

# Specification Coverage - Quick Reference

## UVVM Support VIP

The Specification Coverage feature (aka Requirements Coverage) is an efficient method for verifying the requirement specification.

**Note:** The first page of this QuickRef is just for syntax reference. For an introduction to the Specification Coverage concept, please see page 2.

### 1 UVVM VHDL Methods – see Table 2 for details.

Procedure	Parameters	Examples
<b>initialize_req_cov()</b>	testcase name (string), requirement list file (string), testcase coverage file (string) testcase name (string), testcase coverage file (string) ( <i>→ Note: No spec coverage</i> )	initialize_req_cov("t_base_func", "uart_req_list.csv", "base_func_cov.csv"); initialize_req_cov("t_base_func", "base_func_cov.csv");
<b>log_req_cov()</b>	requirement label (string), [testcase (string)] [, PASS FAIL (t_test_status)]	log_req_cov("UART_REQ_4");
<b>finalize_req_cov()</b>	VOID	Finalize_req_cov(VOID);

### 2 Script Usage – run\_spec\_cov.py

Call the run\_spec\_cov.py from a terminal, using e.g.: **python run\_spec\_cov.py <see Table 6 for script arguments>**

### 3 File Formats – Basic usage

File	Requirement list file ('req_list'), CSV	Testcase coverage file ('tc_cov'), CSV	Specification coverage file ('spec_cov'), CSV
<b>Info</b>	Input to testcase and run_spec_cov.py	Output from testcase, and input to run_spec_cov.py	Output from run_spec_cov.py
<b>Layout</b>	"Requirement label", "Description" [, "Testcase"]	"Requirement", "Testcase", PASS FAIL Preceded by header and succeeded by footer as shown below	"Requirement", COMPLIANT   NON COMPLIANT
<b>Example</b>	FPGA_REQ_1, Baudrate 9k6, T_UART_1 FPGA_REQ_2, Baudrate 9k6, T_UART_1 etc	NOTE: <note> Testcase-name: <name> Delimiter: <single delimiter character> FPGA_REQ_1, T_UART_1, PASS FPGA_REQ_2, T_UART_2, FAIL..... etc SUMMARY: <testcase-name>, FAIL PASS	FPGA_REQ_1, COMPLIANT FPGA_REQ_2, NON_COMPLIANT etc

File	Optional: Requirement map file ('req_map'), CSV	Optional: Testcase coverage file list file (tc_cov_list'), TXT	Optional: Script config file, TXT
<b>Info</b>	Optional input to run_spec_cov.py	Optional input to run_spec_cov.py	Optional input to run_spec_cov.py
<b>Layout</b>	Alt a) "Requirement label", "mapped req label" [, "mapped req label"] Alt b) "Requirement label", "sub-req label", "sub-req label", ...	"path to a Testcase coverage file" "path to another Testcase coverage file" etc ....	--argument value OR -a value etc ....
<b>Example</b>	FPGA_REQ_1, FPGA_REQ_1.a, FPGA_REQ_1.b	T_UART_1.csv ../my_sim_dir/tc_base_func.csv T_base_func.csv	--req_list path/requirement_list.csv -r testcase_coverage_files.txt --strictness 1

NOTE: The CSV separator may be set to any separator character. Default is ',' but may be changed in VHDL (see ch 9.2). Delimiter is written to testcoverage file.

## 4 Definition of 'Testcase' used in this document

- A scenario or sequence of actions - controlled by the test sequencer.
- May test one or multiple features/requirements.
- Typically testing of related functionality, or a logical sequence of events, or an efficient sequence of events
- Important: The minimum sequence of events possible to run in a single simulation execution. Thus if there is an option to run one of multiple test sequences (A or B or C), a set of test sequences (A and B) or all sequences (A+B+C), then all of A, B and C are defined as individual testcases.

## 5 Specification Coverage Concept

An important step of design verification is to check that all requirements have been met. Requirements can be defined very differently depending on application, project management, quality requirements, etc. In some projects, requirements barely exist, and the functionality is based on a brief description. However, in projects where safety and reliability are key the requirement handling is an essential part of the project management flow. In some development standards the requirements and the corresponding testcases that verify the requirements need to be defined, reviewed and accepted by a third-party assessor before even starting the verification flow.

