Linking Multiple Verification Flows Using Automatically Generated Assertions

Jing Li
Broadcom Corporation
3151 Zanker Rd.,
San Jose, CA
jingli@broadcom.com

Nantian Qian
Broadcom Corporation
3151 Zanker Rd.,
San Jose, CA
qian@broadcom.com

Yuan Lu
Nextop Software Inc.
2900 Gordon Ave.,
Santa Clara, CA
yuan@nextopsoftware.com

ABSTRACT

More features and more bandwidth capability enabled in our new generation switch chips create a daunting task for functional verification. Our verification methodology includes a top level test environment and many block level tests for key blocks. Both rely on random stimulus to achieve significant coverage. They are typically independent effort from different groups. This paper reports a new methodology using whitebox assertions and functional cover properties automatically generated by a vendor tool (Bugscope) to link multiple verification flows. Using an Assertion Synthesis technology, Bugscope can automatically generate high quality assertions and cover properties. The main contribution of this paper is that we realize that Bugscope assertions and cover properties generated at block level testing can be used to guide further verification including chip and formal verification. Though the block level verification should cover all functionalities of a block, identifying poorly tested blocks early is an important management task in order to achieve tight schedule. We run Bugscope and generate properties based on block level tests, and simulate the properties at top level. If many cover properties are reached, it indicates that block testing is incomplete as top level tests add many more behaviors. During this process, the generated assertions which capture the block constraints are automatically verified at top level. In a case study, 102 assertions and 43 cover properties are reported by Bugscope. Among them, 35 cover propertied missed at block level are covered at top. This raises a red flag on the block. Meanwhile, one assertion fires and a bug is found.

Categories and Subject Descriptors

B.7.2 **[Integrated circuits]:** Design aids – *verification*.

General Terms

Verification

Keywords

Assertion synthesis, assertion based verification, System Verilog Assertion, constrained random simulation, formal verification.

1. INTRODUCTION

Each new generation of our enterprise switch design enables more features and more bandwidth capability. While some blocks are reused, key components typically needs re-architecture and re-design to accommodate the new features and achieve the line rate and performance requirement. The new components and their interactions with the rest of the system create a daunting task for functional verification [1][2][3][4]. On stimulus side, we do constrained random verification at both block level and top level. For certain self contained blocks, we even apply formal verification to find deep bugs. On observability side, we require block level testing to reach 100% line coverage and >95% conditional coverage. Although we always use the cutting edge methodology and tools, verification is still our biggest cost in the sense of both resource and schedule time. We realize that, with gate count exceeding tens of millions, one of the main issues in our existing verification methodology is that it alone no longer offers sufficient observability.

Assertion-Based Verification (ABV) addresses the observability problem by embedding both black-box and white-box assertions and functional coverage goals in the verification process [5]. ABV is now widely accepted as an effective approach to combat verification complexity [6], and all of our in-house verification tools now support standard assertion languages such as SystemVerilog Assertion (SVA) [7]. Based on their functionality, assertions can be divided into whitebox assertions and blackbox assertions. Whitebox assertions use internal signals and capture detailed design intent while blackbox assertions use interface signals to capture end-to-end functionality. Most blackbox assertions involve complicated calculations and long temporal events while many whitebox assertions are often short and involve fewer operations.

The main barrier of assertion-based verification proliferation has been the high effort required to create enough high quality assertions and functional coverage goals. It is desirable to have one assertion per 10 to 100 lines of RTL, but it is difficult to achieve this desired assertion density without over-burdening RTL designers. Consequently, assertions are not as widely used as they should. In our context, we found that debugging assertions is an extreme painful process. It often takes hours to make sure an assertion is completely correct before it becomes useful. Though we desire to have both whitebox and blackbox assertions in the design, it is not successful in practice to use ABV in our context.

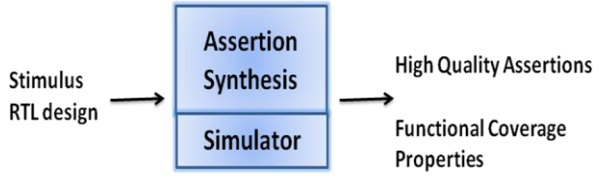

Figure 1 Assertion Synthesis

Bugscope is an EDA tool for automatically generating assertions and

functional coverage goals [9]. Using an Assertion Synthesis technology (see Figure 1), Bugscope takes the RTL design and its tests as input, and generates properties in SVA formats as output. The algorithm guarantees that the generated properties always hold true for the given set of tests. If a property is universally true, it can be classified as an assertion. Otherwise, the property must be an artifact of the tests and its negation represents a functional coverage hole. Note that both assertions and cover properties generated by Bugscope are useful to guide further verification [10].

