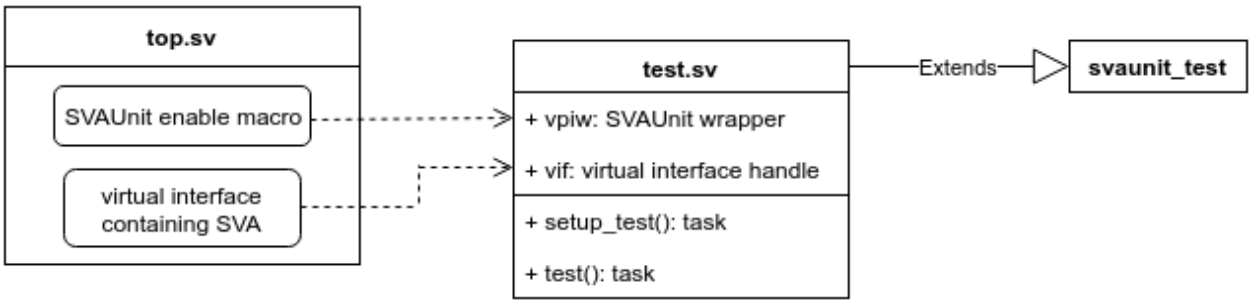
SVAUnit User Guide

Introduction

The following examples are intended as a guide to SVA verification using the SVAUnit framework.



Top

The top module is where the SVAUnit macro needs to be used in order to enable the framework in your project. This will allow you to use the API through the wrapper in the tests you create. You should declare your interfaces containing SVAs here and set handles such that you can access them easier later.

Test

Your test should extend `svaunit_test` and override the `test()` task, which simulates your desired scenario. Optionally, it is good practice to initiate the values in `setup_task()` to avoid signal propagation. Signal propagation occurs due to tests running one after another in the same simulation. Thus, one test might inherit the signal state of the previous test if signals are not initialized in the setup_test(). For every interface that you want to exercise in a test, you should get the corresponding interface handle.

Interface

The interface should contain all the signals you need and the SVAs you want to test. It's no problem to have more than one interface and even nested ones since SVAUnit detects them automatically.

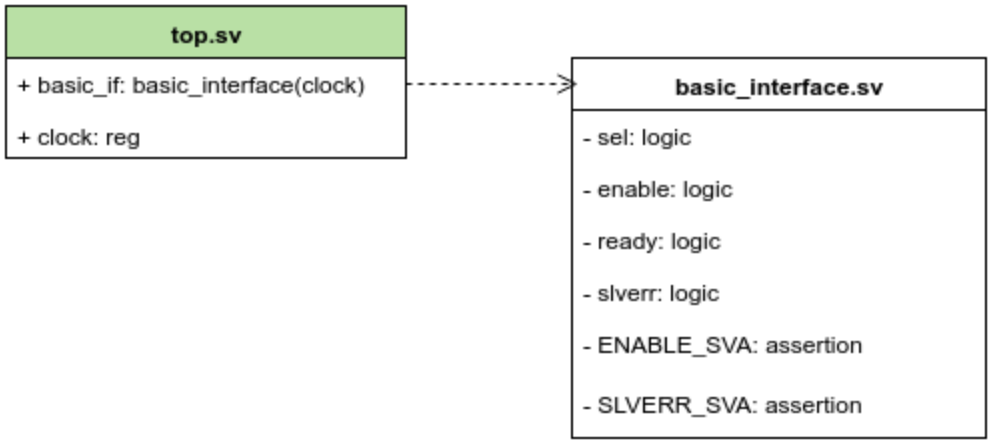
The Basic Example

The interface in this example contains two assertions:

