated)*, the authors estimated the possibility for the sensitized statements to be erroneous while their corresponding PO is correct, denoted as *P(error|CPOi)* as shown in equation (4.1).

$$P(error|CPOi) = P(non\text{-}activated) + P(activated) * P(mask|activated) \quad (4.1)$$

Since *P(non-activated)* and *P(mask|activated)* are generally not high, *P(error|CPOi)* is generally not high, either. For instance, if *P(non-activated)*=0.1 and *P(mask|activated)*=0.3, *P(error|CPOi)* =0.1+0.9*0.3=0.37. For each PO with correct simulation value at each simulation cycle, the SS(CPOi) will be given one point because P(error|CPOi) is generally not high. If a statement gets 5 points, which is denoted as P(error|5CPOs), the probability for it to be erroneous can be estimated as equation (4.2).

$$P(error|5CPOs) = P(error|CPO1)P(error|CPO2)P(error|CPO3)P(error|CPO4)$$
$$P(error|CPO5) \quad (4.2)$$

Assume that each event of the probability is roughly independent to each other. If we take the value of *P(error|CPOi)* calculated previously for each *P(error|CPOi)*, *P(error|5 CPOs)* can be roughly estimated as 0.375 = 0.007. Therefore, **the more points a sensitized statement has; the less possible it is to be a design error**. This is why their CS suitable and capable to represent the confidence level of a statement on its correctness.

By sorting error candidates according to the CS, error candidates obtained in error space identification are displayed in a prioritized order, from the most likely to the least likely one. With the ranked order, the authors expect that true design errors

can be put in the first few lines such that they should be found by designers in a few times of examinations if designers search errors according to the order. In their experimental results, debugging priority indeed can make design errors displayed in the front of the list of error space. Thus, this technique helps reduce the efforts spent on error searching in an *error space*.

4.2 Challenges on Accurate Error Candidate Rank Ordering

Using CS to estimate the likelihood of correctness for error candidates can only work under the assumption that *P(non-activated)* and *P(mask|activated)* are both small. However, this is not always the case. Some HDL operations tend to mask erroneous effects by nature. For example, for a signal ***a*** [15:0], if ***b*** is assigned to be ***a***[0], i.e. ***b*** is left-hand variable of statement "***b*** = ***a***[0]". It is obvious that this bit selection operation "[0]" tends to mask an erroneous effect on signal ***a*** if there is any erroneous value on signal ***a***. If the erroneous effects were masked from being observed at the POs, sensitized statements may get CS points even if design errors hiding within them. We intend to use the following to emphasize this point.

Suppose that the HDL code a designer intends to write is the Verilog HDL code in Figure 4-3(a). The design described in the HDL code has only one PO, **PO1**, on which simulation values are compared against the expected values to check the correctness of the design. The clock period of the clock signal **clk** is assumed to be 10ns. If the HDL code is simulated with the input stimulus shown in Figure 4-3(b), we can obtain the simulation result as represented in Figure 4-3(c), in which **PO1** is equal

to 4 from Time =1 to Time = 25. The simulation result in Figure 4-3(c) can be considered as the specification or the expected value of **PO1** at Time =1 to Time = 25 because the HDL code in Figure 4-3(a) is what the designer intends to write.

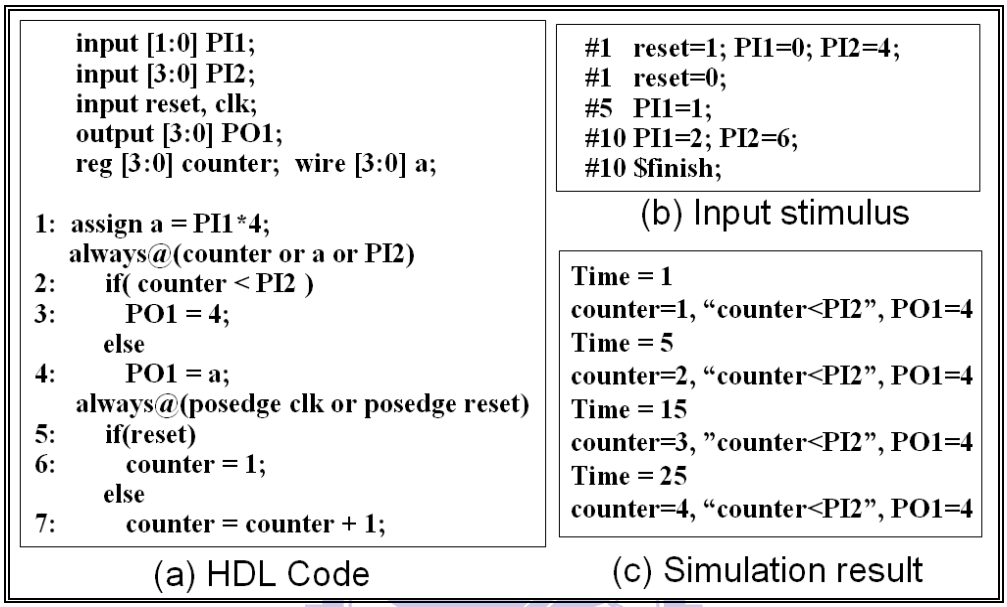


Figure 4-3: A HDL code fragment

However, if the statement at line 7 (denoted as S7) "**counter** = **counter** + 1" is carelessly written into "**counter** = **counter** + 2", the simulation result shall become the one shown in Figure 4-4(a). It can be seen that the simulation value of **PO1** at Time = 25 in Figure 4-4(a) is different from the specification (the expected value of PO1) shown in Figure 4-3(c).

