A Probabilistic Analysis Method for Functional Qualification under Mutation Analysis

Hsiu-Yi Lin, Chun-Yao Wang, Shih-Chieh Chang, Yung-Chih Chen†,
Hsuan-Ming Chou, Ching-Yi Huang, Yen-Chi Yang, and Chun-Chien Shen
Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, R.O.C.
†Department of Electronic Engineering, Chung Yuan Christian University, Chung Li, Taiwan, R.O.C.

***Abstract*—Mutation Analysis (MA) is a fault-based simulation technique that is used to measure the quality of testbenches in error (mutant) detection. Although MA effectively reports the living mutants to designers, it suffers from the high simulation cost. This paper presents a probabilistic MA preprocessing technique, Error Propagation Analysis (EPA), to speed up the MA process. EPA can statically estimate the probability of the error propagation with respect to each mutant for guiding the observation-point insertion. The inserted observation-points will reveal a mutant's status earlier during the simulation such that some useless testcases can be discarded later. We use the mutant model from an industrial EDA tool, Certitude, to conduct our experiments on the OpenCores' RT-level designs. The experimental results show that the EPA approach can save about 14% CPU time while obtaining the same mutant status report as the traditional MA approach.**

I. Introduction

Functional verification is a process to ensure that the implementation is in accord with the design intent [24][26]. When designs are increasingly complex, functional verification are getting more difficult for designers. In general, simulation-based verification methods are the mainstream in the current design flow, especially for large designs. Due to the fact that exhaustive simulation is infeasible, *coverage metrics* are proposed to measure the quality of verification and thus reduce the simulation cost. However, the completeness problem is still an issue. This is because even all coverage metric reports show that there is no bug in the design under verification, it cannot ensure that additional verification tasks would never discover a new bug.

Coverage metrics can be generally classified into two categories, and they are *structural coverage (or code coverage)* [18][23] and *functional coverage* [9][12][23]. Most structural coverage metrics only evaluate the "reachability" of the stimuli to the design content, but do not consider stimuli's abilities to propagate bugs to observation locations. Therefore, the higher scores in structural coverage metrics do not indicate the better qualities of the test vectors. On the other hand, the functional coverage metric considers the semantic interpretation of the functionality to exercise the design. Since verification engineers need to manually set up various verification plans and a set of coverage points based on the specifications, it is a challenge to develop an automatic method to deal with it.

This work was supported in part by the National Science Council of Taiwan under grants NSC 99-2628-E-007-096, NSC 100-2628-E-007-008, and NSC 100-2628-E-007-031-MY3

978-3-9810801-8-6/DATE12/©2012 EDAA

Fault-based verification technique [5–7][10][11] is an effective approach that evaluates the quality of the verification environment. By injecting artificial faults into the original implementation, designers can check whether the verification environment can differentiate the faulty and original designs. If all the artificial faults can be detected, it implies that the verification environment is effective to reveal potential bugs; otherwise, the verification environment exists some weakness. Thus, fault-based verification *does* consider the error propagation issue that the structural coverage does not. The fault-based verification is similar to gate-level fault simulation where the stuck-at fault model is used for calibration of manufacturing test vectors.

Mutation Analysis (MA) is such a fault-based verification technique, originating from the software engineering field in the 1970s [8][13]. MA targets at only a subset of all potential faults and assumes that these faults are sufficient to represent all faults. This is because MA approach is based on two hypotheses: the *Competent Programmer Hypothesis* [8] and *Coupling Effect Hypothesis* [8][20][22]. Competent Programmer Hypothesis states that programmers develop their programs very close to the correct version. Coupling Effect Hypothesis assumes that complex faults can be coupled by simple faults in such a way that a test set detecting all simple faults in a program will detect a very high percentage of the complex faults [22]. As a result, MA approach only considers some simple syntactical changes instead of generating all potential faults. For example, Certitude^{TM} [3][14] is a commercial EDA tool implementing MA approach in RT-level designs for testbench qualification.

In the process of MA, each simple syntactical change is called a *mutant*. For example, given a program segment $a <= b \mid c$, after changing the operator, the new program segment becomes $a <= b \ \& \ c$. The replacement of "|" with "&" is a mutant for the original program. In the traditional process of MA, a set of mutants will be generated first, and then all test data are simulated. If we observe any difference caused by a mutant at the primary outputs, we call the mutant is *killed*, otherwise this mutant is a *living* mutant. The existence of living mutants indicates some weaknesses in the verification environment.

The MA makes the fault-based verification technique become a systematic verification methodology. However, it has suffered from some problems. The surveys of MA in literature [16][21] addressed that the MA induces extremely high computational cost in the state-of-the-art. For each injected

mutant, we need to perform the whole design simulation for all the testcases once. Therefore, the computational cost will rapidly increase due to a great amount of injected mutants and testcases.