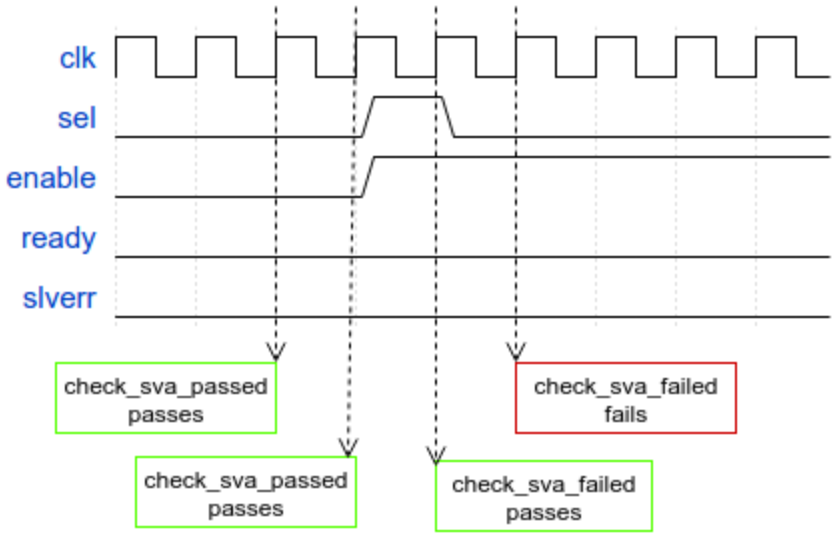
1. “slverr signal should be 0 if no slave is selected or when transfer is not enabled or when slave is not ready to respond” (`SLVERR_SVA`)
2. “sel and enable signals should never be asserted simultaneously” (`ENABLE_SVA`)

For brevity, only the `ENABLE_SVA` assertion will be verified in this example.

This is how the testbench looks like:



You will verify the correct SVA behaviour for the following scenario.



Let’s go through each step in the process.

Step 1: Create the interface and implement the assertions

You need to create an interface that contains your signals and encapsulates the SVAs you need to check with SVAUnit. It’s recommended that you use relevant error message inside `uvm_error in order to debug easier.

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

      // Property definition for valid slverr value when one of sel, enable,
ready signal is de-asserted
      property slverr_property;
      @(posedge clk)
                  !sel || !enable || !ready |-> !slverr;
      endproperty

      // Check that slverr is LOW when one of sel, enable or ready is LOW
      SLVERR_SVA: assert property (my_sva_property) else
      `uvm_error("SLVERR_SVA", "slverr must be LOW when one of sel, enable or
ready is LOW.")

      // Property definition for valid enable and sel values
      property enable_property;
      @(posedge clk)
                  ~(enable & sel);
      endproperty

      // Check that enable and sel are not asserted simultaneously
      ENABLE_SVA: assert property (enable_property) else
      `uvm_error("ENABLE_SVA", "sel and enable can not be asserted
simultaneously.")
endinterface
```

Step 2: Enable SVAUnit framework

You will need to use the `SVAUNIT_UTILS macro in top module to enable the SVAUnit framework. basic_interface will be instantiated in top module and a virtual interface reference will be set in the uvm_config_db in order to have access to it in SVAUnit tests. The run_test() method should be called in order to run the scenarios.

This is how the top module looks for this example:

```
module top;
   // Enable SVAUNIT
   `SVAUNIT_UTILS

   reg clock;

   // basic_interface instance
   basic_interface basic_if(.clk(clock));

   initial begin
      // Set clock initial values
      clock = 1'b0;

      // Register references to the virtual interface to uvm_config_db
      uvm_config_db#(virtual basic_interface)::set(uvm_root::get(), "*",
"BASIC_IF", basic_if);

      // Start test specified with UVM_TESTNAME
      run_test();
   end

   // Clock generation
   always begin
      #5ns clock <= ~clock;
   end
endmodule
```

Step 3: Create a test

In order to create a test, you need to extend `svaunit_test`. Both stimuli and checking of the SVA will be implemented in the `test()` task according to the scenario.
To access the signals, you will need to get the virtual interface handle that you propagated from the uvm_config_db. As a good practice, you should do it in `build_phase()`.
Also, it is a good practice to initialize the signals to a default value and that can be done either in `setup_test()` or `test()` tasks.
Note that all assertions are enabled by default.

Only a subset of the SVAUnit API will be used in this example:

enable_assertion(sva_name) / enable_all_assertions()	enables a specific/all assertion(s)
disable_assertion(sva_name) / disable_all_assertions()	disables a specific/all assertion(s)
check_sva_passed(sva_name, err, …)	checks if the SVA passed successfully
check_sva_failed(sva_name, err, …)	checks if the SVA failed
check_sva_enabled(sva_name, err, …)	checks if the SVA is enabled