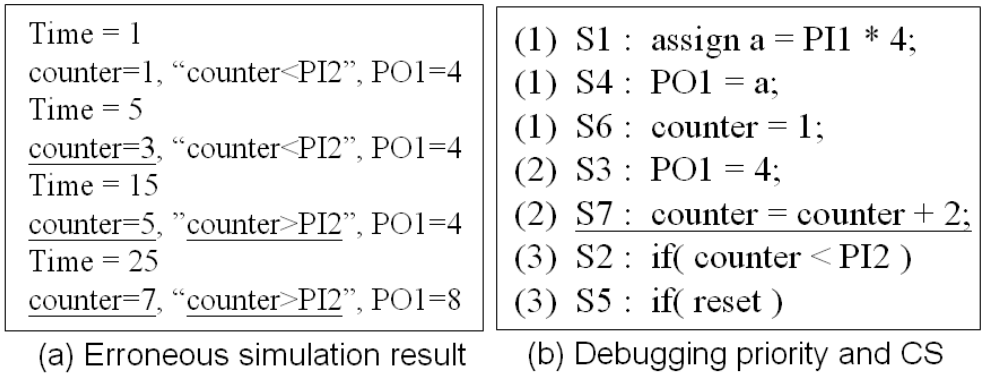


Figure 4-4: Erroneous Simulation Results and *Debugging priority*

According to the definition in [46], **PO1** is an Erroneous Primary Output (EPO) and the clock cycle from t=15 to t=25 is the Error-Occurring clock Cycle (EOC). By using the error space identification approach in [46], an *error space*, {**S1, S2, S3, S4, S5, S6, S7**}, can be obtained.

After obtaining the *error space*, the CS for each error candidate should be calculated for *debugging priority*. Each sensitized statement of a CPO (a primary output with correct simulation) at a time instance before EOC gets one CS point. Finding sensitized statements requires backward tracing from the PO's in the reverse direction of the data flow until 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 to 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 back traverses the True branch and the control signal , which is then reset. The obtained sensitized statements for **PO1** at Time=1 are {**S5**, **S6**} and they both receive one CS point.

As can be seen from the above, 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 Figure 4-4(b) is obtained. The numbers within the 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 at the first line but rather in the fifth in Figure 4-4(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. The reason why design error **S7** is placed at the fifth is because **S7** receives two CS points due to the fact that the erroneous values caused by **S7** are masked twice on its way propagating to **PO1**.

The first *error masking* occurs at Time=*5*. It can be seen that the erroneous statement **S7** causes an incorrect value (3 is different from the correct value 2 shown in Figure 4-3(c) as we 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 value (2) and the incorrect one (3) cause the same result at the output of the operation "**counter**<**PI2**", i.e they are both smaller than the value of **PO2** (4).

Similar *error masking* also occurs at Time=*15*. Although the incorrect value of the *counter* is propagated through the output of "**counter**<**PI2**", i.e., causing it to be FALSE. However, the incorrect result, FALSE, does not alter the value of **PO1** (4) i.e. signal *a* is 4 at Time=15. It is masked by the conditional operation "if(…)….else …."

and can not cause incorrect values at **PO1** at Time=15. Due to the fact that CS does not consider the possible *error masking* that may caused by the operation "**counter**<**PI2**" and the conditional operation "if(…)….else ….", **S7** is given a CS score of two points. This makes **S7** put at line 5 in the candidate list in Figure 4-4(b). The accuracy of the *debugging priority* is reduced due to the lack of considering *error masking* of the CS.

4.3 Probabilistic Confidence Score for Accurate Debugging Priority

Observing the disadvantage of *confidence score* (CS), we intend to estimate of the Likelihood Of Error Masking (LOEM) for a Sensitized Statement (SS) to assess the score the SS can receive. If the LOEM of an arbitrary SS SS_i is quite low, error masking is almost impossible to occur on the paths from SS_i way to 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.

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 Figure 4-3(a) is used as an example and

its CDFG is constructed as shown in Figure 4-5. Vertex "1:*" corresponds to operation "a=PI1*4" in the statement at line 1 (S1). Function $f_{1:*}$ is multiplication "*" and $y_{1:*}$ is signal *a*. Vertex "2:if(…)….else…." corresponds to the operation "if(…) … else ..." at lines 2 to 4. The functionality of vertex "2:if(…)….else…." is quite similar to that of a multiplexer. Vertex PO1 is a special vertex representing the only PO, PO1, of the circuit.

Edge $(v, u) \in \boldsymbol{E}$ indicates that the input of vertex *u* is data dependent on the output of *v*. As shown in Figure 4-5, an edge (*1:**, *4:=*) exists since the operation "4:=" takes the output of vertex "1:*" as its input. The *fanout* of *v* is a set of vertices *u* such that there is an edge from *v* to *u*. The *fanin* 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 $<v_0, v_1, v_2,\ldots, v_k>$ of vertices such that $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in \boldsymbol{E}$.