This UVVM Verification IP is intended for projects where requirements are essential in the workflow but may also be used in a very simple way for projects with lower requirements. Examples of requirements can be seen in Table 1. It is in general a good idea to label the various requirements, and in many projects this would be mandatory. The example in Table 1 shows of course only a subset of all the requirements.

Requirement Label	Description
UART_REQ_1	The device UART interface shall accept a baud rate of 9600kbps.
UART_REQ_2	The device UART interface shall accept a baud rate of 19k2 bps.
UART_REQ_3	The device UART interface shall accept an odd parity
UART_REQ_4	The device reset shall be active low.

*Table 1 Requirement examples.  
(Requirement labels are defined by the user)*

There are lots of acceptable approaches with respect to how much functionality is verified in each testcase and how these are organised. This VIP will allow various approaches from dead simple to advanced. In order to explain the concepts, we start with the simplest case and add step-by-step on that until we have built a full advanced specification coverage system.

## 6 Conceptual introduction and the Simplest possible usage, with a single testcase

For any FPGA / ASIC it is always important to properly specify the design requirements and check that they have all been tested. Normally it is often just ticked off somewhere that a particular requirement is tested – often only once during the development phase, and sometimes just as a mental exercise. It is always better to use a written, repeatable and automated approach. This VIP significantly simplifies such an approach.

When feasible, the simplest structured approach would be to test all requirements in one single self-checking testcase. If so – all you want to do is the following – as illustrated in Figure 1.

1. List all DUT requirements in a requirement list CSV file. (RL)  
This could mean anything from just writing down the requirements directly, to a fully automated requirement extraction from an existing Requirement Specification.  
NOTE: For specification coverage this list is mandatory, but a simplified mode of pure test reporting without the need for a prior listing of the requirements is available (see more info in section 7.1 ).
2. Implement your testcase (T) with all tests required to verify the DUT.  
The test sequencer should initiate coverage using `initialize_req_cov()` (T1), then for each verified requirement call `log_req_cov()` (T2) and then finalise coverage reporting using `finalize_req_cov()` (T3).
3. When the testcase is executed (run), `initialize_req_cov()` (T1) will read the given requirement list file (RL), the given testcase coverage file (TC) is opened, and the testcase name is stored. The header of the testcase coverage file – with NOTE, testcase-name and delimiter is written. The header is not shown in any of the examples, but is shown in the front page file formats.  
Then for each `log_req_cov()` (T2) a separate line is written into the testcase coverage file with a) the given requirement label, b) the name of the testcase, and c) the result of the test.  
The result of the test will be PASS - unless marked as FAIL or unexpected serious alerts (`>= ERROR / TB_ERROR`) have occurred, in which case it will be marked as FAIL.  
Finally when `finalize_req_cov()` (T3) is executed, a closing check of the alert counters is made. If ok, then 'SUMMARY, <Testcase name>, PASS' is written at the end of the Testcase coverage file.  
Otherwise FAIL rather than PASS. If a testcase fails before reaching `finalize_req_cov()`, then no SUMMARY line will be written. This is interpreted as FAIL.  
Note that a given requirement may be tested and reported several times, so that for instance UART\_REQ\_3 may be listed multiple times in the Testcase coverage file (TC).
4. After the testcase has been executed, the overall Specification coverage (SC) can be found by executing the Python script `run_spec_cov.py` (section 10).  
This script traverses the requirement list (RL) and Testcase coverage file (TC) and from that generates the specification coverage (SC).  
Each requirement is listed only once in the specification coverage file. If a requirement has one or more FAIL in a Testcase coverage file, the result is NON\_COMPLIANT for that requirement.  
For a simple scenario with a single testcase, the testcase coverage (TC) file and the specification coverage file (SC) yield the same information, but the specification coverage potentially with fewer lines.