```
class amiq_svaunit_ex_basic_test extends svaunit_test;
	`uvm_component_utils(amiq_svaunit_ex_basic_test)

	// Pointer to interface used to check a scenario
	local virtual basic_interface basic_if;

	function new(string name = "amiq_svaunit_ex_simple_test_unit",
uvm_component parent);
	super.new(name, parent);
	endfunction

	virtual function void build_phase(input uvm_phase phase);
		super.build_phase();

		// Get the interface handle from uvm_config_db
		if(!uvm_config_db#(virtual basic_interface)::get(uvm_root::get(),
	"*", "BASIC_IF", basic_if)) begin
			`uvm_fatal("SVAUNIT_NO_VIF_ERR", $sformatf("SVA interface for
	amiq_svaunit_ex_basic_test is not set!"))
		end
	endfunction
```

```
	virtual task test();
	// Initialize the signals according to the test scenario
		basic_if.sel    <= 1'b0;
		basic_if.ready  <= 1'b0;
		basic_if.enable <= 1'b0;
		basic_if.slverr <= 1'b0;

		// It is good practice to disable all assertions first
		vpiw.disable_all_assertions();

		// And only enable the assertion(s) that will be checked in this
	scenario
		vpiw.enable_assertion("ENABLE_SVA");

		// Drive the signals according to the scenario
		@(posedge basic_if.clk);
		repeat(2) begin
			@(posedge basic_if.clk);
			vpiw.check_sva_passed("ENABLE_SVA", "SVA should have
		passed.");
		end

		// Trigger the error
		basic_if.sel    <= 1'b1;
		basic_if.enable <= 1'b1;
		@(posedge basic_if.clk);
		vpiw.check_sva_failed("ENABLE_SVA", "SVA should have failed");

		// Remove the error
		basic_if.sel <= 1'b0;
		@(posedge basic_if.clk);
		vpiw.check_sva_failed("ENABLE_SVA", "SVA should have failed");
	endtask
endclass
```

Note: To avoid false errors, the checks should be aligned with the sampling moment of the SVA under test.
In the above scenario, the signal values are sampled on the rising edge of clock.

Step 4: Run a test

To run a test, open the sim folder and run the `run_svaunit.sh` script with the necessary arguments.
The arguments you can use are:

-test <name>	specifies a particular test or test suite to run
-seed <value>	specifies a particular seed for the simulation
-i	runs in interactive mode
-tool {ius | questa | vcs}	specifies which simulator to use
-in_reg	specifies if the current invocation is for running a test in regression
-uvm {uvm1.1 | uvm1.2}	specifies the uvm version
-bit[32|64]	specifies the architecture to use (32 or 64 bits)
-compile {yes | no | only} / -c	specifies if compilation should be done
-verbosity <flag>	specifies the verbosity for the messages; the verbosity flag could be one of `UVM_NONE`, `UVM_LOW`, `UVM_MEDIUM`, `UVM_HIGH`, `UVM_FULL` and `UVM_DEBUG`.
-file <name> / -f	specifies the file with an example
-reg	starts a regression

Usage example:

```
./run_svaunit.sh -tool questa -uvm uvm1.1 -f
examples/ex_basic/files.f -top top -test amiq_svaunit_ex_basic_test
-i -c yes
```

Step 5: Report description

SVAUnit's automatically generated report contains the following sections:

- ➔ Project hierarchy: prints the tree containing the tests and sequences; a single test was created in this example, so it is printed as it is

```
------------------- amiq_svaunit_ex_basic_test test suite: Project hierarchy -------------------

amiq_svaunit_ex_basic_test
```

- ➔ Status statistics: prints the final status of each test; the test failed in our case, as expected, with 16/17 checks passed

