Closing Functional and Structural Coverage on RTL Generated by High-Level Synthesis

B. Bowyer
Calypto Design Systems, Inc.
8005 S.W. Boeckman Rd.
Wilsonville, OR 97070

***Abstract*- Most hardware design teams have a verification methodology that requires a deep understanding of the RTL to reach their verification goals, but this type of methodology is difficult to apply to the machine generated RTL from High-level Synthesis (HLS). This paper describes innovative techniques to use with existing methodologies, for example the Universal Verification Methodology (UVM), to close functional and structural coverage on HLS generated code.**

Assuming tests that give 100% functional and structural coverage on synthesizable C++ or SystemC, most verification engineers expect the same tests to give similar coverage on RTL generated by HLS. These tests give 100% functional coverage on the generated RTL, but only give about 80% structural coverage. As HLS runs, it adds states to the FSM, stall states, pipeline ramp-up and ramp-down states, uses one-hot encoded logic and may structure logic in a way that artificially lowers reported coverage. All of these gaps need to be addressed to get near to 100% coverage on the generated RTL.

A set of C++ coding styles and verification technique is proposed for each of these coverage gaps, with a focus on tools and techniques that do not require a deep understanding of the generated RTL. The source code also needs to be designed with well synchronized blocks to simplify stall testing and labels to help analyze FSM coverage.

A Discrete Cosine Transform (DCT) used in the decoder for the High Efficiency Video Codec (HEVC) is used to illustrate the techniques. The DCT is developed in C++ and synthesized into Verilog using Catapult. The C++ source has tests that cover the behavior and give 100% coverage on line, branch and condition coverage as measured by gcov. Initial structural coverage is measured on generated Verilog code using Questa. Questa CoverCheck is used to exclude unreachable conditions and then each technique is applied to show how coverage improves. The final result is near 100% line, branch and Focused Expression Check (FEC) coverage and analysis that shows the remaining coverage bins are unreachable.

ACKNOWLEDGEMENTS

The author wishes to acknowledge Bala Sethuraman for his work to structure RTL to simplify the verification flow, Vishal Sinhga for providing an interesting design and Jim Li for the long hours researching how to manually close coverage on machine generated RTL.

I. INTRODUCTION

The most common goal for using High Level Synthesis (HLS) is to reduce the effort needed to verify hardware[1]. The verification effort is reduced by shifting design and verification from a primarily parallel register transfer model, written in VHDL or Verilog, to a more abstract procedural model. This model is called a High-Level Model (HLM) and is written using Transaction Level SystemC or a C++. The new model contains less detail and often simulates hundreds of times faster than an equivalent model written in synthesizable VHDL or Verilog.

UVM verification techniques, such as coverage points and constrained random simulation, are then applied to the HLM. The faster simulation requires fewer computational resources during testcase development, and a set of test vectors that cover the HLM are recorded so they can be applied to the RTL generated by HLS. An excellent overview of the HLS tools and process is described in [2]. In addition, HLS tools generate a SystemC wrapper and verification agents around the generated RTL that provides the same interface as the original source code, which allows the original test stimulus to be run on the RTL.

The same simulation vectors are then applied to the generated RTL and the functional coverage should be the same 100%, but the structural coverage, defined by FSM, line, branch and Focused Expression Check (FEC) coverage[3], is often disappointing low, often 60-80%. The design and verification teams are faced with the challenge of understanding why the same simulation vectors would generate such different coverage numbers and what to do to fix the problem.

The design techniques in this paper focus on blocking interfaces, which are interfaces that cause the associated process to stall and wait for data to be read or written. Blocking interfaces can be tested in a test environment where each process is randomly stalled. Non-blocking interfaces allow the associated process to continue even if data cannot be read or written, which means those processes can produce different results depending on when data can be read and written. Non-blocking interfaces therefore require a systematic approach to stall to avoid simulation differences between the mostly untimed C++ code and the generated RTL. Detailed description of various communication implementations and their implication for testing can be found in [4].

