Using SystemVerilog Assertions for Functional Coverage

Mark Litterick, Verilab

mark.litterick@verilab.com

ABSTRACT

SystemVerilog Assertions (SVA) can be used to implement relatively complex functional coverage models under appropriate circumstances. This paper explores the issues and implementation of such a functional coverage model to demonstrate both the capabilities of SVA coverage and illustrate coding techniques which can also be applied to the more typical use of SVA coverage, which is to specify key corner cases for the RTL from the designer's detailed knowledge of the structural implementation. This paper is related to previous work published at SNUG Europe 2005 called *Utilizing Vera Functional Coverage in the Verification of a Protocol Engine for the FlexRay™ Automotive Communication System* [1]; readers are encouraged to read both papers.

1 Introduction

Most testbench environments that make use of Assertion Based Verification (ABV) methodologies would typically consider the following aspects of functional coverage:

- High-level functional coverage
  - based on specified features - the requirements for the design
  - implemented by verification engineer
  - implemented in a HLVL like Vera, 'e' or SystemVerilog

- Low-level implementation specific coverage points in RTL
  - capture the designers concept of critical corner cases for verification
  - really the designers requirements from the verification environment
  - implemented by design engineer
  - implemented in SVA

Refer to Chapter 5 of *Assertion Based Design* book [2] for a discussion on the merits of implementing low-level coverage using assertions, and how the two styles complement one another. The *SNUG Europe 2005* paper [1] presents a third alternative level of abstraction. In this case the FlexRay Protocol Specification [3] is defined at a micro-architectural level using Graphical Specification and Description Language (SDL), which enables a functional coverage model to be generated for the *specified requirements* of the RTL. It is important to note the distinction here: these are the specification requirements that are being captured in the functional coverage model and not the implementation corner cases. One possibility, presented in this paper, is implementing the functional coverage model using SystemVerilog Assertions (SVA) [4].

With reference to [1], the following features are required for the functional coverage model irrespective of whether it is implemented in SVA or an HLVL such as Vera:

- following types of coverage points are required
  - state
  - state transitions
  - conditional state transitions
  - state transition sequences
  - SDL trigger coverage
- check for illegal states and transitions relative to the specification
- coverage model should not be implementation dependant
- coding style should closely reflect the SDL, for maintenance and debugging
- ability to query coverage results from the testbench environment for closed loop constrained random stimulus generation or functional checks for directed tests

A full discussion on the role of functional coverage in the overall testbench architecture, and an analysis of the requirements for the FlexRay Protocol Engine SDL coverage, is presented in [1]. The remainder of this paper is focused on how such a coverage model can be implemented using SVA.

2 Definition of Coverage Points in SVA

2.1 State Definitions

Text macros can be used to define the state encoding, using wildcard equivalence operators if required, as shown in Figure 1. Alternative solutions using text macros for just the binary state values or using enumerated types are also possible; however subsequent sections will demonstrate that the proposed style gives a tidy overall result.

```
`define DEFAULT_CONFIG              (vState =?= 7'b000_xxxx)
`define CONFIG                      (vState =?= 7'b001_xxxx)
`define READY                       (vState =?= 7'b011_xxxx)
`define WAKEUP_LISTEN               (vState =?= 7'b010_xx01)
`define WAKEUP_SEND                 (vState =?= 7'b010_xx11)
`define WAKEUP_DETECT               (vState =?= 7'b010_xx10)
`define COLDSTART_LISTEN            (vState =?= 7'b111_0011)
`define COLDSTART_CONSISTENCY_CHECK (vState =?= 7'b111_1010)
`define INTEGRATION_LISTEN          (vState =?= 7'b111_0101)
// etc
```

