**Specification Coverage -** Quick Reference  
UVVM Support Component

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 of this QuickRef.***

|  |  |
| --- | --- |
| **UVVM VHDL Methods – see Table 11 for details** |  |
| **initialize\_req\_cov**(req\_list\_file, coverage\_file) |  |
| **log\_req\_cov**(requirement [, passed]) | **ddddddddddddddddddddddddddddddddddd** |
| **finalize\_req\_cov**(VOID) |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

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 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 list file, CSV** | **Coverage file (from VHDL testbench sim), CSV** | **Coverage summary (from run\_spec\_cov.py), CSV** |
| **Layout** | *“Requirement”; “Description” [; “Testcase”]* | *“Requirement”; “Testcase”; “PASS/FAIL”* | *“Requirement”; “COMPLIANT / NON COMPLIANT”* |
| **Example** | FPGA\_REQ\_1, Baudrate 9k6, TC\_UART\_1  FPGA\_REQ\_2, ……….  ………. | FPGA\_REQ\_1; TC\_UART\_1; PASS  FPGA\_REQ\_2; ………….  ………. | FPGA\_REQ\_1; COMPLIANT  FPGA\_REQ\_2; ……..  ………. |

Optional files

|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **Requirement map file (optional), CSV** | **Test result file list (optional), txt** | **Script config file (optional), txt** |
| **Layout** | *Alt a) “Requirement”, “mapped req” [, “mapped req”]*  *Alt b) “Requirement”, “Sub-req.”; “Sub-req.”; …;”Sub-req.”* | *“path/to/result\_file.csv”* | *--argument value OR -a value* |
| **Example** | FPGA\_REQ\_1; FPGA\_REQ\_1.a; FPGA\_REQ\_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 |



NOTE: The CSV separator may be set to any separator character og string via \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*. Default is comma separated.

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.

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.

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

|  |  |
| --- | --- |
| **Requirement Name** | **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. |

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.

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

For any FPGA or ASIC design 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 the figure below

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 is available to allow all tested requirements to be reported to the coverage file – without the need for prior listing of all requirements (see \*\*\*\*\*)
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 the 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 and fill this into a VHDL table. Also the given testcase coverage file (CF) is opened. The testcase name is stored to be used by log\_req\_cov(). (Testcase name is not required for a single testcase. Default will then be ‘default\_test’.  
   Then for each log\_req\_cov() (T2) a separate line is written into the coverage file with a) the given requirement label, b) the name of the testcase, and c) the result of the check as PASS (unless marked as FAIL or unexpected alerts have occurred, in which case it will be marked as FAIL).  
   Finally when finalize\_req\_cov() (T3) is executed a closing check is made and written to the coverage file.  
   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 coverage file.
4. After the testcase has been executed, the overall coverage summary (CS) can be found by executing the python script run\_spec\_cov.py.

This script traverses the requirements list and testcase coverage file and from that generates the specification coverage summary (CS).  
Each requirement is listed only once in the specification coverage. If a requirement has one or more FAIL in a coverage file, the result is NON\_COMPLIANT for that requirement.  
For a simple scenario with a single testcase the testcase coverage file and the specification coverage files yield the same information, but the specification coverage potentially with fewer lines.