This paper gives an overview of several classes of coverage gaps with specific examples of each and techniques to close these gaps:

- Key differences in how C++ based coverage tools and RTL based coverage tools report coverage
- Coverage gaps that are only exposed after HLS has optimized the design
- Redundancies added by HLS that result in unreachable conditions in the RTL

II. DEFINING COVERAGE IN THE HLS CONTEXT

Functional coverage is a set of criteria defined by the development team for a design. These criteria may be dependent on the structure of the design. For example, the criteria might include testing that all of the elements in a shift register are assigned a non-zero value. These functional elements in a design should not be changed by HLS optimizations or transformations.

Structural coverage is related to how the code is written. Depending on the architecture, the code could be written with different control statements, function calls, loops and other structures. More importantly, HLS optimizations and transformations will sometimes change the structure of the design. So, to compare coverage between the HLM and RTL, this paper uses response to stimuli to compare coverage between the two models.

The C++ language has several constructs with undefined behavior, such as uninitialized variables, out-of-bounds array accesses and shifting by a negative number, so the first step in coverage testing is to confirm the C++ and RTL models give the same response to the functional coverage stimuli. HLS adds new coverage points related to stall, new states and new transitions. To cover the new code generated by HLS, new tests are needed for every FSM state in the RTL to test that the design will both reset and stall properly from that state.

This paper will discuss how to add the new stall and reset tests in section V and how to apply stimuli consistently between the models in section III.

However, when running RTL coverage with this approach, this paper will show that the reported coverage from the RTL tools can be much lower than the reported coverage from the C++ tools. The causes of this difference are then shown to be due to:

1. Differences in the structural reporting between the tools
2. Transformations during HLS that would cause either the C++ or RTL coverage tools to report missing coverage
3. Unreachable combinations of conditions

This paper walks through common sources of these differences between C++ and RTL coverage and approaches to solve them. Further research is needed to determine how well achieving coverage on the C++ code catches bugs and which coverage differences can be safely ignored. Due to the lack of this information the proposed methodology in this paper assumes resolving any coverage difference has the potential to catch a new bug.

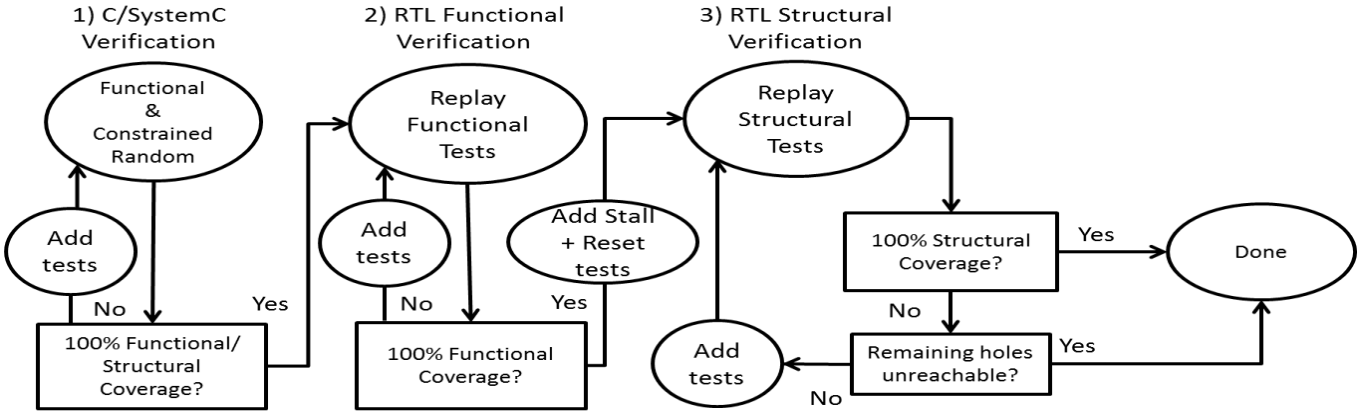

Figure 1. Verification flow for testing RTL generated by HLS

III. TESTING METHODOLOGY

A. Flow Overview

The general flow used to complete functional and structural coverage in RTL is show in Figure 1. This breaks the process into three stages. The first stage is to run directed and constrained random tests until the C coverage tool reports 100% function, line, branch and condition/decision coverage. After the HLM has been synthesized, Step 2 is to run functional verification and then, once functional verification is done, to add reset and stall tests. This process is simplified because HLS will generate adaptors to allow the generated RTL to either run in parallel with or replace the original HLM in the verification environment, depending on how the original test environment was configured.