Figure 1: State Definitions

2.2 State Transition Sequence Definition

The full set of legal state transitions can be defined using SVA sequence statements, some examples of which are shown in Figure 2. Note that since the `ended` method is subsequently used on instances of these sequences, the clock cannot be derived from the context and must be defined in the sequence declaration.

```
sequence seq_DC_C;
  @(posedge clk)
    `DEFAULT_CONFIG ##1
    `CONFIG;
endsequence : seq_DC_C

sequence seq_CCC_IL;
  @(posedge clk)
    `COLDSTART_CONSISTENCY_CHECK ##1
    `INTEGRATION_LISTEN;
endsequence : seq_CCC_IL

// etc
```

Figure 2: State Transition Sequence Definition

2.3 Multiple State Transition Sequence Definition

Multiple state transitions sequences are used in this application as a means of monitoring and checking higher-level protocol operation. The transitions can also be defined using SVA sequence definitions, as shown in Figure 3. The `[*1:$]` syntax in Figure 3 means that state to the left must persist for one or more clocks before the next state in the sequence is attained.

```
sequence seq_POC_CSA_CSI;
  @(posedge clk)
    `READY                              ##1
    `COLDSTART_LISTEN            [*1:$] ##1
    `INITIALIZE_SCHEDULE         [*1:$] ##1
    `INTEGRATION_COLDSTART_CHECK [*1:$] ##1
    `COLDSTART_JOIN              [*1:$] ##1
    `NORMAL_ACTIVE               ;
endsequence : seq_POC_CSA_CSI
// etc
```

Figure 3: Multiple State Transition Sequence Definition

2.4 State Coverage

This is concerned with measuring whether we reached all the states in the specification. Note that this is not the same as code coverage measuring whether or not we acquired all the states in the implementation, since the implementation may have accidentally omitted states (or state transitions). One possible implementation is shown in Figure 4.

```
state_DC  : cover property ( @(posedge clk) `DEFAULT_CONFIG);
state_CCC : cover property ( @(posedge clk) `COLDSTART_CONSISTENCY_CHECK);
// etc
```

Figure 4: State Coverage

2.5 State Transition Coverage

In order to determine if we achieved all possible legal state transitions, each of the previously defined sequences can be covered as shown in Figure 5. Other implementations are possible, for example using the `ended` method, but for simple sequences like these the coverage results are the same.

```
trans_DC_C : cover property ( seq_DC_C );
trans_C_R  : cover property ( seq_C_R );
trans_R_C  : cover property ( seq_R_C );
trans_R_WL : cover property ( seq_CCC_IL );
// etc
```

Figure 5: State Transition Coverage

2.6 Multiple State Transition Coverage

Coverage for multiple state transitions is handled in the same way as single state transitions, an example of which is shown in Figure 6.

```
trans_POC_CSA_CSI : cover property ( seq_POC_CSA_CSI );
// etc
```

Figure 6: Multiple State Transition Coverage

2.7 Conditional State Transition Coverage

One of the main functional coverage goals in [1] is to measure the conditional path transitions between the states for the Protocol Operation Controller (POC). In SDL diagrams of [3] there are multiple possible paths between some adjacent states which are conditional on the value

of protocol variables. Measuring conditional state transitions can be achieved in SVA by evaluating the conditional expression when the corresponding state transition sequence has ended, as shown in Figure 7. The cover statement names in Figure 7 and Figure 8 relate directly to specification tags in the SDL as detailed in [1].

```
F07_13_007_IL : cover property ( @(posedge clk)
                seq_CCC_IL.ended                      &&
                (zStartupNodes <= 0)                  &&
                (vCycleCounter[0] === 1)              );

F07_13_009_IL : cover property ( @(posedge clk)
                seq_CCC_IL.ended                      &&
                (zSyncCalcResult !== `WITHIN_BOUNDS)  &&
                (zStartupNodes > 0)                   &&
                (vCycleCounter[0] === 1)              );

F07_13_011_IL : cover property ( @(posedge clk)
                seq_CCC_IL.ended                      &&
                (vRemainingColdstartAttempts <= 0)    &&
                (zStartupNodes === 0)                 &&
                (vCycleCounter[0] === 0)              );

//etc
```

