SystemVerilog Assertions Verification

Ionuț Ciocîrlan        Andra Radu
Tutorial topics

• Introduction to SystemVerilog Assertions (SVAs)

• Planning SVA development

• Implementation

• SVA verification using SVAUnit

• SVA test patterns
Introduction to SystemVerilog Assertions (SVAs)
Assertions usage

• Assertions package for an UVC

• Assertions in formal proof kit

• Assertions used for RTL generation
  “Revisiting Regular Expressions in SyntHorus2: from PSL SEREs to Hardware”
  (Fatemeh (Negin) Javaheri, Katell Morin-Allory, Dominique Borrione)

• Assertions used for test patterns generation
  “Towards a Toolchain for Assertion-Driven Test Sequence Generation” (Laurence PIERRE)
Assertions and properties

• What is an assertion?

```plaintext
assert (a |-> b)
else $error("Assertion failed!")
```

• What is a property?

```plaintext
property p_example;
  a |-> b
endproperty
```
Simple assertion example

After the rise of request signal, the acknowledge signal should be asserted no later than 3 clocks cycles.

```verbatim
property req_to_rise_p;
    @(posedge clk)
    $rose(req) |-> ##[1:3] $rose(ack);
endproperty

ASSERT_LABEL: assert property (req_to_rise_p)
else `uvm_error("ERR", "Assertion failed")
```
SVAs advantages

• Fast

• Non-intrusive

• Flexible

• Coverable
Types of SystemVerilog Assertions

• Concurrent

Test for a condition at the current time.

```
assert (expression) pass_statement
[else fail_statement]
```

After the assertion of request signal, the acknowledge signal should be asserted no later than 3 clocks cycles.

• Immediate

Test for a condition at the current time.
Planning SVA development
Identify design characteristics

- Defined in a document (design specification)
- Known or specified by the designer
- The most common format is of the form *cause and effect*: antecedent $\rightarrow$ consequent

- Antecedent: $\$rose(req)$
- Consequent: $\#\#[1:3] \$rose(ack)$
Keep it simple. Partition!

- Complex assertions are typically constructed from complex sequences and properties.

\[
\begin{align*}
a & \#1 b[*1:2] \Rightarrow c & \#1 d[*1:2] \Rightarrow \text{fell}(a) \\
\text{sequence} & \text{ seq}(\text{arg1, arg2}); \\
& \text{ arg1 } \#1 \text{ arg2[*1:2];} \\
\text{endsequence} \\
\text{seq}(a, b) & \Rightarrow \text{seq}(c, d) \Rightarrow \text{fell}(a)
\end{align*}
\]
Implementation
Coding guidelines

• Design code shall not be repeated in assertions

• Explicit naming

• Always check ‘X’ occurrences

• Reset considerations
Coding guidelines (contd.)

• Use the correct sampling clock

• Avoid infinite assertions

• Clear assertion messages

• Clear assertion encapsulation
Best practices

• Review the SVA with the designer to avoid DS misinterpretation

• Use \texttt{\$strong} in assertions that may never complete:

\begin{verbatim}
assert property ( req |-> \$strong([1:$] ack));
\end{verbatim}

• Properties should not hold under certain conditions (reset, enable switch)
Best practices (contd.)

• Avoid overlapping assertions that contradict each other

✓ CPU_0: `assert property (WRITE |=> ERROR);`

✓ CPU_1: `assert property (WRITE |=> !ERROR);`

  `assert property (WRITE and CPU==0 |=> ERROR);`

  `assert property (WRITE and CPU==1 |=> !ERROR);`

• Use the `$sampled()` function in action blocks and disable clauses
Assertion example

- AMBA APB protocol specification:

  The bus only remains in the SETUP state for one clock cycle and always moves to the ACCESS state on the next rising edge of the clock.
Assertion example (contd.)

• Antecedent (the SETUP phase)

```verilog
sequence setup_phase_s;
(@(posedge clk)
 $rose(psel) and $rose(pwrite)
 and (!penable) and (!pready);
endsequence
```

• Consequent (the ACCESS phase)

```verilog
sequence access_phase_s;
(@(posedge clk)
 $rose(penable) and $rose(pready)
 $stable(pwrite) and $stable(pwdata) and
 $stable(paddr) and $stable(psel)
 and ##[1: $] $fell(penable);
endsequence
```
Assertion example (contd.)