Finally, once functional, reset and stall tests have been developed, the entire test suite is run on the generated RTL to measure structural coverage.

B. Assumptions, Limitations and Designs Used

This paper does not discuss how the original functional and structural tests are developed. This could be using a UVM-like methodology with constrained random, a graph based methodology [5] or any other technique. As long as the test environment has the correct stimulus the details of this step do not matter for the rest of the methodology to work.

Blocking interfaces allow the test environment to be unaware of the internal design timing. More complex agents that understand the overall timing of the system can be used if there are non-blocking interfaces that require specific timing. The use of blocking interfaces also allows the verification team to separate the testing of functionality from performance testing. This paper does not describe how to do performance testing.

The CatWare blocks[6], which include several different filters and FFTs are used to gather data to compare functional coverage between the source and RTL. An inverse DCT, which is part of the H.265 decoder and has more complex control, is used to compare structural coverage.

IV. FUNCTIONAL COVERAGE

Functional coverage is defined by the test environment and would be the same for any equivalent design run in that test environment. This simplifies functional coverage testing because the design team can assume that their stimuli will achieve 100% functional coverage when applied to the RTL. The key testing in this step is to confirm that the RTL behaves exactly the same as the HLM. Experimentally, fifteen different FFT and filter architectures in the CatWare blocks have been run through the flow in Figure 1, and no configuration showed a difference in the functional coverage between the C++, SystemC and RTL.

Common issues that occur during this testing are related to undefined behavior in the source code. For example, For example, the C++ standard does not define the behavior of uninitialized variables, negative shifts and out-of-bounds array access. The designer should code defensively, using class libraries that avoid common problems, such as AC Datatypes[7] and initializing all variables in the HLM. The developer must also understand and resolve any differences between the HLM and the RTL model before moving on to the next step.

V. STALL AND RESET STRUCTURAL COVERAGE

A) Stall Coverage

The approach for stall coverage is to identify all of the additional states that have been added by the HLS tool and then target tests at these states. This process is dependent on what kind of stall architecture is added by the tool, so the details in this section are specific to Catapult and HLMs that use transaction level interfaces, which do not describe the exact hardware for implementing stall.

First, the verification engineer needs to understand the stall architecture in the generated RTL. For Catapult, a small state machine is built for each IO, and this state is then combined in a “staller” block, which disables flops and the FSM in its associated thread. The stall architecture built by Catapult has the advantage that a test can target each IO separately and still reach 100% structural coverage.

Interfaces can be arbitrarily complex, which affects the stimulus needed, but the concept is the same for all interfaces, so a simple example is used. Assume an input “DIN” to a thread has two control signals, ready and valid, plus a data signal. For DIN, the stall combinations that need to be tested are shown in Table 1. When both ready and valid are high, data is read by the design. If only ready or valid is high, then either the reader or the writer is

stalled. Finally, DIN itself may need to be stalled if it is trying to read and the thread that owns DIN is stalled, usually because an output is blocked, and the thread cannot write its own data.

These stall states can be difficult to reach so Catapult allows an optional stall pin to be added to each thread. This pin is used to achieve coverage and can also be used to stress test a sub-system by randomly stalling blocks.

Table 1. Control pin combinations required to achieve stall coverage.

Ready	Valid	Stall Pin	Covered State
0	0	0	DIN is not active
0	1	0	DIN is waiting and stalls thread
1	1	0	DIN is active
1	1	1	DIN is stalled while active

B) Reset Coverage

Reset coverage simply requires putting the FSM into every state and then asserting reset. The functional coverage tests should cause the FSM to traverse every state, but some additional test might need to be used from the structural coverage suite.

V. OTHER STRUCTURAL COVERAGE

A. Unreachable Code

The most common reason for missing structural coverage is that the missing coverage bin is not reachable for any set of inputs. The designer should first assume the code is unreachable. The first step is to run an unreachability check using a formal tool. For the experiments for this paper, about one line in 1000 is not covered in the RTL and Questa CoverCheck is used and it generally proves about 50% of the unreachable lines are not reachable.