Figure 7: Conditional State Transition Coverage

2.8 SDL Trigger Event Coverage

Another important functional coverage requirement detailed in [1] is measuring the SDL trigger event coverage. Trigger events in the SDL can result in the POC changing state, but the trigger event will have expired before a real RTL implementation actually achieves the new state. Some of these trigger events do not cause a state change, but instead have a secondary effect, such as modifying a protocol variable. From a functional coverage point of view, we care about measuring the occurrence of all relevant triggers in the corresponding states. This can be achieved in SVA by using the `$rose()` system function for the trigger signal when we are in the appropriate state, as shown in Figure 8.

```
F07_11_003_A : cover property ( @(posedge clk)
               `COLDSTART_LISTEN && $rose(header_received_on_A));

F07_11_004_B : cover property ( @(posedge clk)
               `COLDSTART_LISTEN && $rose(symbol_decoded_on_B));

F07_11_005   : cover property ( @(posedge clk)
               `COLDSTART_LISTEN && $rose(CHIRP_on_A));
// etc
```

Figure 8: SDL Trigger Event Coverage

2.9 Detecting Illegal States and Transitions

A side effect of implementing functional coverage in [1] using Vera was that specification of illegal state and transition bins came almost for free (using the `bad_state` and `bad_trans` bin types). Implementation of this checking capability in SVA involves a little more effort as shown in Figure 9 and Figure 10. Some would question whether such a checking operation is really the responsibility of a functional coverage *monitor*, but for reasonably complex coverage models such as this the checking capability provides additional

validation of the coverage statements completeness. In particular if valid state transitions are accidentally omitted by the coverage model, but included in the design, then the defect should manifest itself as a runtime assertion failure.

```
property prop_LEGAL_STATE;
  @(posedge clk)
    (`DEFAULT_CONFIG ||
     `CONFIG         ||
     `READY          ||
      // etc
     `NORMAL_PASSIVE ||
     `HALT);
endproperty : prop_LEGAL_STATE

assert_LEGAL_STATE : assert property (prop_LEGAL_STATE)
  else $error("%m: illegal state");
```

Figure 9: Legal State Property Declaration and Assertion

The coding for the legal states in Figure 9 simply states that the state variable must have a legal code (allowing for the wildcard definitions) at all times, otherwise the assertion fails and generates a simulation error.

```
property prop_LEGAL_TRANS;
  @(posedge clk)
    disable iff (rst)
      (!$stable(vState) |-> (seq_DC_C.ended   ||
                             seq_C_R.ended    ||
                             seq_R_C.ended    ||
                             seq_R_WL.ended   ||
                             // etc
                             seq_CCC_IL.ended ||
                             seq_CJ_NA.ended  ));
endproperty : prop_LEGAL_TRANS

assert_LEGAL_TRANS : assert property (prop_LEGAL_TRANS)
  else $error("%m: illegal transition");
```

Figure 10: Legal Transition Property Declaration and Assertion

The syntax for *prop_LEGAL_TRANS* in Figure 10 means: if *vState* changed (i.e. is not *stable*) that implies that on the same clock edge at least one of the valid state transition sequences must have *ended*. The list includes all the basic state transitions sequences but does not need the multiple state transition sequences. Note that in this case it is appropriate to use the `disable iff` construct to handle reset operation.

3 Implementation of Coverage Infrastructure

The SVA coverage model is defined as a SystemVerilog *module* and can either be instantiated directly in the RTL, or more normally externally bound to an instance of the RTL using the SystemVerilog `bind` statement. The module port list comprises of all the signals and variables that are required to perform the coverage; an excerpt from the module header is shown in Figure 11.

```
module FRPocCovMon_sva
  ( input       clk
  , input       rst
  , input [6:0] vState
  // 8 cluster and node configuration parameters
  , input       pWakeupChannel
  // ...
  // 7 protocol variables
  , input [4:0] vRemainingColdstartAttempts
  // ...
  // 10 process variables
  , input [1:0] zSyncCalcResult
  // ...
  // 36 trigger signals
  , header_received_on_A
  // ...
  );

  // 18 state definitions + coverage
  // 90 state transition sequence definitions + coverage
  // 20 multiple state transition sequence definitions + coverage
  // 70 conditional state transition coverage
  // 220 SDL trigger coverage

endmodule : FRPocCovMon_sva
```

