SVUnit

The SVUnit Authors

Mar 30, 2026

CONTENTS:

1 Contents **3**
1.1 Structure and Workflow . . . 3
1.2 Installation and Setup . . . 4
1.3 Creating a Unit Test Template . . . 4
1.4 Writing Unit Tests . . . 5
1.5 Special Considerations for Unit Testing UVM Components . . . 9
1.6 Running Unit Tests . . . 10
1.7 Simulator Output . . . 13
1.8 Experimental Features . . . 13
1.9 Support and More Information . . . 14
1.10 User Guide Feedback . . . 14

SVUnit is a simple, well-structured verification framework intended for design and verification engineers writing and running tests against Verilog modules, classes or interfaces. By first verifying the individual pieces of a design or testbench in isolation, we can assume much higher quality of the design or testbench as a whole. The feature characteristic of SVUnit is ease-of-use, meaning developers can be up-and-running in an hour or less. SVUnit is open-source under Apache 2.0 licensing making it accessible and productive for any engineer with a Verilog simulator.

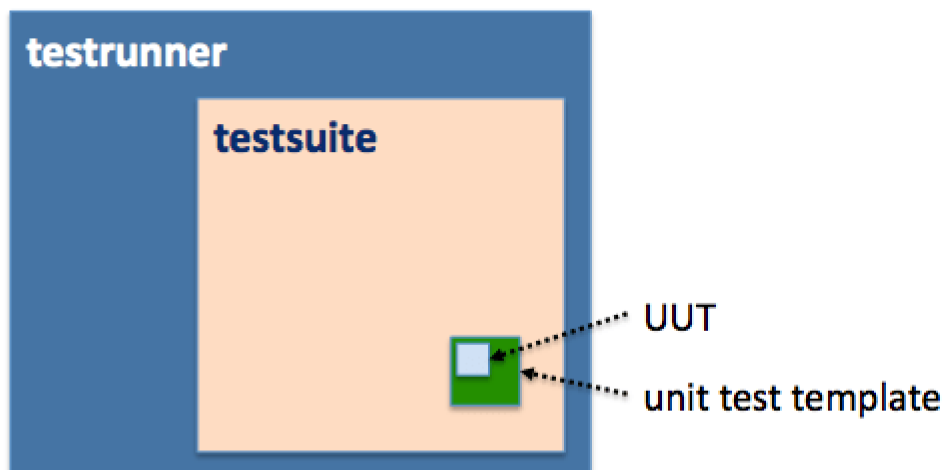
CHAPTER

ONE

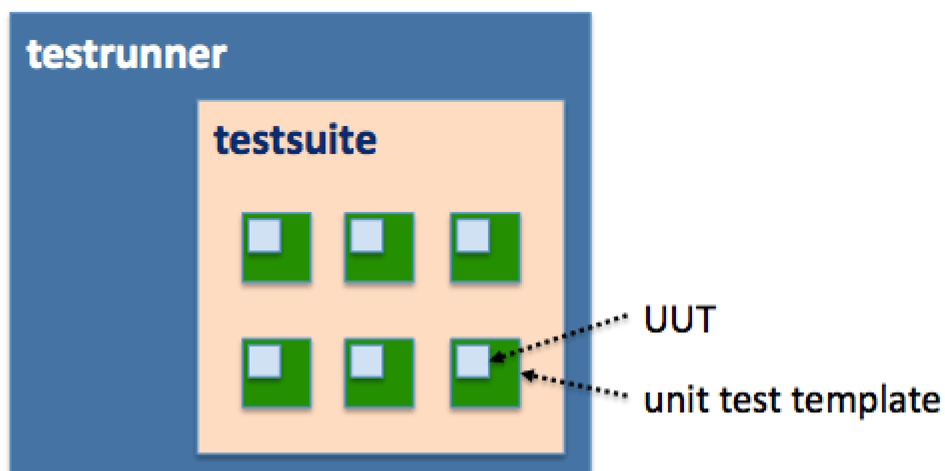
CONTENTS

1.1 Structure and Workflow

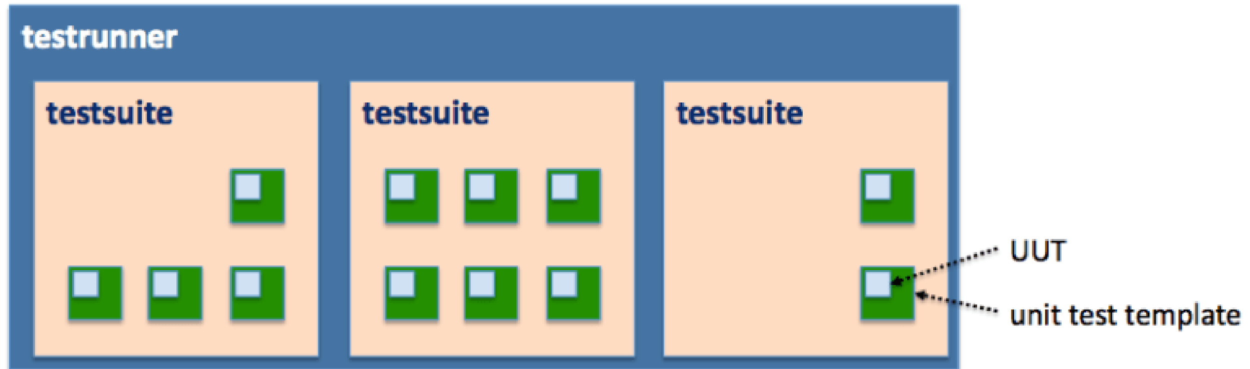
The most basic SVUnit arrangement includes a single unit test template with 1 or more unit tests against a single UUT.



A more likely arrangement is to have several unit test templates in a single directory, each containing unit tests for one of several UUTs. As examples, a design engineer building a subsystem with N modules would have N unit test templates running within a single test suite. Likewise, a verification engineer building a testbench with N transactors would also have N unit test templates running within a single test suite.



A final, large scale arrangement is one with a test suite for each of M subsystems where each subsystem contains any number of unit test templates and the unit test templates for each subsystem are located in different directories.



Hierarchy is derived from file/directory structure where all unit test templates located within the same directory are grouped into a test suite.

