Specification Vs Verification Matrix - Quick Reference

UVVM Support Component

The Specification vs Verification Matrix feature is an efficient method of verifying the requirement specification.

The first page of this QuickRef is just for references - for an introduction to the Specification vs Verification Matrix concept, please see page 2 of this QuickRef.

UVVM VHDL Methods
log_req_cov(requirement, testcase[, passed][, fail_on_alert_mismatch_severity])
start_req_cov(req_to_tc_map_file, output_file)
end_req_cov(VOID)

Script Usage – run_spec_vs_verif.py

Call the run_spec_vs_verif.py from a terminal, using e.g.:

- *python run_spec_vs_verif.py <see Table 12 for script arguments>*

If the output parameter is specified, the output will be placed in this directory. If the output parameter is not specified, the current directory will be used. Python 3.x required.

File Formats – Basic usage

Required files

File	Requirement to Testcase map file	CSV	Output from UVVM test suite	CSV	Output from run_spec_vs_verif.py	CSV
Layout	*"Requirement"; "Description" [; "Testcase"]*		*"Requirement"; "Testcase"; "PASS/FAIL"*		*"Requirement"; "COMPLIANT / NON COMPLIANT"*	
Example	FPGA_SPEC_1; UART all bits 0 ; TC_UART_1 FPGA_SPEC_2; UART start bit polarity; TC_UART_2 FPGA_SPEC_3; UART start bit delay; TC_UART_3		FPGA_SPEC_1; TC_UART_1; PASS FPGA_SPEC_2; TC_UART_2; PASS FPGA_SPEC_3; TC_UART_3; FAIL		FPGA_SPEC_1; COMPLIANT FPGA_SPEC_2; COMPLIANT FPGA_SPEC_3; NON COMPLIANT	

Optional files

File	Requirement to sub-requirement map file (**optional**)	CSV	Test result file list (**optional**)	txt	Script config file (**optional**)	txt
Layout	*"Requirement"; "Sub-requirement"; "Sub-requirement"; …;"Sub-requirement"*		*"path/to/result_file.csv"*		*--argument value OR -a value*	
Example	FPGA_SPEC_1; FPGA_SPEC_1.a; FPGA_SPEC_1.b		path/to/first/ result_file.csv path/to/second/result_file.csv third_result_file.csv		--requirements path/to/requirements.csv -F test_suite_input_files.txt --subreqs path/to/subrequirements.csv	

VHDL 2008 only

Specification vs Verification Matrix Concept

An important step of design verification is to check that all requirements have been met. Requirements can be very different depending on the project management, the product sector and standardized design methodology. In some projects requirements hardly exists, and the functionality is based on a brief description. However, in projects where safety and reliability is key the requirements are an essential part of the project management flow. In some standards the requirements and the corresponding test cases that verifies the requirements need to be defined, reviewed and accepted by a third party assessor before even starting the verification flow. This UVVM Verification Component is intended for projects where requirements are essential in the work flow. Examples of a requirement can be seen in Table 1.

Table 1 Requirement examples

Requirement Number	Description
FPGA_TOP_SPEC_1	The device UART interface shall accept a baud rate of 9600kbps.
FPGA_TOP_SPEC_2	The device reset shall be active low.

A typical verification flow starts with the verification engineer specifying several testcases to verify all requirements. Simplified test cases mapped to the requirements in Table 1 is listed in Table 2. Not that this test case description does not specify in detail how the test is performed. Requirements related to this varies between standards and methodologies.

Table 2 Test case to requirement mapping examples

Test case	Verifies requirement	Description (simplified!)
TC_UART_BAUDRATE	FPGA_TOP_SPEC_1	A set of random UART messages shall be sent to the DUT UART interface, with a baud rate of 9600kbps. It will be verified that all messages are received correctly by the DUT.
TC_TOP_RESET	FPGA_TOP_SPEC_2	The reset will be set low and it will be verified that all I/O are set in their defined reset state. The reset shall then be set high and it will be verified that the DUT can be operated as normal.