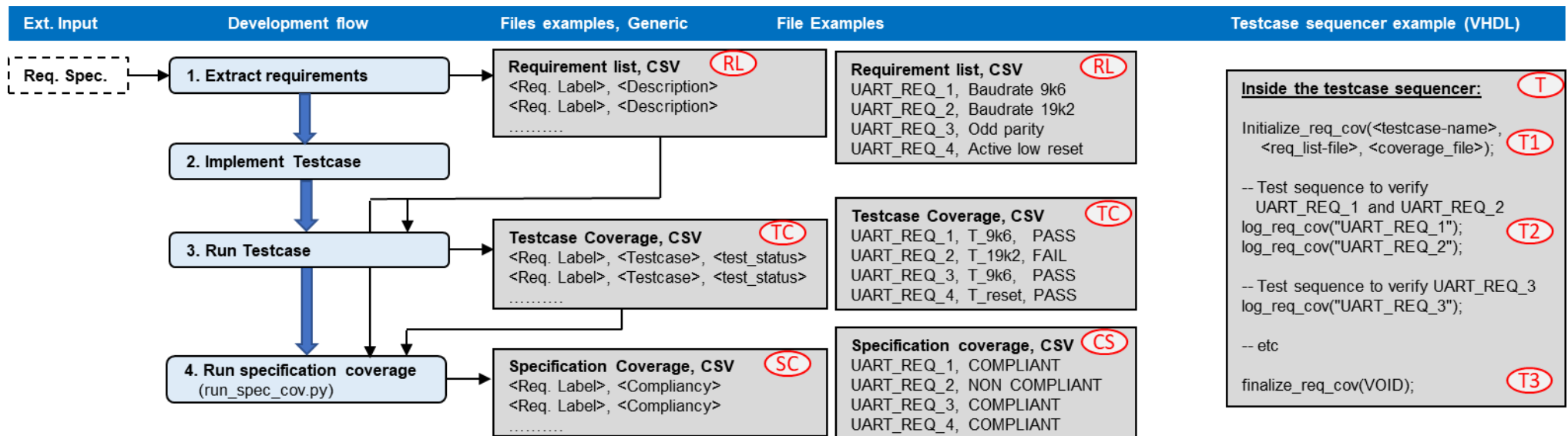


Figure 1: Simplest possible specification coverage (note that testcase coverage files only show the actual requirement coverage lines)

## 7 Simple usage

### 7.1 No requirement mode

A shortcut is supported to allow all tested requirements to be reported to the Testcase coverage file – without the need for a prior listing of the requirements. This shortcut does of course not yield any specification coverage, as no specification is given, but could be useful for scenarios or early testing where only a list of executed tests is wanted. This shortcut mode must be specified via the VHDL configuration record as specified in section 9.2. I.e. allows VHDL testcases to be executed without a requirement list, and thus generates testcase coverage files, but no specification coverage file (the Python script may not be run).

### 7.2 Simple usage, with multiple testcases

Many verification systems will have multiple testcases per DUT. If so, the above simplest approach is not possible.

However, if your tests are split on multiple testcases, but with no requirement to which testcase tests what, then you can apply almost the same simple approach as the simplest case above.

For this scenario, there will be two or more testcases, and so you will have to run all relevant testcases. There will of course still be only a single Requirement list.

For every single testcase the same set of commands will be applied – with `initialize_req_cov()`, one or more `log_req_cov()`, and `finalize_req_cov()`. The only thing to remember here is that `initialize_req_cov()` has to specify separate Testcase coverage files for each testcase. Hence, after simulation you should end up with as many Testcase coverage files as testcases

Note: `log_req_cov()` will as default use the testcase name specified with `initialize_log_req()` when writing to the testcase coverage file. It is however possible to specify a dedicated testcase name as a parameter to `log_req_cov()`. If this testcase name is different from the testcase name already specified by `initialize_log_req()`, then this testcase name is used when logging that specific requirement to the testcase coverage file. All following `log_req_cov()` with no testcase name specified will again use the name given with `initialize_log_req()`. The testcase coverage file may in other words include references to both itself and other testcase names. It is also possible inside a test sequencer to execute `initialize_log_req()` multiple times, but only when the previous `initialize_log_req()` has been terminated with `finalize_req_cov()`.