```
------------------- amiq_svaunit_ex_basic_test : Status statistics -------------------

     *   amiq_svaunit_ex_basic_test SVAUNIT_FAIL (16/17 SVAUnit checks PASSED)
```

- ➔ Exercised SVAs: prints which SVAs were exercised and which not; a single SVA was exercised, as it was discussed in step 1

```
------------------- amiq_svaunit_ex_basic_test: Exercised SVAs -------------------

     1/2 SVA were exercised
            top.basic_if.ENABLE_SVA

     1 SVA were not exercised
            top.basic_if.SLVERR_SVA
```

- ➔ Checks status summary: overall view of the checks used during the simulation; as expected, the last `check_sva_failed` didn't pass while previous checks did

```
------------------- amiq_svaunit_ex_basic_test: Checks status summary -------------------

          CHECK_SVA_EXISTS 9/9 PASSED
          CHECK_SVA_ENABLED 4/4 PASSED
          CHECK_SVA_PASSED 2/2 PASSED
        *   CHECK_SVA_FAILED 1/2 PASSED
```

- ➔ Checks for each SVA statistics: summary of the checks used for each SVA

```
------------------- amiq_svaunit_ex_basic_test: Checks for each SVA statistics -------------------

   *   ENABLE_SVA   16/17 checks PASSED
         *   CHECK_SVA_FAILED 1/2 PASSED
             CHECK_SVA_EXISTS 9/9 PASSED
             CHECK_SVA_ENABLED 4/4 PASSED
             CHECK_SVA_PASSED 2/2 PASSED
```

- ➔ Failed SVAs: lists every failed SVA or prints a success message; our exercised SVA failed, so its name is printed here

```
------------------- amiq_svaunit_ex_basic_test: Failed SVAs -------------------