In this work, we propose a preprocessing technique for accelerating MA process named *Error Propagation Analysis (EPA)*, which analyzes the error propagation ability of each mutant before performing the simulation in RT-level designs. Then, we determine a few *observation-points* [17] to observe the effect of error propagation. The analyzed results can be embedded into the MA cost reduction technique, hence, we can reduce the simulation cost of MA.

II. Cost Reduction Techniques in MA

In the traditional MA process, a set of mutated programs are generated first, then all the testcases are applied to each mutated program to determine if each mutant is killed or not. The simulation cost of this MA approach is much more than that in the original design simulation.

In the latest survey of MA [16], cost reduction techniques are divided into two types: one is *mutant reduction*, and the other is *simulation reduction*. The mutant reduction is to reduce the number of generated mutant without having significant loss of the testbench quality. The simulation reduction is to minimize the simulation cost of a mutated program. In this work, we focus on the issue of simulation reduction. Thus, in the next paragraph, we will introduce the background of a widely used simulation reduction technique, "*Strong, Weak, and Firm Mutation* (*SWFM*)", that is the basis of this work.

SWFM proposed three different degrees of mutation with respect to the observation-points, and they are Strong Mutation, Weak Mutation, and Firm Mutation. In the traditional MA process, when a mutant is "killed" after the simulation, it means that the error effect caused by this mutant is propagated to the primary outputs. This type of mutation is called *Strong Mutation* [8]. Strong Mutation can reveal the potential weaknesses thoroughly, but it suffers from the high simulation cost due to the whole design simulation. To reduce the simulation cost, *Weak Mutation* [15] was proposed. Weak Mutation assumes that each statement in a design is an observation-point. Thus, if a mutant causes any statement's output changed, the mutant is said killed. As a result, Weak Mutation reduces the simulation cost but sacrificing the quality of testbench. When the observation-points lie between the mutated statements (Weak Mutation) and the primary outputs (Strong Mutation), this mutation is called *Firm Mutation* [25], which is a tradeoff between Strong and Weak Mutations. A Verilog example about these three types of mutations is shown in Fig. 1 and Table I.

In Fig. 1, the original operation $b\&c$ is replaced with $b|c$ in line 8, and it is an injected mutant. We apply an input vector $\{a, b, c\} = \{0, 0011, 0001\}$ to both the mutated and the original programs. In this example, we need to simulate this design for 5 clock cycles to get the value of out. The results are shown in Table I. Since these three types of mutations have different observation-points, they may result in different mutant statuses. The Strong Mutation performed the complete simulation, and determined the mutant status after simulating 5 cycles. This mutant status is *living* because the primary

```
1  input clk;
2  input a;
3  input [3:0] b,c;
4  output [3:0] out;
5  reg [3:0] x, y, out;
6  // (x, y, out are initialized to 0)
7  always @ (posedge clk) begin
8     if (a==0)   x = b & c; // mutant: b | c
9     else        x = b ^ c;
10    y = x >> 2;
11    out = y + 1;
12 end
```

Figure 1. An example for demonstrating Strong, Weak, and Firm Mutations.

Table I
THE SIMULATION RESULT WITH THE MUTATED PROGRAM IN FIG. 1

test input {a,b,c} = {0,0011,0001} and the mutant b|c

Mutation type	Observation-Point	Expected Result	Mutated Result	Mutant Status	Determined at # cycle
Strong	out	*out=5*	*out=5*	*Living*	5
Weak	x	*x=0001*	*x=0011*	*Killed*	1
Firm	y	*y=0000*	*y=0000*	*Living*	1

output out of mutated program is the same as that of the original program. In the Weak Mutation, the observation-point is the site of the mutant. Thus, this mutant status is *killed* after one cycle simulation because of the different results between the statements of $x = b\&c$ and $x = b|c$. For the Firm Mutation, however, the location of observation-point has many choices. Assume we set y as the observation-point in this example, then the mutant is still *living* after one cycle simulation by comparing the value y in the mutated and the original programs. From this example, we found that the advantage of Firm Mutation is that it can determine the living mutant after fewer simulation cycles. This is because a living mutant under the Firm Mutation is still a living mutant under the Strong Mutation.

Due to the incomplete simulation, killed mutants under the Weak Mutation are very likely to be living mutants under the Strong Mutation. To avoid sacrificing the quality of results, we need to check whether all killed mutants under the Weak Mutation also can be killed under the Strong Mutation. Although the Weak Mutation Analysis cannot decide the mutants' actual statuses, it can be used to eliminate some useless testcases for the further Strong Mutation. For example, when applying 100 testcases to test a mutant A, we need to perform the whole design simulation 100 times under the Strong Mutation. Suppose we applied the Weak Mutation first, and found that 80 out of 100 testcases cannot kill the mutant A. Then these 80 testcases cannot kill the mutant A under the Strong Mutation, either. Therefore, only the remaining 20 testcases need to be applied for the mutant A under the Strong Mutation. Thus, the total simulation cost can be reduced by first applying the Weak Mutation.

