Coverage Test Technology Based on ONESPIN Verification Platform

YANG Yawen¹, ZHOU Shan¹, KONG Lu¹

¹Technology and Engineering Center for Space Utilization Chinese Academy of Sciences, Beijing, China

Abstract. Coverage test technology is a common software testing technology, which is the basic requirement of software testing. The coverage analysis can quantify the completeness of the test vector. This paper introduces and compares two kinds of mainstream coverage analysis techniques, code coverage and functional coverage. On the basis of existing theories, the working principle of ONESPIN verification platform for VHDL/Verilog/System Verilog language coverage is studied in this paper and the specific testing procedures with a practical example are presented. At last, the covered code, uncovered code and the software defect in the test results are analyzed.

1. Introduction
Along with the rapid growth of the scale and complexity of the integrated circuit design, the complexity of the verification work and the verification time increase. In the process of simulation verification, it is not workable to simulate all the input combinations, and how to balance the simulation verification time and the completeness of the test is a practical problem. Compared with the traditional random verification, currently constraint random test vector can improve the quality and quantity of verification, but it can also result in some areas in the design that are never detected, while others are always repeated. So how do you evaluate the test vectors? When does the validation work end? These questions need to be answered by coverage analysis indicators. Coverage is a necessary indicator to evaluate the quality of verification. Coverage can quantify the completeness of the simulation incentive set. Coverage can diagnose which part of the design has not been fully verified, which can help the generation of subsequent test vectors.

Currently, there are two main methods based on coverage analysis: one is the code coverage analysis, and the other one is the functional coverage analysis. In the following, the first section introduces the two main coverage analysis methods and common coverage collection tools; the second section introduces the working principle and workflow of the ONESPIN verification platform; the third section gives the example to introduce the specific test steps of the ONESPIN verification platform and analyzes the results; at last, the paper makes summary and outlook.

2. Coverage test technology

2.1. Code coverage
Code coverage testing technology originated in software testing is a kind of common white-box testing technology and its analysis object is mainly code. In the test process, the simulation tools, such as
Mentor's Questa and Synopsys's VCS, capture code coverage in the dynamic simulation process by inserting special PLI (Programming Language Interface) tasks in the HDL code to get the code coverage metrics by calculating the proportion of the executed source code, and then the simulation tools provide graphical and report files feedback on statement coverage, branch coverage, condition coverage, expression coverage, toggle coverage, finite state machine coverage [1].

In general, after the tester testing all functional points in the process of functional testing, it's hard to make sure all the source code has been ran. Code coverage testing techniques will be very effective for business scenario. Code coverage can detect the redundant state, the security protection state, the wrong state, and the undetected state of the code, detect the fault and incomplete conditions of the test vectors and increase the maintainability of the code.

As the design scale increases, the code coverage analysis becomes less and less efficient. It can only detect whether the code is fully executed, but cannot detect whether the software’s function is correct, so another coverage analysis method - functional coverage is introduced.

2.2. Functional coverage

Functional coverage focuses on design of architecture, is used to check design’s features, and primarily aims at the overall design’s function rather than the HDL code. Functional coverage can effectively describe the boundary logic and the combination of data and control signals, and indicate which functions have been tested. Functional coverage has become an important criterion for measuring verification work. The key of functional coverage analysis is that verification engineers manually define the verification tasks according to the DUT (Design Under Test). The definition of verification tasks determines the coverage space, and the completeness of verification tasks determines the comprehensiveness of functional coverage analysis. After defining the verification tasks, the verification engineers typically use the verification methodology to describe the verification tasks [2][3]. Currently, simulation software and verification tool can automatically identify professional hardware logic functional verification language, such as SVA (System Verilog Assertion) and PSL (Property Specification Language) and perform corresponding calculations, but as the design scale is getting bigger, the functional points are getting more, the test scenarios are getting more complex, the signal numbers as well as the possible combinations of the signals are explosive growth [4][5]. The key problems of verification are obvious as following:

- It takes a lot of manpower and time to build and debug the verification environment.
- It takes a lot of manpower and time to design test case set, and it is difficult to ensure the completeness and correctness of the test case set.
- It is difficult to assess the adequacy of the functional verification coverage, that is, the logical functional domain represented by RTL code is greater than/equal to/less than the functional test case set.

