SystemVerilog Assertions Verification with SVAUnit

Ionuţ Ciocîrlan
Andra Radu

AMIQ Consulting
Bucharest, Romania

www.amiq.com

ABSTRACT

SystemVerilog Assertions are one of the central pieces in functional verification for protocol checking or validation of specific functions. In order to benefit from assertion advantages (fast, synthesizable, non-intrusive, coverable), one must be sure that assertions work as specified. Verification engineers need to make sure that SVAs pass in normal conditions and fail under error conditions. This implies some tedious work to create the scenario that properly triggers an assertion. Besides the stimuli generation, one should also implement checks to ensure that the assertion under test is triggered at the right time. All this preferably without contaminating the assertions with the validation code.

SVAUnit is a SystemVerilog library that addresses this by decoupling assertion validation code from assertion definition code, simplifying the generation of stimuli and providing the ability to reuse scenarios. It also includes a self-checking mechanism and automatic test status report.

Table of Contents

1. Introduction 4

2. SVAs and accompanying challenges 4

3. Introducing SVAUnit 5

4. The big picture 6

5. Building blocks 7
5.1. SVAUnit Testbench 7
5.2. SVAUnit Test 8
5.2.1. Pre_test() method 10
5.2.2. Test() method 11
5.3. SVAUnit Test Suite 13

6. Batteries included 14
6.1. APIs for controlling SVAs 14
6.2. APIs for checking SVAs 16
6.3. APIs for printing reports 17

7. SVAUnit flow 18

8. Reaping the rewards 19

9. Conclusions 22

10. Availability 22

11. References 23

Table of Figures

Figure 1. Simple SVA example 4
Figure 2. SVAUnit components 6
Figure 3. SVA Scenario 11
Figure 4. SVAUnit flow 18
Figure 5. SVAUnit tree report 19
Figure 6. SVAUnit Test status report 19
Figure 7. SVAUnit report on SVAs 20
Figure 8. SVAUnit check status report 20
Figure 9. SVAUnit report on checks used 21
Figure 10 Example of SVAUnit error 21
Figure 11. Example of SVAUnit error for a test with parameters 21

Table of Code examples

Code example 1. SVAUnit Testbench for simple interface 7
Code example 2. SVAUnit Testbench for multiple parameterized interfaces 8
Code example 3. SVAUnit Test 9
Code example 4. SVAUnit Test with parameters 10
Code example 5. pre_test() method 10
Code example 6. SVA example 11
Code example 7. test() method 12
Code example 8. SVAUnit Test Suite 13

Table of Tables

Table 1. APIs for controlling SVA 14
Table 2. APIs for controlling the entire set of SVAs 15
Table 3. APIs for checking SVA state 16
Table 4. APIs for printing reports 17

1. Introduction

The paper begins with a short description of SystemVerilog Assertions and their role in the verification world. It will also address some of the challenges encountered in validating SystemVerilog Assertions.
This will set the stage for presenting the capabilities of SVAUnit, what it is and what are the main features that can help the user improve its efficiency when working with SVAs.
The paper also contains an overview of the SVAUnit package and its main building blocks. The architectural details of each component and their role inside the SVAUnit package will also be presented.
Knowing how the SVAUnit framework is built, the paper offers a description of each control method available.
Last, but not least, the paper provides a step by step example for using the SVAUnit package in order to reach the goal of having complete and verified SVAs with minimum effort.

2. SVAs and accompanying challenges

SystemVerilog Assertions (SVAs) are a fundamental part of verifying that the design-under-test complies with a given protocol or validating its specific functions. Simply put, an assertion is a check against the specification of a design that we want to make sure it never violates.
Based on its purpose, an SVA can vary from a simple statement that a certain property must be true, to complex expressions meant to check that the intent of the design is met over simulation time.