After all testcases passes, the verification engineer will write a report stating that all the requirements have been verified. One of the key components in a verification report is the compliancy statements, which is a requirement to test case mapping. An example of this can be seen in Table 3.

Table 3 Requirement to test case mapping in the verification report

Requirement Number	Tested by test case	Compliancy
FPGA_TOP_SPEC_1	TC_UART_BAUDRATE	COMPLIANT
FPGA_TOP_SPEC_2	TC_TOP_RESET	COMPLIANT

In many cases it is sufficient to do this process manually. However, when working on larger projects with multiple re-iterations or test cases that are constantly changing, it is a good idea to automate this process. By outsourcing requirement compliancy reporting to the VHDL testbench, the verification engineer does not have to constantly evaluate each testcase to verify that it still addresses the requirement it is supposed to verify. Additionally, it will save time for the verification engineer if a report is generated where each requirement is listed as compliant or non-compliant. This list can be used directly in the verification report.

The UVVM Specification vs Verification Matrix support component is an extension to the UVVM package, consisting of a set of new VHDL methods for logging requirement compliancy and a post-processing Python script for generating a report based on a requirement specification and the simulation results. Figure 1 shows an example of basic Specification vs Verification Matrix usage. Table 4 gives a description of each step in Figure 1. See the front-page of this QuickRef for a description of the input-files used in each step.