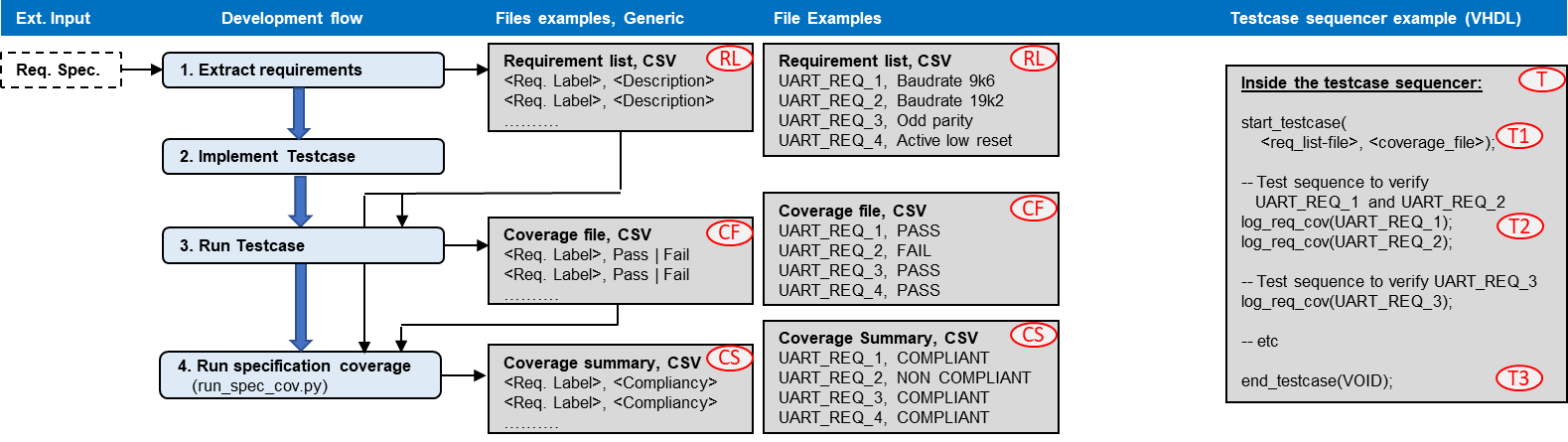


Figure 1: Simplest possible specification coverage

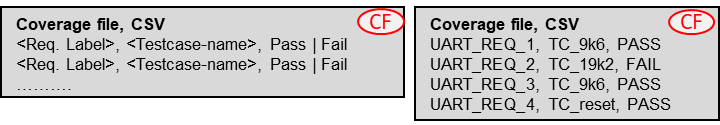
Simple usage, with multiple testcases

Many ver

Simple usage, with multiple testcases

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

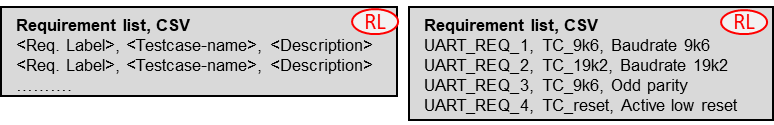
However, if your tests are split on multiple testcases, but with no requirement to which testcase tests what, then you can apply almost the same simple approach as the simplest case above.

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 coverage files for each testcase. Hence, after simulation you should end up with as many coverage files as testcases. In these coverage files the testcase name will be provided on every requirement status line as shown to the right..

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 coverage files.

Then the coverage summary is generated exactly as before.

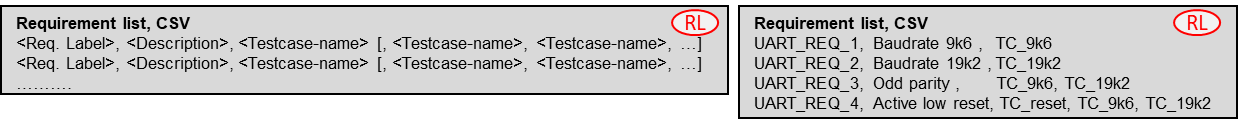
Multiple testcases – with strict requirement test vs testcase relation

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.

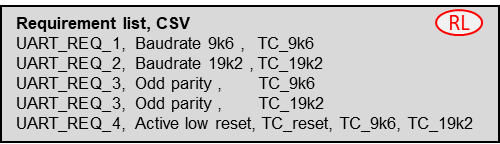
The coverage file will of course then also include the testcase name as shown above. Other than that the inputs and outputs will be the same. Specifying testcase names in the requirement list will however not force a testcase check by itself. In order to check that a test is performed in the specified testcase ‘shared\_req\_vs\_cov\_strict\_testcase’ must be set to TRUE.

One major difference however, is that if the requirement is not tested in the specified testcase a testcase mismatch will be flagged.

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

In the previous examples all tests in all testcases had to pass for the overall specification coverage summary to pass.  
There may however be situations where a given requirement is tested in multiple testcases and it is sufficient that only 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.

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

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 TC\_9k6 and TC\_19k2.