               *   ENABLE_SVA
```

Note that the failed tests are marked with an asterisk (*).

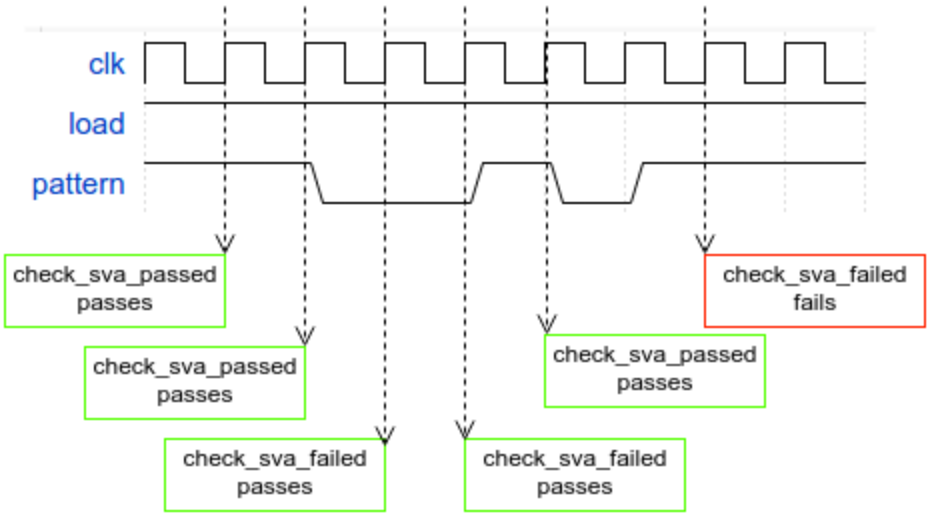
The Extended Example

This example focuses on more advanced features of SVAUnit. It will not be step-based but the approach will be similar to the basic example.

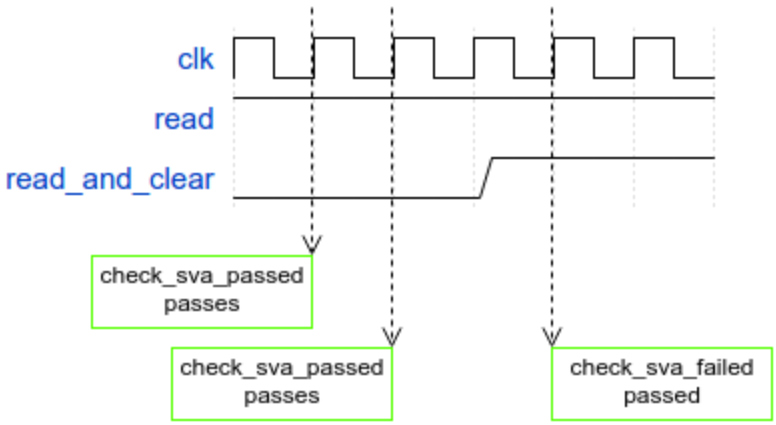
There will be two assertions that you will be testing through different scenarios, each one in a separate interface.

The SVAs are:

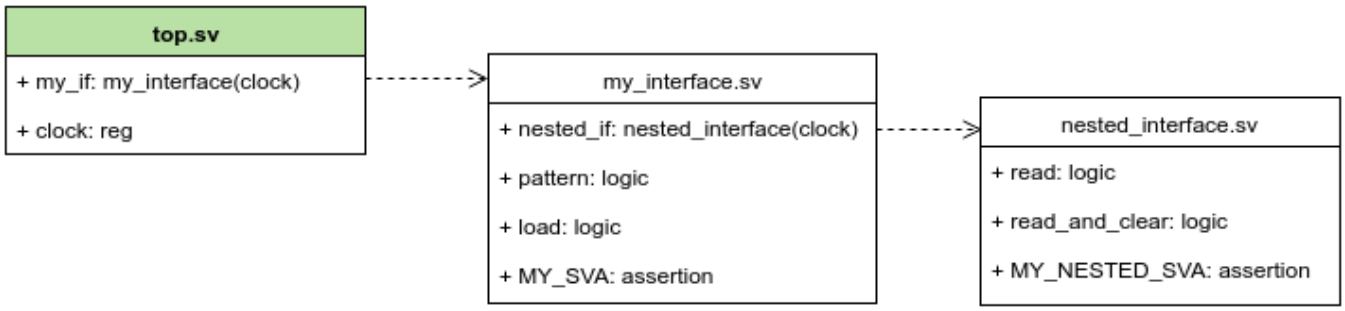
1. “if load is 1, pattern should also be 1 on the same clock cycle” (`MY_SVA`)



2. “read and read_and_clear signals should never be asserted simultaneously” (`MY_NESTED_SVA`)



This is how the testbench looks like:



The interfaces will have a nested hierarchy with nested_interface instantiated under my_interface.
SVAUnit is capable of analyzing all types of instance topologies.

```
interface my_interface (input clk);
      // nested interface instance
      nested_interface nested_if(.clk(clk));

      logic load;
      logic pattern;

      // Property definition for valid load and pattern values
      property my_sva_property;
      @(posedge clk)
                  load |-> pattern;
      endproperty

      // Check that if load is 1, pattern should also be 1 on the same clock
cycle
      MY_SVA: assert property (my_sva_property) else
      `uvm_error("MY_SVA", "If load is 1, pattern should also be 1.")
endinterface



interface nested_interface (input clk);
      logic read;
      logic read_and_clear;

      // Property definition for valid read and read_and_clear values
      property my_sva_property;
      @(posedge clk)
```

```
                ~(read & read_and_clear);
    endproperty

    // Check that read and read_and_clear can not be HIGH at the same time
    MY_NESTED_SVA: assert property (my_sva_property) else
    `uvm_error("MY_NESTED_SVA", "read and read_and_clear can not be asserted
simultaneously.")
endinterface
```

You need to set the virtual interface handle for each interface.

