Hindawi Publishing Corporation
VLSI Design
Volume 2015, Article ID 256474, 9 pages
http://dx.doi.org/10.1155/2015/256474

Research Article

Functional Testbench Qualification by Mutation Analysis

Kai Huang,[1] Peng Zhu,[2] Rongjie Yan,[3] and Xiaolang Yan[2]

[1]*Department of Information Science and Electronic Engineering, Zhejiang University, Hangzhou 310027, China*
[2]*Institute of Very Large Scale Integrated Circuit Design, Zhejiang University, Hangzhou 310027, China*
[3]*Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100080, China*

Correspondence should be addressed to Rongjie Yan; yrj@ios.ac.cn

Received 2 February 2015; Revised 16 April 2015; Accepted 26 April 2015

Academic Editor: Avi Ziv

The growing complexity and higher time-to-market pressure make the functional verification of modern large scale hardware systems more challenging. These challenges bring the requirement of a high quality testbench that is capable of thoroughly verifying the design. To reveal a bug, the testbench needs to activate it by stimulus, propagate the erroneous behaviors to some checked points, and detect it at these checked points by checkers. However, current dominant verification approaches focus only on the activation aspect using a coverage model which is not qualified and ignore the propagation and detection aspects. Using a new metric, this paper qualifies the testbench by mutation analysis technique with the consideration of the quality of the stimulus, the coverage model, and the checkers. Then the testbench is iteratively refined according to the qualification feedback. We have conducted experiments on two designs of different scales to demonstrate the effectiveness of the proposed method in improving the quality of the testbench.

1. Introduction

Functional verification of modern hardware systems always requires the largest amount of resources and human efforts during the design cycle [1]. Simulation based verification is the predominant approach in hardware verification [1], which uses a testbench to verify the design under verification (DUV). There are three main components in a testbench as shown in Figure 1: (1) stimulus to activate the DUV; (2) a set of coverpoints to observe the behavior of the DUV and collect coverage information; and (3) a set of checkers to check internal states and outputs of the DUV. To reveal a bug, the corresponding circuits must be activated first, and then the erroneous results must be propagated to some ports which are properly checked by checkers. To thoroughly verify a hardware system, we need sufficient stimulus to activate all corners of the DUV, a sufficient coverage model to ensure that all important functions of the DUV are executed, and a sufficient set of checkers to guarantee that all results of the executed functions are adequately checked. All the three components should be considered to build a high quality testbench [2–4].

A coverage model, built according to coverage metrics such as code coverage and user defined functional coverage [5, 6], is the main way to evaluate the thoroughness of verification. Unfortunately, these coverage metrics focus only on the quantity of the activated coverpoints and ignore the propagation and the sufficiency of the checkers [4]. Meanwhile, the quality of the functional coverage model and the checkers heavily depends on the experience of verification engineers. It is easy to omit some corner scenarios in the coverage model, and it is practically impossible to encode all correct behaviors in the checkers. It is even harder to build a good checker, for some modern hardware systems that may exhibit a degree of indeterminism, and the golden output is not always known [7, 8]. Therefore, building a high quality functional coverage model and a set of high quality functional checkers is a challenging task.

Mutation analysis is originally used to design new test data and evaluate the quality of existing test data in software testing [9]. It modifies a program syntactically, for example, by using a wrong operator. Each mutated program is a mutant. Mutants are created from well-defined mutation operators that mimic typical programming errors. Tests kill

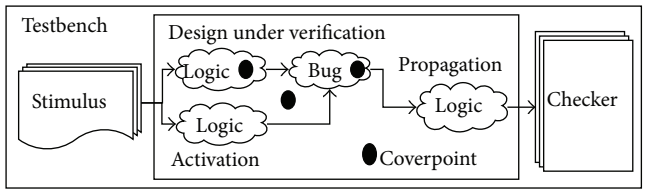

Figure 1: Simulation based verification testbench.

mutants by capturing the behavior of the original version that is different from that of the mutant. Error detecting ability of test suites is measured by the percentage of killed mutants. Note that mutation analysis assumes that the quality of the checkers is sufficient which usually is not real. The effectiveness of mutation analysis in hardware verification has been confirmed by recent works in diverse fields and levels of abstraction, such as RTL [10–12], SystemC [13–19], and embedded C software [20]. Commercial EDA tool for hardware description languages (HDL) mutation analysis has already been available [21].

To address aforementioned weaknesses of existing functional verification techniques, this paper presents a method to qualify the three components in the testbench by mutation analysis. The main contributions of our work include three aspects: (1) assessing both the integrity and quality of the coverage model and the set of checkers, (2) proposing a testbench quality metric which quantifies the quality of the testbench, and (3) proposing a simulation mechanism to accelerate the process of testbench qualification.

The rest of this paper is organized as follows. Section 2 reviews the existing related work. Section 3 explains the testbench qualification technique in detail. Section 4 presents experimental results to demonstrate the usefulness of the proposed technique. Section 5 concludes the paper.

2. Related Work

Mutation analysis has been actively studied for over three decades in software testing community and it has been applied to various software programming languages [9].