As described above, our existing methodology involves multiple flows to achieve high quality verification, including block level testing, top level testing and formal verification. In this study, we demonstrate a new and more effective coverage driven random simulation flow. We also show how to effectively integrate multiple verification flows by using assertion synthesis technology.

2. CONSTRAINED RANDOM SIMULATION

Figure 2 illustrates the block diagram of a typical packet processing engine in a switch design. In our case, the entire switch design contains over 500K lines of RTL Verilog, the packet processor contains approximately 30K lines of RTL Verilog and the filtering block contains approximately 3K lines of Verilog.

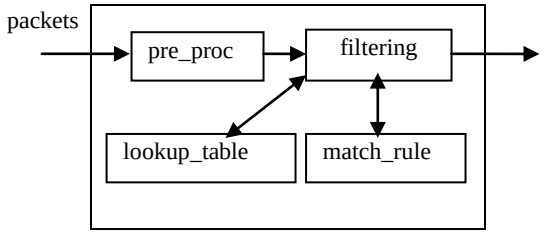

Figure 2. A Typical Packet Processing Engine

The test environment contains both block level tests for the packet processor and top level tests for the entire switch design. At block level, there are over 2000 direct and random tests in the regression and it takes 10 hours to finish them.

With the 2000 tests, 100% line coverage and >95% conditional coverage have been achieved. Based on this set of regression test, Bugscope generates 145 properties for the filtering block, which is about 5% of the RTL line count. Among them, 43 are classified as coverage goals and 102 are classified as assertions. Some of the properties are listed in Table 1 as examples.

Table 1 Bugscope Property Examples

assert !(buffer_empty && filter_fifo_rd)
assert {pkt_valid, sop} != 0 |-> @pkt_length != pkt_length
assert onehot0({key_pkt, bypass_pkt, invalid_tag})
cover (eop && state != DATA1)
cover (multicast_pkt && cur_multicast_pkt)

The first assertion says no read while buffer empty, i.e. no fifo underflow. In a typical SVA context, this assertion will be written as follows.

```
property not_underflow;
   @( posedge clk) disable iff ( !rst_n)
      !(buffer_empty && filter_fifo_rd)
endproperty : not_underflow
assert_not_underflow : assert property (not_underflow) else
begin
   $display ("ERROR: buffer is underflowed");
end
```

In order for human to read easily, Bugscope outputs the properties in the format listed in Table 1 instead of lengthy executable SVA format. After the properties are classified, the final executable assertions or cover properties will be outputted in SVA format. In Bugscope's shorthand notation, there are two new operators introduced besides Verilog operators: |-> is the SVA implication and @ is Bugscope's next state operator. For example, @packet_length denotes packet_length's value at next cycle and @packet_length == packet_length denotes the fact that packet_length doesn't change its value for two cycles.

The second assertion says in a valid packet, the packet length must be updated. The third assertion implies that key packet, bypass packet and invalid tag must be mutually exclusive. Note that onehot0() has the same meaning with SVA's system function $onehot0(). The fourth coverage property says the internal finite state machine is never in DATA1 at the end of a packet while the last coverage property says that we never send back-to-back multicast packets into the block. Note that the two coverage holes denoted by the last two coverage properties are not detected with either code coverage or conditional coverage.

With careful analysis, designer realizes that the first coverage property (cover (eop && state != DATA1)) listed in Table 1 exposes a functional coverage hole that points to a case where a sequence of undersized packet has not tested. Then we hook up all the assertions and coverage properties to the top level random environment. The FIFO underflow assertion *!(buffer_empty && filter_fifo_rd)* is triggered after a top level random test is added to patch the coverage hole, and points out a new bug in the filtering RTL. Interestingly, the top level checker does not fire for the added test because the error condition does not propagate to the output in this particular random test. As a matter of fact, it is extremely difficult for top level random tests to propagate this bug to the checker output. Without the assertion and the cover property, the bug is very likely to slip through our verification process in both block level testing as well as top level testing.