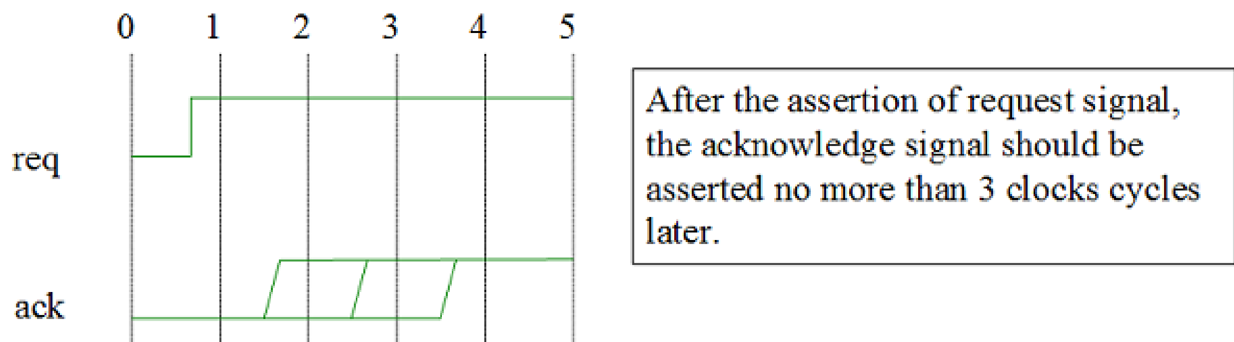

Figure 1. Simple SVA example

There is a saying which seems valid for the functional verification field as well: *"be careful of the environment you choose, for it will shape you".* Assertions can increase the value of the verification done when they function as intended but can also work against it if left unverified (e.g. issues in the checking logic usually hide the ones in the design).
The question we try to answer is how to assure that an SVA works as specified.
The most common approach would be to start developing a scenario in which the SVA is triggered, but that is not enough. Making sure the SVA triggers as expected and at the correct moment requires additional code that will most likely be non-reusable and can hide issues or omit scenarios.
There is also an alternative to validate the SVA by running it against a proven design, but that seems a luxury not everybody can enjoy and still does not cover all the possible issues. For each

of the approaches presented above, tedious work has to be invested in checking the correctness of each SVA.
A more standardized solution is needed, one that can do all the above but still be reusable from project to project and at the same time ready to use out of the box. This was the driving force for developing SVAUnit, a package for verifying SystemVerilog Assertions that will be addressed in the following chapters.

3. Introducing SVAUnit

SVAUnit combines the unit testing paradigm of the software world with the powerful feature of assertions from hardware verification languages like SystemVerilog. SVAUnit represents a structured framework for unit testing that allows the user to decouple the assertion validation code from the definition code.
It provides the ability to tackle verification completeness head on in the early steps of the code development process. Assertions can be validated as they are written without having to resort to time consuming and non-reusable checking logic.
SVAUnit is an UVM compliant package written in SystemVerilog. It provides a base class to develop unit tests and suites in order to prove that assertions execute as they are intended.
SVAUnit appeals to modularity by providing the means to encapsulate each SVA testing scenario inside an unit test.
Increasingly complex environments require reusability. SVAUnit provides the ability to reuse scenarios and extend the SVA verification to multiple tests in the same simulation through SVAUnit test suites.
All the functionality can be easily controlled and supervised using a simple API.
Although the SVAUnit package can be smoothly integrated with an existing verification environment, the only real requirement is an interface containing the SVAs to be verified.

