Domain Coverage Metric for SoC Validation

Xue-Xiang Wang and Jun Yang

Abstract—The importance of system-on-chip (SoC) validation continues to grow with the increase of design size. An innovative domain coverage metric is proposed to measure the completeness and quality of validation approach. Domain methodology is based on a geometrical analysis of the domain boundary and takes advantage of the fact that the point on or near the boundary is the most sensitive to domain errors. The coverage tool has been implemented using Verilog procedural interface (VPI) and applied to validation of SoC under design. Results show that the domain coverage can detect many design faults which statement and path coverage can not.

Index Terms—Coverage, domain, system-on-chip validation.

1. Introduction

In spite of recent technology advancement in design automation, hardware design verification is still a challenging work. Among various design stages, verification of register-transfer-level (RTL) design is often left to engineer’s insight and judgment due to the lack of tools. The process of verifying correctness of system-on-chip (SoC) designs consumes between 60% and 80% of design effort.

Many approaches are for verifying a design before production. Popular approaches include physical prototype, emulation, hardware acceleration, simulation and formal verification. Physical prototype is an expensive and ineffective verification method for SoC design. Formal verification promises the ability to prove that a given design meets its specification. However, as the design size grows, most techniques are affected by the state explosion problem. Model checkers can verify circuits with hundreds of latches, but many industrial circuits are at least an order of magnitude larger[1]. Emulation systems use reconfigurable hardware (typically FPGA) to implement essentially equivalent functionality in hardware. However, limitation on the visibility into an emulated system makes the debug of the hardware description very difficult, especially in the early development stage. Hardware acceleration for verification comprises of custom hardware that is dedicated to specific simulation application. As a result, this approach is extremely expensive and the reuse ability is very poor.


Statement coverage measures how many of the total lines of code be executed by the test suite, and 100 percent coverage means all the source codes be executed but the correctness of the device under test (DUT) is not implied. State machine coverage measures state visitation and state transition. Condition coverage measures how sub-expressions in statements are evaluated during the simulation[8]. Toggle coverage monitors each net and register for any value transition from 0 to 1 and 1 to 0 to obtain coverage results with regard to the behavior and structure of the design. Prior to statement coverage, observed coverage tells not only which line is executed during simulation, but also whether the execution of the line makes any difference to the values on the outputs of the design.

Corner case is generally the most difficult to activate by test vector. Fig. 1 illustrate a common design fault, line 4 “if(counter_r>4'b0111)” is incorrect, the correct is “if(counter_r>4'b0100)”. Although the test suite obtain high statement coverage, observed coverage, condition coverage, toggle coverage and state machine coverage, this kind of faults can not be revealed, because these faults lie on or near the boundaries. The boundaries should be tested...
Nodes represent statements, and edges represent possible outcomes of decision statements, such as if, if-else or case statement. They are thought as being labeled with outcomes of decision statement, such as if, if-else or case statement. If and if-else statements have two out-going edges, the first edge represents the predicate's boundary but not path's boundary, so the weak domain coverage is not an effective metric to measure the completeness and quality of validation. In this paper, an innovative strict domain coverage metric is proposed and applied to measure the completeness and quality of RTL validation approach.

The rest of this paper is organized as follows. Section 2 describes the domain terminology; the domain fault model is introduced in Section 3; Section 4 proposes the test point selecting strategy, domain coverage metric is presented in Section 5; the experiments results and conclusions are given in Section 6 and Section 7 respectively.

### 2. Domain Terminology

In Verilog, a program can be described by modules executed concurrently. A module consists of a number of concurrently executing processes. Process can run continuously like always or continuous assignment, or run only once like initial process.

The control flow graph (CFG) of a process (always or continuous assignment) is a directed graph $G=<V, E, D>$, where $V$ is the set of nodes of $G$, and $E$ is the set of edges. Nodes represent statements, and edges represent possible flows of the control between statements. If a node of $G$ has more than one edge leaving, then the edge represents the outcomes of decision statement, such as if, if-else or case statement. They are thought as being labeled with appropriate predicates. If and if-else statements have two out-going edges, case statement is rewritten in terms of if-else statements. $B$ and $D$ are specific nodes, denoting respectively begin node and end node of the process.

The CFG of a process is shown in Fig. 2. TriggerLevel and FullLevel are constants, FifoCounter and Time are variables. There are four paths from always begin node to the end node. Which path to be executed depends on the value of variables FifoCounter and Time. If “Time<TimeOut” and “(FifoCounter>=TriggerLevel) && (FifoCounter<FullLevel)” are both true, as the shadowed domain in Fig. 3, the path, from begin node through node 5, 7 and 9 to end node, will be executed. Every path has an executing condition, which is termed path condition. Predicate node (if, if-else) has two out-going edges, the Boolean value of predicate expression determines which edge will be executed. If the expression’s Boolean value is true, left edge will be executed, else right edge. Therefore, a path corresponds to a path condition and if only a condition is satisfied, the path can be executed. A path condition is the conjunction of all individual predicates encountered along the path. The expression of a predicate may be a complex expression, which includes logical operator ($\&\&$, $\|\$). References [11], [12], and [13] assumed that predicate expression is only a simple expression. In RTL description, this assumption is not suitable, because complex expression is widely used to describe the behavior of hardware. In this paper, we do not limit predicate expression.