• The property can be expressed as:

```verilog
property access_to_setup_p;
  @(posedge clk)
  setup_phase_s |=> access_phase_s;
endproperty
```

• The assertion will look like:

```verilog
assert property (access_to_setup_p)
else `uvm_error("ERR", "Assertion failed")
```
SVA Verification with SVAUnit
SVA Verification Challenges

Easy to:
- Update
- Enhance
- Disable

Clear separation between validation and SVA definition code

Results should be:
- Deterministic
- Repeatable
Introducing SVAUnit

- Structured framework for Unit Testing for SVAs
- Allows the user to decouple the SVA definition from its validation code
- UVM compliant package written in SystemVerilog
- Encapsulate each SVA testing scenario inside an unit test
- Easily controlled and supervised using a simple API
SVAUnit Infrastructure

- **SVAUnit Testbench**
  - Enables SVAUnit
  - Instantiates SVA interface
  - Starts test

- **SVAUnit Test**
  - Contains the SVA scenario

- **SVAUnit Test Suite**
  - Test and test suite container
Example of SVAUnit Testbench

```verilog
module top;
  // Instantiate the SVAUnit framework
  `SVAUNIT_UTILS ... 

  // Interface with the SVAs we want to test
  an_if dut_if(.clk(clock));

  initial begin
    // Register interface with the uvm_config_db
    uvm_config_db#(virtual an_if)::
      set(uvm_root::get(), "*", "VIF", dut_if);

    // Start the scenarios
    run_test();
  end

  ... 
endmodule
```

© Accellera Systems Initiative
Example of SVAUnit Test

```verilog
class ut1 extends svaunit_test;
  // The virtual interface used to drive the signals
  virtual an_if vif;

  function void build_phase(input uvm_phase phase);
    // Retrieve the interface handle from the uvm_config_db
    if (!uvm_config_db#(virtual an_if)::get(this, "", "vif", vif))
      `uvm_fatal("UT1_NO_VIF_ERR", "SVA interface is not set!")
    // Test will run by default;
    disable_test();
    endfunction

  task test();
    // Initialize signals
    // Create scenarios for AN_SVA
    endtask
endclass
```
Example of SVAUnit Test Suite

```verilog
class uts extends svaunit_test_suite;
  // Instantiate the SVAUnit tests
  ut1 ut1;
  ...
  ut10 ut10;

  function void build_phase(input uvm_phase phase);
    // Create the tests using UVM factory
    ut1 = ut1::type_id::create("ut1", this);
    ...
    ut10 = ut10::type_id::create("ut10", this);

    // Register tests in suite
    `add_test(ut1);
    ...
    `add_test(ut10);
  endfunction

endclass
```

© Accellera Systems Initiative
SVAUnit Test API

**CONTROL**
- disable_all_assertions();
- enable_assertion(sva_name);
- enable_all_assertions();
- ...

**CHECK**
- fail_if_sva_does_not_exists(sva_name, error_msg);
- pass_if_sva_not_succeeded(sva_name, error_msg);
- pass/fail_if(expression, error_msg);
- ...

**REPORT**
- print_status();
- print_sva();
- print_report();
- ...

© Accellera Systems Initiative
SVAUUnit Flow

1. Create SVAUnit Testbench
2. Create an SVAUnit Test
3. Implement test() method

4. Create an SVAUnit Test Suite
5. Instantiate test in Test Suite
6. Register tests in test suite

7. Simulate
8. Scan report
Example APB interface

```verilog
top { 
  interface apb_if; 
  logic clk; 
  logic sel; 
  logic write; 
  logic enable; 
  logic ready; 
  logic[31:0] addr; 
  logic[31:0] wdata; 

  // APB sequences definitions 
  // APB property definition 
  // APB assertion definition 
} 

derivative apb_if; 
  logic addr; 
  logic wdata; 
  logic enable; 
  logic ready; 
  logic write; 
  logic sel; 
  logic clk; 
} 
```
Example APB sequences

• Antecedent (the SETUP phase)

```vhd
sequence setup_phase_s;
  @(posedge clk)
  $rose(psel) and $rose(pwrite) and (!penable) and (!pready)
endsequence
```

• Consequent (the ACCESS phase)

```vhd
sequence access_phase_s;
  @(posedge clk)
  $rose(penable) and $rose(pready)
  $stable(pwrite) and $stable(pwdata) and
  $stable(paddr) and $stable(psel)
  and ##[1: $] $fell(penable);
endsequence
```
Example APB property & assertion

• The property can be expressed as:

```verilog
property access_to_setup_p;
  @(posedge clk)
  setup_phase_s |=> access_phase_s;
endproperty
```

• The assertion will look like:

```verilog
APB_PHASES: assert property (access_to_setup_p)
else `uvm_error("ERR", "Assertion failed")
```
APB – SVA test steps

1. Enable the APB SVA
2. Initialize the interface signals
3. Generate setup phase stimuli
4. Generate access phase stimuli
5. SVA checks based on generated stimuli
Enable SVA and initialize signals

...  