1.2 Installation and Setup

Download to setup is about a 3 minute procedure:

1. Download the latest version of SVUnit from GitHub
2. Extract the SVUnit archive:

```
unzip svunit-code-master.zip
```

3. Source the SVUnit setup script (bash users source Setup.bsh. csh users source Setup.csh):

```
cd svunit-code-master
source Setup.bsh
```

1.3 Creating a Unit Test Template

New unit test templates are created using create_unit_test.pl. Usage of create_unit_test.pl is as follows:

```
Usage:  create_unit_test.pl [ -help | -uvm | -out <file> | -overwrite | uut.sv ]

Where -help                   : prints this help screen
      -uvm                    : generate a uvm component test template
                                IMPORTANT: do not use '-uvm' unless the UUT is derived from
↪a uvm_component
      -out <file>             : specifies a new default output file
      -overwrite              : overwrites the output file if it already exists
      -class_name <name>      : generate a unit test template for a class <name>
      -module_name <name>     : generate a unit test template for a module <name>
      -if_name <name>         : generate a unit test template for an interface <name>
      uut.sv                  : the file with the unit under test
```

A template can be generated by using any of the -class_name, -module_name, -if_name switches or by specifying a verilog file that contains a definition of the UUT. For example, you can generate a new unit test template for a class called 'foo' with:

```
create_unit_test.pl -class_name foo
```

Likewise, if you already have a definition for class 'foo' in 'foo.v', you can generate a corresponding unit test template with:

```
create_unit_test.pl foo.v
```

The default output for create_unit_test.pl is written to ./<name>_unit_test.sv. The default can be overridden, however, using the -out <file> switch. A different file name and/or directory can be specified as required. The file name, however, must follow the <name>_unit_test.sv format.