A path condition defines a path domain, which is the set of variables making path condition satisfied and causing the path executed. A path domain is surrounded by a boundary. Section of the boundary is called border, which is formed by a simple expression of the path condition. A border of a domain may either be closed or open with respect to that domain. A closed border belongs to the path domain and comes from a simple expression that contains $\geq$, $\leq$, or $\approx$. An open border is not part of the path domain and comes from a simple expression that contains $\neq$, $>$, or $<$. The given border, which corresponds to a simple predicate expression, is the border to be tested and may be correct or not. When a domain error occurs along a path, it may be thought as being caused by a border that is different from the correct. In order to find whether the given border is correct, we need a strategy to select a number of test points. References [11] to [13] have proposed different strategies for selecting test points. In this paper, we propose a new strategy. The first type of point required by domain testing is known as ON point, which lies on the given border. The second type of point required is known as OFF point, which lies slightly off the given border on the open side of it. The distance between the border and the OFF point is 1. The conception is illustrated in Fig. 3; points A and B are the OFF point and the ON point, respectively. Small arrows are used to indicate that which side of the borders is closed.
3. Domain Fault Model

Vector is mostly used to present variables, so we should first map the vector type variable reg[msb:lsb] and wire[msb:lsb] to scalar type variable. The following formulas are used to map vector to scalar.

For reg[msb:lsb],
\[
\text{reg}[msb : lsb] = \text{wire}[0] \times 2^l + \text{reg}[l] \times 2^{l-1}.
\]

For wire[msb:lsb],
\[
\text{wire}[msb : lsb] = \text{wire}[0] \times 2^l + \text{wire}[l] \times 2^{l-1}.
\]

where \( l = \text{msb} - \text{lsb} \).

After mapping vector to scalar, the way to handle the RTL description is the same with the software testing\(^{[14]}\). All the variables in RTL description are mapped to positive integers, so we only need to consider the positive integer spaces and ignore the negative integer spaces.

Domain errors are caused by an incorrectly specified predicate. Domain errors are divided into two types: Domain Fault I and Domain Fault II. An operand fault in predicate expression, which causes a border shifted from its correct position, is termed Domain Fault I. An operator fault in predicate expression is termed Domain Fault II.

4. Test Points Selecting Strategy

4.1 Strategies

For revealing the potential design errors, a set of test points are required.

A path condition is the conjunction of all individual predicates encountered along the path. In Fig. 4, the path condition, making line 11 to be executed, is the boolean values of expressions \( E_1, E_2, \ldots, E_n \). Suppose there be \( n \) predicates along a path, we define the path condition \( \alpha \)
\[
\alpha := (E_1) \wedge (E_2) \wedge \cdots \wedge (E_n) \quad (1)
\]

where \( E_i \) is the \( i \)th predicate expression, which combines operands with operators. \( E_i \) may be simple or complex expression. For checking whether domain, Faults I and II occur, we should select test points. For simple expressions, we select test points according to following strategies I, II and III. For complex expressions, we translate them into simple expressions. All \( E_i \) can be translated into simple expressions in (1)\(^2\).

We select test points for simple expression through the following three strategies.

Strategy I: For testing a border, which corresponds to a simple expression containing equality or nonequality \((==, !=)\) operator, one ON point and two OFF points are required. This is depicted in Fig. 5 (a). Border K corresponds to a simple expression containing equality or nonequality \((==, !=)\) operator such as if(reg[2:0] == 3'b111), point A and C are OFF points, point B is ON point.

Strategy II: For testing a border, which corresponds to a simple expression containing inequality \( (<, >, <=, >=) \) operator and \( N \) variables, one OFF point and \( N \) ON points are required. This is depicted in Fig. 5 (b). Border K corresponds to a simple expression containing inequality \( (<, >, <=, >=) \) operator and 2 variables (2-dimension) such as if( reg_a[2:0] + reg_b[2:0]<3'b111), point A is OFF point, points B and C are ON points.

Strategy III: For testing a border, which corresponds to a simple expression not containing relation operator \((==, !=, <, >, <=, >=)\), such as predicating if(A[1]&B[1]), one ON point \((A[1]&B[1] \text{ true})\) and one OFF point \((A[1]&B[1] \text{ false})\) are required.