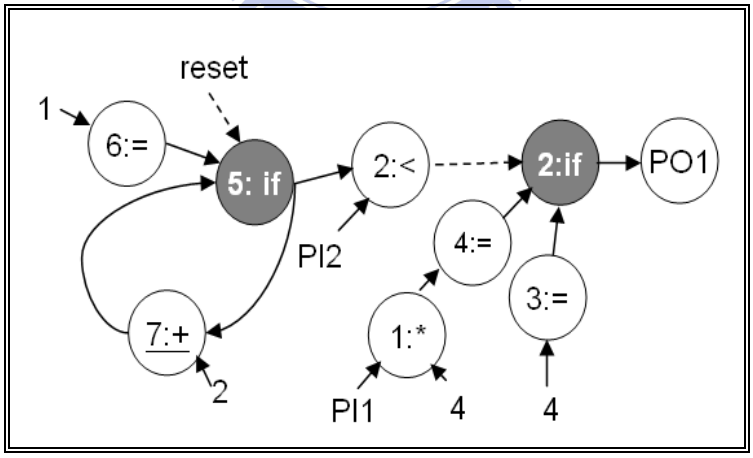


Figure 4-5: The CDFG of the HDL code in Figure 4-1

Suppose that verification finds incorrect circuit behavior at the n_{th} positive edge

of the clock signal $t=c_n$[4]. This special positive edge of the clock is called Error-Occurring Edge (EOE). Assume that the faulty DUV has m POs {PO_1, PO_2, …, PO_m} and n-1 clock cycles pass before the EOE ($t=c_n$).

To introduce how we model *error masking* 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 clock before $t=c_n$. Otherwise, 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 EOE, the incorrect value w is *masked* by some vertices on the paths from statement v at $t=t_i$ (denoted as $v@t=t_i$) to PO_j at $t=c_k$ (denoted as $PO_j@t=c_k$), causing the simulation value of PO_j to be the same as the correct value at $t=c_k$.

$$f_{v@t=t_i \rightarrow PO_j@t=c_k}(w) = CV(PO_j@t=c_k) \tag{4.3}$$

where $f_{v@t=t_i \rightarrow PO_j@t=c_k}$ is the function of the paths from v in time frame $t=t_i$ to PO_j in time frame $t=c_k$ and $CV(PO_j@t=c_k)$ 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 them 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 $t=c_k$

[4] 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 DUV is a falling-edge triggered or double-edge triggered design, the modeling and the computation algorithm can be easily changed to fit to it.

before EOE, the function of the path(s) from vertex v at $t=t_i$ that goes 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 above description can be modeled in (4.4).

$$\bigcap_{j=1}^{m}\bigcap_{k=0}^{n-1} f_{v@t=t_i \to PO_j @t=c_k}(w) = CV(PO_j @t = c_k) \tag{4.4}$$

We now consider the likelihood that the incorrect value w truly exists on y_v but is masked from causing any incorrect values on POs at any time instances before EOE. We first notice that all the possible values of y_v that can satisfy (4.4) forms a special set of values. We call it the *Masked Value Set* (MVS) of vertex v at time instance $t=t_i$ (denoted as MVS($v@t=t_i$)). Its formula is given in (4.5).

$$MVS(v@t=t_i) = \{x \mid \bigcap_{j=1}^{m}\bigcap_{k=0}^{n-1} f_{v@t=t_i \to PO_j @t=c_k}(x) = CV(PO_j @t = c_k)\} \tag{4.5}$$

Each element in MVS($v@t=t_i$) retains the correct values of all POs at all the rising edges of clock before the EOE, no matter 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 MVS($v@t=t_i$). This justifies the existence of MVS($v@t=t_i$). If 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 MVS($v@t=t_i$) and *error masking* 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 to become an incorrect value that remains unrevealed at all the rising edges of the clock before EOE. The correctness of statement v is less obvious. In other words, the more elements MVS($v@t=t_i$) contains; the more likely that the simulated value of v at $t=t_i$ is a masked incorrect value. Hence, we define the Likelihood Of Error Masking (LOEM) of statement v at time instance $t=t_i$ as (4.6). Its complement is the likelihood that an erroneous value of v at $t=t_i$ is propagated to at least one PO before EOE and observed (the Likelihood Of Error Propagating (LOEP) of v at $t=t_i$). We show its formula in (4.7)

$$LOEM(v@t=t_i)=\frac{|MVS(v@t=t_i)|-1}{2^{BW}-1} \tag{4.6}$$

$$LOEP(v@t=t_i)=1-\frac{|MVS(v@t=t_i)|-1}{2^{BW}-1} \tag{4.7}$$

where BW is the bit width of the output of variable v.

In the given input value change dump file, the output variable y_v of an arbitrary statement v can have many times of value changes, say l times, at different time instances before 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 LOEP($v@t=t_i$). The Probabilistic Confidence Score of v (PCS(v)) is defined as the maximum among these LOEP values, as described in (4.8).

$$PCS(v)=MAX\{LOEP(v@t=t_i\}, \text{where} t_i \in \{t=t_1, t=t_2,\ldots, t=t_l\} \tag{4.8}$$

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 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 how to efficiently compute the *Masked Value Sets* of each error candidate at different time instances before an EOE.

4.4 An Efficient Probabilistic Confidence Score Calculation Algorithm

The proposed PCS computation algorithm is a topology-based analysis with *time frame expansion* to handle 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 $t=t_i$ that goes to any PO in each time frame before EOE. This path-oriented computation scheme is defined in (4.9), which can be derived from (4.5).

$$MVS(v@t=t_i)=\bigcap_{j=1}^{m}\bigcap_{k=0}^{n-1}\{x \mid f_{v@t=t_i\rightarrow PO_j@t=c_k}(x)=CV(PO_j@t=c_k\} \qquad (4.9)$$

The set $\{x \mid f_{v@t=t_i\rightarrow PO_j@t=c_k}(x)= CV(\mathrm{PO}_j@t=c_k)\}$ is defined as the *Masked Value*

Set of vertex v at time instance $t=t_i$ with respect to POj at $t=c_k$ (denoted as MVS($v@t=t_i$)$_{POj@t=ck}$). 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 (4.9), if it is possible to derive MVS($v@t=t_i$)$_{POj@t=ck}$ for each PO$_j$ at each time frame $t=c_k$, then intersecting these sets yields MVS($v@t=t_i$). After deriving MVS($v@t=t_i$), PCS of vertex v can be obtained according to formula (4.7) and (4.8).