4. The big picture

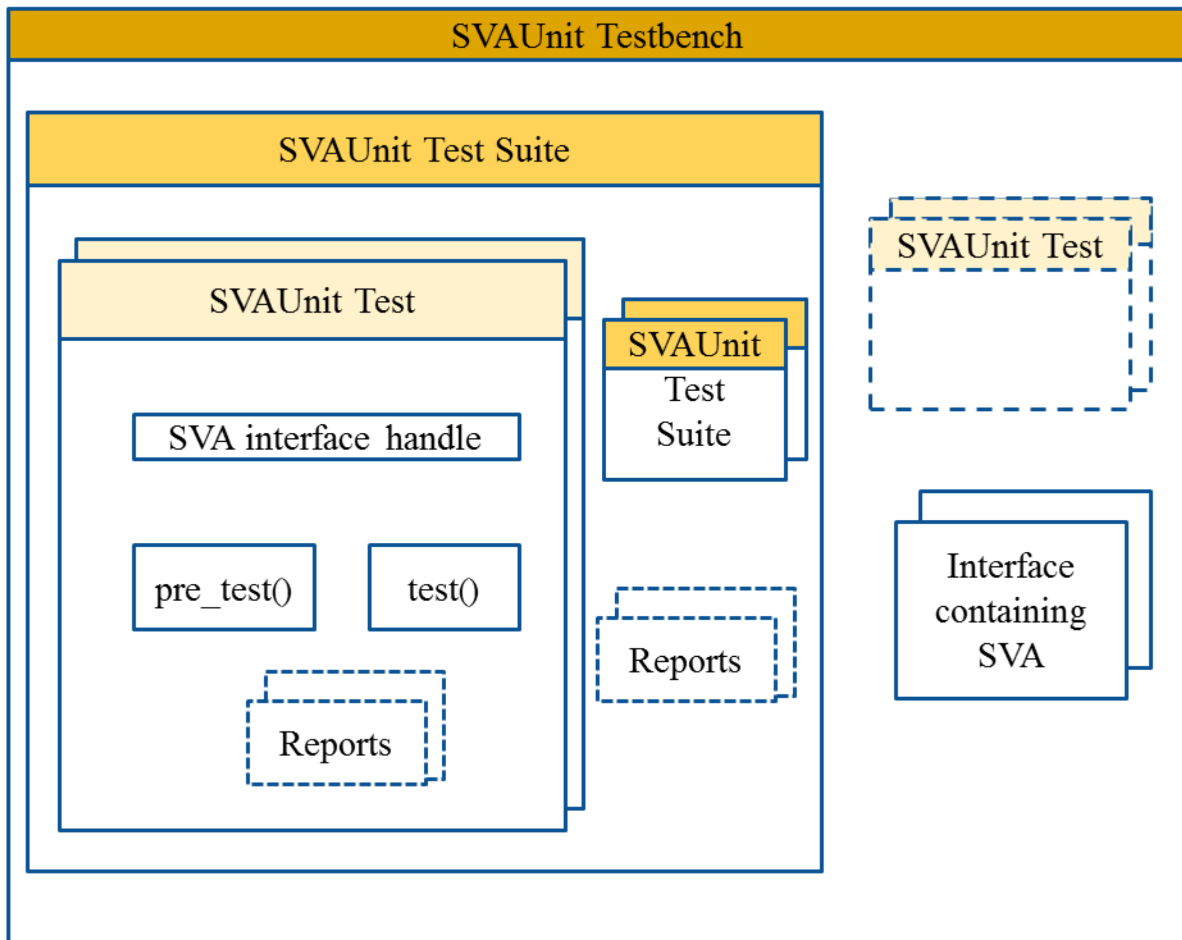

Figure 2. SVAUnit components

The main building blocks of the SVAUnit package are:

- The **SVAUnit Testbench** instantiates the interface containing the SVA and represents the starting point for an **SVAUnit Test** or **SVAUnit Test Suite**.
- The **SVAUnit Test** represents a scenario used to check an SVA. It can be run standalone and/or inside a test suite.
- The **SVAUnit Test Suite** represents a collection of **SVAUnit Tests** and/or **Test Suites** used to verify a set of SVAs.

5. Building blocks

Architectural details of each building block will be presented in the following pages as well as code examples. The SVAUnit package contains code templates enabling the user to concentrate on test scenario rather than building the infrastructure.

5.1. SVAUnit Testbench

The SVAUnit Testbench represents a SystemVerilog module where the SVAUnit package is used.
You can define it as a separate module and integrate in your verification environment or you can simply upgrade your top module that has access to the SVA interface.
The SVAUnit framework is enabled as soon as one instantiates the `` `SVAUNIT_UTILS `` which will handle all the "heavy lifting" in a manner that is transparent from the user's perspective.
The interface containing the SVA must be instantiated in the SVAUnit Testbench and a virtual interface reference must be set in the `uvm_config_db` in order to have access to it later on.
Following you can find an example on how a simple SVAUnit Testbench can look like:

```
module top;
  `SVAUNIT_UTILS
  reg clock;

  my_if dut_if(.clk(clock));

  initial begin
    uvm_config_db#(virtual my_if)::set(uvm_root::get(), "*",
    "VIF", dut_if);
  end

  initial begin
    run_test();
  end

  initial begin
    clock = 1'b0;
  end

  always #1 clock = ~clock;
endmodule
```