Although the Weak Mutation effectively reduces the simulation cost, the remaining simulation cost for the Strong Mutation is still large. A killed mutant under the Weak Mutation but living under the Strong Mutation means that this mutant causes the error in the mutated statement but cannot be propagated to the primary outputs. We called such a mutant *Non-Propagation Mutant (NPM)*. The Firm Mutation is proposed to mainly target at this propagation issue. It can also be designed to insert the internal observation-points to find the potential NPMs earlier. However, the determination of the locations and the number of the observation-points is a

critical issue. This is because too many observation-points will increase the simulation overhead, and improper observation-points will decrease the probability of finding potential NPMs. To solve this problem, we propose a probabilistic analysis to automatically determine the observation-points based on the Firm Mutation mechanism, which will be detailed in Section III.

III. Proposed Approach

A. Overview

The EPA intends to statically analyze the structure of HDL design and evaluate the error propagation ability for a given mutant. We propose *Mutant Controllability (MC)* to represent the changing probability of each signal. We also propose the MC estimation formulae for various operations that are used to determine the MCs of the signals in the propagation path with respect to each mutant. Using the estimated MC, we derive the Decreasing Rate (DR) of MC, and select the locations with the largest DR for inserting observation-points.

In the Firm Mutation approach, the inserted observation-points will affect a mutant's status during the simulation. As a result, the testcases that cannot kill the mutant are excluded from the original testcase set for speeding up the succeeding Strong Mutation simulations. Finally, we collect the living mutants under the Firm Mutation, and analyze their related MCs to point out the possible causes for their liveness. Note that our approach in this work targets at the word-level operations in Verilog designs, but it also can be used in other design languages in a similar way.

The functional qualification flow equipped with the proposed EPA is shown in Fig. 2, where the EPA (the gray area) includes MC estimation, observation-point selection, and Firm Mutation Analysis. After performing the EPA, the effort of performing the Strong Mutation Analysis can be reduced to that only dealing with killed mutants and the corresponding testcases under the Firm Mutation.

B. Terminology

1) Mutant Controllability (MC): When MA is performed, the signals in the propagation path from the site of mutant to the primary outputs will be possibly changed. If one of the primary output changes, the mutant is killed as desired. However, if the mutant effect is blocked such that all primary outputs are intact, the mutant is living. Here, we define the *Mutant Controllability (MC)* of each signal as its changing probability in its propagation path with respect to a given mutant. Specifically, $MC(m, s)$ represents the changing probability of a signal s with respect to a mutant m. $MC(m, s) = 1$ if the signal s is on the mutated statement. $MC(m, s) = 0$ if the mutant m does not change the value of the signal s. The signal with a higher MC means that the mutant is able to change the signal with a higher probability. For the example in Fig. 3, the mutant m is on $tmp1$, hence, we set $MC(m, tmp1) = 1$. Since the error effect will be propagated and the signals x, y are on the propagation path from m, we can derive the values of $MC(m, x)$ and $MC(m, y)$ after the MC calculation. On the other hand, $tmp2$ is not on the propagation path, hence, we can directly set $MC(m, tmp2) = 0$.

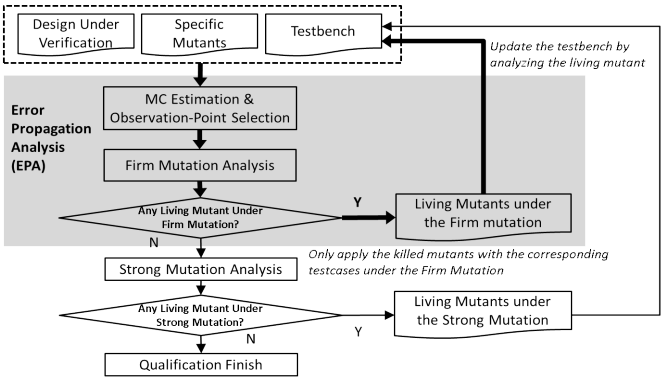

Figure 2. Functional qualification flow with the Error Propagation Analysis.

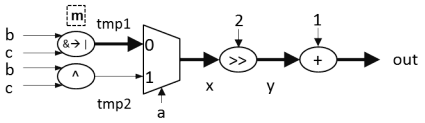

Figure 3. A propagation path from the mutant m.