The Python script `run_spec_cov.py` will be run in the same way as before, but needs to be given a list of all the relevant Testcase coverage files. Then the Specification coverage is generated exactly as before.

### 7.3 Multiple testcases – with strict requirement vs testcase relation

For most applications where high quality and confidence is required it is mandatory to specify up front in which testcase a given requirement will be tested. In these cases, the requirement list must be extended to include the testcase in which a requirement will be tested, - as shown to the right.

#### Requirement list, CSV

```
<Req. Label>, <Description>, <Testcase-name>
<Req. Label>, <Description>, <Testcase-name>
.....
```

RL

#### Requirement list, CSV

```
UART_REQ_1, Baudrate 9k6 , T_9k6
UART_REQ_2, Baudrate 19k2 , T_19k2
UART_REQ_3, Odd parity , T_9k6
UART_REQ_4, Active low reset, T_reset,
```

RL

Specifying a testcase name in the requirement list will however not force a requirement vs

testcase check by itself. In order to check that a requirement is tested in the specified testcase the switch ‘–strictness 1’ must be used when calling `run_spec_cov.py` as described in section 10.

If the switch is set then for example the requirement ‘UART\_REQ\_3, Odd parity, T\_9k6’:

- Will be marked as COMPLIANT in the specification coverage file if UART\_REQ\_3 is checked positive in testcase T\_9k6. May also be checked elsewhere.
- Will be marked as NON-COMPLIANT in the specification coverage file if UART\_REQ\_3 is not checked in testcase T\_9k6 (but for instance only in T\_19k2).

NOTE: An even stricter check could be applied is calling `run_spec_cov.py` with the switch –strictness 2.

This will result in NON-COMPLIANT even if a requirement is executed in the given testcase when it is also tested in a non-given testcase. E.g. if UART\_REQ\_3 is tested in testcase T\_9k6, but **also** in T\_19k2.

Default strictness is –strictness 0, i.e. neither of the above strict checking.

Testcase names are not case sensitive.

## 8 Advanced usage

### 8.1 Advanced usage – Only one of multiple testcases for a given requirement must pass

In the previous example all tests in all testcases had to pass for the overall specification coverage to pass.

There may however be situations where a given requirement is tested in multiple testcases and it is sufficient that only any one of these pass – given of course that none of the others have failed, but just haven't been executed. A fail in any executed testcase will always result in a summary fail.

This approach could for instance be used to qualify for a lab test release.

A requirement for this approach to work is to remove all old testcase coverage files before running your test suite generating new testcase coverage files. The `run_spec_cov.py` option `-clean` may be used for this.

#### Requirement list, CSV

```
<Req. Label>, <Description>, <Testcase-name> [, <Testcase-name>, <Testcase-name>, ...]
<Req. Label>, <Description>, <Testcase-name> [, <Testcase-name>, <Testcase-name>, ...]
.....
```

RL

#### Requirement list, CSV

```
UART_REQ_1, Baudrate 9k6 , T_9k6
UART_REQ_2, Baudrate 19k2 , T_19k2
UART_REQ_3, Odd parity , T_9k6, T_19k2
UART_REQ_4, Active low reset, T_reset, T_9k6, T_19k2
```

RL

The example above shows that UART\_REQ\_3 is covered by both testcases T\_9k6 and T\_19k2, and that UART\_REQ\_4 may in fact be tested by any of the three testcases.

### 8.2 Advanced usage – All (or some) of multiple testcases for a given requirement must pass

This is basically the opposite of the above and is easy to achieve by just adding lines in the requirement list for all wanted combinations of requirements and testcases. The example to the right states that UART\_REQ\_3 must pass in both T\_9k6 and T\_19k2.