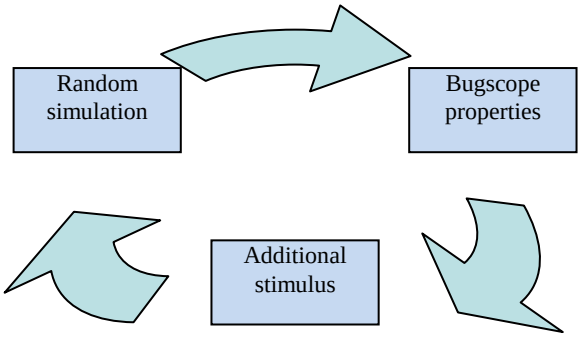

Figure 3. Coverage Driven Simulation Flow

Two key concerns in a constrained random simulation flow are

addressed by using Bugscope. One is to detect/mitigate the risks of deficiencies in a top level checker. Checkers that mask errors can lead to serious bugs that are not even corner case. White-box assertions complement checkers. They improve the observability of bugs. In addition, they reduce our debug turn-around time. The other key concern is how to measure the quality of random stimulus. Structural code coverage is insufficient because the result heavily depends on RTL syntax. Instead, white-box functional coverage goals by Bugscope can be used to drive test stimulus development. We propose a new coverage driven flow based on Bugscope (see Figure 3). We use Bugscope to find whitebox functional coverage holes in our random environment. Then we add stimulus to address these coverage holes. Note that the process is similar to traditional coverage driven verification. The only difference is that we have a better coverage metrics than structural coverage metrics. Consequently, we enhance the traditional coverage driven methodology by adding additional whitebox cover properties (see Figure 3). Note that this enhanced methodology doesn't require any changes in the existing flow except adding Bugscope's cover properties and assertions.

3. LINKING BLOCK AND TOP LEVEL SIMULATION

Our current verification methodology includes a top level test environment and many block level tests for key blocks. Both top level and block level rely on random stimulus to achieve significant coverage. They are typically owned by different engineers and often from different groups. Due to historical reasons, our top level random environment is more mature and stable comparing with block level testbenches which are often ad hoc and owned by private engineers.

The block level interface offers more controllability, and ideally block level tests should cover all functionalities of a block. From a management perspective, identifying poorly tested blocks or poorly developed block level tests early is one of the most important tasks in order to meet tight development schedule. Unfortunately, RTL verification success is measured by effort spent on worst case instead of average case. One poor verified block will delay the schedule no matter how good the other blocks are verified. To our knowledge, there is no good metrics to identify bad verification practice early. For example, structural code coverage cannot tell such information as all of our block level tests must hit high code coverage. Otherwise, they won't be integrated into chip level. In our previous practice, we highly depend on individual engineer's experience and expertise. In other words, our approach is subjective and therefore sometimes fails to find the "black hole".

At the same time, block assumptions and constraints are the area which typically introduces difficult bugs. It is very important to validate these assumptions and constraints made by block testing at top level. As a matter of fact, it is top level verification primary task to validate such assumptions and constraints. In our traditional methodology, only RTL is integrated into top level testing. No information about block level testing is captured. In other words, top level testing is totally independent from block level effort.

There are two requirements for top level testing to capture interface constraint bugs: 1) the top level test must activate the bug; 2) the top level test must propagate the bug to the checker boundary. Propagation is often difficult due to top level's complexity. This becomes extremely difficult in our switch designs because the switch by nature allow to discarding packets. A packet which activates a bug can be easily trapped and dropped later before reaching checker boundary. Therefore, to our knowledge, there is no good approach to address this problem.

In this paper, we would like to introduce a new methodology which addresses both issues using Bugscope. The idea works as follows.

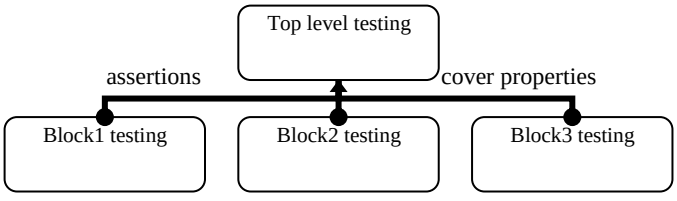

Figure 4. Use Block level properties at top level

Run Bugscope and generate properties based on total block level test suit, and simulate both assertions and the cover properties in top level environment (see Figure 4). In this case, the properties capture the dynamic behaviors of block level tests. If few of the cover properties are violated, it indicates that block testing is complete relative to top level testing. If many properties are violated, it indicates that block testing is incomplete as top level tests add many more behaviors. For example, in the above packet processing engine block, we find that 35 out of 43 cover propertied reported by Bugscope are covered at top level. At the same time, assertions obtained from block level can be used as internal monitor to catch interface constraint bugs. Note that these assertions are extracted based on block level testing. So they are only true with respect to the block level testing. When block level constraints are violated, these assertions may fire and find bugs directly. Based on our experience, we propose a bottom-up simulation methodology as follows.