Code example 1. SVAUnit Testbench for simple interface

Using parameterized interfaces or multiple interfaces at once is fully supported.
The next code example shows an SVAUnit Testbench containing several instances of a parameterized interface.

```
module top;
  ...

  my_if#(100) dut_if(.clk(clock));

  initial begin
    uvm_config_db#(virtual my_if#(100))::set(uvm_root::get(),
    "*", "VIF", dut_if);
  end
  ...
endmodule

module top;
  ...

  generate
    genvar if_param;

    for(if_param = 100; if_param < 200; if_param++) begin
      my_if#(if_param) dut_if(.clk(clock));

      initial begin
        uvm_config_db#(virtual
        my_if#(.if_param(if_param)))::set(uvm_root::get(),
        "*", $sformatf("vif%0d", if_param), dut_if);
      end
    end
  endgenerate

  ...
endmodule
```

Code example 2. SVAUnit Testbench for multiple parameterized interfaces

5.2. SVAUnit Test

The SVAUnit Test class inherits `uvm_test`, which means it will benefit from UVM base test features. The SVAUnit Test is used to describe and implement one scenario that verifies one or more aspects of an SVA (this is a recommendation, not a requirement).
The interface containing the SVAs under test is accessible through the `uvm_config_db` since it was set from the SVAUnit Testbench.

The test class contains two important methods: `pre_test()` and `test()`. The `pre_test()` method should contain the verification scenario initialization.
The verification scenario should be defined inside the `test()` method. The scenario contains the SVA stimuli generation and checking of the SVA state. An example of an SVAUnit Test is provided below.

```
class ut1 extends svaunit_test;
  virtual my_if vif;

  function void build_phase(input uvm_phase phase);
    if (!uvm_config_db#(virtual my_if)::get(this, "", "vif",
vif))
      `uvm_fatal("UT1_NO_VIF_ERR", "SVA IF is not set!")

      // The test is configured to run by default
      disable_test();
  endfunction

  task pre_test();
    // Initialize signals
  endtask

  task test();
    // Create scenarios for AN_SVA
  endtask
endclass
```

Code example 3. SVAUnit Test

One can enable or disable a test through a call to `enable_test()` or `disable_test()` method in the `build_phase()`.
The SVAUnit package offers the possibility to create parameterized tests by using `` `SVAUNIT_TEST_WITH_PARAM_UTILS `` macro call inside a SVAUnit Test class.

An example of an SVAUnit Test with parameters is provided below:

```
class ut2#(type if_t=int) extends svaunit_test;
  `SVAUNIT_TEST_WITH_PARAM_UTILS

  virtual if_t vif;

  function void build_phase(uvm_phase phase);
    if (!uvm_config_db#(virtual if_t)::get(this, "", "vif",
vif))
      `uvm_fatal("UT2_NO_VIF_ERR", "SVA IF is not set!")
  endfunction

  task pre_test();
    // Initialize signals
  endtask

  task test();
    // Create scenarios for AN_SVA
  endtask
endclass
```

Code example 4. SVAUnit Test with parameters

Each SVAUnit Test has a name that can be retrieved using the `get_test_name()` method. The name will be constructed as *parent_test_suite_name.test_name*.

5.2.1. Pre_test() method

The `pre_test()` method should commonly contain the signals initializations, this to ensure the signal values are not propagated when running multiple tests. The method also provides a good place to enable or disable assertions through SVAUnit package provided API.
An example of the `pre_test()` method can be found below:

```
task pre_test();
  disable_all_assertions();
  vif.sel    = 1'b0;
  vif.enable = 1'b0;
  vif.ready  = 1'b0;
  vif.slverr = 1'b0;
endtask
```

Code example 5. pre_test() method

5.2.2. Test() method

The `test()` method contains the scenario to verify the SVA. Stimuli are driven on the interface signals and SVAUnit checks are used to detect if the behaviour of the user-defined SVAs matches the expected one. For example, for an SVA described as *„slverr signal should be 0 if no slave is selected or when transfer is not enabled or when slave is not ready to respond“*, the SVA code could look like this:

```
interface my_if (input clk);
  ...
  logic sel;
  logic enable;
  logic ready;
  logic slverr;

  property an_sva_property;
    @(posedge clk)
    !sel || !enable || !ready |-> !slverr;
  endproperty
  AN_SVA: assert property (an_sva_property) else
    `uvm_error("AN_SVA", "AN_SVA failed")
endinterface
```

Code example 6. SVA example

Signal wise the scenario will look like in the figure below, where the green arrow means that the SVA should have succeeded and the red one that the SVA should have failed.