// Enable the APB SVA
vpiw.disable_all_assertions();
vpiw.enable_assertion("APB_PHASES");

// Initialize signals

task initialize_signals();
    apb_vif.addr <= 32'b0;
    apb_vif.wdata <= 32'b0;
    apb_vif.write <= 1'b0;
    apb_vif.enable <= 1'b0;
    apb_vif.sel <= 1'b0;
endtask

...
Generate setup phase stimuli

... task generate_setup_phase_stimuli(bit valid);
...
  // Stimuli for valid SVA scenario
  valid == 1 ->
  write == 1 && sel == 1 && enable == 0 && ready == 0;

  // Stimuli for invalid SVA scenario
  valid == 0 ->
  write != 1 || sel != 1 || enable != 0 || ready != 0;

  ...
endtask

...
Generate access phase stimuli

... task generate_access_phase_stimuli(bit valid);
...

// Constrained stimuli for valid SVA scenario
valid == 1 ->
wdata == apb_vif.wdata && addr == apb_vif.addr &&
write == 1 && sel == 1 && enable == 1 && ready == 1;

// Constrained stimuli for invalid SVA scenario
valid == 0 -> wdata != apb_vif.wdata || addr != apb_vif.addr ||
write != 1 || sel != 1 || enable != 1 || ready != 1;

... endtask ...
...
SVA state checking

... 

if (valid_setup_phase)
    if (valid_access_phase)
        vpiw.fail_if_sva_not_succeeded("APB_PHASES",
            "The assertion should have succeeded!");
    else
        vpiw.fail_if_sva_succeeded("APB_PHASES",
            "The assertion should have failed!");
else
    vpiw.pass_if_sva_not_started("APB_PHASES",
            "The assertion should not have started!");

...
Error reporting

Name of SVAUnit check

SVAUnit test path

Name of SVA under test

Custom error message

UVM_ERROR @ 55000 ns [SVAUNIT FAIL IF SVA SUCCEEDED_ERR]: [x_z.suite.addr_x_z_test::x_z_addr_ut AMIQ_APB_ILLEGAL_ADDR_VALUE_ERR] The assertion should have failed
Hierarchy report

UVM_INFO @ 56000 ns [protocol_ts]:
protocol_ts
  protocol_ts.protocol_test1
  protocol_ts.protocol_test2
  protocol_ts.x_z_suite
    x_z_suite.addr_x_z_test
    x_z_suite.slverr_x_z_test
    x_z_suite.sel_x_z_test
    x_z_suite.write_x_z_test
    x_z_suite.strb_x_z_test
    x_z_suite.prot_x_z_test
    x_z_suite.enable_x_z_test
    x_z_suite.ready_x_z_test
Test scenarios exercised

protocol_ts test suite: Status statistics

* protocol_ts FAIL (2/3 test cases PASSED)

* protocol_ts.x_z_suite FAIL (0/8 test cases PASSED)
  protocol_ts.protocol_test2 PASS (13/13 assertions PASSED)
  protocol_ts.protocol_test1 PASS (13/13 assertions PASSED)

UVM_INFO @ 56000 ns [protocol_ts]:

3/3 Tests ran during simulation

  protocol_ts.x_z_suite
  protocol_ts.protocol_test2
  protocol_ts.protocol_test1
SVAs and checks exercised

-----------------------------
protocol_ts test suite : SVA and checks statistics
-----------------------------

AMIQ_APB_ILLEGAL_SEL_TRANSITION_TR_PHASES_ERR 13/13 checks PASSED
  SVAUNIT_FAIL_IF_SVA_SUCCEEDED_ERR 1/1 times PASSED
  SVAUNIT_FAIL_IF_SVA_NOT_SUCCEEDED_ERR 2/2 times PASSED
  SVAUNIT_FAIL_IF_SVA_DOES_NOT_EXISTS_ERR 7/7 times PASSED
  SVAUNIT_PASS_IF_SVA_IS_ENABLE_ERR 3/3 times PASSED