```
module top;
   // Enable SVAUNIT
   `SVAUNIT_UTILS

   reg clock;

   // my_interface instance
   my_interface my_if(.clk(clock));

   initial begin
      // Set clock initial values
      clock = 1'b0;

      // Register references to the virtual interfaces to uvm_config_db
      uvm_config_db#(virtual my_interface)::set(uvm_root::get(), "*", "MY_IF",
my_if);
      uvm_config_db#(virtual nested_interface)::set(uvm_root::get(), "*",
"MY_NESTED_IF", my_if.nested_if);

      // Start test specified with UVM_TESTNAME
      run_test();
   end

   // Clock generation
   always begin
      #5ns clock <= ~clock;
   end
endmodule
```

SVAUnit API

The SVAUnit API can be split into three categories.

API for controlling SVAs

These methods control the enabling and disabling of the assertions.

reset_assertion(sva_name)	Sets the SVA back to the initial state.
disable_assertion(sva_name)	SVA won't start unless enabled.
enable_assertion(sva_name)	SVA can now start.
kill_assertion(sva_name, sim_time)	SVA started at sim_time will not be finished.
disable_step_assertion(sva_name)	Any step callback for this SVA will not be triggered.
enable_step_assertion(sva_name)	Enables step assertion.
reset_all_assertions() disable_all_assertions() enable_all_assertions() kill_all_assertions(sim_time) disable_step_assertions() enable_step_assertions()	Similar to the above ones, but apply to all the 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_of_all_assertions()
system_off_all_assertions()	The SVA system will not start again 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.

API for checking SVAs

If one check fails then the corresponding test fails. The check API requires the name of the SVA under test. Optionally, you can provide a custom error message to override the default one.

check_sva_exists(sva_name, err_msg, ...)	Checks if the SVA exists.
check_sva_enabled(sva_name, err_msg, ...)	Checks if the SVA is enabled.

check_sva_disabled(sva_name, err_msg, …)	Checks if the SVA is disabled.
check_sva_passed(sva_name, err_msg, ...)	Checks if the SVA finished successfully.
check_sva_failed(sva_name, err_msg, ...)	Checks if the SVA failed.
check_all_sva_passed(sva_name, err_msg, ...)	Checks if all the SVAs finished successfully.
check_that(expression, err_msg, ...)	Checks if the expression is true.

API for printing reports

The reports are printed after a test or test suite has finished.

print_status()	Displays the status of a test or a test suite.
print_sva()	Displays 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()	Displays how many checks are used and how many are not along with their names.
print_sva_and_checks()	Displays all SVAs along with the checks used to test them.
print_tests()	Displays the tests that have run in the simulation. It can be used inside a test suite.
print_tree()	Displays the created SVAUnit topology.
print_report()	Displays all the above reports.
print_sva_info(sva_name)	Displays informations regarding the selected SVA.

Advanced tests

A test must always extend `svaunit_test`. In `build_phase` you should get your virtual interface that you are going to test using `uvm_config_db`. you also have the option to disable the test here, which is enabled by default.

The test scenario can be simulated either in a test (in its `test()` task) or a sequence (in its `body()` task). If you decide to write a test, you don't need to create sequences and you can run it as it is (the basic example).
However, if you decide to write a sequence-based scenario, you have two options:

- ➔ create a test which encapsulates the sequence and in its `test()` task, start the sequence by either calling `` `uvm_do `` or `start()`
- ➔ add the sequence to the test suite, and a test will be auto-generated according to your scenario

As a good practice, you should initialize the signals in your tests to avoid signal value propagation. When you have more tests they will run sequentially and the signals could propagate from one test to another.
It is recommend that only one assertion is enabled per test for easy debugging, but SVAUnit can handle any number of enabled assertions per test.
When referring to SVAs inside checks, you can use either full paths ("`top.my_if.MY_SVA`") or only their name ("`MY_SVA`").

Test without sequences

The whole scenario that you want to test will be included in the `test()` task. The structure of the test is identical with the one in the basic example. For this example, we'll be exercising `MY_SVA` in this test, created in a class name `amiq_svaunit_ex_extended_test_no_sequence` .

