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

Bryan Bowyer
Calypto Design Systems, 8005 SW Boeckman Rd., Bldg. E, Wilsonville, OR 97070

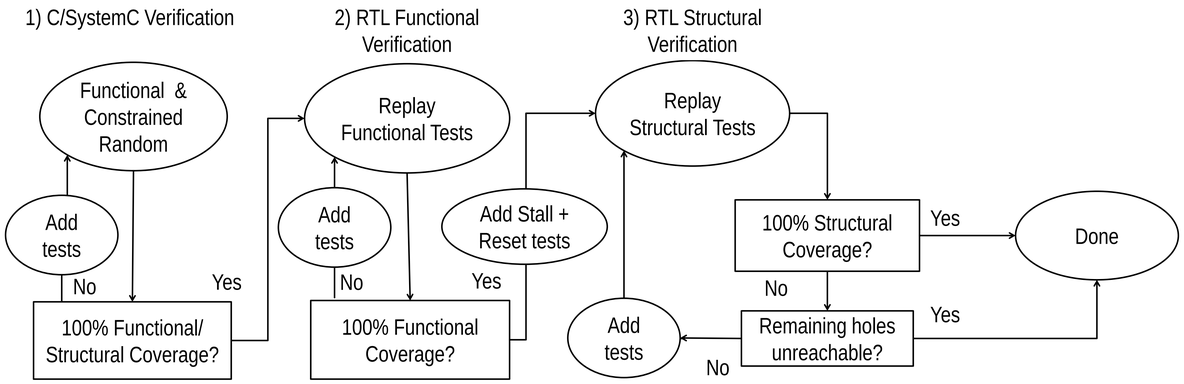

Figure 1. Verification flow for testing RTL generated by HLS

Introduction

The most common goal for using HLS is to reduce hardware verification effort. Verification effort is reduced by applying UVM techniques to test a High-Level Model (HLM) written in synthesizable C++ or SystemC, which allows verification development and debug to be done using faster simulation. The code is synthesized with HLS to generate RTL. The generated RTL is tested by replaying the original tests and then applying techniques to close coverage.

This paper describes a methodology to close the RTL coverage gaps found when replaying the coverage tests. Common causes of RTL coverage gaps are:

- Differences between C++ and RTL coverage tools
- C++ coding style
- Stall and reset behavior only testable in the RTL
- Coverage gaps exposed by HLS optimizations
- Redundancies added to the RTL by HLS optimizations

Coverage gaps can be closed by using coding style, formal unreachability tools, additional tests or manual inspection of the RTL code.

Defining Coverage in the HLS Context

Two types of coverage, functional and structural, are used to confirm that the source code is well tested. Functional coverage is defined by the development team and includes all of the design elements they decide are important to tests. Structural coverage is related to how the code is written and is determined by running tools.

The structural coverage is more difficult to manage because the structure of the generated RTL is different than the structure of the HLM. Designers must consider if their source code contains constructs that have undefined behavior, like uninitialized variables or other constructs that will be reported differently by C++ vs. RTL coverage tools.

Testing Methodology

The overview of the testing flow is shown in Figure 1. In step 1) C/SystemC Verification, coverage is achieved on the HLM. During step 2) RTL Functional Verification, the original functional tests are replayed and additional functional tests are added. Next reset and stall tests are added.

In step 3) RTL Structural Verification, the structural tests are replayed and two checks are used to confirm if new tests are needed. First an RTL structural coverage tool is used to find coverage holes. The coverage holes are checked for reachability either using a formal unreachability tool or by hand. Testing is finished once all remaining coverage holes are known to be unreachable.

Functional Coverage

Designs with interfaces which stop and wait to read or write data, also called blocking interfaces, should not have their functionality changed by HLS. Additional or modified functional tests should only be needed when the design does not stop to wait for data.

Undefined behavior in the HLS can lead to errors when running the functional tests. These errors are due to the C compiler and HLS making different optimization decisions and can be fixed using C++ coding styles that do not use undefined behavior.

Stall and Reset Structural Coverage

Table 1. Test combinations used 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

HLS adds states to the design during synthesis. The HLM tests do not test stalling in or resetting from these states. HLS adds a stall pin to each process to simplify stall testing. Table 1. shows how to use the stall pin to achieve coverage for an interface with a ready/valid handshake.

Unreachable Code

The most common reason for structural coverage gaps is code combinations that are unreachable. Unreachability tools can find some of these gaps, but others need to be analyzed and, if possible, fixed by hand.

Source code examples that lead to unreachable RTL:

1) Path that skips loop is never exercised. Assumes `dynamic_bound` >= 0.

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

To fix, use code that can never skip the loop.

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

2) If/else structures converted to muxes. Combinations of conditions not tested in C++, but tested in RTL.

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

Mux tree feeding “out” cannot write constant zero, from `cond = false` to `out`.

3)A condition in a loop is only true in the first or last iteration:

```
for ( ac_int<N,false> iter = 0; /* */ ; iter++ )
{
  INTERNAL:if ( iter == 0 ) {
    /*condition body*/
  }
  /* loop body */
  if (iter >= ac_int<N,false>(dynamic_bound – 1)
) break;
}
```

This can be fixed by moving the INTERNAL:if out of the loop.

```
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;
}
```

Missing Stimulus

When HLS unrolls loops or inlines functions, the C++ code is duplicated and optimized separately. The RTL coverage tools can report missing coverage for the duplicate code. The C++ coverage tools do not see this duplication, and could report the code as fully covered. Additional stimulus needs to be added to achieve coverage.

Results

The techniques in this paper are applied to hardware for the Discrete Cosine Transform (DCT) used in the HEVC decoder. The code is 490 lines of code in written C++ using the coding styles described in this paper. The test stimuli are developed and 100% structural coverage is achieved as reported by gcov. Next, the code is synthesized with Catapult and C++ tests are replayed on the RTL using Questa to check coverage.

	Total Bins	Coverage Holes			
Type		Initial	Gray box libs	Cover Check	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 Coverage Holes	4988	279	153	107	25

After replaying the C++ tests, there are 279 holes and a total of 4988 coverage bins. Catapult includes several HDL libraries for component like FIFOs that cannot be fully covered by block level tests. These libraries include their own coverage tests, allowing them to be excluded from the overall coverage and reducing the total coverage holes to 153. Next, Questa CoverCheck is run to find unreachable holes and these are excluded, to reach 107 holes. Finally, the stall and reset tests are added to reach 25 holes. These 25 holes were inspected manually to determine that they are unreachable.

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.