1. Run Bugscope whenever tests for a block is nearly complete
   - Ahead of code coverage setup
   - Allow designer time to classify assertions and coverage properties
2. Add assertions and coverage to top level
   - Check block level assumptions
   - Test quality evaluation for managers
3. Improve block tests to patch coverage holes
4. Repeat Step 1-3 until coverage converges

Conversely, run Bugscope and generate both assertions and cover properties based on top level tests, and simulate the properties in block level tests. In this case, the properties capture the behaviors of top level tests. If few cover properties of a block are covered at block level, it indicates that the block testing is incomplete as it adds few behaviors beyond top level testing. If many properties of many blocks are covered at block level, it indicates good block level testing and a possible need to enhance top level testing. At the same time, the assertions generated from top level often capture the real assumptions when blocks are integrated. These assertions must be followed at block level testing. Any trigger of these assertions may directly indicate incorrect understanding of specification and therefore find bugs in testbenches. We call this a top-down simulation methodology.

Both bottom-up and top-down approaches using Bugscope is feasible and useful. Which approach to be used mainly depends on which effort finishes first. In most cases, block level testing finishes before top level testing. Therefore, we typically use bottom-up approach. However, there are cases that block level effort is not planned initially. If top level testing finds a lot of issues in some blocks, we

often decide to engage a new block level testing to ensure its quality. Then we can use the above top-down approach.

4. LINKING SIMULATION AND FORMAL FLOWS

Running formal [11] and semi-formal verification [12][13] typically requires intensive engineering effort. The two most time consuming tasks are defining interface constraints and writing formal properties. We find that careful planning of constraints and deciding which properties to verify is often critical to using formal verification tools successful or not.

We use assertions and coverage goals generated by Bugscope as properties to drive formal engines. These properties are white-box properties involving interface as well as internal design signals. They typically have a much smaller cone of influence than end-to-end properties using only interface signals. As a result, the white-box properties are often easier for formal engine to converge.

In a typical formal verification setup, we would have focused on the 3K line filter block due to formal capacity limitation. Using automatically generated white-box properties, we are able to move to a higher level of abstraction and use the whole packet processing engine instead (see Figure 5). Note that the whole packet processing block includes 30K lines of RTL and several large memories which make it difficult for formal tools to converge for any end-to-end assertions. Therefore, we decide not to apply formal verification to this block initially. However, because Bugscope properties are whitebox and often involve smaller cone of logic, it is much easier to converge even with current formal tools.

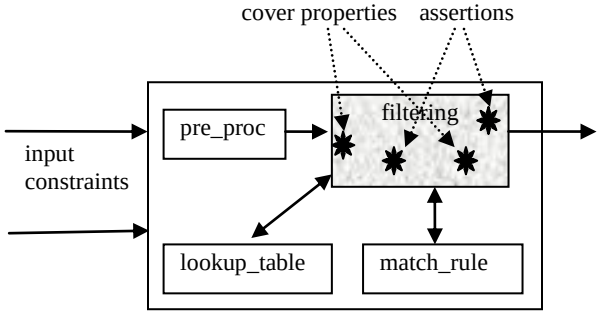

Figure 5. A New Formal Verification Approach

The benefit of moving the formal verification boundary higher is obvious: the interface constraints at packet processing block are much less painful to develop than those for the filter block. There are only two main input buses to be modeled formally while the internal filtering block talk with a few other blocks and they need to sync up in order to behave correctly. If we write constraints at filtering block level, the effort will be prohibitly large. On the other hand, the two input buses are totally independent. Either of them follows a simple protocol in which data coming at different cycles are virtually independent. It takes less than a day to setup the constraints for the packet processing engine while it may take a week to model constraints for the filter block. Intuitively, we use other blocks in the whole packet processing block as default constraints instead of manually abstracting them and write the constraints. The key benefit of this methodology is its low bar of investment and fast and high return.

Given the coverage goal and assertion previously mentioned, a formal tool has proved 68 of the 102 assertions and covered all 43 coverage properties. In addition, it found a counterexample of an assertion that pointed to the bug.