2) Control Data Flow Graph (CDFG): In this work, the input is a Verilog design represented as a CDFG [4][19]. A CDFG representation is composed of four types of nodes: *operational nodes*, *control nodes*, *call nodes*, and *storage nodes*. Operational nodes are responsible for arithmetic, logical, or relational operations. Control nodes are responsible for branch conditions such as *if-else* or *case* statements. Call nodes and storage nodes represent calls to subprogram and assignment operations associated with variables and signals, respectively. Since most mutant effects are propagated through the operational nodes and control nodes, our MC estimation formulae only focus on these two types of nodes. Although storage nodes do not affect the propagation of mutant effects, they are the MC recomputation points for the sequential part of designs. Fig. 4 shows a mapping example from a Verilog design to its CDFG.

C. Mutant Controllability Estimation Formulae

MC estimation formulae are used to estimate how difficult a mutant effect can be propagated out. Operational nodes can be divided into unary and binary operational nodes and are discussed as follows:

1) Unary operational nodes: Consider a unary operation "$out = op\ a$", where a is an operand, op is a unary operator as shown in Fig. 5(a). The MC estimation formula for a unary operation is shown as EQ(1).

$$MC(m, out) = MC(m, a) \times [1 - P_{mask}(v'_a, op)] \quad (1)$$

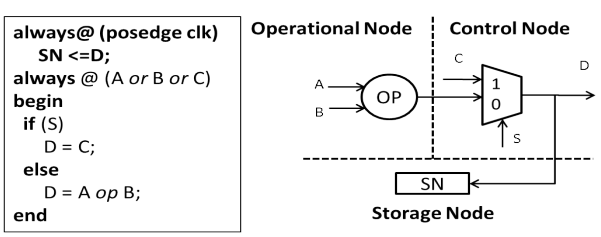

Figure 4. An example about the mapping from a Verilog design to its CDFG.

In EQ(1), we get the MC of out from the MC of a. Here, suppose that for the operand a, v_a is its correct value and v'_a is the new value caused by the injected mutant. $P_{mask}(v'_a, op)$ represents the probability of the "same" result in the operations of op v_a and op v'_a. This probability is called *masking probability*. Assume v_{out} and v'_{out} are the results of the unary operation with respect to the input v_a and v'_a, then $P_{mask}(v'_a, op) = P(v_{out} = v'_{out})$. To derive this masking probability, we need to consider the operator op and the width w of the operand a.

To simplify the computation, we assume that v_a and v'_a are evenly distributed from 0 to $2^w - 1$, and $v_a \neq v'_a$ (the mutant effect has been activated). For example, suppose the op is a unary AND "&", which will output 1 iff all bits of the operand are 1; 0 otherwise. The masking probability is the probability that v_{out} and v'_{out} are the same. Thus, $P_{mask}(v'_a, \&)$ is derived as $P_1((v_{out} = 1) \wedge (v'_{out} = 1)) + P_2((v_{out} = 0) \wedge (v'_{out} = 0))$. Because only one vector $\{111...1\}$ results in the output 1, it is not possible that $\&v'_a = v'_{out} = \&v_a = v_{out} = 1$. As a result, P_1 is 0. On the other hand, the probability of $\&v_a = v_{out} = 0$ is $1 - \frac{1}{2^w}$, and since $v'_a \neq v_a$, the probability of $\&v'_a = v'_{out} = 0$ is $1 - \frac{2}{2^w}$. Also, since these two events are independent, P_2 is the product of them. Hence, we can get $P_{mask}(v'_a, \&) = P_1 + P_2 = 0 + (1 - \frac{1}{2^w})(1 - \frac{2}{2^w})$. Finally, we get $MC(m, out) = MC(m, a) \times [1 - (1 - \frac{1}{2^w})(1 - \frac{2}{2^w})]$. This result shows that the $MC(m, out)$ will be smaller when w is larger, which implies that the error effect will be difficultly propagated. The masking probabilities of the common unary operations are summarized in the bottom of Table II.

2) Binary operational nodes: Consider a binary operation "$out = a$ op b", where a and b are the operands. The estimation formula is shown as EQ(2).

$$\begin{aligned} MC(m, out) = & MC(m, a) \times \overline{MC}(m, b) \times [1 - P_{mask}(v'_a, v_b, op)] + \\ & \overline{MC}(m, a) \times MC(m, b) \times [1 - P_{mask}(v_a, v'_b, op)] + \\ & MC(m, a) \times MC(m, b) \times [1 - P_{mask}(v'_a, v'_b, op)] \end{aligned} \tag{2}$$

In EQ(2), $\overline{MC}(m, a)$ and $\overline{MC}(m, b)$ represent the probability of the correct values on the input signals and can be derived from $1 - MC(m, a)$ and $1 - MC(m, b)$, respectively. In a binary operation as Fig. 5(b) shows, there are three conditions that the error effects will be propagated out with respect to the incorrect values on a, b, or both. Because the masking probabilities in these three conditions are different, we separately consider these conditions and multiply them with their corresponding MC or $\overline{MC}$ values. The first term in EQ(2) shows that v'_a is an incorrect value and v_b is a correct value, hence, the equation is involved with the factors $MC(m, a)$ and $\overline{MC}(m, b)$. Similarly, the second term is involved with $\overline{MC}(m, a)$ and $MC(m, b)$ with respect to v_a and v'_b. The last term represents the condition of the simultaneous incorrect values v'_a and v'_b.