Existing files will not be overwritten unless the -overwrite switch is used.

1.3.1 Integrating the UUT

The template generated by create_unit_test.pl includes an instance of the UUT as well as other important parts of the infrastructure. The UUT does not, however, include any pin level connectivity so connectivity is handled by users. For example, if a template is created for a module named 'test_module', an instance of 'test_module' is created as:

```
//===================================
// This is the UUT that we're
// running the Unit Tests on
//===================================
test_module my_test_module();
```

If test_module has two outputs, oBus and oPin, and one input, iPin, a user connects them as normal Verilog IO thereby making them accessible within the unit test template:

```
//===================================
// This is the UUT that we're
// running the Unit Tests on
//===================================
reg        iPin;
wire [7:0] oBus;
wire       oPin;
test_module my_test_module(.iPin(iPin),
                           .oBus(oBus),
                           .oPin(oPin));
```

1.4 Writing Unit Tests

The unit test template includes embedded instructions on structure and position of unit tests:

```
//===================================
// All tests are defined between the
// SVUNIT_TESTS_BEGIN/END macros
//
// Each individual test must be
```

(continues on next page)

(continued from previous page)

```
// defined between `SVTEST(_NAME_)
// `SVTEST_END
//
// i.e.
//   `SVTEST(mytest)
//     <test code>
//   `SVTEST_END
//===================================
`SVUNIT_TESTS_BEGIN



`SVUNIT_TESTS_END
```

Multiple unit tests can be defined in a unit test template. All must be defined between the `SVUNIT_TESTS_BEGIN and `SVUNIT_TESTS_END macros. Individual unit tests are defined using the `SVTEST(<name>) and `SVTEST_END macros. The test <name> must be a valid verilog code block label (i.e. any alphanumeric starting with [_a-zA-Z]). For example, a template with 2 tests called test0 and test1 would be declared as:

```
`SVUNIT_TESTS_BEGIN

`SVTEST(test0)
`SVTEST_END

`SVTEST(test1)
`SVTEST_END

`SVUNIT_TESTS_END
```

The macros expand to a Verilog code block so any code that is legal within a code block can be used within a unit test. Other required variables, declarations, functions, tasks, etc must be defined outside the BEGIN/END macros. For example, the function helper() can be defined and used within a unit test as:

```
`SVUNIT_TESTS_BEGIN

`SVTEST(test0)
`SVTEST_END

`SVTEST(test1)
  helper();
`SVTEST_END

`SVUNIT_TESTS_END

function helper();
endfunction
```

SVUnit imposes no limit on the number of unit tests that can be defined with a unit test template.

1.4.1 SVUnit Reporting Macros

SVUnit includes an integrated reporting mechanism such that the exit PASS/FAIL status of every unit test is collected, reported and used to report a cumulative result. To set PASS/FAIL status, SVUnit defines several logging macros that are integrated with the reporting structure:

```
`define FAIL_IF(exp)
`define FAIL_UNLESS(exp)

`define FAIL_IF_EQUAL(a,b)
`define FAIL_UNLESS_EQUAL(a,b)

`define FAIL_IF_STR_EQUAL(a,b)
`define FAIL_UNLESS_STR_EQUAL(a,b)
```

The most commonly used macros are `FAIL_IF and `FAIL_UNLESS that take a single boolean expression as input. The `FAIL_IF_EQUAL and `FAIL_UNLESS_EQUAL macros exit based on an '===' comparison of boolean inputs a and b. Likewise, `FAIL_IF_STR_EQUAL and `FAIL_UNLESS_STR_EQUAL do a string comparison between inputs a and b.