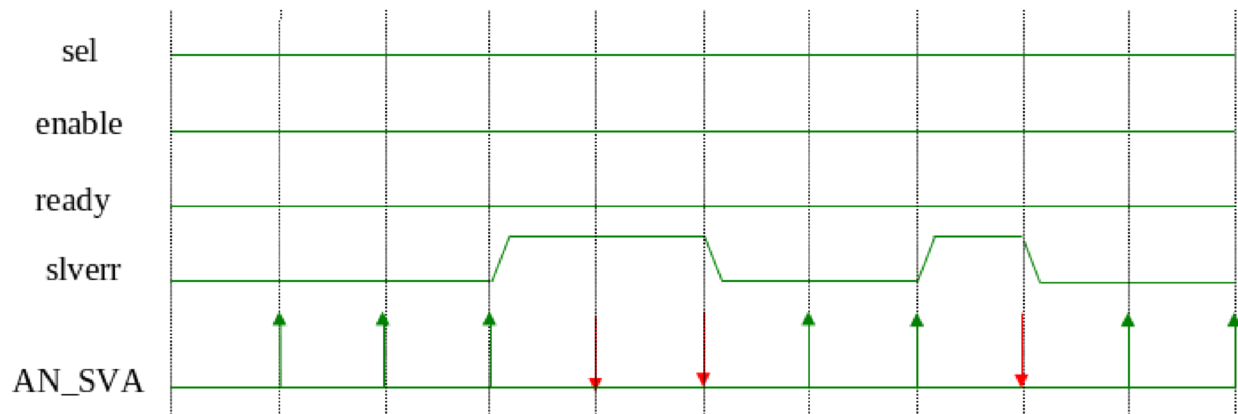

Figure 3. SVA Scenario

This scenario can be translated into the `test()` method as seen below:

```
task test();
  disable_all_assertions();
  enable_assertion("AN_SVA");

  repeat(2) @(posedge vif.clk);
  repeat(2) begin
    @(posedge vif.clk);
    fail_if_sva_not_succeeded("AN_SVA", "SVA should have
    succeeded");
  end

  // Trigger the error scenario
  vif.slverr <= 1'b1;

  repeat(2) begin
    @(posedge vif.clk);
    fail_if_sva_succeeded("AN_SVA", "SVA should have failed");
  end

  // End the error scenario
  vif.slverr <= 1'b0;
  repeat(2) begin
    @(posedge vif.clk);
    fail_if_sva_not_succeeded("AN_SVA", "SVA should have
    succeeded");
  end

  // Trigger the error scenario
  vif.slverr <= 1'b1;
  @(posedge vif.clk);
  fail_if_sva_succeeded("AN_SVA", "SVA should have failed");

  // End the error scenario
  vif.slverr <= 1'b0;
  repeat(2) begin
    @(posedge  vif.clk);
    fail_if_sva_not_succeeded("AN_SVA", "SVA should have
    succeeded");
  end
endtask
```

Code example 7. test() method

5.3. SVAUnit Test Suite

The SVAUnit Test Suite comes in handy when more than one SVAUnit Test are required. It inherits the `svaunit_test` class and is a run container for other test or test suites. Tests can be easily added inside the test suite using the `add_test()` method after they are instantiated and created. They will run in the same order they were added inside the test suite. A test can be excluded from running by disabling it through the `disable_test()` method. All these actions should be done inside the `build_phase()`'s body.

```
class uts extends svaunit_test_suite;
  ut1 unit_test1;
  ...
  ut2#(my_if#(100)) unit_test2;

  function void build_phase(input uvm_phase phase);
    unit_test1 = ut1::type_id::create("unit_test1", this);
    ...
    unit_test2 = ut2#(my_if#(100))::type_id::create("unit_test2",
this);

    add_test(unit_test1);
    ...
    add_test(unit_test2);
    unit_test2.disable_test();
  endfunction
endclass
```

Code example 8. SVAUnit Test Suite

6. Batteries included

The SVAUnit packages comes equipped with APIs to control the behaviour of an SVA, check its state or to print out reports containing results and statistics from the SVAUnit tests or test suites. These APIs can be used both inside SVAUnit tests and SVAUnit Test Suites. They are accessible in the `test()` method.

6.1. APIs for controlling SVAs

These APIs will control the status and behaviour of an SVA. They can be used to control either a single SVA or the entire list of SVAs.