For example, suppose the operator op is "+" and the width of the operands is w_+. To derive the masking probability for each term in EQ(2), we assume that the error effect will cause a change with Δ_a and Δ_b such that $v'_a = v_a + \Delta_a$ and $v'_b = v_b + \Delta_b$, where $\Delta_a \neq 0$ and $\Delta_b \neq 0$. Thus, both $P_{mask}(v'_a, v_b, +)$ and $P_{mask}(v_a, v'_b, +)$ will be 0 because of $v_a + v_b \neq v'_a + v_b = (v_a + \Delta_a) + v_b$ and $v_a + v_b \neq v_a + v'_b = v_a + (v_b + \Delta_b)$.

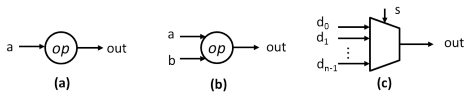

Figure 5. (a) A unary operational node. (b) A binary operational node. (c) A control node.

Table II
MASKING PROBABILITIES FOR COMMON OPERATIONS IN VERILOG

Types	Operation	$P_{mask}(v'_a, v_b, op)$	$P_{mask}(v_a, v'_b, op)$	$P_{mask}(v'_a, v'_b, op)$
Arithmetic	$a + b$	0	0	$\frac{1}{2^{w_a}}$
	$a * b$	$\frac{1}{2^w}$	$\frac{1}{2^w}$	0
	$a \% b$	$\frac{b}{2^w - 1}$	b is constant	
Relational	$a > b, a < b$	$\frac{1}{2}$		
	$a > b$	$\frac{2^w - b}{2^w}$	b is constant	
	$a < b$	$\frac{b}{2^w}$	b is constant	
Equality	$a == b, a != b$	$1 - \sum_{i=1}^{2^{w+1}-2} \frac{2i}{(2^w+1)^2}$		
Bit-wise	$a \& b, a \mid b, a \wedge b$	$\sum_{i=1}^{w} \binom{w}{i} \frac{1}{2^i}$		
Shift	$a >> b$	$\frac{2^b - 1}{2^w - 1}$	b is constant	
Unary	~a	1	unary operation	
	$\wedge a$	$\frac{1}{2}$	unary operation	
	$\&a, \mid a$	$(1 - \frac{1}{2^w})(1 - \frac{2}{2^w})$	unary operation	

For the last term, $v_a + v_b = v'_a + v'_b = v_a + v_b + \Delta_a + \Delta_b$ is true iff $\Delta_a = -\Delta_b$ is true. Therefore, the masking probability $P_{mask}(v'_a, v'_b, +)$ in the last term is the same as the probability of $\Delta_a = -\Delta_b$, which is $\frac{1}{2^{w_+}}$. Finally, we get $MC(m, out) = MC(m, a) \times MC(m, b) \times (1 - \frac{1}{2^{w_+}})$. The masking probabilities of the common binary operations are also summarized in Table II.

3) Control nodes: Consider a control node which contains n inputs $\{d_0, d_1..., d_{n-1}\}$, a data output out, and a selector s with the range from 0 to $n - 1$. There are two conditions that the error effect will be propagated to the data output out: one is the error effect is on the selector s and the incorrectly selected input has a different value with that of the expected input; the other is the error effect is on d_i and $s = i$. To simplify our computation, we assume s is an integer evenly distributed from 0 to $n-1$, which means that each input signal will be selected with the same probability. Additionally, the width of each d_i is w and its value is also evenly distributed from 0 to $2^w - 1$. The estimation formula for the control nodes is shown as EQ(3).

$$\begin{aligned} MC(m, out) = & (MC(m, s) \times \frac{2^w - 1}{2^w}) \ + \\ & (\overline{MC}(m, s) \times \frac{\sum_{i=0}^{n-1} MC(m, d_i)}{n}) \end{aligned} \tag{3}$$

We divide EQ(3) into two terms with respect to whether the selector is erroneous or not. The first term is for the case that the error effect is on the selector s. Suppose i, j are the correct and incorrect values of the selector s, we only need to consider whether the expected input d_i and the incorrectly selected input d_j have the same value. Since d_i, d_j are evenly distributed from 0 to $2^w - 1$, the probability of $d_i \neq d_j$ is $\frac{2^w - 1}{2^w}$. Therefore, the first term is $MC(m, s) \times \frac{2^w - 1}{2^w}$ where $MC(m, s)$ is the probability of erroneous selector. The second term is for the case that the value of the selector s is correct. The error effect will be propagated when the selected input has an incorrect value. Because we assume each input will be selected with the same probability, we use the averaged MC value $\frac{\sum_{i=0}^{n-1} MC(m,\ d_i)}{n}$ to represent the probability of the

error effect on the selected input. Thus, the second term is the product of $\overline{MC}(m,s)$ and the averaged MC value among all inputs.