The verification platform of the Germany ONESPIN company combines the formalized verification engine with the ABV based functional verification methodology, which can effectively solve the above problems.

3. ONESPIN verification platform

3.1. General concepts and verification principle

Germany ONESPIN verification platform provides formal functional verification for RTL modules. The verification platform provides Windows/Linux versions, supports VHDL/Verilog/System Verilog language and supports SVA/PSL/ITL/TIDAL/OVL verification language library [6]. The architecture of ONESPIN verification platform is as “Figure. 1”, which has the following characteristics:

- The most technically advanced formal platform available
- High performance, capacity, optimized formal model
- High proof depth, convergence, broad range algorithms
- Automate engine application for best results and heuristic director
3.2. Verification flow
The verification platform includes three main tools: INSPECT, CERTIFY and VERIFY. ONESPIN verification can be performed in accordance with the process as “Figure. 2”.

1) Use INSPECT to find the low-level defects in the RTL code, and make a preliminary analysis of the dead code and the uncovered code.
2) Use professional hardware logic functional verification language such as SVA or PSL to accurately describe the functional and timing requirements, and then use CERTIFY to check the completeness and correctness of the functional requirement suite.
3) Use VERIFY to perform functional verification, check the consistency of the RTL code and the functional requirement, and generate detailed coverage information. Using VERIFY, it is possible to answer the following key questions of verification:
   a. How good are my assertions – which parts of the RTL code are actually covered and passed? Which parts are covered but failed? Which parts are not covered and should be targeted by additional assertions?
   b. How good are my constraints – is there some unintended over-constraining?
   c. How much of my verification plan has been completed?

All the coverage data can be exported to standard UCIS format data, and for all the problems and the errors found, VERIFY can automatically generate the counterexample, and display in a waveform diagram which linkage with code and show the process of the problems.

4. Application example
This section applies this platform to verify specific instance, analyze and statistic the results.

4.1. The design overview
The selected design is a voting system. The key signals in the design are explained in “Table 1”.

Table 1. The Key signals in the design.

| Signals | I/O   | Explanation                                              |
|---------|-------|----------------------------------------------------------|
| 1. i_clk| input | clock                                                   |
| 2. i_rst_n| input | reset, low is active                                    |
| 3. x[2:0]| input | multi-channel base signal                               |
| 4. x_q[2:0]| input | multi-channel quality signal                            |
| 5. bp[2:0]| input | multi-channel secondary signal                          |
| 6. bv[2:0]| internal | internal signal, judge which channel is selected       |
| 7. om [1:0]| output | om [1]: the quality bit output of the voting om [0]: the logic result of the voting |
4.2. Specific operation steps

Step I: The platform reads and compiles the RTL code, and then run tool - INSPECT to detect defects. The results are as “Figure. 3” and “Figure. 4”. As shown in the results, there are no low-level defects and dead code in the design.

Step II: Analyze functional requirement and use SVA to describe the requirement manually. The functional requirement and the corresponding SVA description are as shown in “Table 2” [7].