Figure 11: SVA Coverage Module

The `bind` statement is usually contained in a separate file or as part of the top-level testbench environment and it is implementation specific. If certain protocol variables are inaccessible in a particular implementation, for example because they are stored in RAM, then the corresponding port on the module can have a void binding (i.e. it is unconnected) as shown for *vRemainingColdstartAttempts* in Figure 12. Void bindings for coverage monitor ports result in lower coverage results since it is not possible to determine if the corresponding conditional state transition path was exercised.

```
bind tb.pe.poc
  FRPocCovMon_sva
    FRPocCovMon_sva_i
      ( .clk                          (clk)
      , .rst                          (rst)
      , .vState                       (vState)
      , .pWakeupChannel               (pWakeupChannel)
      , .vRemainingColdstartAttempts ()
      , .zSyncCalcResult              (zSyncCalcResult)
      , .header_received_on_A         (header_received[0])
      // ...
      );
```

Figure 12: SVA Bind for Coverage Module

The testbench environment can determine the coverage results for closed-loop constrained random stimulus generation, or checking multi-state sequences, by using the built-in system functions for SystemVerilog real-time coverage access as defined in the *Coverage API* section of the LRM [4]. Figure 13 shows an example of how to query the coverage value for a labeled cover statement using the `$coverage_control` and `$coverage_get` system functions.

```
integer num;
if ($coverage_control(  `SV_COV_CHECK,
                        `SV_COV_ASSERTION,
                        `SV_COV_MODULE,
                        $root.tb.pe.poc.trans_POC_CSA_CSI) == `SV_COV_OK )
  begin
    num = $coverage_get(`SV_COV_ASSERTION,
                        `SV_COV_MODULE,
                        $root.tb.pe.poc.trans_POC_CSA_CSI);
    case(num)
      `SV_COV_OVERFLOW : $fatal("non-integer coverage value");
      `SV_COV_ERROR    : $fatal("error extracting coverage value");
      default          : $display("trans_POC_CSA_CSI coverage = %0d",num);
    endcase
  end
else
  $fatal("coverage cannot be obtained for trans_POC_CSA_CSI");
```

Figure 13: Querying Real-Time SVA Coverage

4 Conclusion

This paper has demonstrated that it is possible to implement a relatively complex functional coverage model using SystemVerilog Assertions. While the full benefit of such an approach may be limited to applications where the micro-architectural requirements for the state coverage are well defined by the specification, as is the case for the SDL specification of the FlexRay Protocol Engine, the paper also illustrates the power of the SVA coverage constructs. The coding style and solutions provided can also be used in the typical low-level assertion-based coverage implemented by design and verification engineers in ABV driven projects.

While there are some disadvantages of this approach when compared to implementations in using object-oriented HLVL, in particular a lack of scalability to abstract stimulus coverage, there are a number of specific advantages which should not be overlooked. In particular:

- mixed-language simulators allow SVA to be mixed with any HDL or HLVL
- SVA coverage model can be used in simulation environments with no HLVL coverage
- implementation and use of SVA coverage model requires no object-oriented programming knowledge or skills

5 References

[1] M.Litterick and M.Brenner, *Utilizing Vera Functional Coverage in the Verification of a Protocol Engine for the FlexRay Automotive Communication System*, SNUG Europe 2005, available from http://www.verilab.com/download.htm

[2] H.Foster, A.Krolnik and D.Lacey, *Assertion-Based Design 2nd Edition*, Kluwer Academic Publishers, ISBN: 1-4020-8027-1

[3] FlexRay Consortium, *FlexRay Communications System Protocol Specification, Version 2.0, 2004*, available from http://www.flexray-group.org

[4] Accellera, *SystemVerilog 3.1a Language Reference Manual*, available from http://www.accellera.org