In recent years, mutation analysis has been applied to languages for system-level hardware modeling and verification, such as SystemC [13–19]. Several works have applied fault models for SystemC, such as perturbing SystemC TLM descriptions in [16, 17], introducing mutation operators for concurrent SystemC designs [18]. Other works have looked into error injection and fault localization in SystemC. For example, SCEMIT injects errors automatically into SystemC models [14]. Le et al. introduce an automatic fault localization approach for TLM designs based on bounded model checking [15]. Verification quality on SystemC is also investigated. The work in [13] develops mutation analysis based coverage metrics to attack the verification quality problem for concurrent SystemC programs. Functional qualification is introduced to measure the quality of functional verification of TLM models [19].

Additional to the application in system-level modeling and verification, mutation analysis has been applied also to HDL, such as Verilog and VHDL [10–12, 22]. The work in [10] qualifies the error detecting ability of test cases by mutation analysis and automatically improves validation data. Mutation analysis performed at TLM is reused at RTL to help designers in optimizing the time spent for simulation at RTL and improving the RTL testbench quality [11]. Liu and Vasudevan measure branch coverage through mutated guard in the symbolic expression during symbolic simulation [12]. HIFSuite [22] provides a framework that supports many fundamental activities including mutation analysis.

Recently, mutation analysis is applied to guide the process of stimulus generation for hardware verification to efficiently kill more mutants [23, 24]. Xie et al. propose a search based approach and defined an objective cost function to solve the problem of automatic simulation data generation targeting HDL mutants [23]. They extend the work in [23] by representing a simulation flow with two phases towards an enhanced mutation analysis score [24].

Coverage discounting technique [2–4] maps survived mutants to deficiently covered coverpoints and removes these coverpoints from coverage results, which leads to a decreased but better quality coverage results. One of the shortcomings of this technique is that it concerns the quality of testbench but focuses only on the quality of the existing coverage results and ignores the integrity of the coverage model, which will be addressed in this paper.

Observability-based coverage is first introduced in [25] to address the activation-only nature of other coverage metrics. Since then, attempts have been tried to generalize or extend the technique [26, 27]. Observability-based coverage metrics are a form of implicit metric which introduces data-flow evaluation. We consider statements to be covered only if they are executed and the result of that execution has a dynamic data flow to the output. However, it is always an assumption that the output is properly checked, and the presence and quality of the checker are never actually evaluated. Observability-based coverage metrics require specialized simulation tools with extensive instrumentation to record dynamic data-flows that are not available in most cases. As the observability concept only works with implicit metrics, it cannot be combined with arbitrary functional coverpoints.

3. Testbench Qualification

This section is organized as follows. The first subsection presents the basic idea of testbench qualification. The second and the third subsections describe refinement of the coverage model and the checkers, respectively. Then we propose a testbench quality metric by considering all three components in the testbench. The last subsection presents a simulation mechanism to accelerate the process of testbench qualification.

3.1. The Basic Idea. Verification engineers (VE) build the testbench manually according to the test plan and design specification, which is error prone and may lead to low quality elements in the testbench. According to the types of faults, we can divide a coverage model (resp., a set of checkers)

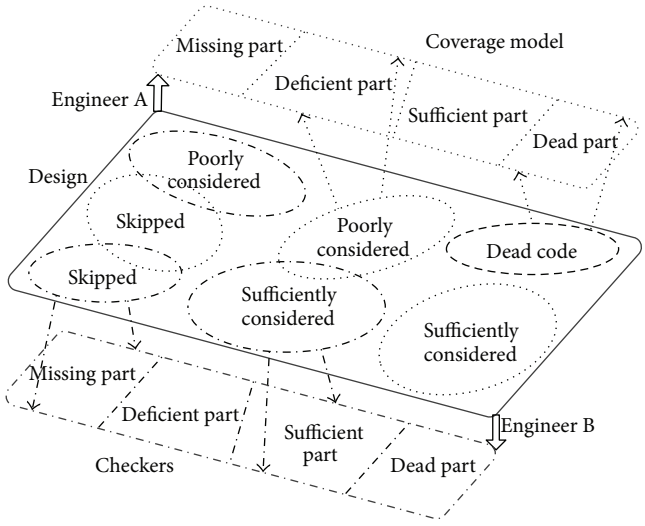

Figure 2: Fault classification of the coverage model and the checkers.

into four parts, as shown in Figure 2 (some dotted lines are not depicted for clarity): (1) the missing part (some relevant features of the design are skipped by the verification engineers because of careless), (2) the deficient part (some features of the design are poorly considered by the verification engineers), (3) the sufficient part (features of the design those are adequately handled by the verification engineers), and (4) the dead part (some features of the design can be dead codes which will never be executed). Although the partitions of the coverage model and the checkers are identical, a specific partition of the coverage model and the checkers can base on different parts of the design which usually overlap with each other, as shown in Figure 2. Because the coverage model and the checkers are possibly built by different engineers who can treat the design differently, or by the same engineer at different time, it is almost impossible to consider the design exactly the same at different time. The testbench qualification method aims at improving the testbench by eliminating the missing part and refining the deficient part.

Injecting error into the design may corrupt some combinational logics or/and sequential logics in the design. In the simulation results of the mutant, comparing to the simulation results of the original design, some unactivated functions may activate and some activated functions may be suppressed. With a well-built coverage model, all relevant functions are monitored by coverpoints. Therefore, with the error, some unfired coverpoints may fire and some fired coverpoints may not fire, which results in changes in the coverage results. In more detail, these changes can be one fired coverpoint being suppressed, or several unfired coverpoints being fired, or the former two events happen together. While with a deficient coverage model, it monitors only a subset of the relevant functions, which possibly not include the corrupted functions. In this situation, the coverage results may remain unchanged. For convenience, we use two terms, fluctuation and stable, in the rest of this paper. We define fluctuation as changes in the coverage results. If the coverage results remain identical, we define this is a stable coverage results.