We may observe that the definition and the derivation of Probabilistic Confidence Score are based on LOEP and Masked Value Set, which were introduced in section 3.2 before. Thus, the computation algorithm for MVS's and LOEPs can be applied to PCS computation with some modifications. If there is exactly one path from v at $t=t_i$ to a PO PO$_j$ at $t=c_k$, the induction-based computation approach introduced in section 3.3.1 and section 3.3.2 can be applied to compute exact MVS($v@t=t_i$)$_{POj@t=ck}$. If there are multiple paths from v at $t=t_i$ to PO$_j$ at $t=c_k$, the quick estimation approach introduced in section 3.3.3 that guarantees lower-bound LOEP estimations will be applied. In addition, to avoid unnecessary back-tracing, the bounded traversal strategy introduced in section 3.3.4.1 is also applied. The entire PCS computation algorithm is represented as the pseudo code shown in Figure 4-6, which incorporates each part mentioned above.

The input of this algorithm are 1) the Design Under Validation (DUV) described in a HDL, 2) the value change dumpfile during simulation, 3) the Error-occurring Edge (EOE), and 4) an *error space* obtained by any *error space* identification approach.

```
PCS_computation (DUV, Dumpfile, EOE, Error space)
1: 3-address Code Generation and Conditional statement odification
2: CDFG Construction
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)
8:    Calculate PCS with the computed MVSs


MVS_Com_for_Vertex(PreviousMVS, vertex v, StartPO, StartTime, time t_j)
//*** Modification for incorporating MVS computation for multiple paths ***
1:    if traversed for first time in traversal starting from StartPO 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 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) ∩ PreviousMVS
9:    else //Multiple 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 multiple 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
17:            return
18:      Compute CurrentMVS, which is { x | f_v(x) ∈ PreviousCVS }
19:      MVS_Com_for_Vertex(CurrentMVS, u, StartPO, StartTime, t_h)
```

Figure 4-6: Pseudo-code of PCS Computation Algorithm

During traversal(s) that starts from a PO (StartPO) at a time instance (StartTime), if vertex v is visited for the first time, a *single path* case is temporarily assumed. The

PreviousMVS will be intersected with MVS($v@t=t_i$), which is already the intersection of many PreviousMVSs. However, if this vertex v is found traversed for two or more times in the traversal starting from StartPO at StartTime, there are multiple paths from v at $t=t_i$ to StartPO at StartTime. Then, the previously recorded MVSforRecovery($v@t=t_i$) is used to chancel intersections made in this traversal before.

During the MVS computation process, if the condition at line 5 of the *MVS_Com_for_vertex* subroutine is met, according to **Theorem 2** introduced in section 3.4, additional traversal and MVS computation cannot affect the already computed MVS. Thus, an immediate return from subroutine *MVS_Com_for_vertex* at line 6 is made and precious computation time is saved.

The preparation phases of this algorithm are shown at lines 1 and 2. The *3-address code generations* and the *conditional statement modification* introduced in section 3.3.3 must be conducted first for the information required in MVS computation for control vertices (conditional statements). The detailed *conditional statement modification* algorithm can be found in [19]. Next, a CDFG based on the input DUV described in a HDL is constructed.

The example in Figure 4-3 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 Figure4-3 is constructed as shown in Figure 4-7(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.