The remaining lines need to be analyzed by hand. This might seem a difficult task on machine generated code, but remember the goal is only to trace the logic back to prove it has unreachable conditions. In practice, only about 30 minutes were needed to trace back a redundant line of code.

The analysis of several designs uncovered a set of redundancies that are present in the source code. The three most common cases are discussed below.

1) When writing loops, the all paths in, out and over the loop should be reachable. Consider the following loop where the input `dynamic_bound` controls the number of loop iterations.

```
for ( int iter = 0; iter < dynamic_bound ; iter++ ) {
  /* loop body */
}
```

The `for` loop should only be written in the above style if all values of `dynamic_bound` are exercisable during simulation. Assume the minimum value for `dynamic_bound` is four. The path that skips the loop is only exercised if `dynamic_bound` is zero. The missing path is not caught by C++ coverage tools, but the RTL code will have coverage gaps in the loop control logic. The code can be re-written to move the break condition to the end and guarantee that the loop body is always executed.

```
for ( ac_int<N,false> iter = 0; /*intentionally left blank*/ ; iter++ ) {
  /* loop body */
  if (iter >= ac_int<N,false>(dynamic_bound - 1) ) break;
}
```

The `int` datatype is replaced with a N bit AC integer datatype. The second template parameter determines if the datatype is signed, so this datatype is unsigned. The coding style above guarantees that the loop iterates at least one time. Casting the right-hand side of the break condition to `ac_int<N,false>` avoids a potential coverage gap related to logic generated in case the right-hand side becoming negative. Using this coding style should eliminate any redundancies related to the loop having a dynamic bound.

2) During HLS, the if/else structures in the source code are converted into mux trees. Redundancies present in the relationship between different conditions in the source may not be caught by gcov, but will be present in the condition coverage in the RTL. Catapult is able to optimize the mux tree in most cases to remove the redundancy, so there is no coding style to follow. In some cases, such as the example below, there will not be an obvious coding style change and the coverage hole should be waived.

```
data_to_sat = condition ? input.read() : 0;
if ( data_to_sat > N )
  data_to_sat = N;
if ( condition )
  out.write(data_to_sat);
```

In this case, a single mux could be constructed to assign out to "N", "0" or "input". However, the constant value "0" is never assigned to out. In many cases, Catapult would optimize away this input to the mux, but the optimization may not work in all cases. In practice, most cases where this occurs involve a constant conditionally assigned to a variable.

3) A condition inside of a loop that depends on the loop iterator can lead to redundancies. Catapult adds FSM states, visible as "fsm_output" in the RTL, and pipeline states, visible as "stage_var" in the RTL, to the design. The code below will cause redundancies because the `Internal:if` statement is only true in the first iteration of the loop:

```
for ( ac_int<N,false> iter = 0; /*intentionally left blank*/ ; iter++ ) {
  Internal:if ( iter == 0 ) {
    /*condition body*/
  }
  /* loop body */
  if (iter >= ac_int<N,false>(dynamic_bound - 1) ) break;
}
```

The `Internal:if` statement is only true in the first iteration of the loop. The FSM transition into the loop also captures the same information. When the `Internal:if` statement is scheduled at the start of the loop, Catapult would generate logic to confirm both that the `Internal:if` statement is true and that the design is in the correct FSM state:

```
mux_ctl = (iter==0) & (FSM_state == state_before_loop);
```

Both of these expressions will always become true at the same time, leading to holes in the condition coverage. This must be waived by hand based on finding a hole that involves and FSM state and loop iterator, crossprobing in Catapult back to the source to confirm the location of the `Internal:if` statement. When the design is pipelined, the `state_var` is used to control which stages of the pipeline are active. The `Internal:if` statement in a pipelined loop would also be ANDed with the appropriate pipeline stage. In this example, the `Internal:if` statement can be moved out of the loop body to remove the redundancy.

```
Internal{
    /*condition body*/
}
for ( ac_int<N,false> iter = 0; /*intentionally left blank*/ ; iter++ ) {
  /* loop body */
  if (iter >= ac_int<N,false>(dynamic_bound - 1) ) break;
}
```