API for controlling a single SVA	Description
`reset_assertion(sva_name);`	This will set the SVA back to its initial enabled or disabled state.
`disable_assertion(sva_name);`	The SVA will not start. If it was already started, it will not be finished. By default, each SVA is enabled.
`enable_assertion(sva_name);`	The SVA will start again if it was disabled.
`kill_assertion(sva_name, sim_time);`	The SVA started at `sim_time` will not be finished. The SVA will remain enabled without being set back to its initial state.
`disable_step_assertion(sva_name);`	Any step callback for this SVA will be not triggered.
`enable_step_assertion(sva_name);`	This will start triggering the step callback for this SVA. By default step callback for all assertions is disabled.

Table 1. APIs for controlling SVA

Assertion step represents callbacks triggered only when an event occurs; the assertion will advance at that event.

APIs for controlling the entire set of SVAs	Description
`reset_all_assertions();`	Resets all assertions.
`disable_all_assertions();`	Disables all assertions.
`enable_all_assertions();`	Enables all assertions.
`kill_all_assertions(sim_time);`	Kills all assertions started at sim_time.
`disable_step_all_assertions();`	Disable step for all assertions.
`enable_step_all_assertions();`	Enable step for all assertions.
`system_reset_all_assertions();`	The entire SVA system will be set back to its initial state. The step callbacks will be removed.
`system_on_all_assertions();`	The entire SVA system will be restarted after the suspension of the system with `system_off_all_assertions()`.
`system_off_all_assertions();`	The SVA system will not start again and if any SVA state has started, it will not be finished.
`system_end_all_assertions();`	SVA system will be disabled. All callbacks will be removed.

Table 2. APIs for controlling the entire set of SVAs

6.2. APIs for checking SVAs

These APIs are used to check the SVA state and are the powerhouse of the SVAUnit package. The SVA state must be checked at least one clock cycle after the SVA was triggered in order for its state to be correctly retrieved from the simulator.

APIs for checking SVA state	Description
`pass/fail_if_sva_does_not_exists (sva_name, error_msg);`	The test will pass/fail if the given SVA name does not exists.
`pass/fail_if_sva_enabled (sva_name, error_msg);`	The test will pass/fail if the given SVA is enabled.
`pass/fail_if_sva_succeeded (sva_name, error_msg);`	The test will pass/fail if the SVA succeeded.
`pass/fail_if_sva_not_succeeded (sva_name, error_msg);`	The test will pass/fail if the SVA has failed.
`pass/fail_if_sva_started_but_not_finished(sva_name, error_msg);`	The test will pass/fail if the SVA has started but did not finished.
`pass/fail_if_sva_not_started (sva_name, error_msg);`	The test will pass/fail if the SVA has not started.
`pass/fail_if_sva_finished (sva_name, error_msg);`	The test will pass/fail if the SVA has finished.
`pass/fail_if_sva_not_finished (sva_name, error_msg);`	The test will pass/fail if the SVA has not finished.
`pass/fail_if_all_sva_succeeded (error_msg);`	The test will pass/fail if all the SVAs from an interface succeeded.
`pass/fail_if(expression, error_msg);`	The test will pass/fail if the expression is true.

Table 3. APIs for checking SVA state

The `fail_if_sva_does_not_exists` and `pass_if_sva_enabled` are always used when the other checks are used, except for the `pass/fail_if` check. Furthermore, the `fail_if_sva_does_not_exists` check will be performed when a control API is used for an SVA.

6.3. APIs for printing reports

The APIs for printing reports are called when a test suite or test has finished, depending on which has been run. Failing tests or SVAs will be flagged inside the report using an asterisk („*“).

APIs for printing reports	Description
`print_status();`	This can be used to display the status of a test suite or test.
`print_sva();`	This can be used to display how many assertions are tested and how many are not, along with the SVA names. It will also display the statistics for the coverage statements written for the SVA.
`print_checks();`	This can be used to display how many checks are used and how many are not used along with the check names.
`print_sva_and_checks();`	This can be used to display all SVAs along with the checks used to test them.
`print_tests();`	This displays the tests that have run in the simulation. It can be used inside a test suite.
`print_tree();`	This can be used to display the SVAUnit topology created.
`print_report();`	This can be used to display all the above reports.
`print_sva_info(sva_name);`	This can be used to display information regarding the selected SVA.