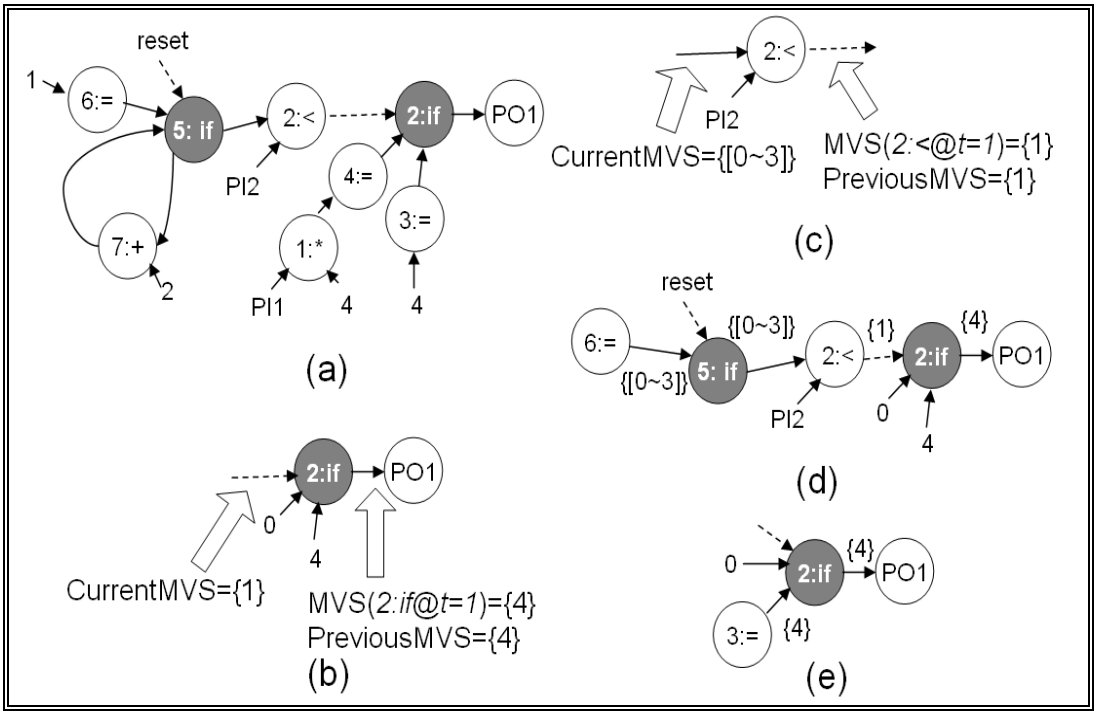


Figure 4-7: Computation processes starting from PO1 at t=1

When subroutine *MVS_Com_for_vertex* is called for the first time, the traversal reaches vertex "2:if" in time frame t=1 also for the first time. As shown in Figure 4-7(b), the recorded MVS(2: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 fanin vertices "2:<" and "3:=" for further backward traversals. We decide to traverse "2:<" before traversing to "3:=" and compute CurrentMVS. Due to the fact that PreviousMVS is {4}, the MVS computation for conditional statements will be used and we obtain CurrentMVS {1}. The computation process is shown in Figure 4-7(b).

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 Figure 4-7(c). Similar computations is repeated by recursively vertex by vertex vertex "6:=" in time frame *t=1* is reached. The computation results along the traversal from "2:if" to "6:=" are

shown in Figure 4-7(d), where each set of integers next to an edge is the recorded MVS. Since vertex "6:=" in time frame *t=1* has no *fanin* vertex, the computation will traverse another *fanin* vertex "3:=" of vertex "2:if." The repetitious calling of subroutine *MVS_Com_for_Vertex* can produce the results shown in Figure 4-7(e).

After completing the traversals and MVS computations starting from PO1 in time frame *t=1*, continue the backward-traversal based MVS computation from PO1 in time frame t=5 can produce the results shown in Figure 4-8(a) and (b). When the computation reaches vertex "5:if" in time frame t=1, PreviousMVS {[0~15]} will include MVS(5:if@t=1)={[0~3]}. The bounding traversal condition is satisfied and the traversal is bounded here.

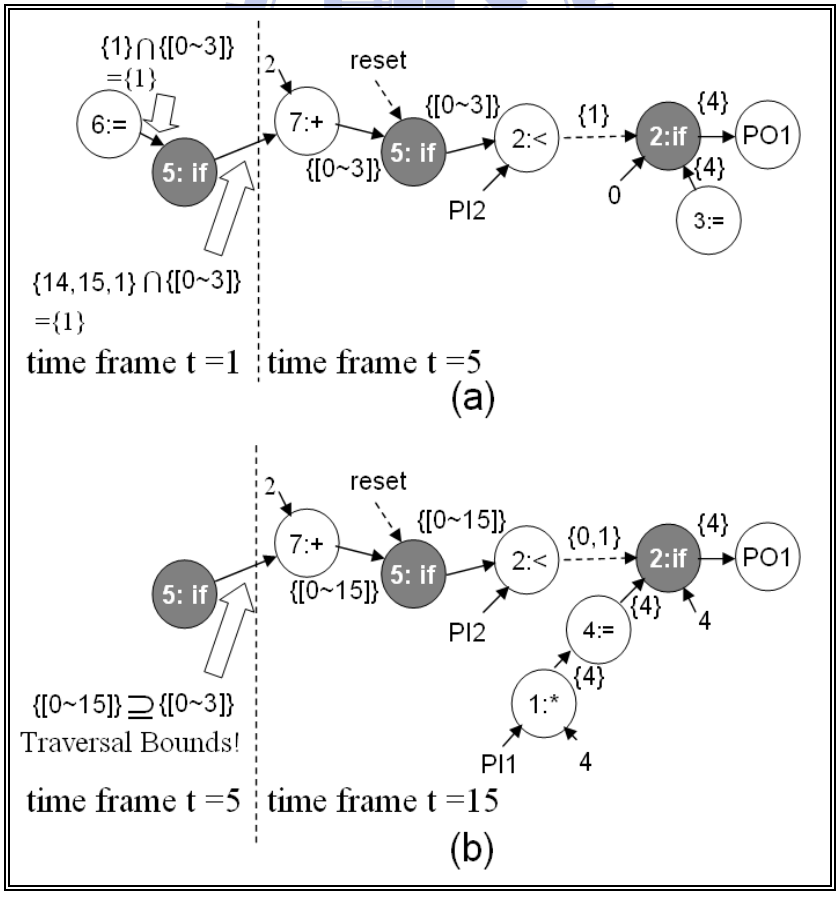


Figure 4-8: Computation starting from PO1 at t=5 and t=15

After all the POs at all the positive clock edges before the EOE are applied in MVS computation, PCS is calculated with formulas based on the computed MVSs. With PCS, a *debugging priority* with PCS (in round brackets) is obtained, as shown in Figure 4-9. 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 succeed immediately. In the experimental results in the next section, it is also proven that the proposed PCS can efficiently deliver *debugging priority* with high accuracy, which greatly reduces both the time and efforts required for design error searches in the input *error space*.