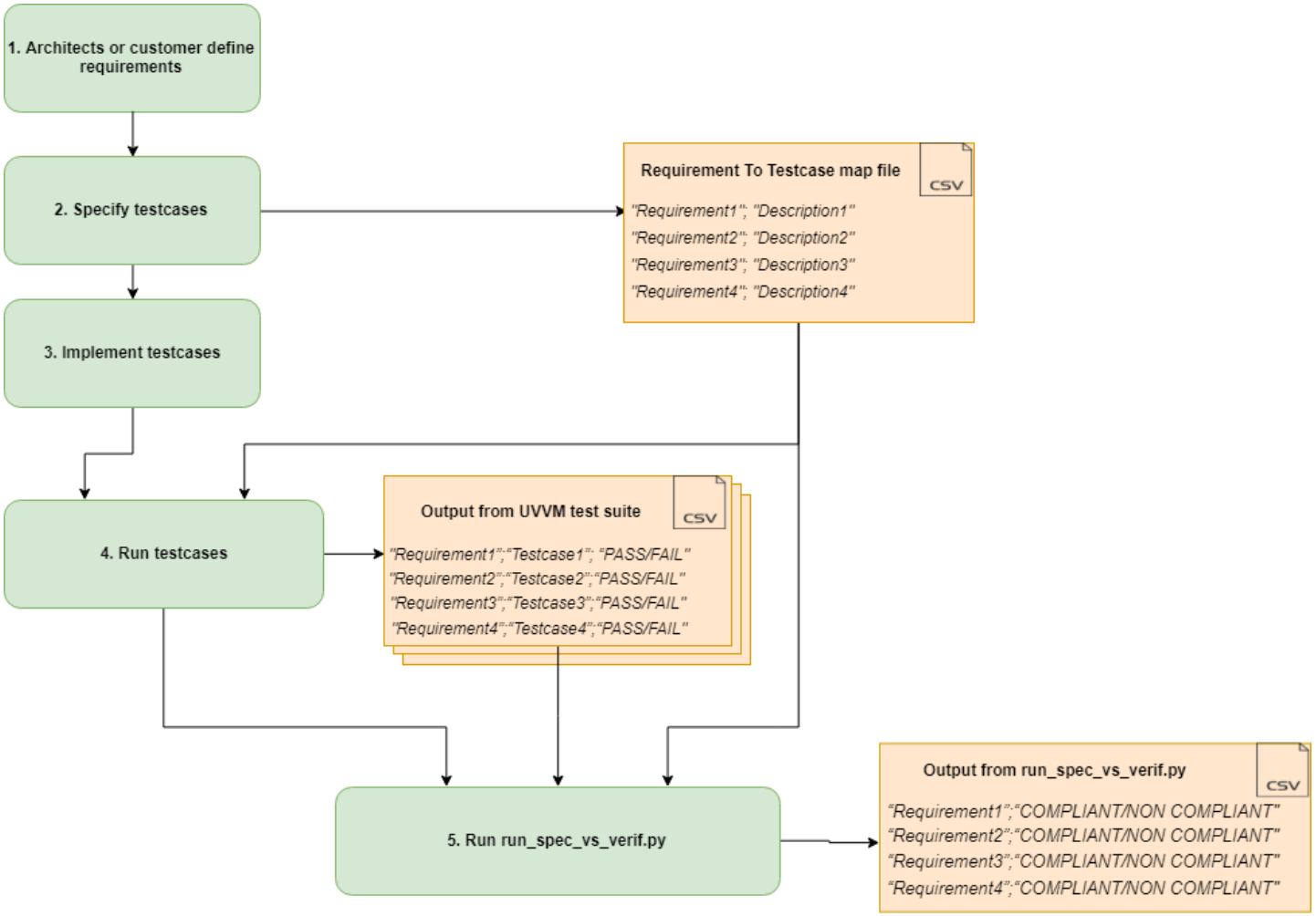

Figure 1 Basic Specification vs Verification Typical Usage

Table 4 Basic Specification vs Verification Typical Usage Description

Step	Description

1. Architects or customer define requirements	This step typically occurs prior to the FPGA designer involvement. The system architect or the customer defines system level requirements for the design, that will later be used to create a detailed design description. These requirements are often broad and incomplete.
2. Specify testcases	In this step the designer will specify testcases that will cover all the requirements specified in the previous steps. As part of this step, the designer will create a CSV-file with a mapping from requirement to testcase, as shown in Figure 1. More information on this CSV-file can be seen on the first page of this QuickRef. Note that the "Test case" input is optional unless the *shared_req_vs_cov_strict_testcase_checking* variable is set true, as described in Table 9.
3. Implement testcases	After all the testcases have been defined, the designer will implement the testcases in VHDL. The VHDL functions for starting and ending the requirement coverage procedure shall be used at the beginning and end of the testcases, and the log_req_cov() procedure shall be used to log results. Typically, the log_req_cov() function will be used where a requirement is verified in the testcase. E.g. if a requirement states that the DUT shall set signal X when signal Y is received, the log_req_cov() is called with either pass or fail as the *passed*-argument when this feature has been tested. The log_req_cov() procedure will also check if there are any alert mismatches, and if so the requirement will be marked as failed. The VHDL functions are documented in Table 11.
4. Run testcases	When the tests have been implemented, all testcases shall be run. The testcases can be run sequentially, in parallel, or as soon as each testcase is developed. In the latter case, step 3 and 4 will be run simultaneously. The tests that are run in this step will generate one or more result files where the log_req_cov() results are stored. The output files are specified with the *start_req_cov()* function, as described in Table 11.
5. Run run_spec_vs_verif.py	The final step, after all tests have been run, is to run the Python script *run_spec_vs_verif.py*. This script will check the output from step 2 against the output file(s) from step 4 and create a report file where all requirements are marked as COMPLIANT or NON-COMPLIANT. A description of the script parameters can be found in Table 12.

For a practical example of the specification vs verification matrix concept, see the example design under the *demo* directory.

Advanced Concept Usage

In some cases, the designer would want to extend or even split up broad requirements in order to have simple, testable requirements. This is allowed in the specification vs verification matrix concept by extending step 1, as illustrated by Figure 2. In this case, a new CSV file with a mapping from (broad) requirement to (sub) requirements is created as part of step 1b, and used later on when running the Python script in step 5. A description of how to use this feature is documented in Table 11.