1.4.2 Setting Test Exit Status

The reporting macros can be used to verify outputs and response of a UUT and set test exit status according. For example, we can use the `FAIL_IF and `FAIL_UNLESS_EQUAL macros to verify oBus and oPin are driven to the proper state:

```
`SVUNIT_TESTS_BEGIN

`SVTEST(test0)
  `FAIL_IF(oPin !== 1)
`SVTEST_END

`SVTEST(test1)
  `FAIL_UNLESS_EQUAL(oBus, 'h6)
`SVTEST_END

`SVUNIT_TESTS_END
```

If the conditions described by the macros in either test0 or test1 are not satisfied, the test fails with an assert error and is reported as having failed. Tests are killed immediately at first failure so any code appearing after the failing assert statement does not execute.

1.4.3 Interacting with the UUT

Tests can interact with the UUT using simple procedural assignments to inputs or through helper functions and tasks for more complex interactions. For example, if the state of the oPin and oBus outputs is conditional based on the state of the iPin, oPin and oBus can be verified by driving iPin as necessary:

```
`SVTEST(test0)
  iPin = 1;
  #0 `FAIL_IF(oPin !== 1)
`SVTEST_END
```

(continues on next page)

(continued from previous page)

```
`SVTEST(test1)
  helper();
  #0 `FAIL_UNLESS_EQUAL(oBus, 'h6)
`SVTEST_END

function helper();
  iPin = 1;
endfunction
```

Note: The addition of the #0 assumes oPin and oBus are asynchronous outputs that require a delta cycle be consumed before they reach their intended state.

1.4.4 Test Setup and Teardown

For behaviour that is repeated before and after every test, the setup() and teardown() tasks in the unit test template are intended to group any logic that is repeated before and/or after every test - the setup() task is run before every test and the teardown() task is run after every test. For example, if the default state of iPin is logic 1, that assignment can be done in the setup task rather than each individual test:

```
//===================================
// Setup for running the Unit Tests
//===================================
task setup();
  svunit_ut.setup();
  /* Place Setup Code Here */
  iPin = 1;
endtask
```

As a result of moving the 'iPin = 1' assignment to the setup() task, test0 and test1 can be simplified to:

```
`SVTEST(test0)
  #0 `FAIL_IF(oPin !== 1)
`SVTEST_END

`SVTEST(test1)
  #0 `FAIL_UNLESS_EQUAL(oBus, 'h6)
`SVTEST_END
```

It is recommended that common initialization code be contained in the setup() task. Reset sequence or register initialization, for example, is common logic that should be included in the setup() task. As well, it is recommended that any general cleanup of the UUT or unit test harness be grouped in the teardown() task to avoid polluting the state space for subsequent tests (i.e teardown() is for "cleaning the slate").

1.5 Special Considerations for Unit Testing UVM Components

UVM components require additional infrastructure found in the svunit_uvm_mock_pkg. An additional switch, therefore, is included with create_unit_test.pl to generate a UVM component specific test case template. If, for example, class blah is derived from uvm_component, create_unit_test.pl would be invoked as:

```
create_unit_test.pl -uvm -class_name blah
```

The UVM component test case template includes a UUT wrapper that can be used to instantiate any additional unit test infrastructure, like connections to TLM FIFOs or analysis ports, if required. The UUT wrapper in the test case template is output as:

```
class blah_uvm_wrapper extends blah;

  `uvm_component_utils(blah_uvm_wrapper)
  function new(string name = "blah_uvm_wrapper", uvm_component parent);
    super.new(name, parent);
  endfunction

  //===================================
  // Build
  //===================================
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    /* Place Build Code Here */
  endfunction

  //==================================
  // Connect
  //=================================
  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    /* Place Connection Code Here */
  endfunction
endclass
```

The template also includes calls to functions svunit_activate_uvm_component, svunit_deactivate_uvm_component, svunit_uvm_test_start and svunit_uvm_test_finish in the build, setup and teardown tasks to integrate the UVM runflow with the sequential test running behaviour of SVUnit:

```
//===================================
// Build
//===================================
function void build();
   svunit_ut = new(name);

   my_blah = blah_uvm_wrapper::type_id::create("", null);

   svunit_deactivate_uvm_component(my_blah);
endfunction



//===================================
```