Table 1: Reactions from the coverage model and checkers to the injected error.

Reactions from a coverage model	Reactions from checkers	Possible reasons
Fluctuated	Killed	Sufficient part
Fluctuated	Survived	Deficient checkers
Stable	Killed	Missing coverpoints
Stable	Survived	Several possible reasons

In a well-built testbench, there should be a corresponding relationship between the features of a design, the set of coverpoints in the coverage model, and the set of checkers. That is, for every feature in the design, there should be some coverpoints to ensure that it is actually implemented and executed and some checkers to ensure that the behavior of the feature is correct. Therefore, if some errors are injected into a design and the injection disturbs some functions of the design, these disturbed functions should result in fluctuation in the coverage results and be detected by the checkers. In other words, the design, the coverage model, and the set of checkers should behave consistently. If the expected result is not observed, either the coverage model or the checkers may be inadequate. Therefore, the quality of the coverage model and the checkers can be analyzed according to their reaction to the error injection during mutation analysis.

We summarize four possible combinations of reactions from the coverage model and the checkers when simulating a mutant. In Table 1, the first column lists possible reactions from a coverage model, which may lead to either fluctuated or stable coverage results. The second column presents possible reactions from checkers, which may either kill the mutant or let it survive. The last column provides possible reasons of each combination. In the first row, when a mutant is killed and the coverage results fluctuate, the related coverpoints and checkers are sufficient because they behave consistently. In the last row, when the mutant is alive and the coverage results remain stable, there are several possible reasons: (1) the mutant is not activated, (2) the coverage model and the set of checkers are both deficient, and (3) the error is injected into meaningless dead code. Complete analysis of this situation is quite heavy and left as a future work. We will mainly investigate the combinations of the two situations in the second and third rows, for example, missing coverpoints and deficient checkers, to refine the coverage model and the set of checkers.

3.2. Coverage Model Refinement. When a mutant is killed by the checkers but coverage results remain stable, some coverpoints are missing in the coverage model. As shown in Figure 3, only two coverpoints (CP1 and CP2) are used while there are more functions that need coverpoints to monitor. The coverage results before and after the injection of the error may always record CP1 and CP2 being covered, since the functions monitored by CP1 and CP2 are possibly not infected by the error. However, the checkers kill the mutant. A killed mutant means some functions of the design are indeed corrupted by the injected error, as the

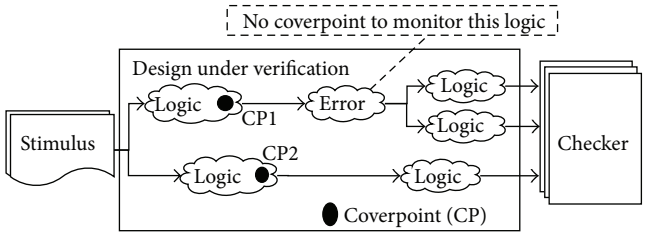

Figure 3: Insufficient coverage model.

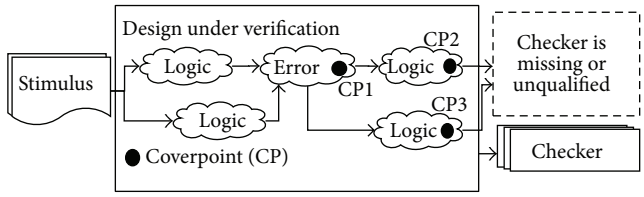

Figure 4: An insufficient checker.

```
(1) input x; output y;
(2) always @ (x) begin
(3)     if (x > 0) //the mutant replaces the > by <
(4)         y = 1 + x; //coverpoint2 is added here
(5)     else if (x == 0)
(6)         y = 2; //monitored by coverpoint1
(7)     else
(8)         y = 1 − x; //coverpoint3 is added here
(9) end
(10) assertion: y > 1; //the checker
(11) //test vectors: x1 = 0; x2 = 1; x3 = 2;
(12) //original outputs: y1 = 2; y2 = 2; y3 = 3;
(13) //mutant outputs: y1 = 2; y2 = 0; y3 = −1;
```

Listing 1: Coverage model qualification example.

checkers have detected it. The stable coverage results from the corrupted functions come from the missing coverpoints to monitor these corrupted functions. The coverage model can be qualified and improved by finding out and eliminating these missing coverpoints.

A well-built coverage model is constructed by a set of coverpoints which encode the entire features of the design at a certain level of abstraction that is determined by the verification requirement. The set of coverpoints should be able to monitor all behaviors of the design and be sensitive enough to the injected errors which cause wrong behaviors at that certain level of abstraction. For instance, the nontrivial functions that we plan to verify are a set $\{F1, F2, \ldots, Fn\}$. Accordingly, the coverage model should consist of a set of coverpoints $\{C1, C2, \ldots, Cn\}$ to monitor and ensure that all functions in $\{F1, F2, \ldots, Fn\}$ are implemented and exercised. The coverage model refinement is a repetitive process of qualification and improvement of the existing coverage model, involving the following steps: (1) simulating the original design and producing the original coverage results, (2) simulating mutants and producing coverage results from the mutants, (3) adding missing coverpoints if a mutant is killed but coverage results remain stable, and (4) repeating previous steps until the quality of the testbench satisfies the predefined threshold.