D. MC Estimation Considering the Sequential Circuits

When considering sequential circuits, we can see that there exists feedback loops in a CDFG. Because a mutant is likely to affect some signals after a few cycles, we need to recompute the MC values of signals in the feedback loop for several times. Consider the feedback case as shown in Fig. 6, after deriving the $MC(m,E)$ with a given mutant m, the value of $MC(m,A)$ will be updated in the next clock cycle. Because assignment operations do not affect the propagation of error effects, we get updated $MC(m,A) = M(m,E)$. Therefore, we need to recompute $MC(m,D)$ for the operation $D = A\ op\ B$. To avoid too much computation cost for the feedback, we terminate the recomputation when the difference between the current and the next MC values is under a given threshold.

E. The Observation-Point Selection

For any node in the CDFG, if the error effect can be observed on its inputs but not on its output, this node is an *error-masking node*. If there exists one error-masking node on each propagation path of a mutant, this mutant is an NPM. The observation-point is used to observe the mutants' statuses under the Firm Mutation. Since we desire to reduce the unnecessary simulation effort, a proper observation-point should be lied on the output of an error-masking node.

To determine these proper observation-points, we must find the error-masking nodes first. Because of the enormous number of nodes in the CDFG, finding all the error-masking nodes for an NPM is time-consuming. Thus, we use the results of the MC estimation to find the potential error-masking nodes.

Next, we detail the observation-point selection method. In the first step, we extract all the propagation paths of a given mutant without considering feedback signals. For a given mutant m, assume that there are n propagation paths from m to the primary outputs. $PP_{m,i}$ denotes the i^{th} propagation path of the mutant m where $1 \le i \le n$. $PP_{m,i}$ consists of some ordered signals on this propagation path. $|PP_{m,i}|$ denotes the number of signals on this path. For example, considering a design segment in Fig. 7, if there exists an error effect on A from a mutant m, we can identify two propagation paths $PP_{m,1} = \{A, tmp, X, Y, O_1\}, PP_{m,2} = \{A, tmp, X, Y, O_2\}$, and $|PP_{m,1}| = |PP_{m,2}| = 5$.

We propose the Decreasing Rate (DR) as shown in EQ(4) to represent the degree of the error-masking effect between adjacent signals on a given propagation path $PP_{m,i}$.

$$DR_{s_k}(PP_{m,i}) = \frac{MC(m,s_{k-1}) - MC(m,s_k)}{MC(m,s_{k-1})}, 2 \le k \le |PP_{m,i}| \tag{4}$$

In EQ(4), $DR_{s_k}(PP_{m,i})$ represents the DR of the k^{th} signal, s_k, on the $PP_{m,i}$. For example, given a $PP_{m,i}$ with three signals $\{a,b,c\}$, $DR_{s_3}(PP_{m,i})$ is the DR of the third signal c on the $PP_{m,i}$ and it can be derived by $\frac{MC(m,b)-MC(m,c)}{MC(m,b)}$. The larger $DR_{s_k}(PP_{m,i})$ implies the node between the signals s_{k-1} and s_k is a potential error-masking node. Therefore, we select the signals with the largest DR in each propagation path

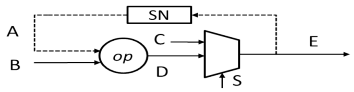

Figure 6. The feedback case.

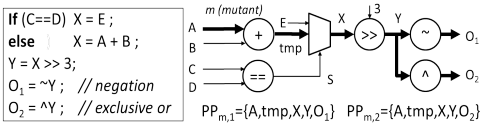

Figure 7. An example of propagation paths.

to be the observation-point candidates. Finally, we choose the most frequently selected signals among all candidates to be the final observation-points.

IV. Experimental Results

We implemented the EPA analyzer in C++ and the testbench in Verilog, respectively. The experiments were conducted on an Intel Core™2 Quad 2.5 GHz Linux platform (Ubuntu 10.10).