```
virtual task test();
      vpiw.disable_all_assertions();
      vpiw.enable_assertion("MY_SVA");

      // Initialization for this particular test
      my_if.pattern <= 1'b1;
      my_if.load    <= 1'b1;


      @(posedge my_if.clk);
      repeat(2) begin
      @(posedge my_if.clk);
      vpiw.check_sva_passed("MY_SVA", "The assertion should have passed");
      end

      // Trigger error
      my_if.pattern <= 1'b0;
      repeat(2) begin
      @(posedge my_if.clk);
      vpiw.check_sva_failed("top.my_if.MY_SVA", "The assertion should have
failed");
```

```
        end

        // Remove the error
        my_if.pattern <= 1'b1;
        @(posedge my_if.clk);
        vpiw.check_sva_passed("MY_SVA", "The assertion should have passed");
        my_if.pattern <= 1'b0;
        @(posedge my_if.clk);
        my_if.pattern <= 1'b1;
        @(posedge my_if.clk);
        vpiw.check_sva_failed("MY_SVA", "The assertion should have failed");
        @(posedge my_if.clk);
endtask
```

The `pre_test()` task has been deprecated since the 2.0 release.

Test with sequences

SVAUnit supports sequences as a stimuli driver and SVA checking. As mentioned in the previous step, you also have the option to create a sequence-based scenario, which you can encapsulate in a test or use as it is in a test suite. In order to use this feature, you will need to create a class that extends `svaunit_base_sequence` and includes the scenario in its `body()` task.
Note that you do not have to declare any sequencer, SVAUnit takes care of that.

Example sequence for your nested SVA:

```
class amiq_svaunit_ex_extended_test_sequence extends svaunit_base_sequence;
        `uvm_object_utils(amiq_svaunit_ex_extended_test_sequence)

        // Reference to virtual interface containing MY_SVA
        local virtual nested_interface my_nested_if;

        // Constructor where the nested interface handle is populated from
uvm_config_db

        // Create scenarios for MY_NESTED_SVA
        virtual task body();
                vpiw.disable_all_assertions();
                vpiw.enable_assertion("MY_NESTED_SVA");

                // Initialization for this particular test
                my_nested_if.read           <=1'b1;
```

```
            my_nested_if.read_and_clear <=1'b0;

            @(posedge my_nested_if.clk);
            repeat(2) begin
            @(posedge my_nested_if.clk);
            vpiw.check_sva_passed("MY_NESTED_SVA", "The assertion should have
      passed");
            end

            my_nested_if.read_and_clear <=1'b1;
            @(posedge my_nested_if.clk);
            vpiw.check_sva_failed("top.my_if.nested_if.MY_NESTED_SVA", "The
      assertion should have failed");
      endtask
endclass
```

The test code is:

```
class amiq_svaunit_ex_extended_test_sequence extends svaunit_test;
      `uvm_component_utils(amiq_svaunit_ex_extended_test_sequence)

      // Pointer to sequence used to check a scenario
      local amiq_svaunit_ex_extended_sequence seq;

      // Constructor
      // Build phase which creates the sequence
      virtual function void build_phase(input uvm_phase phase);
      super.build_phase(phase);
            seq = amiq_svaunit_ex_extended_sequence::type_id::
                                          create("seq", this);
      endfunction

      // Create scenarios for MY_NESTED_SVA
      virtual task test();
      seq.start(sequencer);
            // alternatively: `uvm_do(seq)
      endtask
endclass
```

Parameterized tests

You can also have parameterized tests if you use the `SVAUNIT_TEST_WITH_PARAM_UTILS macro inside your test class. In this example, the default argument is 10, but you can set it to your needs.

```
class amiq_svaunit_ex_extended_test_with_parameter#(int unsigned A_PARAM = 10)
extends svaunit_test;
	`uvm_component_param_utils(amiq_svaunit_ex_extended_test_with_parameter#(
A_PARAM))
	`SVAUNIT_TEST_WITH_PARAM_UTILS

	// Reference to virtual interface containing MY_SVA
	local virtual nested_interface my_nested_if;

	// Constructor and build_phase where the nested interface handle is
populated from uvm_config_db

	// Create scenarios for MY_NESTED_SVA
	virtual task test();
		vpiw.disable_all_assertions();
		vpiw.enable_assertion("MY_NESTED_SVA");