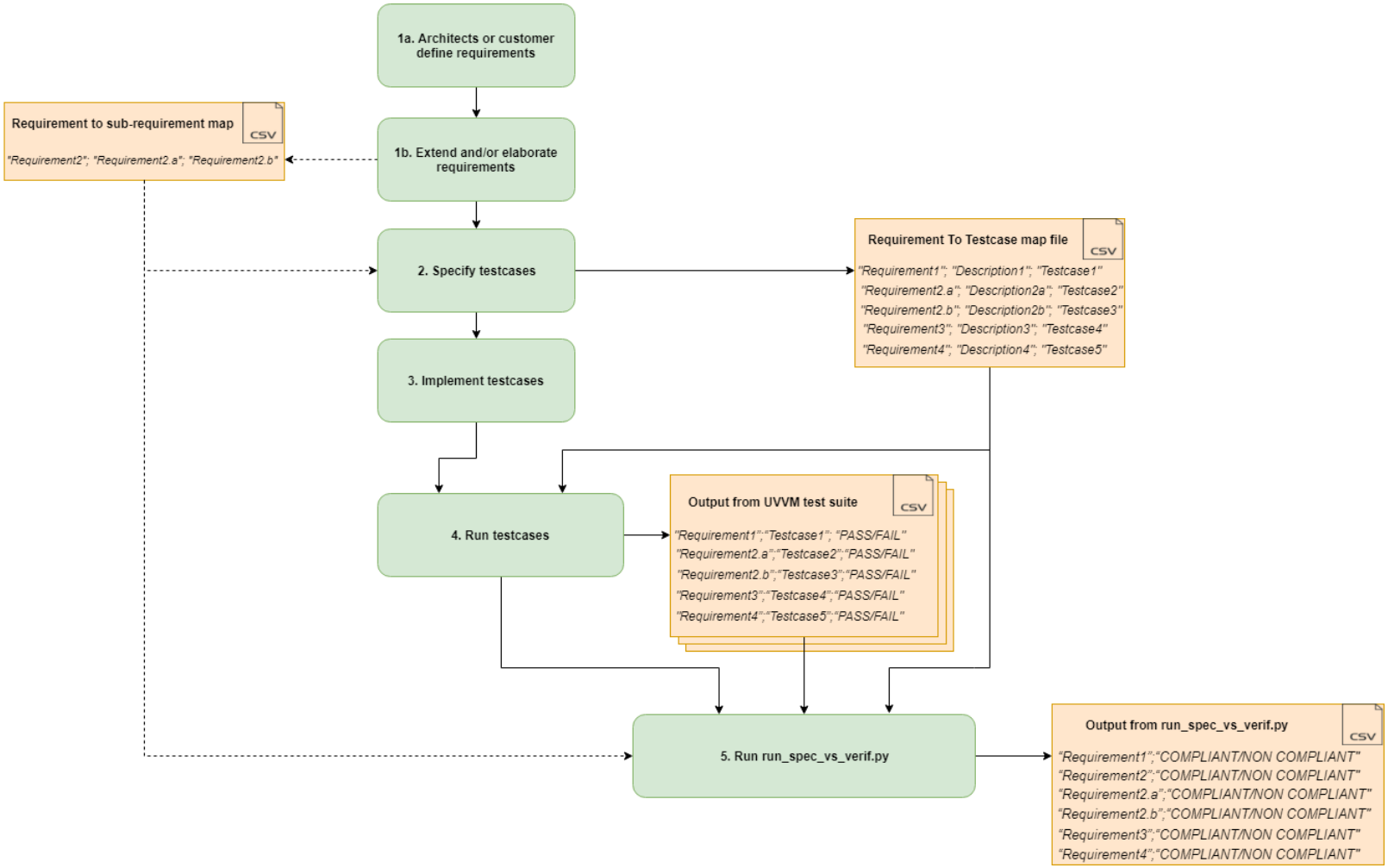

Figure 2 Advanced Specification vs Verification Typical Usage

Table 5 Advanced Specification vs Verification Typical Usage Description

Step	Description
1a. Architects or customer define requirements	See Table 4, step 1.
1b. Extend and/or elaborate requirements	As mentioned in the above step, the requirements defined by the customer or architect often specify broad requirements. In these cases, the designer may create sub-requirements for each step in the broad requirement. For example, if the requirement states: - FPGA_SPEC_1: "UART shall handle baud-rate of 9600bps and data shall be sent LSB first. This requirement could be replaced and split into two sub-requirements: - FPGA_SPEC_1.a: "UART shall handle baud-rate of 9600bps." - FPGA_SPEC_1.b: "UART shall handle data being sent LSB first" An optional map of requirement to sub-requirements can be created during this step. This mapping can be used later by the script to check and report the original requirement and not just the sub-requirement. In some cases, the designer will also want to add additional requirements to the requirement list in this step, e.g. if a requirement is relevant for how the design is implemented.
2. Specify testcases	See Table 4, step 2. In the cases where a requirement has been split into sub-requirement, the original requirement shall not be part of this map-file.
3. Implement testcases	See Table 4, step 3.
4. Run testcases	See Table 4, step 4.
5. Run run_spec_vs_verif.py	See Table 4, step 5. In addition to the requirements, the sub-requirements will also be listed as either compliant or non-compliant.

When minimum one test case must pass

In some cases, a requirement will require that at least one of many test cases must pass for the requirement to be compliant. However, not all test cases must pass for the requirement to be compliant. This is allowed by the Specification vs Verification extension. To instruct the tool that only one test case is required to pass, the requirement is specified as shown in the table below, entered in the Requirement to Testcase map file. In this example, the requirement FPGA_SPEC_3 is tested by test case TC_UART_3.a, b, c and d, and at least one of these test cases needs to pass for the requirement to be compliant.