To integrate the MA and our EPA approach into the verification environment, we establish a framework to automate this technique as shown in Fig 8. The EPA analyzer analyzes the mutated design and determines its observation-points. It also automatically generates the monitors to observe the simulation data on the observation-points. Here, we use a random generator to generate the testcases for the designs. The comparator compares the simulation data between the original and the mutated designs, and reports the results. Because the generated monitors can be easily configured to observe different signals for performing the Strong, Weak, or Firm Mutation, this framework makes the testbench reusable for different mutated designs. The threshold value for terminating the recomputation in feedback loops is set 0.1.

The commercial EDA tool, Certitude [14], has proposed various types of mutants. In our experiments, we adopt three types of mutants, and they are *changing operator*, *dead assignment*, and *condition stuck-at-true (false)*. Our simulation engine is Icarus Verilog [1], an open-source Verilog simulator. The benchmark comes from the OpenCore website [2]. We only conduct the experiments for three designs in our preliminary implementation. The results of EPA are shown in Table III. Columns 2 and 3 show the number of lines in the benchmarks and the number of injected mutants, respectively. Column 4 is the spent CPU time of the EPA approach.

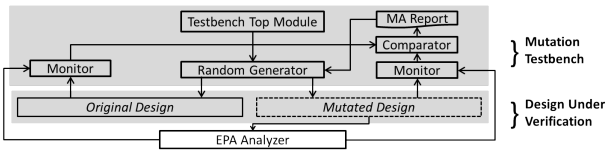

Figure 8. The framework of our implementation.

Table III
THE EPA RESULTS.

Design	RTL #lines	$\|IM\|$	$T_{epa}(s)$
double_fpu	1910	100	11
ecpu_alu	1232	150	10
reed_solomon_decoder	3802	300	31

Table IV
THE WEAK AND FIRM MUTATION ANALYSIS RESULTS.

Design	$\|TC\|$	$\|IM\|$	$\|TSR\|$	Preprocessing Method					Strong Mutation Analysis		
				Pre. Types	$\|RSR\|$	$RSR\%$	M_p	T_p	M_s	$T_s(+T_p)$	$T\%$
double_fpu	1000	100	1.0×10^5	Weak	3.72×10^4	37.20	23	132	43	1412	1
				Firm	4.07×10^4	**40.70**	**29**	296	43	**1136**	0.80
ecpu_alu	1000	150	1.5×10^5	Weak	4.37×10^4	29.13	37	73	61	412	1
				Firm	4.92×10^4	**32.80**	**39**	81	61	**391**	0.94
reed_solomon_decoder	100	300	3.0×10^4	Weak	1.36×10^4	45.33	103	181	171	2701	1
				Firm	1.82×10^4	**60.67**	**112**	258	171	**2273**	0.84
Averaged CPU time reduction of the Strong Mutation Analysis with the Firm Mutation preprocessing											0.86

To compare the performance of our preprocessing method, we apply the Weak and the Firm Mutation to the benchmark, respectively, followed by the Strong Mutation. The Firm Mutation is embedded in the proposed EPA approach. The simulation results are shown in Table IV. Columns 2 and 3 list the number of testcases (|TC|) and the number of injected mutants (|IM|), respectively. Column 4 is the number of the total simulation rounds (|TSR|). Here, we use a Simulation Round (SR) to represent a unit cost in the mutation simulation, which is the time cost of applying a testcase to test a mutant. Thus, the total number of SRs in performing the MA without any preprocessing method is the product of the number of testcases and the number of injected mutants. Columns 5 to 9 show the results of the two MA preprocessing methods, Weak and Firm Mutation. Because both preprocessing methods can find the living mutants and the corresponding useless testcases, the numbers of SRs and the number of injected mutants applying for the further Strong Mutation are reduced. Columns 6 and 7 show the number of the reduced SRs (|RSR|) and the percentage of that to the |TSR| (RSR%). Column 8 is the number of living mutants reported by the preprocessing methods. Column 9 is the required CPU time measured in second. Note that the Firm Mutation is embedded into the EPA approach, therefore, the time for the EPA as shown in Table III has been accounted in Column 9. Columns 10 to 12 show the results of overall flow including the preprocessing method. Column 10 is the number of living mutants (M_s). Column 11 is the total CPU time including the preprocessing time. Column 12 shows the ratio of total CPU time of the Strong Mutation embedding the Firm Mutation against the Weak Mutation for the preprocessing ($T\%$).

According to Table IV, we found that the Weak Mutation cost less CPU time as compared to the Firm Mutation for the preprocessing. It is because for the Weak Mutation, it can determine a mutant is killed when the mutated statement was activated, without considering the propagation issue. On the other hand, the Firm Mutation embedded in our EPA method identified more living mutants and the corresponding useless testcases for the succeeding Strong Mutation. Thus, although the Firm Mutation cost more CPU time, its larger number of RSRs makes computation cost in the further Strong Mutation less as shown in Column 11. The averaged CPU time reduction in the last row shows that the Firm Mutation embedded in our EPA method can save 14% CPU time on average.