#### Requirement list, CSV

```
UART_REQ_1, Baudrate 9k6 , T_9k6
UART_REQ_2, Baudrate 19k2 , T_19k2
UART_REQ_3, Odd parity , T_9k6
UART_REQ_3, Odd parity , T_19k2
UART_REQ_4, Active low reset, T_reset, T_9k6, T_19k2
```

RL

## 8.3 Advanced usage – Requirement mapping

Requirement mapping just maps one or more requirements to another requirement. This is intended for two different use cases:

### 8.3.1 Mapping of project requirements to IP or legacy requirements

Assuming you already have a UART IP, that has been properly verified, and the provided UART testbench already has full UVVM specification coverage support, with a requirement list file and testcases generating Testcase coverage files.

Assuming you then have a project with its own UART requirements, that hopefully more or less matches that of the IP, but the requirement labels and combinations may be different. Then you don't want to modify the provided UART testbench in order just to get the "right" requirement labels.

A much better approach is to map the project requirements to the UART IP requirements.

From the UART IP	From the PROJECT requirements
<b>Requirement list, CSV</b> <span style="border: 1px solid red; border-radius: 50%; padding: 2px;">RL</span> UART_REQ_1, Baudrate 9k6 , T_9k6 UART_REQ_2, Baudrate 19k2 , T_19k2 UART_REQ_3, Odd parity , T_9k6 UART_REQ_4, Active low reset, T_reset,	<b>Requirement list, CSV</b> <span style="border: 1px solid red; border-radius: 50%; padding: 2px;">RL</span> UART_REQ_A, Baudrate 9k6 and 19k2 UART_REQ_B, Odd parity

Now assume the already shown UART requirement list is that of the IP, and that we have similar project UART requirements, we may have a scenario as shown above.

Here we can see that UART\_REQ\_B of the project matches that of UART\_REQ\_3 of the UART IP, and that UART\_REQ\_A of the project is actually not covered by any single UART IP requirement, but in fact must include both UART\_REQ\_1 and UART\_REQ\_2. The map would for this case be as shown to the right.

<b>Requirement map file, CSV</b> <span style="border: 1px solid red; border-radius: 50%; padding: 2px;">RM</span> UART_REQ_A, UART_REQ_1, UART_REQ_2 UART_REQ_B, UART_REQ_3
---

Making the requirement map file is of course a manual job, which could be simple or complex depending on how much the two requirement lists differ in structure.

The requirement map file is only used as an input to the run\_spec\_cov.py Python script, as is also the Testcase coverage file from the UART IP verification using the IP (or legacy) testbench. The Python script will check that

- For project requirement UART\_REQ\_A, both UART\_REQ\_1 and UART\_REQ\_2 have passed
- For project requirement UART\_REQ\_B, UART\_REQ\_3 has passed
- Etc...

The report from run\_spec\_cov.py will show compliancy for the project requirement (e.g. UART\_REQ\_A), but also for the "sub-requirement(s)" (e.g. UART\_REQ\_1 and UART\_REQ\_2).

### 8.3.2 Mapping of requirements to multiple sub-requirements

Often the original requirements are too complex (or compound), so that it is difficult to tick off a requirement as checked until a whole lot of different things are tested. An example of that could be a UART requirement like the one shown to the right – with a single UART\_REQ\_GENERAL requirement.

If you want to split this into more specific requirements you have several options, with some potential options listed below and illustrated to the right.

- a) Rename the requirements
- b) Extend the names
- c) Extend the names using a record like notation

All of these and more are of course possible, but the problem is that they don't show the relation to the original requirement.

**Requirement list, CSV**  
UART\_REQ\_GENERAL, Baudrates 9k6 and 19k2, odd and even parity

**a) Potential new Requirement list, CSV**  
UART\_REQ\_BR\_A, Baudrate 9k6  
UART\_REQ\_BR\_B, Baudrate 19k2  
UART\_REQ\_ODD, Odd parity  
UART\_REQ\_EVEN, Even parity

**b) Potential new Requirement list, CSV**  
UART\_REQ\_GENERAL\_BR\_A, Baudrate 9k6  
UART\_REQ\_GENERAL\_BR\_B, Baudrate 19k2  
UART\_REQ\_GENERAL\_ODD, Odd parity  
UART\_REQ\_GENERAL\_EVEN, Even parity