Consider Listing 1 as an example, which is a code block of the DUV with one checker (line 10), and the coverage model only contains coverpoint1 to monitor line (6). After simulation of the test vector ($x1 = 0; x2 = 1; x3 = 2$) against the original design, the output is $y1 = 2; y2 = 2; y3 = 3$. The coverage model records that coverpoint1 is covered and the checker does not fire.

Then an error is injected into line (3) which replaces ">" by "<". After simulation of the test vectors ($x1 = 0; x2 = 1; x3 = 2$) against the mutant, the output is $y1 = 2; y2 = 0; y3 = -1$. The coverage model records that coverpoint1 is covered just the same as the original design. But the checker does fire when $x2 = 1$ and $y2 = 0$. Therefore, this mutant is killed. In this example, the coverage results remain stable, but the checker fires and kills the mutant. The discrepancy between reaction of the coverage model and reaction of the checkers indicates that some coverpoints are missing. Because the code is simple, it is straightforward to know that coverpoints should be added to line (4) and line (8). After improving the coverage model by adding coverpoint2 and coverpoint3 to line (4) and line (8), respectively, the firing of the checker is consistent with the fluctuation of coverage results. That is, the original coverage results record that coverpoint1 and coverpoint2 are covered, but the mutant coverage results record that coverpoint1 and coverpoint3 are covered.

3.3. Checker Refinement. The features of the design, the coverpoints in the coverage model, and the set of checkers are tightly related. Anything happened in one component can either trace back to some reasons or result in some consequences in other components. Being at the later stage in the testbench, a perfect set of checkers should be capable of reflecting any activity happened in the design and the coverage model.

When a mutant survives but the coverage results fluctuate, the set of checkers is insufficient. More precisely, the fluctuated coverage results reflect that some functions of the design are corrupted by the injected errors, but the checkers are incapable of detecting the wrong behavior. The possible reasons can be either missing checkers or low quality checkers. As shown in Figure 4, three coverpoints (CP1, CP2, and CP3) are used to monitor the functions that are tightly related to the injected error. With the original design, the coverage results may recode that CP1 and CP2 are covered. However, the injection of the error can cause misbehavior of the design and lead to a fluctuated coverage results that recode CP1 and CP3 being covered. While the coverage results fluctuated, the mutant is alive, because the checker is either missing or unqualified.

As it is impractical to encode all functions of a design into a perfect set of checkers at the beginning, we also need an iterative improvement process to deliver adequate

```
(1) input x; output y;
(2) always @ (x) begin
(3)     if (x > 0) //mutant1 replaces the > by <
(4)         y = 1 + x; //monitored by coverpoint1
(5)     else if (x == 0)//mutant2 replaces the condition by true
(6)         y = 2; //monitored by coverpoint2
(7)     else
(8)         y = 1 − x; //monitored by coverpoint3
(9) end
(10) Original assertion: none;
(11) The first added assertion: y > 1;
(12) The final assertion: (x == 0)? y == 2: y == 1 + abs(x);
(13) //test vectors: x1 = −2; x2 = 0;
(14) //original outputs: y1 = 3; y2 = 2;
(15) //mutant1 outputs: y1 = −1; y2 = 2;
(16) //mutant2 outputs: y1 = 2; y2 = 2;
```

Listing 2: Checker qualification example.

checking ability. Sometimes, the link between checkers and features of the design can be missing or weak that they can only check partial properties of a function rather than the entire function. These missing checkers should be added, and weak checkers should be refined until they can reveal both functional and performance bugs.

A set of well-built checkers should be as sensitive as a well-built coverage model to the injected errors. The checkers refinement process is as follows: (1) simulating the original design and producing the original coverage results, (2) simulating mutants and producing coverage results from the mutants, (3) adding the missing checkers or improving the deficient checkers if a mutant is alive but coverage results fluctuate, and (4) repeating previous steps until the quality of the testbench satisfies the predefined threshold.

Listing 2 is another example to explain the checker refinement technique. The code block in Listing 2 is the same as that in Listing 1 except that there are more coverpoints and without any checkers at the beginning, after simulation of the test vector ($x1 = -2; x2 = 0$) against the original design, the coverage model records that coverpoint2 and coverpoint3 are covered.

In mutant1, an error is injected into line (3) which replaces ">" by "<". After simulation of the test vector ($x1 = -2; x2 = 0$) against mutant1, the output is $y1 = -1; y2 = 2$. The coverage results are changed to coverpoint1 and coverpoint2 which are covered. But the mutant is survived because there is no checker to kill it. Therefore, some checkers are missing, and we add a checker ($y > 1$) as line (11) in Listing 2 to kill mutant1.

In mutant2, instead of the error injected in line (3), another error is injected into line (5), which replaces the condition express (in==0) with true. After simulation of the test vector ($x1 = -2; x2 = 0$) against mutant2, the output is $y1 = 2; y2 = 2$. The checker in line (11) ($y > 1$) cannot kill this mutant, which needs refinement. A more elaborate checker in line (12) can kill this mutant. After the refinement, the checker encodes the entire function of the design and deliveries good error detecting ability.