(continues on next page)

(continued from previous page)

```
// Setup for running the Unit Tests
//===================================
task setup();
   svunit_ut.setup();
   /* Place Setup Code Here */

   svunit_activate_uvm_component(my_blah);

   //-----------------------------
   // start the testing phase
   //-----------------------------
   svunit_uvm_test_start();
endtask


//===================================
// Here we deconstruct anything we
// need after running the Unit Tests
//===================================
task teardown();
   svunit_ut.teardown();
   //-----------------------------
   // terminate the testing phase
   //-----------------------------
   svunit_uvm_test_finish();

   /* Place Teardown Code Here */

   svunit_deactivate_uvm_component(my_blah);
endtask
```

The activate/deactivate functions are used to isolate the component in the uvm_domain such that multiple uvm_components can be tested in series. The start/finish functions are used to invoke phase jumping such that unit tests against against a component can be run iteratively.

1.6 Running Unit Tests

SVUnit unit tests are run using runSVUnit. Usage of runSVUnit is as follows:

```
Usage:  runSVUnit [-s|--sim <simulator> -l|--log <log> -d|--define <macro> -f|--filelist
↪<file> -U|-uvm -m|-mixedsim <vhdlfile>
                  -r|--r_arg <option> -c|--c_arg <option> -o|--out <dir> -t|--test <test>
↪ --filter <filter>]
  -s|--sim <simulator>     : simulator is either of questa, modelsim, riviera, ius,␣
↪xcelium, vcs, dsim or xsim
  -l|--log <log>           : simulation log file (default: run.log)
  -d|--define <macro>      : appended to the command line as +define+<macro>
  -f|--filelist <file>     : some verilog file list
  -r|--r_arg <option>      : specify additional runtime options
  -c|--c_arg <option>      : specify additional compile options
```

(continues on next page)

(continued from previous page)

```
  -e|--e_arg <option>      : specify additional elaboration options
  -U|--uvm                 : run SVUnit with UVM
  -o|--out                 : output directory for tmp and simulation files
  -t|--test                : specifies a unit test to run (multiple can be given)
  -m|--mixedsim <vhdlfile> : consolidated file list with VHDL files and command line␣
↪switches
  -w|--wavedrom            : process json files as wavedrom output
     --filter <filter>     : specify which tests to run, as <test_module>.<test_name>
  -h|--help                : prints this help screen
```

1.6.1 Choosing a Simulator

SVUnit can be run using most commonly used EDA simulators using the '-s' switch. Supported simulators currently include Mentor Graphics Questa, Cadence Incisive, Synopsys VCS and Aldec Riviera PRO.

1.6.2 Logging

By default, simulation output is written to run.log. The default location can be overridden using the '-l' switch.

1.6.3 Specifying Command-line Macros

SVUnit will pass command line macro defines specified by the '-d' switch directly to the simulator.

1.6.4 Adding Files For Simulation

Through the use of `include directives, both the unit test template and corresponding UUT file are included in compilation making it possible to build and verify on simple designs without any need to specify or maintain file lists. As designs grow, however, more files can be added using standard simulator file lists and the '-f' switch.

Note: The file svunit.f is automatically included for compilation provided it exists. Thus, files can be added to svunit.f without having to specify '-f svunit.f' on the command line.

1.6.5 Adding Run Time and/or Compile and/or Elaboration Options

It is possible to specify compile and run time options using the '-c_arg', '-e_arg' and '-r_arg' switches respectively. All compile, elaboration and run time arguments are passed directly to the simulator command line.

1.6.6 Enable UVM Component Unit Testing

For verification engineers unit testing UVM-based components, the '-U' switch must be specified to include relevant run-flow handling.

1.6.7 Specifying a Simulation Directory

By default, SVUnit is run in the current working directory. However, to avoid mixing source files with simulation output, it is possible to change the location where SVUnit is built and simulated using the '-o' switch. It is an error to use the '-o' switch to runSVUnit that doesn't exist.