V. CONCLUSION

In this paper, we propose an Error Propagation Analysis approach that deals with the error propagation issues in Mutation Analysis. This approach can probabilistically analyze the HDL designs and add the proper observation-points. Based on the Firm Mutation approach, we can determine the mutant status by monitoring these selected observation-points. Therefore, the Firm Mutation can discover more living mutants and discard the useless testcases of each mutant to reduce the simulation cost of the further Strong Mutation. The experimental results show that our approach is more efficient than the Weak Mutation, which does not consider the error propagation issue.

REFERENCES

[1] *Icarus Verilog*. [Online]. Available: http://iverilog.icarus.com/
[2] *OpenCores*. [Online]. Available: http://opencores.org/
[3] *SpringSoft*. [Online]. Available: http://www.springsoft.com/
[4] S. Anellal and B. Kaminska, "Scheduling of a control and data flow graph," in *Proc. IEEE Int. Symp. on Circuits and Systems*, May 1993, pp. 1666 – 1669.
[5] J. Arlat, A. Costes, Y. Crouzet, J. Laprie, and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," *IEEE Trans. Computers*, vol. 42, no. 8, pp. 913 – 923, Aug. 1993.
[6] A. Benso, A. Bosio, S. Di Carlo, and R. Mariani, "A functional verification based fault injection environment," in *Proc. Defect and Fault-Tolerance in VLSI Systems*, 2007, pp. 114 – 122.
[7] A. Benso, M. Rebaudengo, M. Reorda, and P. Civera, "An integrated HW and SW fault injection environment for real-time systems," in *Proc. Defect and Fault Tolerance in VLSI Systems*, 1998, pp. 117 – 122.
[8] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Trans. Computers*, vol. 11, no. 4, pp. 34 – 41, Apr. 1978.
[9] L. Drucker, "Functional coverage metrics–the next frontier," in *EETimes*, Aug. 2002.
[10] F. Ferrandi, F. Fummi, and D. Sciuto, "Implicit test generation for behavioral VHDL models," in *Proc. Int. Test Conference*, 1998, pp. 587 – 596.
[11] A. Fin and F. Fummi, "A VHDL error simulator for functional test generation," in *Proc. Design, Automation and Test in Europe*, 2000, pp. 390 – 395.
[12] A. Gluska, "Coverage-oriented verification of banias," in *Proc. Design Automation Conference*, 2003, pp. 280 – 285.
[13] R. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Software Engineering*, vol. SE-3, no. 4, pp. 279 – 290, July 1977.
[14] M. Hampton and S. Petithomme, "Leveraging a commercial mutation analysis tool for research," in *Proc. Testing: Academic and Industrial Conference Practice and Research Techniques - Mutation*, 2007, pp. 203 – 209.
[15] W. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. Software Engineering*, vol. SE-8, no. 4, pp. 371 – 379, July 1982.
[16] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 1 – 32, 2010.
[17] T. Lv, H. wei Li, and X. wei Li, "Automatic selection of internal observation signals for design verification," in *Proc. VLSI Test Symposium*, 2009, pp. 203 –208.
[18] J. C. Miller and C. J. Maloney, "Systematic mistake analysis of digital computer programs," *Commun. ACM*, vol. 6, pp. 58 – 63, Feb. 1963.
[19] R. Namballa, N. Ranganathan, and A. Ejnioui, "Control and data flow graph extraction for high-level synthesis," in *Proc. IEEE Annual Symposium on VLSI*, Feb. 2004, pp. 187 – 192.
[20] A. Offutt, "The coupling effect: fact or fiction," in *Proc. Symp. Software Testing, Analysis, and Verification*, 1989, pp. 131 – 140.
[21] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," in *Proc. Mutation*, 2000, pp. 45 – 55.
[22] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Trans. Software Engineering and Methodology*, vol. 1, pp. 5 – 20, Jan. 1992.
[23] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design and Test of Computers*, vol. 18, no. 4, pp. 36 – 45, Jul./Aug. 2001.
[24] C.-Y. Wang, S.-W. Tung, and J.-Y. Jou, "On automatic verification pattern generation for soc with port order fault model," *IEEE Trans. Computer-Aided Design*, vol. 21, no. 4, pp. 466–479, April 2002.
[25] M. Woodward and K. Halewood, "From weak to strong, dead or alive? an analysis of some mutation testing issues," in *Proc. Workshop Software Testing, Verification, and Analysis*, 1988, pp. 152 –158.
[26] S.-C. Wu, C.-Y. Wang, and Y.-C. Chen, "Novel probabilistic combinational equivalence checking," *IEEE Tran. Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 4, pp. 365–375, April 2008.