3.4. Testbench Quality Metric. After refining the coverage model and the set of checkers, we propose a testbench quality metric to qualify the whole testbench. A metric that qualifies the whole testbench should consider (1) the stimulus, (2) the coverage model, and (3) the set of checkers. The metric is defined in (1), where $\text{quality}_{\text{tb}}$, $\text{quality}_{\text{sti}}$, $\text{quality}_{\text{cov}}$, and $\text{quality}_{\text{che}}$ are quality of testbench, stimulus, coverage model, and the set of checkers, respectively, and w_{sti}, w_{cov}, and w_{che} are weights that reflect different importance of various components with the constraint $w_{\text{sti}} + w_{\text{cov}} + w_{\text{che}} = 1$. The weights are selected feasibly to meet the current verification requirement. A lower weight is set for a component when you are confident about its quality. Otherwise, a higher weight is set

$$\text{quality}_{\text{tb}} = w_{\text{sti}} \times \text{quality}_{\text{sti}} + w_{\text{cov}} \times \text{quality}_{\text{cov}} + w_{\text{che}} \times \text{quality}_{\text{che}}, \tag{1}$$

$$\text{quality}_{\text{sti}} = \text{compact\_factor}_{\text{sti}} \times \text{integrity\_factor}_{\text{sti}}, \tag{2}$$

$$\text{quality}_{\text{cov}} = \frac{\text{sufficient\_factor}}{\text{base}}, \tag{3}$$

$$\text{quality}_{\text{che}} = \text{compact\_factor}_{\text{che}} \times \text{integrity\_factor}_{\text{che}}. \tag{4}$$

The $\text{quality}_{\text{sti}}$ is defined as the product of $\text{compact\_factor}_{\text{sti}}$ and $\text{integrity\_factor}_{\text{sti}}$ as in (2). Being the ratio of the number of tests having activated different coverpoints over the total number of tests, $\text{compact\_factor}_{\text{sti}}$ reflects the compactness of the stimulus. $\text{integrity\_factor}_{\text{sti}}$ is the ratio of the number of covered coverpoints over the total number of coverpoints, which represents the completeness of the stimulus. The range of $\text{quality}_{\text{sti}}$ lies between 0 and 1. When it approaches 0, all the tests are useless and none of them can active any coverpoints. When it approaches 1, the test suit is compact and complete.

The $\text{quality}_{\text{cov}}$ is defined as the ratio of the sufficient_factor over the base as in (3). sufficient_factor is the number of coverpoints whose coverage results fluctuate when simulating killed mutants. It reflects the size of the sufficient part of the coverage model that is properly checked. base is the total number of coverpoints. The range of $\text{quality}_{\text{cov}}$ is also between 0 and 1. When it approaches 0, all the existing coverpoints are useless and none of relevant functions are monitored. When it approaches 1, all the existing coverpoints are useful and monitoring some relevant functions.

The $\text{quality}_{\text{che}}$ is defined as the product of $\text{compact\_factor}_{\text{che}}$ and $\text{integrity\_factor}_{\text{che}}$ as in (4). The former, being the ratio of number of the fired checkers over the total number of checkers, reflects the compactness of the set of checkers. And its value lies between 0 and 1, where 0 means that all the checkers have not taken effect, and 1 means that all the checkers have indeed taken effect and there is no redundant checkers at all. The latter is the ratio of the number of killed mutants with fluctuated coverage results over the total number of mutants with fluctuated coverage results. $\text{integrity\_factor}_{\text{che}}$ reflects the completeness of the set of checkers and the value lies between 0 and 1, where 0 means that all the existing checkers are useless, and 1 means that all the existing checkers are useful.

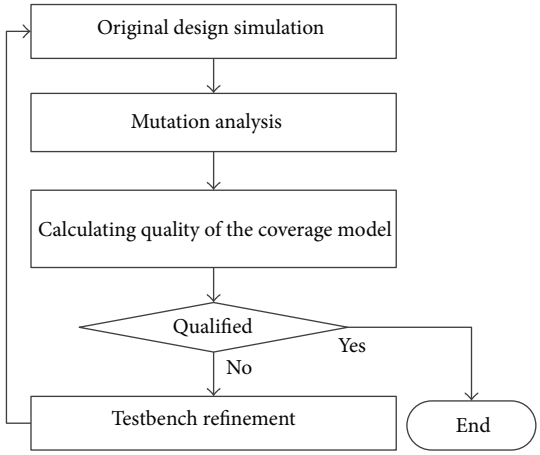

Figure 5: The work flow.

As the sum of weighted $\text{quality}_{\text{sti}}$, $\text{quality}_{\text{cov}}$, and $\text{quality}_{\text{che}}$, the range of $\text{quality}_{\text{tb}}$ is also between 0 and 1, where 0 means that the testbench is of low quality, and 1 means that the testbench is of high quality.