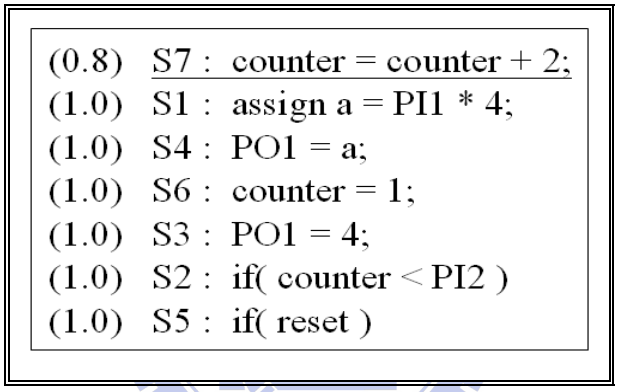


Figure 4-9: *Debugging priority* and the PCS

4.5 Experimental Results

The experiments are conducted on a subset of ITC'99 benchmark [1] and four other designs written in Verilog HDL. Number of lines (#line) of all the designs and the number of variables (#var) are presented in Table 4-1.

For every design case, one statement is randomly chosen and injected with an artificial design error based on typical bugs that designers usually induce [47]. With the injected error, a simulation is run until some incorrect values occur on POs. Then,

the *error space* identification approach proposed in [46][5] is applied to obtain an *error space*. After that, CS calculation in [46] and our proposed PCS calculation are both applied to derive two respective debugging priorities for the same *error space.*

With a *debugging priority*, error candidates that a digital circuit designer has to examine before he/she can find a design error are often less than blindly searching. In a sense, we can think that the size of the *error space* is thus reduced as a result of a *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* are also different. To compare the effectiveness of the two debugging priorities, a quantitative metric called Effective Size Ratio (ESR) is formulated as "the rank of the injected error/the number of error candidates in the *error space*". The two debugging priorities sorted with CS and with PCS have their own ESRs, respectively. Smaller ESR means the error has better rank with respect to the size of the *error space*. That also implies that the effective size reduction contributed by the corresponding *debugging priority* is larger and the effort required for design error searching in the *error space* is less.

[5] 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 4-1 A Comparison of Confidence Score (CS) and Probabilistic Confidence Score (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

With each design case, the above experimental processes are repeated for fifty times. In each repetition, the ESR of CS and PCS are calculated and recorded. The average ESR values with respect to CS and PCS are also presented in the columns "Avg_ESR_CS" and "Avg_ESR_PCS", respectively. The number of times, in which the ESR values of CS and PCS appear in three ESR value ranges, are also recorded to show the distribution of ESR values. The three ESR ranges are "ESR<0.2 (0~0.2)", "0.2<ESR<0.5 (0.2~0.5)", and "0.5<ESR (0.5~1.0)". The number of times is presented in the column "#case_CS" and "#case_PCS", respectively.

In Table 4-1, it can be observed that when PCS is used to obtain a *debugging*

priority, in all the design cases, the average ESR values are all less than 0.2 and are also less than the average ESR values of CS. For example, in design "B02", if CS is used to derive *debugging priority*, 38 times out of 50 times the ESR value is less than 0.2. In other words, our created errors are placed in the first twenty percent in the displayed list of error space for 38 times out of 50 experimental times. But, if PCS is applied instead, in each of the 50 repetitions, the injected error always appears in the first twenty percent. If a designer conduct error searching on design "B02" with the *debugging priority* sorted with PCS, he/she will locate the error by checking less than twenty percent of the derived error candidates. At least eighty percent of searching effort is saved. Moreover, it can be seen that ESR values of PCS is never greater than 0.5 in the fifty repetitions. This means that even in the worst case of the fifty repetitions, a *debugging priority* sorted with PCS can still save more than half the amount of efforts needed for design error searching in the error space. By contrast, the CS method was not found to 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 PCS is much greater than the one with respect to CS. The ratio Avg_ESR_PCS to Avg_ESR_CS shown in the column "ESR Ratio" is about 0.49 on average and 0.38 in the best case, meaning that with PCS, a 50% further size reduction, on average, is possible and a 62% size reduction is also achieved in the best case, as compared to CS. Therefore, on average, the proposed PCS method should save much more time/effort needed in the error searching process in the error space than CS. The cost of this improvement is little computation time, as compared to CS. The computation time needed for the two measurements, PCS and

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

4.6 Summary

This chapter presents a probabilistic measurement, PCS, to derive an accurate and reliable *debugging priority* for quick error searching among error candidates in an *error space*. Instead of assuming that the erroneous effects caused by some activated errors are seldom masked, the proposed PCS takes *error masking* into consideration and estimates the Likelihood Of Error-Propagating (LOEP) of an error candidate. The idea is that if the LOEP is high, *error masking* 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 *error masking* becomes quite possible. The 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 for 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, *debugging priority* sorted with the proposed PCS can effectively speed up error searching process in the input *error*

space. As compared to CS, the proposed PCS-based *debugging priority* can save more than half of the efforts (or time) needed for error searching process in an *error space* in our experiments, at the cost of little extra computation time. The time saving contributed by the proposed PCS method should usually much larger than the extra computation time the PCS calculation needs. Therefore, the gain of the proposed PCS can often outweigh the cost of extra computation time the PCS needs.