Table 6 Test case mapping – Minimum one test case must pass

File	Requirement to Testcase map file	CSV
Layout	*"Requirement"; "Description"; "Testcase 1" [; "Testcase 2"; "Testcase 3"; …; "Testcase N"]*	
Example	FPGA_SPEC_1; UART all bits 0 ; TC_UART_1 FPGA_SPEC_2; UART start bit polarity; TC_UART_2 FPGA_SPEC_3; UART start bit delay; TC_UART_3.a; TC_UART_3.b; TC_UART_3.c;TC_UART_3.d	

When multiple requirements are tested by one test case

This is supported by the tool as shown in the table below.

Table 7 Test case mapping – One test case verifies multiple requirements

File	Requirement to Testcase map file	CSV
Layout	*"Requirement"; "Description"; "Testcase 1" [; "Testcase 2"; "Testcase 3"; …; "Testcase N"]*	
Example	FPGA_SPEC_1; UART all bits 0 ; TC_UART_1 FPGA_SPEC_2; UART start bit polarity; TC_UART_2 FPGA_SPEC_3; UART start bit delay, minimum; TC_UART_2 FPGA_SPEC_4; UART start bit delay, maximum; TC_UART_2	

When one requirement maps to multiple test cases

It is also allowed by the tool to specify that one (complex) requirement shall be tested by multiple test cases. This is specified by adding the requirement on multiple lines, as shown in the table below. However, it is recommended that such cases are handled by splitting the complex requirement is split into smaller, testable sub-requirements that have one test case per sub-requirement. This visualizes exactly what parts of the requirement that is verified by the test case. In the example below, TC_UART_3.a and TC_UART_3.b must pass for FPGA_SPEC_3 to be compliant.

Table 8 Test case mapping – Multiple test cases per requirement

File	Requirement to Testcase map file	CSV
Layout	*"Requirement"; "Description"; "Testcase 1" [; "Testcase 2"; "Testcase 3"; …; "Testcase N"]*	
Example	FPGA_SPEC_1; UART all bits 0 ; TC_UART_1 FPGA_SPEC_2; UART start bit polarity; TC_UART_2 FPGA_SPEC_3; UART start bit delay, minimum; TC_UART_3.a FPGA_SPEC_3; UART start bit delay, maximum; TC_UART_3.b	

VHDL Package

A vital part of the specification vs verification matrix concept is the VHDL testbench methods. These methods are described in Table 11. The methods are located inside the *spec_vs_verif_methods.vhd* file in the *src/* directory of this VIP.