AMIQ_APB_ILLEGAL_SEL_TRANSITION_DURING_TRANSFER_ERR 13/13 checks PASSED
  SVAUNIT_FAIL_IF_SVA_NOT_SUCCEEDED_ERR 1/1 times PASSED
  SVAUNIT_FAIL_IF_SVA_SUCCEEDED_ERR 2/2 times PASSED
  SVAUNIT_FAIL_IF_SVA_DOES_NOT_EXISTS_ERR 7/7 times PASSED
  SVAUNIT_PASS_IF_SVA_IS_ENABLE_ERR 3/3 times PASSED
Tools integration
Availability

- SVAUnit is an open-source package released by AMIQ Consulting
- We provide:
  - SystemVerilog and simulator integration codes
  - AMBA-APB assertion package
  - Code templates and examples
  - HTML documentation for API

https://github.com/amiq-consulting/svaunit
SVA test patterns
Simple implication test

- \( a \text{ and } b \implies c \)

```cpp
repeat (test_loop_count) begin
    randomize(stimuli_for_a, stimuli_for_b, stimuli_for_c);

    interface.a  <= stimuli_for_a;
    interface.b  <= stimuli_for_b;
    @(posedge an_vif.clk);

    interface.c  <= stimuli_for_c;
    @(posedge interface.clk);

    @(posedge interface.clk);
    if (stimuli_for_a == 1 && stimuli_for_b == 1)
        if (stimuli_for_c == 1)
            vpiw.fail_if_sva_not_succeeded("IMPLICATION_ASSERT",
                                             "The assertion should have failed!");
        else
            vpiw.fail_if_sva_succeeded("IMPLICATION_ASSERT",
                                      "The assertion should have succeeded!");
    else
        vpiw.pass_if_sva_not_started("IMPLICATION_ASSERT",
                                      "The assertion should not have started!");
end
```
Multi-thread antecedent/consequent

• $\text{rose}(a) \#\#[1:4] b \quad \rightarrow \quad \#\#[1:3] c$

```vhls
repeat (test_loop_count) begin
  // Generate valid delays for asserting b and c signals
  randomize(delay_for_b inside {[1:4]}, delay_for_c inside {[1:3]});
  interface.a <= 1;

  repeat (delay_for_b)
    @(posedge interface.clk);
  interface.b <= 1;

  vpiw.pass_if_sva_started_but_not_finished("MULTITHREAD_ASSERT",
      "The assertion should have started but not finished!");

  repeat (delay_for_c)
    @(posedge interface.clk);
  interface.c <= 1;

  vpiw.pass_if_sva_succeeded("MULTITHREAD_ASSERT",
      "The assertion should have succeeded!");
end
```

© Accellera Systems Initiative 45
Multi-thread
antecedent/consequent (contd.)

• $\text{\texttt{\$rose(a) \#\#[1:4] b |-> \#\#[1:3] c}}$

repeat (test_loop_count) begin

  // Generate invalid delays for asserting b and c signals
  randomize(delay_for_b inside {0,[5:10]}, delay_for_c inside {0,[4:10]});
  interface.a <= 1;

  repeat (delay_for_b)
    @(posedge interface.clk);
  interface.b <= 1;

  vpiw.pass_if_sva_not_succeeded("MULTITHREAD_ASSERT",
    "The assertion should have failed!");

  repeat (delay_for_c)
    @(posedge interface.clk);
  interface.c <= 1;

  if (delay_for_b < 5)
    vpiw.fail_if_sva_succeeded("MULTITHREAD_ASSERT",
      "The assertion should have failed!");
end
Consecutive repetition

• a |→ b[*1:2]  # #1 c

```verilog
repeat (test_loop_count) begin
    randomize(stimuli_for_a, stimuli_for_c, number_of_b_cycles <= 2);

    interface.a  <= stimuli_for_a;

    repeat (number_of_b_cycles) begin
        randomize(stimuli_for_b)
        interface.b  <= stimuli_for_b;
        if (stimuli_for_b == 1) number_of_b_assertions += 1;

        @(posedge interface.clk);
    end

    if (stimuli_for_a == 1 && number_of_b_assertions == number_of_b_cycles &&
        number_of_b Assertions > 0)
        vpiw.pass_if_sva_started but not finished("IMPLICATION ASSERT",
            "The assertion should have started but not finished!");
        @(posedge interface.clk);

... // (continued on the next slide)
```
Consecutive repetition