5. FUTURE WORK

Testbench acceleration [14] is becoming more and more popular. Cadence Palladium [15], Mentor Graphics Veloce [16] and Eve's ZeBu [17] all provide such capability. Though it has superior performance benefit comparing with the traditional simulation based approach, it still cannot replace simulation in most verification context. One primary reason is that simulation is debug friendly and easier to converge because of various mature debugging and coverage tools. In order for testbench acceleration tools to become a signoff approach, it is very important for them to have a notion of coverage. Note that such coverage must be synthesizable in order for accelerators to accept. Traditional line coverage or conditional coverage cannot be applied directly. In contrast, the automatically generated assertions and coverage properties by Bugscope can be applied in accelerators.

Bugscope can output both synthesizable and nonsynthesizable properties. The tool understands the definition of synthesizable properties and can output them according to user's requests. Given the latest testbench accelerators start to support SVA assertions, Bugscope assertions can be integrated into testbench accelerators. In our future work, we will investigate various areas in testbench acceleration by using Bugscope's automated assertions and cover properties:

- Use Bugscope functional cover properties as coverage signoff for testbench accelerators. The goal is for testbench accelerator to cover all the missing whitebox cover properties from block level tests;
- Use Bugscope assertions as an extra monitor to patch checker holes missed by our end-to-end checkers;

6. CONCLUSION

There are two contributions of this paper. First, we propose a new coverage driven verification methodology based on automatically generated assertions and cover properties by assertion synthesis technology. Second, the main contribution of this paper is that we realize that Bugscope assertions and cover properties generated at block level testing can be used to guide further verification including chip and formal verification. The new methodology allows us to uncover corner cases bug and identify functional coverage holes. By reusing the assertions and coverage holes across multiple verification flows, the methodology allows us to measure and leverage the quality of different test environments. Such an integrated verification platform is critical for verifying complex SoCs.

7. REFERENCES

[1] P. Mishra, and N. D. Dut, *Functional Verification o Programmable Embedded Architectures, A Top-down Approach*, Springer, USA 2005.

[2] Andreas Meyer, *Principles of Functional Verification*, Newnes Publishers, 2005.

[3] Stuart Sutherland, *Adding Last-minute Assertions to a Design and Verification Project: the Good, the Bad and "Would I Do It Again?"*, DVCon 2009.
[4] Ashish Chandra, Subir Roy, G. Sheshadri, *A Novel Approach to Complex Interrupt Controller Verification*, Design Automation Conference 2010.
[5] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design, 2nd ed*. Kluwer Academic Publishers, 2004.
[6] Ping Yeung, *Assertion-Based Verification of ARM Core-Based Designs*, Design Strategies and Methodologies, Vol.3, No.5, 2004.
[7] IEEE ***Std 1800-2005,*** IEEE Computer Society, 2005.
[8] IEEE ***Std 1850-2005***, IEEE Computer Society, 2005.
[9] Nextop Software Inc. http://www.nextopsoftware.com.
[10] P. Chatterjee, S. Godil, P. Nelson, Y. Lu, *Utilizing Assertion Synthesis to Achieve An Automated Assertion-Based Verification Methodology for Complex Graphics Chip Designs*, Design Automation Conference, 2010.
[11] Edmund M. Clarke, Jr., Orna Grumberg and Doron A. Peled, *Model Checking*, MIT Press, 1999.
[12] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Yunshan Zhu, *Symbolic Model Checking without BDDs*, TACAS 1999.
[13] Synopsys, *Magellan – Hybrid RTL Formal Verification*, http://www.synopsys.com/TOOLS/VERIFICATION/FUNCTIONALVERIFICATION/Pages/Magellan.aspx
[14] Shabtay Matalon, Leonard Drucker, Maya Bar, Michael Stellfox, *Building Transaction-Based Acceleration Regression Environment using Plan-Driven Verification Approach*, http://www.cdnusers.org/community/incisive/Vtp_dvcon2007_tbaregression.pdf
[15] Cadence Design System, *Incisive Enterprise Palladium Series with Incisive XE Software*, http://www.cadence.com
[16] Mentor Graphics, *Mentor Graphics Veloce Delivers 400X Acceleration for OVM Driven Verification*, Whitepaper, http://www.mentor.com
[17] Eve Emulation & Verification Engineering, *Next Generation System Validation Using Transactors*, Whitepaper, http://www.eve-team.com