Table 4. APIs for printing reports

7. SVAUnit flow

The first and only real requirement to use the SVAUnit framework is to have an interface containing the SVAs that need to be validated.

The rest is just a simple matter of putting the discussed building blocks and APIs together with the interface and one can start developing test scenarios as shown in the SVAUnit flow diagram below.

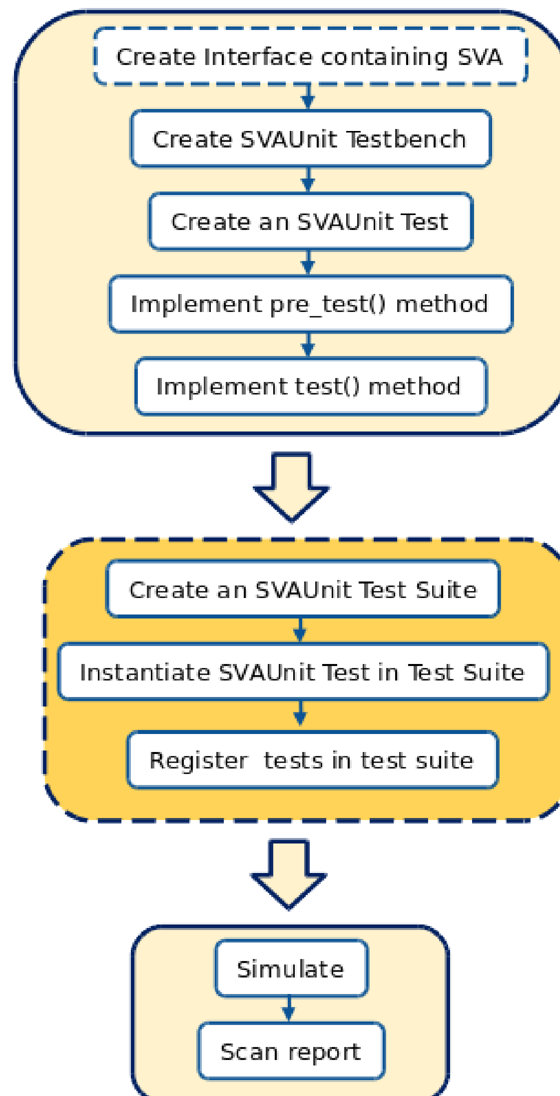

Figure 4. SVAUnit flow

8. Reaping the rewards

Now that one knows how to start using the SVAUnit package, it is a good time to have a quick walkthrough on how SVAUnit's reports are printed out.
The complete end of simulation report is structured into several layers, starting with the SVAUnit's topology.

```
        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
```

Figure 5. SVAUnit tree report

The above report contains information on which tests and/or test suites make up our main test suite. The SVAUnit topology can vary from running a simple test to complex test suites several layers deep.
The next report will present the status of each test and/or test suite run in our simulation.

```
-------------------- 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
```

Figure 6. SVAUnit Test status report

The test or test suite in which the SVA checks have failed will be flagged with a wildcard in the report.

Besides presenting the status of the simulation, the report also contains both the SVAs that have been tested and those that have not.

```
------------------- protocol_ts test suite : SVAs statistics --------------------

       3/30 SVA were exercised
               AMIQ_APB_ILLEGAL_ADDR_VALUE_ERR
               AMIQ_APB_ILLEGAL_SEL_TRANSITION_DURING_TRANSFER_ERR
               AMIQ_APB_ILLEGAL_SEL_TRANSITION_TR_PHASES_ERR

       27 SVA were not exercised
               AMIQ_APB_ILLEGAL_SEL_VALUE_ERR
               AMIQ_APB_ILLEGAL_WRITE_VALUE_ERR
               AMIQ_APB_ILLEGAL_PROT_VALUE_ERR
               AMIQ_APB_ILLEGAL_ENABLE_VALUE_ERR
               AMIQ_APB_ILLEGAL_STRB_VALUE_ERR
               AMIQ_APB_ILLEGAL_READY_VALUE_ERR
               AMIQ_APB_ILLEGAL_SLVERR_VALUE_ERR
               AMIQ_APB_ILLEGAL_SEL_VALUE_POST_RESET_ERR
               AMIQ_APB_ILLEGAL_ENABLE_VALUE_POST_RESET_ERR
               AMIQ_APB_ILLEGAL_SLVERR_VALUE_POST_RESET_ERR
               AMIQ_APB_ILLEGAL_SEL_LEGAL_VALUES_ERR
```