B. Missing Stimulus

HLS performs two transformations that affect how the coverage tools view the design, loop unrolling and function inlining. These transformations duplicate code, which changes how the coverage tools view the code. Consider the following example:

```
for ( int i =0; i < 3; i++ ) {
  if ( A[i] ) {

  }
}
```

In source simulation, the "if" statement is covered when A[0] or A[1] or A[2] is true and false. However, after the loop is unrolled, there are three separate "if" statements, one for each iteration of the loop. This means that A[0], A[1] and A[2] all must be both true and false while the loop is running to cover all the "if" branches for all loop iterations.

When the same changes were performed on the C code by hand, gcov also flagged the same missing coverage. This implies that a C++ coverage tool could be enhanced to find these cases. For now, these cases must be thought of by the verification engineer as part of a functional coverage methodology or by analyzing the coverage holes in

the RTL. In this case, the presence of an unrolled loop or function that is inlined multiple times can be identified by looking for messages from Catapult.

VI. RESULTS

The methodology described in this paper is applied to the DCT used in an HEVC decoder. The source code is written using 490 lines of C++, which follow the coding styles described in this paper. A test environment is developed using directed tests and constrained random data to achieve 100% coverage as reported by gcov.

Next the code is synthesized with Catapult to generate 15735 lines of Verilog. Many of the mathematical expressions take multiple lines, which is why the line count is only 3801 in the data below. After applying the C++ stimulus to the RTL, there are 279 holes, or about 94% coverage. The coverage tests for the library components that ship with Catapult are applied to the code to "gray box" the components and exclude them from the coverage data. Next CoverCheck is run to identify unreachable coverage bins. The result of these steps is to reduce the number of coverage holes to 107, or about 98% coverage.

The next step is to add stall and reset testing and to manually evaluate the remaining coverage holes. Stall testing is done in parallel with manual evaluation because some holes could be related to stall. After all of the new tests are added, only 25 coverage holes remain, or about 99.5% coverage.

Finally, the remaining 25 coverage holes are manually waived. Eleven of these holes are simple sequential redundancies and would be good targets for future optimization and or formal unreachability tools. The remaining holes are related to the relationship with C++ state and the pipeline state variables added by Catapult.

Table 2. – Example progress on code coverage

		Coverage Holes			
Type	Total Bins	Initial	Gray box libs	CoverCheck	Stall and Reset
Line	3801	75	10	2	0
Branch	473	48	8	2	0
FEC Expression	693	148	131	99	25
FEC Condition	21	8	4	4	0
Total	4988	279	153	107	25

VII. CONCLUSION

This paper shows a consistent and repeatable process for closing coverage on RTL generated by HLS. The HLM is developed following a set of coding guidelines to avoid coverage problems in the generated RTL. Functional coverage test stimulus is developed and then additional stimulus is added to achieve structural coverage in C++ or SystemC. Stall and reset tests are added to the HLM stimulus, and used to measure the functional and structural coverage in the RTL. The initial RTL coverage is refined by using unreachability tools like CoverCheck. If required, the remaining coverage holes can be closed by hand. By following this process, a verification engineer can consistently close coverage on RTL generated by HLS.

REFERENCES

[1] S. McCloud, "High Level synthesis Report 2011," pp. 3

[2] Coussy, P.; Gajski, D. D.; Meredith, M.; Takach, A. (2009). "An Introduction to High-Level Synthesis". IEEE Design & Test of Computers 26 (4): 8–17. doi:10.1109/MDT.2009.69

[3] R. Salemi, "What the Heck if FEC?," eejournal.com, Sept, 2011

[4] B. Bailey, G. Martin, A. Piziali, *ESL Design and Verification*, pp. 248-254, 2007.

[5] M. Andrews, B. Hristov, "Portable Stimulus Models for C/SystemC, UVM and Emulation," DVCon 2015

[6] Calypto, "Catware Library datasheet," calypto.com

[7] M. Fingeroff, *High-level Synthesis Blue Book*, pp. 35-58, 2010