| Functional requirement | SVA |
|------------------------|-----|
| 1. Channel voting: if \( b[p][i] \) is enable and \( x_q[i] \) is unable, then \( bv[i] \) is active. | \( \text{if } b[p][i] \text{ is enable and } x_q[i] \text{ is unable, then } bv[i] \text{ is active.} \) |
| 2. Logic voting: if all bits of \( bv \) are enable and at least two bits of \( x \) are enable, then the value of \( om \) is 2'b01. | \( \text{if all bits of } bv \text{ are enable and at least two bits of } x \text{ are enable, then the value of } om \text{ is 2'b01.} \) |
| 3. Logic voting: if \( bv \) and \( x \) don’t meet the conditions in 2 above, then the value of \( om \) is 2'b00. | \( \text{if } bv \text{ and } x \text{ don’t meet the conditions in 2 above, then the value of } om \text{ is 2'b00.} \) |
| 4. Logic voting: if at least two bits of \( bv \) are enable and at least one bit of \( x \) is enable, then the value of \( om \) is 2'b01. | \( \text{if at least two bits of } bv \text{ are enable and at least one bit of } x \text{ is enable, then the value of } om \text{ is 2'b01.} \) |
| 5. Logic voting: if \( bv \) and \( x \) don’t meet the conditions in 4 above, then the value of \( om \) is 2'b00. | \( \text{if } bv \text{ and } x \text{ don’t meet the conditions in 4 above, then the value of } om \text{ is 2'b00.} \) |
| 6. Logic voting: if at least two bits of \( bv \) are unable, then the value of \( om \) is 2'b11. | \( \text{if at least two bits of } bv \text{ are unable, then the value of } om \text{ is 2'b11.} \) |

Step III: Run tool - CERTIFY to check the SVA, the result is as “Figure. 5”. As shown in the result, the SVA requirement suite is not complete. Though the waveform generated by CERTIFY, the requirement description of reset behavior and status output signal \( o \_task\_finish \) is not found in the SVA requirement suite. The supplementary functional requirement and the corresponding SVA description are as shown in “Table 3”.

Run tool - CERTIFY to check the SVA again, all seven type of check are successful, that is to say the SVA requirement suite is complete and correct. Take a type check result for example which is shown as “Figure. 6”.

Figure 3. INSPECT result for low-level defects

Figure 4. INSPECT result for dead code
Figure 5. CERTIFY result

Figure 6. CERTIFY result for contradiction

Table 3. The supplementary functional requirement and SVA.

| Functional requirement | SVA |
|------------------------|-----|
| 7. Reset: if i_rst_n is active, then the value of om is 2'b00 and the value of o_task_finish is 1'b0. |
| 8. status output signal: if i_rst_n is not active, the state of o_task_finish is same as the state of signal i_task_start. |

Step IV: Use tool - VERIFY to perform functional verification and generate coverage information. The verification results, the coverage information and the counterexample are shown as “Figure. 7”, “Figure. 8” and “Figure. 9”. In the 1st round verification, there is an inconsistency between the code and the SVA. 21.05% statements and 14.29% branches in the code has not been reached.

Figure 7. The 1st verification result

Figure 8. The 1st coverage information

Figure 9. Counterexample waveform

The designers analyze and modify the code based on the waveforms and reports provided by VERIFY, and then submit the code to the verification engineers.

Step V: Use tool - VERIFY to perform regression verification. The final verification result and the coverage information are shown as “Figure. 10” and “Figure. 11”. The results mean the code is consistent with the SVA, the code and the function domains are covered.
5. Conclusion
In this paper, an automated coverage method is proposed based on ONESPIN. Compare to traditional method, verification engineers can use ONESPIN verification platform in the coverage test process without manually construct test vectors, without manually analyze functional assertion’s quality, without manually build verification environment. The whole verification process is highly automated and highly visible, which greatly reduces the verification time and effectively solves the common problems in the coverage test.

6. References
[1] Mentor Graphics, Questa SIM Reference Manual, 2011.
[2] Martinez A, Barron R, and Molina H 2014, “Automated functional coverage directed for complex system,” VLSI-SOC.
[3] Harry Foster, “Trends in functional verification: a 2014 industry study,” DAC, 2015.
[4] Hassan Sohofi, Zainalbedin Navabi, “Assertion-Based Verification for System-Level Designs,” ISQED, 2014.
[5] Rott J S, Marriott P, and Graham M 2015, “Navigating the Functional Coverage Black Hole: Be More Effective At Functional Coverage Modeling,” In Design and Verification Conference (DVCon).
[6] OneSpin, User Manual: OneSpin 360™ Version 2016.12.
[7] OneSpin, OneSpin SystemVerilog Assertions Quick Reference Card.