- $a \rightarrow b[*1:2] \#\#1 c$

... // (continued from previous slide)

```vhdl
interface.c <= stimuli_for_c;
@ (posedge interface.clk);

if (stimuli_for_a == 1)
    if (number_of_b_assertions != number_of_b_cycles ||
        number_of_b_assertions == 0 ||
        stimuli_for_c == 0)
        vpiw.fail_if_sva_succeeded("IMPLICATION_ASSERT",
            "The assertion should have failed!");
    else
        vpiw.fail_if_sva_not_succeeded("IMPLICATION_ASSERT",
            "The assertion should have succeeded!");

end // end of test repeat loop
```
Repetition range with zero

- **a \rightarrow b^{*0:2} \#1 c**

```verilog
repeat (test_loop_count) begin
    randomize(stimuli_for_a, stimuli_for_c, number_of_b_cycles <= 2);

    interface.a <= stimuli_for_a;

    repeat (number_of_b_cycles) begin
        randomize(stimuli_for_b)
        interface.b <= stimuli_for_b;
        if (stimuli_for_b == 1) number_of_b_assertions += 1;

        @(posedge interface.clk);
    end

    if (stimuli_for_a == 1 && number_of_b_assertions == number_of_b_cycles)
        && number_of_b_assertions > 0)
        vpiw.pass_if_sva_started_but_not_finished("IMPLICATION_ASSERT",
            "The assertion should have started but not finished!");
    @(posedge interface.clk);

    ...
    // (continued on the next slide)
```
Repetition range with zero

• a |-> b[*0:2] ##1 c

```vhdl
...  
// (continued from previous slide)

interface.c <= stimuli_for_c;
@(posedge interface.clk);

if (stimuli_for_a == 1)
    if (number_of_b_assertions != number_of_b_cycles ||
        number_of_b_assertions == 0 ||
        stimuli_for_c == 0)
        vpiw.fail_if_sva_succeeded("REPETITION_RANGE0_ASSERT",
                                "The assertion should have failed!");
    else
        vpiw.fail_if_sva_not_succeeded("REPETITION_RANGE0_ASSERT",
                                     "The assertion should have succeeded!");

end // end of test repeat loop
```
Sequence disjunction

• \( a \rightarrow (b \ ##1 \ c) \ or \ (d \ ##1 \ e) \)

```vhdl
repeat (test_loop_count) begin
  randomize(stimuli_for_a, stimuli_for_b, stimuli_for_c, stimuli_for_d, stimuli_for_e);

  interface.a <= stimuli_for_a;
  @(posedge interface.clk);
  fork
    begin
      Stimuli for branch: (b ##1 c)
      SVA state check based on branch stimuli
    end
  begin
    Stimuli for branch: (c ##1 d)
    SVA state check based on branch stimuli
  end
  join
end
```
Sequence disjunction (contd.)

- \[ a \mid\Rightarrow (b \;\#\#1\; c) \; or \; (d \;\#\#1\; e) \]

```
...

// Stimuli for branch (b \#\#1 c)
fork
    begin
        interface.b  <= stimuli_for_b;
        @(posedge interface.clk);

        interface.c  <= stimuli_for_c;
        @(posedge interface.clk);

        @(posedge interface.clk);
        // SVA state check based on branch stimuli
        sva_check_phase(interface.a, interface.b, interface.c);
    end
join
```
Sequence disjunction (contd.)

• \(a \implies (b \#1 c) \text{ or } (d \#1 e)\)

```vhdl
... // Stimuli for branch (d \#1 e)
fork
  begin
    interface.b <= stimuli_for_d;
    @(posedge interface.clk);

    interface.c <= stimuli_for_e;
    @(posedge interface.clk);

    @(posedge interface.clk);
    // SVA state check based on branch stimuli
    sva_check_phase(interface.a, interface.d, interface.e);
  end
join
```
Sequence disjunction (contd.)

• \( a \implies (b \#\#1 c) \textbf{ or } (d \#\#1 e) \)

```// SVA state checking task used in each fork branch

task sva_check_phase(bit stimuli_a, bit stimuli_b, bit stimuli_c);
  if (stimuli_a)
    if (stimuli_b && stimuli_c)
      vpiw.pass_if_sva_succeeded("DISJUNCTION_ASSERT",
                                 "The assertion should have succeeded");
    else
      vpiw.fail_if_sva_succeeded("DISJUNCTION_ASSERT",
                                "The assertion should have failed");
  endtask
```
Conclusions

- SVAUnit decouples the checking logic from SVA definition code
- Safety net for eventual code refactoring
- Can also be used as self-checking documentation on how SVAs work
- Quick learning curve
- Easy-to-use and flexible API
- Speed up verification closure
- Boost verification quality
Q & A