Chapter 5

Conclusions and Future Works

Simulation-based functional validation is still one of popular means to verify a digital hardware design described in a HDL. In simulation-based validation, the circuit behavior of an implementation described in a HDL can only be compared against the expected behavior or the specification on Observation Points (OPs). Even if some design errors are executed and activated, the erroneous effects caused by the design errors are still required to be propagated to the OPs for error detection.

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

The proposed probabilistic observability measures can be used to replace tags for the application of observability-based code coverage metric. In addition, since hard-to-observe points can be identified using our observability measure, designers can insert assertions in those locations to find hard-to-observe bugs more easily. This observability-driven assertion insertion is simple, but should be very effective.

The proposed observability measures for HDL descriptions can also be applied to assist debugging faulty HDL designs when a discrepancy between the simulation values of the OPs and the expected values occurs. The probabilistic observability measures can be used as a new probabilistic confidence score, which has better capability of estimating the likelihood of correctness for error candidates in error space. The experimental results shown in section 4.5 confirm that the proposed PCS measurement is indeed accurate in estimating the likelihood of correctness such that accurate debugging priority can be obtained. As a result, debugging priority sorted with the proposed PCS can effectively speed up error searching process in the input error space. As compared to CS in [46], the PCS-based debugging priority can save more than half of the efforts (or time) needed for error searching process in an error space in our experiments, at the cost of little extra computation time. The time saving contributed by the proposed PCS method should usually be much larger than the extra computation time the PCS calculation needs. Therefore, the gain of the proposed PCS can often outweigh the cost of extra computation time the PCS needs.

One possible future research direction is to generate a test vector set that creates some highly transparent sensitized paths to propagate potential incorrect values of the exercised statements to OPs for higher observability-based coverage. Other possible future improvements may include 1) more accurate observability estimation approaches for multiple paths, 2) a more accurate probabilistic observability measure by considering the probability distribution of each signal, and 3) integrating our dump-file based observability and PCS-based HDL debugging approach with commercial HDL simulator to form an efficient verification/debugging framework.

These future directions may provide a comprehensive solution for the observability issue during simulation-based functional validation.

Bibliography

[1] S.Tasiran, K. Keutzer, “Coverage Metrics for Functional Validation of Hardware Designs,” IEEE Design and Test of Computers, vol. 18, no. 4, pp. 36-45, July-August 2001.

[2] Collett International Research, Inc., http://www.collett.com.

[3] V. Bhagwati and S. Devadas, “Automatic Verification of Pipelined Micro -processors”, Proceedings of IEEE/ACM Design Automation Conference, pp.603-608, June 1994.

[4] R. B. Jones, D. L. Dill, and J. R. Burch, “Efficient Validity Checking for Processor Verification”, Proceedings of IEEE/ACM International Conference on Computer-Aided Design, pp.2-6, November 1995.

[5] R. E. Bryant, and Y.-A. Chen, “Verification of Arithmetic Functions with Binary Moment Diagrams”, Proceedings of IEEE/ACM Design Automation Conference, pp.535-541, June 1995.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, “Symbolic Model Checking: 10^{20} States and Beyond,” Information and Computation, pp. 428-439, August 1992.

[7] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, “Symbolic Model Checking without BDDs,” Proceedings of International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, pp.201-206, May 1999.

[8] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi,

"Benefits of Bounded Model Checking at an Industrial Setting," Proceedings of International Conference on Computer Aided Verification, vol. 2102, pp. 436–453, July 2001.

[9] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking Using SAT Procedures instead of BDDs," Proceedings of IEEE/ACM Design Automation Conference, pp. 317-320, November 1999.

[10] P. Bjesse, T. Leonard, and A. Mokkedem, "Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers," Proceedings of International Conference on Computer Aided Verification, vol. 2102, pp. 454-464, July 2001.

[11] H. Foster, A. Krolnik, and D. Lacey, Assertion-Based Design, Kluwer Academic Publishers, 2003.

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

[13] R.C. Ho and M.A. Horowitz, "Validation Coverage Analysis for Complex Digital Designs," Proceedings of IEEE/ACM International Conference Computer-Aided Design, pp. 322-325, November 1996.

[14] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets, "A Study in Coverage-Driven Test Generation," Proceedings of IEEE/ACM 36th Design Automation Conference, pp. 970-975, June 1999.

[15] J. Shen, and J.A. Abraham, "A RTL Abstraction Technique for Processor Microarchitecture Validation and Test Generation," Journal of Electronic Testing: Theory and Application, vol. 16, nos. 1-2, pp. 67-81, February 1999.

[16] S. Ur and Y. Yadin, "Micro Architecture Coverage Directed Generation of Test

Program,” Proceedings of IEEE/ACM Design Automation Conference, pp. 175-180, June 1999.

[17] D. Moundanos, J.A. Abraham, and Y.V.Hoskote, “Abstraction Technique for Validation Coverage Analysis and Test Generation,” IEEE Transactions of Computers, vol. 47, no.1, pp. 2-13, January 1998.

[18] S. Devadas, A. Ghosh, and K. Keutzer, "An Observability-based Code Coverage Metric for Functional Simulation," Proceedings of International Conference on Computer-Aided Design, pp. 418-425, November 1996.

[19] F. Fallah, S. Devadas, and K. Keutzer, "OCCOM: Efficient Computation of Observability-based Code Coverage Metrics for Functional Simulation," IEEE Transactions on Computer-Aided Design, vol 20, pp. 1003-1015, August 2001.

[20] F. Fallah, I. Ghosh, and M. Fujita, "Event-driven Observability Enhanced Coverage Analysis of C Programs for Functional Validation," Proceedings of IEEE Asian and South Pacific Design Automation Conference, pp.123-128, January 2003.