4.2 Procedures of Test Points Selecting

Test points are selected by the above strategies, the detail procedure is clarified by Fig. 6.
5. Domain Coverage Metric

Coverage Metric is used to measure the quality and the completeness of validation. We generate test points by strategies I, II and III before simulation, monitor the value of variable for the purpose of checking whether test point is activated during simulation, and calculate the domain coverage after simulation. Every test point has a weight \( w \) indicating the importance of test point. After simulation, we sum up the number of test points that have been activated during simulation, and calculate the DUT’s coverage according to following formula:

\[
\text{coverage} = \frac{\sum \text{activated test point} \times w}{\sum \text{test point} \times w} \times 100\%.
\]

6. Experiments

We implemented the domain coverage tool using about 8600 lines C code through the PLI2.0 (VPI). Fig. 7 illustrates the relation of callbacks and Verilog simulator. User only needs to put a system task or function in the Verilog source description. Our domain coverage tool has been built on Candence NC-Verilog5.0/Verilog-XL5.0 simulator and Synopsys VCS 7.1.1 simulator. Our tool can work smoothly with Candence’s simulator, but there is a problem when working with the VCS simulator, because the VCS7.1.1 simulator does not support statement callback and can not obtain expression’s value through the function `vpi_get()`. At present, our tool can only work with Candence simulator.

![Diagram](image)

Fig. 7. Framework of domain coverage tool.

Our experiment was carried out under the environment of SUN Fire-v880 server, SUN OS5.8, NC-Verilog5.0 simulator and Specman Elite4.2. Test vector is generated by constrained random using Specman. DUTs used in the experiment include standard benchmark and test circuits of industrial SoC under design. The standard benchmark circuit is texas97-benchmarks-MPEG System Decoder. SoC integrate DMA controller, EMI (external memory interface), LCD controller, PWM, AC97 controller, RTC and an arbiter circuit.

We have compared the domain coverage with statement coverage and path coverage\(^8\). The results are shown in Table 1. It illustrates that the statement coverage and path coverage both reach sufficient level; however, the domain coverage is still low. In our project, domain coverage revealed more design faults than statement and path coverage. Fig. 8 shows the errors detected by statement coverage, path coverage and domain coverage in arbiter circuit validation approach. Although statement and path coverage reach very high level, the errors detected are just about 8 and 20, respectively. When the domain coverage reaches about 85%, 30 errors have been revealed. So we can say domain coverage is powerful to find potential design errors.

Table 1: Coverage comparisons

<table>
<thead>
<tr>
<th>DUT</th>
<th># of line</th>
<th># of path</th>
<th>Domain cov.</th>
<th>Statement cov.</th>
<th>Path cov.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUT</td>
<td># of test points</td>
<td>Cov(%)</td>
<td>Cov(%)</td>
<td>Cov(%)</td>
<td>Cov(%)</td>
</tr>
<tr>
<td>Mpeg</td>
<td>1150</td>
<td>383</td>
<td>6588</td>
<td>56.33</td>
<td>100</td>
</tr>
<tr>
<td>Dmac</td>
<td>8316</td>
<td>622</td>
<td>2565</td>
<td>57.59</td>
<td>100</td>
</tr>
<tr>
<td>Lcdc</td>
<td>6292</td>
<td>1594</td>
<td>3689</td>
<td>74.61</td>
<td>100</td>
</tr>
<tr>
<td>Pwm</td>
<td>1003</td>
<td>142</td>
<td>274</td>
<td>42.82</td>
<td>100</td>
</tr>
<tr>
<td>AC97</td>
<td>3438</td>
<td>423</td>
<td>2039</td>
<td>42.82</td>
<td>100</td>
</tr>
<tr>
<td>RTC</td>
<td>1731</td>
<td>142</td>
<td>274</td>
<td>42.82</td>
<td>100</td>
</tr>
<tr>
<td>Arbiter</td>
<td>1255</td>
<td>138</td>
<td>635</td>
<td>33.33</td>
<td>100</td>
</tr>
</tbody>
</table>
7. Results

The importance of validation continues to grow with the increase of design size. Most of commercial coverage only provides simple coverage metrics, such as statement coverage. These coverage metrics are too rough to reveal design faults. In this paper, we adopted three coverage metrics, first applying the simplest statement coverage metric, and then using the path coverage metric, last employing sophisticated domain coverage metric. Results show that the domain coverage can detect many design faults that statement and path coverage can not.

References


Xue-Xiang Wang was born in Xinjiang Province, China, in 1972. She received the M.S. degree in electronics in 2001 from the Southeast University of China. She is currently pursuing the Ph.D. degree with the Southeast University. Her research interests include SoC design and memory subsystem.

Jun Yang was born in Jiangsu Province, China, in 1976. He received the Ph.D. degree in electronics in 2002 from the Southeast University of China. He is currently a professor with the Southeast University. His research interests include SoC design and memory subsystem.