UVVM Adaptation Package Configuration

Table 9 UVVM Adaptations Shared Variables

Variable	Type	Default	Description
shared_req_vs_cov_strict_testcase_checking	boolean	false	When this shared variable is set true, the Specification vs Verification Matrix checking will require that the input "*Requirement to Testcase map file*" contains the test case field. Only the test case specified in this field will be able to verify the requirement it is supposed to test. For example, if the *Requirement to Testcase map file* contains the following line: *FPGA_SPEC_1; Start bit polarity test; TC_UART_1* Then it will not be sufficient to run *log_req_cov* from any test case. It will have to be reported with: *log_req_cov("FPGA_SPEC_1", "TC_UART_1")* in order to pass. A mismatch between requirement and test case will result in a warning, and the requirement will not be marked as compliant.

Table 10 UVVM Adaptations Constants

Constant	Type	Default	Description
C_REQ_TC_MISMATCH_SEVERITY	t_alert_level	warning	Alert level used when the log_req_cov() procedure does not find the specified requirement or testcase in the requirement to testcase input file.
C_DEFAULT_RESULT_FILE_NAME	string	"resultfile.csv"	Default file name for the result file, if none is specified in the start_req_cov() procedure.
C_CSV_DELIMITER	character	';'	Character used as delimiter in the CSV files.
C_MAX_NUM_REQUIREMENTS	natural	1000	Maximum number of requirements in the req_to_tc_map file used in start_req_cov(). Increase this number if the number of requirements exceeds 1000.
C_MAX_NUM_TC_PR_REQUIREMENT	natural	20	Maximum number of testcases allowed per requirement. This is applicable when one requirement is verified by one or more test cases (as described above, under "When minimum one test case must pass").

The specification vs verification matrix implementation uses two new message Ids, as described in the table below. All message Ids are located in the adaptations package. The specification vs verification matrix implementation uses the shared message id panel for all logging.

Message Id	Description
ID_FILE_PARSER	Id used for CSV parser messages.
C_DEFAULT_RESULT_FILE_NAME	Id used for all messages that are not directly related to CSV parsing.

VHDL Methods Details

Table 11 VHDL Methods

Procedure	Parameters and examples	Description

log_req_cov()	requirement(string), testcase(string) [, passed(boolean) [, fail_on_alert_mismatch_severity(t_fail_on_alert_mismatch_severity)]
	
	Examples
	-- Will pass if no unexpected alert occurred
	log_req_cov("SPEC_FPGA_1", "Testcase_1");
	
	-- Will fail since passed argument is set to false
	log_req_cov("ESA_FPGA_UART_1", "UART_BAUDRATE_16k_Test", false, WARNING);
	
	-- Intended usage of the passed argument
	if(v_test_result > v_requirement_limit) then
	log_req_cov("ESA_FPGA_UART_2", "UART_Test");
	else
	log_req_cov("ESA_FPGA_UART_2", "UART_Test", false);
	end if;
		Evaluates and logs the specified requirement. The procedure checks the global alert mismatch status, and if an alert mismatch is present the requirement will be marked as failed. If there are no alert mismatches, the requirement will be marked as passed, unless the *passed* input is set to false.
		The result is written to both the transcript and a result file, specified in the *start_req_cov()* procedure. The *log_req_cov()* will look up the specified requirement and testcase in the req_to_tc_map_file specified in *start_req_cov()*, and use the description from this entry as log message. The procedure will also issue a warning if the specified requirement and testcase was not found.
		
		- *requirement:* String with the requirement tag. Must match a requirement tag in the file specified in start_req_cov().
		- *testcase*: String with the testcase tag. Must match a testcase tag in the file specified in start_req_cov().
		- *passed:* Boolean value with the result. True indicates passed, false indicated not passed.
		- *fail_on_alert_mismatch_severity:* Which alert mismatche severity on the global alert counter that will cause a requirement to be considered failed. Can be either WARNING or ERROR.
		