3.5. Simulation Mechanism. The overall work flow of the proposed method is depicted in Figure 5. First, normal simulation of the original design is conducted, and the $\text{quality}_{\text{sti}}$ can be determined by the collected coverage results. Then errors are injected and mutants are simulated during mutation analysis, and the $\text{quality}_{\text{che}}$ can be determined by mutation score and coverage results of mutant simulation. In this stage, we check the simulation results of the mutants to determine the possibility of coverage model refinement and checks refinement. If any one of the two refinements is possible, we record the type of the refinement, the test case, and the mutant, which will be used in the testbench refinement stage. Subsequently, we calculate the $\text{quality}_{\text{cov}}$ to determine the $\text{quality}_{\text{tb}}$. Finally, the process terminates if the quality of the testbench satisfies the predefined threshold. Otherwise, we refine the testbench and repeat the previous steps. In the testbench refinement stage, we investigate the possible refinements recorded in the mutation analysis stage. We refine the coverage model as discussed in Section 3.2 if the coverage results remain stable while the mutant is killed or refine the checkers as discussed in Section 3.3 if the coverage results fluctuated while the mutant is alive. There can be one or more possible refinements which may lead to addition of one or more coverpoints or checkers. But the adding is not always happening, since the refinement is manually done. In the iterative work flow, with the refined checkers produced in the previous iteration, we can further refine the coverage model in the current iteration. Because a better checker can possibly kill more mutants, which can result in more instances that the coverage results remain stable while the mutant is killed. Similarly, we can further refine the checkers with the better coverage model given by the previous iteration. So the coverage model and the checkers are incrementally refined iteration by iteration.

Usually, a quality threshold is predefined and the testbench qualification process can stop if it is achieved. Instead of calculating the exact $\text{quality}_{\text{cov}}$, approximating it as fast as possible can achieve the threshold earlier and accelerate the testbench qualification process. We provide two criteria to accelerate the calculation of an approximated $\text{quality}_{\text{cov}}$. (1) Criterion to select stimulus: select the one covering the most coverpoints. This criterion provides high probability to increase the numerator of $\text{quality}_{\text{cov}}$ as fast as possible. (2) Criterion to select mutant: select the one that is killed and activated by the least stimulus. Such a mutant usually is related to some corner features, and they can lead to higher $\text{quality}_{\text{cov}}$ when the coverage results reach the ceiling.

4. Experimental Results

4.1. Experimental Setup. We use two designs with different scales to show the effectiveness and scalability of the proposed method. The first is an interconnection module (ICM) that connects four cores and maintains cache coherence operations in a multicore processor. The second is a commercial RISC CPU. The fault injection and mutation analysis are performed with self-built scripts. We select a set of mutation operators to limit the population of mutants. According to the mutation operators, the fault injection script parses the design files to locate syntaxes to inject errors first and then inject errors at these located syntaxes to produce mutants. The mutation analysis script simulates every test case in the test suit on each mutant first and then checks the simulation results against the simulation results from the original design to determine whether the refinement of the coverage model or the checkers is possible. If possible, we record the type of refinement (the coverage model or the checkers), the test case, and the mutant in a text file which is used in the testbench refinement stage. Finally, according to the flow given in Figure 5, we build a top level script, which is responsible for calculating $\text{quality}_{\text{sti}}$, $\text{quality}_{\text{cov}}$, $\text{quality}_{\text{che}}$, and $\text{quality}_{\text{tb}}$, based on the fault injection script and the mutation analysis script. The weights w_{sti}, w_{cov}, and w_{che} are assigned with 0.2, 0.4, and 0.4, respectively. The values of w_{cov} and w_{che} are set higher than that of w_{sti}, because this paper focuses on the qualification of the coverage model and the checkers. The threshold of the testbench quality is 0.9. We carry out the experiments on a computer with an Intel i5 dual-core CPU at 1.8 GHZ and 8 G RAM.

4.2. Constrained Random Stimulus on the ICM. In this experiment, we have used 20 constrained random test cases. Each random test case consists of 100 memory accessing instructions in every thread. There are 400 instructions in a test case, for the setup with 4 threads for 4 cores. All instructions in these test cases access a limited range of address space to increase the probability of cache coherence interaction between different cores. The testbench aims at verifying all kinds of cache coherence operations supported by the ICM. A well-built coverage model that satisfies the verification requirements should consist of coverpoints (bins or sequences in system Verilog) to monitor all possible cache

Table 2: Results under random stimulus.

Iteration	0	1	2	3	4	5	6	7	8	9
Coverpoints	12	17	23	30	33	41	48	53	58	60
Checkers	4	5	7	8	9	11	12	13	15	16
Testbench quality	0.51	0.55	0.58	0.62	0.69	0.76	0.79	0.82	0.88	0.91

Table 3: Results under directed stimulus.

Iteration	0	1	2	3	4	5	6
Coverpoints	12	23	32	40	49	57	62
Checkers	4	6	7	9	11	14	16
Testbench quality	0.51	0.59	0.65	0.72	0.79	0.86	0.92

coherence operations between every individual core and the ICM. The total number of coverpoints is 64 in the 4-core processor. But only a subset of well-built coverage model is developed at the beginning of the experiments, which will be subsequently refined during testbench qualification. A well-built set of checkers should be able to thoroughly check results of all kinds of cache coherence operations. The total number of checkers is 16 in the 4-core processor. We develop a subset of well-built checker set at the beginning of the experiments for the same reason of the coverage model. There are 623 mutants in total.

During the testbench qualification process, coverpoints are added when a mutant is killed but the coverage results remain stable. And checkers are added or refined when a mutant survives but the coverage results fluctuate. With the improved testbench, more mutants are killed. Table 2 presents the experimental results. The first row is the index of each iteration, and iteration 0 is the initial state. The second and third rows show the numbers of the coverpoints and checkers, respectively. The number of coverpoints does not reach 64. The reason is that some coverpoints are interrelated, and injected errors can lead to the fluctuation in the coverage results with a subset or the entire set of coverpoints. Along with the improvement of the coverage model and the set of checkers, the quality of the testbench is improved, as shown in the last row. Finally, we take 9 iterations to satisfy the testbench quality threshold. The simulation time is a little more than 5 hours, which excludes the manual refinement time.