		// Initialization for this particular test
		my_nested_if.read           <=1'b1;
		my_nested_if.read_and_clear <=1'b0;

		@(posedge my_nested_if.clk);
		my_nested_if.read_and_clear <=1'b1;

		repeat(A_PARAM) begin
		@(posedge my_nested_if.clk);
		vpiw.check_sva_failed("MY_NESTED_SVA", "The assertion should have
	failed.");
		end
	endtask
endclass
```

Test suites

For easier management of the testing environment, SVAUnit has the ability to encapsulate more tests in a test suite. You need to create a class that extends `svaunit_test_suite` and in its `build_phase()` you need to use the `` `add_test() `` macro for every test or sequence you want to run.
A test suite supports all scenarios mentioned in the previous sections:

- ➔ a sequence
- ➔ a test that runs a sequence
- ➔ a test that simulates the scenario without a sequence

Here is an example:

```
class amiq_svaunit_ex_extended_test_suite extends svaunit_test_suite;
   `uvm_component_utils(amiq_svaunit_ex_extended_test_suite)

   function new(string name = "amiq_svaunit_ex_extended_test_suite",
uvm_component parent);
      super.new(name, parent);
   endfunction

   virtual function void build_phase(input uvm_phase phase);
      super.build_phase(phase);

      // Register unit tests and sequences to test suite
      `add_test(amiq_svaunit_ex_extended_test_no_sequence)
      `add_test(amiq_svaunit_ex_extended_sequence)
      `add_test(amiq_svaunit_ex_extended_test_sequence)
      `add_test(amiq_svaunit_ex_extended_test_with_paramete#(3)
      `add_test(amiq_svaunit_ex_extended_test_with_paramete#(4)
      `add_test(amiq_svaunit_ex_extended_test_sequence)
   endfunction
endclass
```

The `amiq_svaunit_ex_simple_test_suite` contains and starts the following tests:

- ➔ a test without using a sequence (`test_no_sequence`) that exercises `MY_SVA` and should fail at the last `check_sva_failed`
- ➔ a sequence (`sequence`) that exercises `MY_NESTED_SVA` and should finish successfully
- ➔ two tests (`test_sequence`) using a sequence (`sequence`)
- ➔ two parameterized tests (`test_with_parameter`) that exercises `MY_NESTED_SVA` and should finish successfully

Report filtering

In some cases, like the extended example, the output could be large and hard to understand. As shown in **Step 4: Running tests**, you have the option to set the verbosity of the `uvm` messages.

UVM_NONE	prints the checks for each SVA statistics and the list of failed SVAs
UVM_LOW	prints the status statistics, the checks status summary and the above ones

UVM_MEDIUM (default)	prints the test suite hierarchy, the exercised SVAs and the above ones
UVM_DEBUG	prints debug messages and the above ones

Running the extended example with
`./run_svaunit.sh -tool questa -uvm uvm1.1 -f examples/ex_basic/files.f -top top -test amiq_svaunit_ex_basic_test -i -c yes -verbosity UVM_NONE`
will produce the next output:

```
------------------- amiq_svaunit_ex_advanced_test_suite: Checks for each SVA statistics --------------------

        MY_NESTED_SVA   95/95 checks PASSED
                CHECK_SVA_EXISTS 51/51 PASSED
                CHECK_SVA_ENABLED 22/22 PASSED
                CHECK_SVA_PASSED 10/10 PASSED
                CHECK_SVA_FAILED 12/12 PASSED

    *   MY_SVA   24/25 checks PASSED
            *   CHECK_SVA_FAILED 2/3 PASSED
                CHECK_SVA_EXISTS 13/13 PASSED
                CHECK_SVA_ENABLED 6/6 PASSED
                CHECK_SVA_PASSED 3/3 PASSED


UVM_INFO @ 286000 ns [amiq_svaunit_ex_advanced_test_suite]:

------------------- amiq_svaunit_ex_advanced_test_suite: Failed SVAs --------------------

            *   MY_SVA
```

As expected, there is only one failed check, `check_sva_failed`, for `MY_SVA` from the test without sequences.