		Defaults
		passed <= true; fail_on_alert_mismatch_severity <= ERROR

start_req_cov()	req_to_tc_map_file(string), output_file(string) **Example** start_req_cov("c:/test_folder/my_req_to_testcase_map.csv", "output_file_path.csv"); start_req_cov(GC_REQ_MAP_PATH, GC_RESULT_PATH); -- Using generic as path	Starts the requirement coverage process in a test. The requirement to test mapping file specified in the *req_matrix_path* path is used to check that the requirement exists, and the description is retrieved from the file and logged to the transcript. - *req_to_tc_map_file:* String with the path to the requirement to testcase map CSV file. - *output_file*: String with the location where the output file will be placed. If a file exists in the specified path, the results will be appended to the file. **Defaults** output_file <= C_DEFAULT_RESULT_FILE_NAME
end_req_cov()	VOID(t_void) **Example** end_req_cov(VOID);	Ends the requirement coverage process in a test. This procedure writes a closing check to the output file specified in start_req_cov. This check is used later by the run_spec_vs_verif.py script.

Additional Documentation

Additional documentation about UVVM and its features can be found under “/uvvm_vvc_framework/doc/”.

Compilation

This VHDL package may only be compiled with VHDL 2008. It is dependent on the following libraries

- ***UVVM Utility Library (UVVM-Util), version 2.6.0 and up***
- ***UVVM VVC Framework, version 2.3.0 and up***

Before compiling the Specification vs Verification Matrix component, make sure that uvvm_vvc_framework and uvvm_util have been compiled.
See UVVM Essential Mechanisms located in uvvm_vvc_framework/doc for information about compile scripts.

Compile order for the Specification vs Verification Matrix Component:

Compile to library	File	Comment
bitvis_vip_spec_vs_verif	csv_file_reader_pkg.vhd	Package for reading and parsing of CSV input files
bitvis_vip_spec_vs_verif	spec_vs_verif_methods.vhd	Specification vs Verification Matrix component implementation

Simulator compatibility and setup

This VVC has been compiled and tested with Modelsim version 10.4b and Riviera-PRO version 2018.02.
For required simulator setup see UVVM-Util Quick reference.

Post-processing Script

The final step of the Specification vs Verification usage shown in Figure 1 and Figure 2 is to run a post-processing script to evaluate all the simulation results. This script is called *run_spec_vs_verif.py*. This script requires Python 3.x. The script can be called with the arguments listed in Table 12 from any command line.

Table 12 Script Arguments

Argument	Short form	Example	Description
--requirements	-r	--requirements path/to/requirements.csv -r path/to/requirements.csv	Points to the requirement list from Step 3 in Table 4. This argument is mandatory.
--resultfile	-f	--resultfile path/to/resultfile.csv -f path/to/resultfile.csv	Points to the resultfile from the VHDL simulation, discussed in Step 5 in Table 4. Either this argument OR the --resultfiles/-F argument is mandatory. Both can't be used at the same time.
--resultlistfile	-l	--resultlistfiles path/to/fileinput.txt -l path/to/fileinput.txt	Points to a text file that contains the paths to all result files. The files listed in this file will be merged to a single resultfile. This argument can be used if the simulations have been run in more than one testbench. Either this argument OR the --resultfile/-f argument is mandatory. Both can't be used at the same time. Example resultfiles file contents: *path/to/first/result.csv* *path/to/second/result.csv* *path_to_third_result.csv*
--output	-o	--output path/to/outputs -o path/to/outputs	Path to directory where the output described in Step 6 in Table 4 shall be stored. This argument is optional. If the argument is not used, the output directory will be the current directory.
--subrequirements	-s	--subrequirements path/to/subrequirements.csv -s path/to/subrequirements.csv	Points to the requirement to sub-requirement map file, described in Step 2 in Table 4. This argument is optional. If this argument is omitted, the script assumes that no sub-requirements exists.
--config	-c	--config path/to/configfile.txt -c path/to/configfile.txt	Optional configuration file where all the arguments can be placed. This argument will override all other arguments. The configuration file does not need to have the .txt extension. All arguments shall be added on a new line. Example configuration file contents: *--requirements path/to/requirements.csv* *-F test_suite_input_files.txt* *--subreqs path/to/subrequirements.csv*

The output of the post-processing is a CSV file where all requirements and sub-requirements are listed and marked as either compliant or non-compliant. The format of this file is shown on the front-page of this QuickRef. The post-processing script will also print a transcript to file, where it is indicated whether or not the script succeeded.