**c) Potential new Requirement list, CSV**  
UART\_REQ\_GENERAL.BR\_A, Baudrate 9k6  
UART\_REQ\_GENERAL.BR\_B, Baudrate 19k2  
UART\_REQ\_GENERAL.ODD, Odd parity  
UART\_REQ\_GENERAL.EVEN, Even parity

Showing this relationship is quite simple in UVVM, by using exactly the same mechanism as for the IP or legacy scenario in the previous example. Thus all you have to do is to make your own requirement list and then map the original requirement to a set of requirements in your new list.

**Requirement map file, CSV**  
UART\_REQ\_GENERAL, UART\_REQ\_GENERAL.BR\_A, UART\_REQ\_GENERAL.BR\_B, UART\_REQ\_GENERAL.ODD, UART\_REQ\_GENERAL.EVEN

- Or even simpler just name the subrequirements A,B,C,....

**Requirement map file, CSV**  
UART\_REQ\_GENERAL, UART\_REQ\_GENERAL.A, UART\_REQ\_GENERAL.B, UART\_REQ\_GENERAL.C, UART\_REQ\_GENERAL.D

As for the IP scenario this approach allows the report to show both the original requirement and its new sub-requirements.

## 9 VHDL Package

A vital part of the specification coverage concept is the VHDL testbench methods. These methods are described in Table 2. The methods are located inside the *spec\_cov\_methods\_pkg.vhd* file in the *src/* directory of this VIP.

### 9.1 VHDL Methods Details

Procedure	Parameters and examples	Description
initialize_req_cov()	testcase (string), req. list file (string), testcase coverage file (string) or testcase (string), testcase coverage file (string)  <b>Examples</b> initialize_req_cov("T_UART_9k6", "c:/my_folder/requirements.csv", ".cov_9k6.csv"); initialize_req_cov("T_UART_9k6", "requirements.csv", "cov_9k6.csv");	Starts the requirement coverage process in a testcase.  The requirement list file is optional, but without it specification coverage is of course not possible, and run_spec_cov.py may not be executed.  The testcase_coverage_file is opened – and the header is written with NOTE, Testcase-name, Delimiter
log_req_cov()	requirement(string) [, testcase(string)] [, test_status(t_test_status)]  <b>Examples</b> -- Will pass if no unexpected alert occurred log_req_cov("UART_REQ_1", "T_UART_9k6");  -- Will fail since passed argument is set to false log_req_cov("UART_REQ_1", "T_UART_9k6", FALSE);	Evaluates and logs the specified requirement. The procedure checks the global alert mismatch status, and if an alert mismatch is present on ERROR, FAILURE, TB_ERROR or TB_FAILURE the requirement will be marked as FAIL. If there are no such alert mismatches, the requirement will be marked as PASS, unless the test_status is explicitly set to FAIL. The result is written to both the transcript (and log) and the testcase coverage file (specified in the initialize_log_req() command) The log_req_cov() will look up the specified requirement and testcase in the req_list specified in initialize_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.  <ul style="list-style-type: none"> <li>- <i>requirement</i>: String with the requirement label. Must as default match a requirement label in the given requirement list.</li> <li>- <i>testcase</i>: Optional: String with the testcase name. Default: Testcase name from initialize_log_req()</li> <li>- <i>test_status</i>: Optional: Enter PASS or FAIL depending on status of test. Default: PASS</li> </ul>
finalize_req_cov()	VOID(t_void)  <b>Example</b> finalize_req_cov(VOID);	Ends the requirement coverage process in a test. If alert status is OK – appends a line to testcase coverage file: 'SUMMARY, <Testcase name>, PASS' If alert status is not OK – appends a line to testcase coverage file: 'SUMMARY, <Testcase name>, FAIL' This line is used later by the run_spec_cov.py script. If simulation never reached this command, e.g. if failed, then no summary line is written – indicating FAIL

Table 2 VHDL Methods



## 9.2 Specification Coverage configuration record: shared\_spec\_cov\_config

This record is located in the local adaptation package 'bitvis\_vip\_spec\_cov/src/local\_adaptations\_pkg.vhd'

The configuration record is applied as a shared\_variable 'shared\_spec\_cov\_config' to allow different configuration for different DUTs.