[21] F. Fallah, S. Devadas, and K. Keutzer, "Functional Vector Generation for HDL Models Using Linear Programming and Boolean Satisfiability," IEEE Transactions on Computer-Aided Design, vol. 20, pp. 994-1002, August 2001.

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

[23] L. H. Goldstein, "Controllability/Observability Analysis," IEEE Transactions on Circuits and Systems, vol. 26, pp685-693, September 1979.

[24] C. H. Chen and D. G. Saab, "A Novel Behavioral Testability Measure," IEEE Transactions on Computer-Aided Design, vol. 12, pp. 1960-1970, December 1993.

[25] S. Bhattacharya, S. Dey, and F. Brglez, "RT-level Transformations for Gate-level Testability," Proceedings of European Conference on Design Automation, pp. 162-169, February 1993.

[26] P. A. Thaker, V. D. Agrawal, M. E. Zaghloul, "Validation Vector Grade (VVG): A New Coverage Metric for Validation and Test," Proceedings of IEEE VLSI Test Symposium, pp. 182-188, April 1999.

[27] B. Murray and J. P. Hayes, "Hierarchical Test Generation Using Precomputed Test for Modules," IEEE Transactions on Computer-Aided Design, vol. 9, pp. 594-602, June 1990.

[28] S. Ravi, G. Lakshminarayana, and N. K. Jha, "TAO: Regular Expression-based Register-transfer Level Testability Analysis and Optimization," IEEE Transactions on VLSI Systems, vol. 9, pp. 824-832, December 2001.

[29] J. Voas, "PIE: A Dynamic Failure-based Technique," IEEE Transactions on Software Engineering, vol 18, pp. 717-727, August 1992.

[30] S.Y. Huang, K. T. Cheng, K. C. Chen, and D. I. Cheng, “Error Tracer: A fault Simulation-based Approach to Design Error Diagnosis”, Proceedings of IEEE International Test Conference, pp. 974-981, October 1997.

[31] D.W. Hoffmann, and T. Kropf, "Efficient Design Error Correction of Digital Circuits", Proceedings of IEEE International Conference on Computer Design, pp. 465-472, September 2000.

[32] M. Tomita, and H. H. Jiang, "An Algorithm for Locating Logic Design Errors", Proceedings of IEEE/ACM Design Automation Conference, pp. 468-471, June 1990.

[33] P.Y. Chung, Y.M. Wang, and I. N. Hajj, "Diagnosis and Correction of Logic

Design Errors in Digital Circuits", Proceedings of IEEE/ACM Design Automation Conference, pp503-508, June 1993.

[34] A. Smith, A. Veneris, M.F. Ali, and A. Viglas, “Fault Diagnosis and Logic Debugging Using Boolean Satisfiability,” IEEE Transactions on Computer-Aided Design, vol. 24, pp1606-1621, October 2005.

[35] J. R. Lyle, and M. Weiser, "Automatic Bug Location by Program Slicing", Proceedings of the Second International Conference on Computers and Applications, Beijing, China, pp. 877-883, June, 1987.

[36] M. Weiser, "Programmers Use Slices When Debugging", Communications of ACM, vol. 25, No. 7, pp.446-452, 1982.

[37] B. Peischl and F. Wotawa, “Automated Source-level Error Localization in Hardware Designs,” IEEE Design and Test Computer, vol. 23, no. 1, pp.8-19, January-February 2006.

[38] M. Khalil, Y. Le Traon, and C. Robach, “Towards an Automatic Diagnosis for High-level Validation”, Proceedings of IEEE International Test Conference, pp. 1010-1018, October 1998.

[39] C. H. Shi, and J. Y. Jou, "An Efficient Approach for Error Diagnosis in HDL Designs", Proceedings of IEEE International Symposium on Circuits and Systems, pp. IV-732- IV-735, May 2003.

[40] B.R. Huang, T.J. Tsai, and C.N. Liu, "On Debugging Assistance in Assertion-based Verification," Proceedings of the 12th Workshop on Synthesis and System Integration of Mixed Information Technologies, pp. 290-295, October 2004.

[41] Y.C. Hsu, B. Tabbara, Y.A. Chen, and F. Tsai, "Advanced Technique for RTL

Debugging," Proceedings of IEEE/ACM Design Automation Conference, pp.362-367, June 2003.

[42] J.P. Roth, “Diagnosis of Automata Failures: A Calculus and a Method,“ IBM, Journal on Research and Develop, vol. 10, pp. 278-291, June 1980.

[43] F. Fallah, “Coverage-Directed Validation of Hardware Models,” Dissertation of M.I.T, 1999.

[44] J. Voas and K. Miller, "Software Testability: The New Verification," IEEE Transactions on Software Engineering, vol. 12, pp. 17-28, May 1995.

[45] J. Voas, G. McGraw, L. Kassab, and L. Voas, "A “Crystal Ball” for Software Reliability," IEEE Computers, vol. 30, pp. 29-36, June 1997.

[46] T.Y. Jiang, C.N. Liu, and J. Y. Jou, “Effective Error Diagnosis for RTL Designs in HDLs”, Proceedings of IEEE 11th Asia Test Symposium, pp. 362-367, November 2002.

[47] A. Offutt and G. Rothermel, “An Experimental Evaluation of Selective Mutation,” Proceedings of International Conference on Software Engineering, pp. 100-107, May 1993.