4.3. Directed Stimulus on the ICM. A good quality metric should be able to distinguish the quality of stimulus. A stimulus with high quality is elaborately designed to activate, propagate, and detect bugs in the DUV. Therefore, dedicated directed stimulus has higher probability to expose the weakness of the testbench than random stimulus. Therefore, we manually build 20 directed test cases each with 400 instructions, which are dedicated to test cache coherence operations. The testbench qualification process is reconducted on the ICM and we expect that the quality threshold should be satisfied with fewer iterations. The experimental results under directed stimulus is given in Table 3. The directed stimulus takes 6 iterations which cost less than 3 hours in simulation to satisfy the threshold and this confirms the expectation.

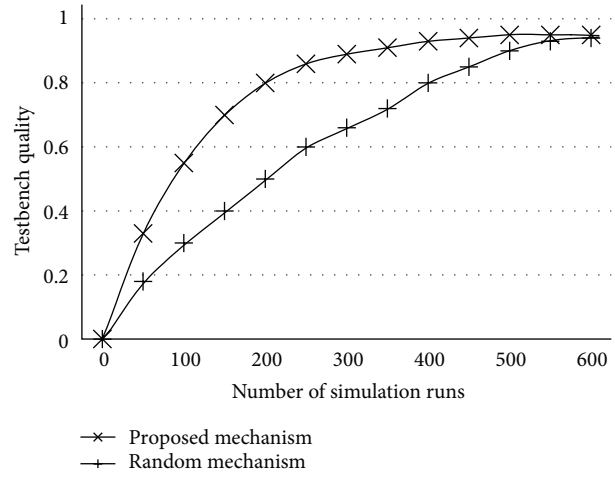

Figure 6: Comparisons of different simulation mechanisms.

4.4. Comparison of Different Simulation Mechanism on the ICM. We use a random selection mechanism as the comparison, which randomly selects test cases and mutants when calculating the quality of the coverage model. The difference between the speeds of the testbench quality improvement with different simulation mechanism will demonstrate the efficiency of the proposed simulation mechanism.

This experiment uses 50 random test cases to stimulate the ICM, and each test case consists of 100 instructions for each thread. We present the trends of the testbench quality under different simulation mechanism with 50 random test cases in Figure 6. The testbench quality under the proposed simulation mechanism increases much faster at the beginning stage than that under the random selection mechanism, meaning that the proposed simulation mechanism can achieve the quality threshold with fewer simulation runs.

4.5. Experiments on a Commercial CPU. To show that the proposed method is feasible for larger scale design, we adopt a commercial RISC CPU whose instruction set architecture consists of more than 150 instructions. We intend to use 20 constrained random test cases to thoroughly verify all kinds of instruction types, and each test case consists of 100 instructions. Operands of instruction are constrained as some rare values to incur interesting corner scenarios, for example, the address of memory accessing instruction is close to page boundary.

In this case, a sufficient coverage model should have coverpoints to monitor each individual instruction type, and a sufficient set of checkers should properly investigate outputs of all execution units. Only a subset of the sufficient one is

Table 4: Results from the commercial CPU.

Iteration	0	1	2	3	4	5	6	7	8	9	10
Coverpoints	30	39	52	61	74	81	91	105	123	131	143
Checkers	10	13	15	16	18	20	22	24	26	29	32
Testbench quality	0.48	0.55	0.60	0.63	0.68	0.71	0.75	0.79	0.83	0.86	0.91

constructed in the initial stage of the experiment to show the effect of the qualification. The improving progresses of the coverpoints, the checkers, and the quality of testbench are provided in Table 4. We can see that when the proposed method is applied to the CPU, it is as effective as the case to the ICM. Simulation in this experiment consumes almost 7 hours.

5. Conclusion

The growing complexity of modern hardware systems and time-to-market pressure require more efficient and high quality functional verification. These requirements necessitate a high quality testbench that can thoroughly verify the DUV in limited time. Currently, the thoroughness of function verification is dominantly measured by structural code coverage and user defined functional coverage. However, all of these coverage metrics concentrate on activation of the design but ignore the propagation and detection aspects that are indispensable to expose bugs. This paper qualifies the whole testbench by considering activation, propagation, and detection process. In particular, we have (1) improved the integrity and quality of the coverage model and the set of checkers through mutation analysis, (2) presented a metric to measure the quality of the testbench, and (3) proposed a simulation mechanism to satisfy the metric faster. Experimental results demonstrate the effectiveness of the proposed method. The future work can be extended in the following directions: (1) detail discussion about the situation that a mutant is survived and the coverage results remain stable and (2) formalizing the relationship between the design, the coverage model, and the set of checkers.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] Y. Shuo, R. Wille, and R. Drechsler, "Improving coverage of simulation-based verification by dedicated stimuli generation," in *Proceedings of the 17th Euromicro Conference on Digital System Design (DSD '14)*, pp. 599–606, Verona, Italy, 2014.

[2] P. Lisherness and K.-T. Cheng, "Improving validation coverage metrics to account for limited observability," in *Proceedings of the 17th Asia and South Pacific Design Automation Conference (ASP-DAC '12)*, pp. 292–297, Sydney, Australia, February 2012.