Any test sequencer may then set the complete record as required – or even just parts of it like shared\_spec\_cov\_config.csv\_delimiter := ',';

Record elements	Type	Default	Description
missing_req_label_severity	t_alert_level	TB_WARNING	Alert level used when the log_req_cov() procedure does not find the specified requirement label in the requirement list, given that a requirement list is given in the initialize_log_req() command.
csv_delimiter	character	','	Character used as delimiter in the CSV files. This will also be written into all testcase coverage files. run_spec_cov.py will find the delimiter there.
max_requirements	natural	1000	Maximum number of requirements in the req_map file used in initialize_req_cov(). Increase this number if the number of requirements exceeds 1000.
max_testcases_per_req	natural	20	Maximum number of testcases allowed per requirement. This is applicable when one requirement is verified by one or more testcases.
csv_max_line_length	Positive	256	Maximum length of each line in any CSV file. (i.e. max number of characters for all values and separators in total)

Table 3 Specification coverage adaptation

The specification coverage implementation uses three new message ids, as described in the table below. All message ids are located in uvvm\_util adaptations package. The specification coverage implementation uses the shared message id panel for all logging.

Message Id	Description
ID_FILE_OPEN_CLOSE	Id used for any file open and close operation
ID_FILE_PARSER	Id used for CSV parser messages.
ID_SPEC_COV	Id used for all messages that are not directly related to CSV parsing.

Table 4 Message ID usage

## 9.3 Additional Documentation

Additional documentation about UVVM and its features can be found under “/uvvm\_vvc\_framework/doc/”.

### 9.3.1 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 Coverage 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_cov	Local_adaptations_pkg.vhd	Local file for user adaptations
bitvis_vip_spec_cov	csv_file_reader_pkg.vhd	Package for reading and parsing of CSV input files
bitvis_vip_spec_cov	spec_coverage_pkg.vhd	Specification Coverage component implementation

Table 5 Compile order

### 9.3.2 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.

## 10 Post-processing Script

The final step of the Specification Coverage usage is to run a post-processing script to evaluate all the simulation results. This script is called *run\_spec\_cov.py*. The script requires Python 3.x. The script can be called with the arguments listed in Table 6 from any command line. The CSV delimiter is fetched by the Python script from the testcase coverage file headers.

Note: All files may be referenced with absolute paths or relative to working directory.

Argument	Short form	Example	Description
--req_list	-r	--req_list my_path/requirements.csv	Points to the requirement list. This argument is mandatory. (For the case without a requirement list – this script may not be executed, and wouldn't make sense anyway.)
--test_cov	-t	--test_cov my_testcase_cov.csv --test_cov my_coverage_files.txt	Points to the testcase coverage file generated by the VHDL simulation. May also point to a file list including references to multiple testcase coverage files The format of such a file would be just each file on a separate line – potentially prefixed by a relative or absolute path
--req_map_list	-m	--req_map_list path/to/subrequirements.csv	Optional: Points to the requirement map file, described in section 8.3. If this argument is omitted, the script assumes that no sub-requirements exists.
--spec_cov	-s	--spec_cov uart_spec_cov.csv	Name (and optional path) of the specification_coverage file,
--clean		--clean	Will clean any/all given testcase coverage files – as specified using --test_cov. No short form - to avoid unwanted clean
--strictness		-- strictness 1	Default strictness is 0 (when not applied). 1 is stricter and 2 is much stricter (see section 7.3). No short form – to avoid wrong strictness
--config	-c	--config 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: --req_list my_path/requirements.csv --test_cov my_testcase_cov_files.txt --spec_cov my_spec_cov.csv

Table 6 Script Arguments

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.

### INTELLECTUAL PROPERTY

Disclaimer: This IP and any part thereof are provided "as is", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with this IP.