1.6.8 Specifying Unit Tests to be Run

By default, runSVUnit finds and simulates all unit test templates within a given parent directory. For short runs, this is recommended practice. However, if simulation times grow to the point where they are long and cumbersome, it is possible to specify specific unit test templates to be run using the '-t' switch. For example, if a parent directory has 12 unit test templates but you only want to run mine_unit_test.sv, you can use the '-t' switch as:

```
runSVunit -t mine_unit_test.sv -s <your simulator>
```

The '-t' switch can be used to specify multiple unit test templates as:

> runSVunit -t mine_unit_test.sv -t yours_unit_test.sv -s <your simulator>

It's also possible to restrict which individual tests should run. This is done using the '–filter' option.

The following call runs only some_test defined in some_testcase:

```
runSVUnit --filter some_testcase.some_test
```

The following call runs all tests called some_test regardless of which testcase they are defined in:

```
runSVUnit --filter *.some_test
```

The following call runs all tests defined in some_testcase:

```
runSVUnit --filter some_testcase.*
```

The previous command is conceptually similar to using the '-t' option. While the runtime behavior is the same, it is slightly different in terms of what gets compiled. Using '-t' selects what gets compiled and by extension limits what can be run. Using '–filter' only affects which of the tests that were compiled should run, but doesn't control what gets compiled. Both options are useful, as they serve different purposes. The '-t' option is helpful when API changes would require modifications to many unit test files, but you would like to update them one after the other. It is also a very blunt tool, as compilation can only be handled at the file level. The '–filter' option can be used to focus on finer subsets of tests.

1.6.9 Listing Test Names

It is possible to list available tests without actually running them:

```
runSVUnit --list-tests
```

1.7 Simulator Output

Using built-in logging macros, the logged SVUnit output shows step-by-step run status for each test, unit test template and test suite as well as a cumulative result for the testrunner.

```
INFO:  [0][__ts]: Registering Unit Test Case test_module_ut
INFO:  [0][testrunner]: Registering Test Suite __ts
INFO:  [0][__ts]: RUNNING
INFO:  [0][test_module_ut]: RUNNING
INFO:  [0][test_module_ut]: test0::RUNNING
INFO:  [0][test_module_ut]: test0::PASSED
INFO:  [0][test_module_ut]: test1::RUNNING
INFO:  [0][test_module_ut]: test1::PASSED
INFO:  [0][test_module_ut]: PASSED (2 of 2 tests passing)

INFO:  [0][__ts]: PASSED (1 of 1 testcases passing)

INFO:  [0][testrunner]: PASSED (1 of 1 suites passing) [SVUnit v3.6]
```

test suite name
unit test template name
unit test name
unit test PASS/FAIL status
unit test template cumulative status
test suite cumulative status
test runner cumulative status

1.8 Experimental Features

SVUnit provides a more streamlined way of defining tests, which requires less boilerplate code. This is an experimental feature, which is still under development and is not part of the public API.

To activate this feature, use the *–enable-experimental* option of the *runSVUnit* script.

Here's an example of such a streamlined test:

```
package factorial_test;

  import factorial::*;

  `include "svunit.svh"
  `include "svunit_defines.svh"


  `TEST_BEGIN(handles_zero_input)
    `FAIL_UNLESS(factorial(0) == 1)
  `TEST_END


  `TEST_BEGIN(handle_positive_input)
    `FAIL_UNLESS(factorial(1) == 1)
    `FAIL_UNLESS(factorial(2) == 2)
```

(continues on next page)

(continued from previous page)

```
    `FAIL_UNLESS(factorial(3) == 6)
    `FAIL_UNLESS(factorial(8) == 40320)
  `TEST_END

endpackage
```

1.9 Support and More Information

To connect with other users, ask questions and share experience, join the SVUnit User Group . Everyone is welcome!

1.10 User Guide Feedback

It'd be a big help for me to get your feedback and direction so I can make sure the user guide is actually helping SVUnit users. Please let me know what you think!