[3] P. Lisherness, N. Lesperance, and K.-T. Cheng, "Mutation analysis with coverage discounting," in *Proceedings of the 16th Design, Automation and Test in Europe Conference and Exhibition (DATE '13)*, pp. 31–34, IEEE, Grenoble, France, March 2013.

[4] P. Lisherness and C. Kwang-Ting, "Coverage discounting: a generalized approach for testbench qualification," in *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT '11)*, pp. 49–56, Napa Valley, Calif, USA, November 2011.

[5] R. Grinwald, E. Harel, M. Orgad et al., "User defined coverage—a tool supported methodology for design verification," in *Proceedings of the Design Automation Conference (DAC '98)*, pp. 158–163, San Francisco, Calif, USA, 1998.

[6] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design & Test of Computers*, vol. 18, no. 4, pp. 36–45, 2001.

[7] A. Meixner and D. J. Sorin, "Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 1, pp. 18–31, 2009.

[8] A. Adir, A. Nahir, and A. Ziv, "Concurrent generation of concurrent programs for post-silicon validation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 8, pp. 1297–1302, 2012.

[9] J. Yue and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[10] Y. Serrestou, V. Beroulle, and C. Robach, "Functional verification of RTL designs driven by mutation testing metrics," in *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD '07)*, pp. 222–227, Lübeck, Germany, August 2007.

[11] V. Guarnieri, G. Di Guglielmo, N. Bombieri et al., "On the reuse of TLM mutation analysis at RTL," *Journal of Electronic Testing*, vol. 28, no. 4, pp. 435–448, 2012.

[12] L. Liu and S. Vasudevan, "Efficient validation input generation in RTL by hybridized source code analysis," in *Proceedings of the 14th Design, Automation and Test in Europe Conference and Exhibition (DATE '11)*, pp. 1–6, Grenoble, France, March 2011.

[13] A. Sen and M. S. Abadir, "Coverage metrics for verification of concurrent SystemC designs using mutation testing," in *Proceedings of the 15th IEEE International High Level Design Validation and Test Workshop (HLDVT '10)*, pp. 75–81, IEEE, Anaheim, Calif, USA, June 2010.

[14] P. Lisherness and K.-T. Cheng, "SCEMIT: a systemc error and mutation injection tool," in *Proceedings of the 47th ACM/IEEE Design Automation Conference (DAC '10)*, pp. 228–233, ACM, Anaheim, Calif, USA, June 2010.

[15] H. M. Le, D. Grosse, and R. Drechsler, "Automatic TLM fault localization for SystemC," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 8, pp. 1249–1262, 2012.

[16] N. Bombieri, F. Fummi, and G. Pravadelli, "A mutation model for the SystemC TLM 2.0 communication interfaces," in *Proceedings of the Design, Automation and Test in Europe (DATE '08)*, pp. 396–401, Munich, Germany, March 2008.

[17] N. Bombieri, F. Fummi, and G. Pravadelli, "On the mutation analysis of systemC TLM-2.0 standard," in *Proceedings of the 10th International Workshop on Microprocessor Test and Verification (MTV '09)*, pp. 32–37, Austin, Tex, USA, December 2009.

[18] A. Sen, "Mutation operators for concurrent systemC designs," in *Proceedings of the 10th International Workshop on Microprocessor Test and Verification: Common Challenges and Solutions (MTV '09)*, pp. 27–31, Austin, Tex, USA, December 2009.

[19] N. Bombieri, F. Fummi, G. Pravadelli et al., "Functional qualification of TLM verification," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '09)*, pp. 190–195, Nice, France, 2009.

[20] S. Bouvier, N. Sauzede, F. Letombe et al., "A practical approach to measuring and improving the functional verification of embedded software," in *Proceedings of the Design Verification Conference (DVC '12)*, June 2012.

[21] M. Hampton and S. Petithomme, "Leveraging a commercial mutation analysis tool for research," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques—MUTATION (TAICPART-MUTATION '07)*, pp. 203–209, Windsor, UK, September 2007.

[22] N. Bombieri, G. Di Guglielmo, M. Ferrari et al., "HIFSuite: tools for HDL code conversion and manipulation," *Eurasip Journal on Embedded Systems*, vol. 2010, Article ID 436328, 2010.

[23] T. Xie, W. Mueller, and F. Letombe, "HDL-mutation based simulation data generation by propagation guided search," in *Proceedings of the 14th Euromicro Conference on Digital System Design (DSD '11)*, pp. 608–615, Oulu, Finland, September 2011.

[24] X. Tao, W. Mueller, and F. Letombe, "Mutation-analysis driven functional verification of a soft microprocessor," in *Proceedings of the IEEE International SOC Conference (SOCC '12)*, pp. 283–288, IEEE, Niagara Falls, NY, USA, September 2012.

[25] F. Fallah, S. Devadas, and K. Keutzer, "OCCOM-efficient computation of observability-based code coverage metrics for functional verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 8, pp. 1003–1015, 2001.

[26] T. Lv, J.-P. Fan, X.-W. Li, and L.-Y. Liu, "Observability statement coverage based on dynamic factored use-definition chains for functional verification," *Journal of Electronic Testing*, vol. 22, no. 3, pp. 273–285, 2006.

[27] P. Lisherness and K.-T. Cheng, "An instrumented observability coverage method for system validation," in *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT '09)*, pp. 88–93, San Francisco, Calif, USA, November 2009.