Figure 7. SVAUnit report on SVAs

The report contains information on which checks have been performed on the SVAs and their status at the end of the simulation.

```
---------------- protocol_ts test suite : Checks statistics --------------------

    4/20 Checks were exercised

            SVAUNIT_FAIL_IF_SVA_DOES_NOT_EXISTS_ERR 17/17 times PASSED
            SVAUNIT_PASS_IF_SVA_IS_ENABLE_ERR 7/7 times PASSED
       *    SVAUNIT_FAIL_IF_SVA_SUCCEEDED_ERR 3/4 times PASSED
            SVAUNIT_FAIL_IF_SVA_NOT_SUCCEEDED_ERR 3/3 times PASSED

    16/20 Checks were not exercised

             SVAUNIT_FAIL_IF_SVA_IS_ENABLE_ERR
             SVAUNIT_FAIL_IF_SVA_STARTED_BUT_NOT_FINISHED_ERR
             SVAUNIT_FAIL_IF_SVA_NOT_STARTED_ERR
             SVAUNIT_FAIL_IF_SVA_FINISHED_ERR
             SVAUNIT_FAIL_IF_SVA_NOT_FINISHED_ERR
             SVAUNIT_FAIL_IF_ERR
             SVAUNIT_FAIL_IF_ALL_SUCCEEDED_ERR
             SVAUNIT_PASS_IF_SVA_DOES_NOT_EXISTS_ERR
             SVAUNIT_PASS_IF_SVA_SUCCEEDED_ERR
```

Figure 8. SVAUnit check status report

Information on which checks have been used is available in the following format:

```
--------------- 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

    *   AMIQ_APB_ILLEGAL_ADDR_VALUE_ERR   4/5 checks PASSED
            *   SVAUNIT_FAIL_IF_SVA_SUCCEEDED_ERR 0/1 times PASSED
                SVAUNIT_FAIL_IF_SVA_DOES_NOT_EXISTS_ERR 3/3 times PASSED
                SVAUNIT_PASS_IF_SVA_IS_ENABLE_ERR 1/1 times PASSED
```

Figure 9. SVAUnit report on checks used

Although the desired outcome of the simulation is debateable from each user's perspective, an SVA that does not behave according to the specified scenario will be detected and cause the simulation to fail with an error message as below.

```
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] SVA should have failed
```

Figure 10 Example of SVAUnit error

The error message will contain the name of the failing check, the enclosing test's name along with its type and the name of the SVA under validation.

In case parameters are used, the error messsage will also print them out like in the example below.

```
UVM_ERROR @ 55000 ns [SVAUNIT_FAIL_IF_SVA_SUCCEEDED_ERR]: [x_z_suite.addr_x_z_test::
x_z_addr_ut#(10) AMIQ_APB_ILLEGAL_ADDR_VALUE_ERR] SVA should have failed
```

Figure 11. Example of SVAUnit error for a test with parameters

9. Conclusions

SVAUnit is a seamlessly plug and play package that allows the user to validate SVAs behaviour in an isolated manner.
It provides a safety net for eventual code refactoring and removes the headaches of having to manually debug the possible issues.
By separating the checking logic of the SVA from its definition code it helps reduce the level of possible issues introduced in the actual validation code.
Work once, reap the rewards several times. SVAUnit provides the ability to reuse testing scenarios leaving the user free to concentrate on more complex scenarios instead of churning away through repetitive tasks.
It can even be seen as a form of self-checking documentation on how verified SVAs should work thus supporting code and knowledge sharing between several users.
With an extensive library of APIs and a quick learning curve, the SVAUnit package can speed up verification closure while at the same time increase verification quality.

10. Availability

SVAUnit is released by AMIQ Consulting as an open source project and can be used as it is or further extended to support new features. It contains SystemVerilog and simulator integration source code, the AMBA-APB assertion package, SVAUnit test examples and code templates.

11.References

[1] AMIQ Consulting Blog, www.amiq.com/consulting/blog

[2] UVM Accellera standard, http://www.accellera.org/downloads/standards/

[3] 1800-2012 - IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language, http://standards.ieee.org/findstds/standard/1800-2012.html