Advanced usage – Requirement mapping

Requirement mapping just maps one or more requirements to another requirement.

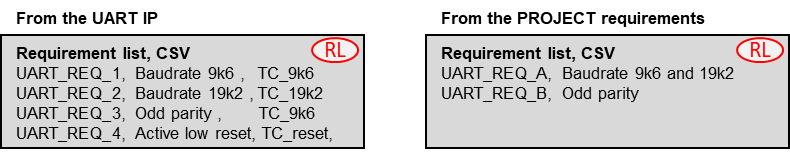
This is intended for two different use cases:

**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 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.

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.

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 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).

**Mapping of requirements to multiple sub-requirements**

**Requirement list, CSV**

UART\_REQ\_GENERAL, Baudrates 9k6 and 19k2, odd and even parity

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.

If you want to split this into more specific requirements you have several options, with some potential options shown below.

**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

**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

**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

1. Rename the requirements
2. Extend the names
3. 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.

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.

VHDL Package

A vital part of the specification coverage concept is the VHDL testbench methods. These methods are described in Table 11. The methods are located inside the *spec\_cov\_methods\_pkg.vhd* file in the *src/* directory of this VIP.

UVVM Adaptation Package Configuration

NB: Gjør om til record. Kun som konstant C\_SPEC\_COV.\* (trenger vel ikke noe som shared variable)

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 initialize\_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 initialize\_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 testcases (as described above, under “When minimum one testcase must pass”). |
| C\_CSV\_FILE\_MAX\_LINE\_LENGTH | Positive | 256 | Maximum length of each line in the req\_to\_tc\_map CSV file. |
| fail\_on\_alert\_mismatch\_severity |  |  |  |
| shared\_req\_vs\_cov\_strict\_testcase\_checking | boolean | false | When this shared variable is set true, the Specification Coverage checking will require that the input “*Requirement to Testcase map file*” contains the testcase field. Only the testcase 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\_REQ\_1; Start bit polarity test; TC\_UART\_1*  Then it will not be sufficient to run *log\_req\_cov* from any testcase. It will have to be reported with:  *log\_req\_cov(“FPGA\_REQ\_1”, “TC\_UART\_1”)*  in order to pass. A mismatch between requirement and testcase will result in a warning, and the requirement will not be marked as compliant. |

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

|  |  |
| --- | --- |
| **Message Id** | **Description** |
| 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. |

VHDL Methods Details

Table 11 VHDL Methods

|  |  |  |
| --- | --- | --- |
| **Procedure** | **Parameters and examples** | **Description** |
| initialize\_req\_cov() | req\_to\_tc\_map\_file(string), output\_file(string)    **Example** initialize\_req\_cov(“c:/test\_folder/my\_req\_to\_testcase\_map.csv”, “output\_file\_path.csv");  initialize\_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 |
| 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 *initialize\_req\_cov()* procedure. The *log\_req\_cov()* will look up the specified requirement and testcase in the req\_to\_tc\_map\_file 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.   |  |  | | --- | --- | | - *requirement:* | String with the requirement tag. Must match a requirement tag in the file specified in initialize\_req\_cov(). | | - *testcase*: | String with the testcase tag. Must match a testcase tag in the file specified in initialize\_req\_cov(). | | - *passed:* | Boolean value with the result. True indicates passed, false indicated not passed. | | *- fail\_on\_alert\_mismatch\_severity:* | Which alert mismatch 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 |
| finalize\_req\_cov() | VOID(t\_void)  **Example** finalize\_req\_cov(VOID); | Ends the requirement coverage process in a test. This procedure writes a closing check to the output file specified in initialize\_req\_cov. This check is used later by the run\_spec\_cov.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 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 | csv\_file\_reader\_pkg.vhd | Package for reading and parsing of CSV input files |
| bitvis\_vip\_spec\_cov | spec\_coverage\_methods.vhd | Specification Coverage 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 Coverage 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\_cov.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 2 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 4 in  Table 4. Either this argument OR the --resultlistfile/-l argument is mandatory. Both can’t be used at the same time. |
| --resultlistfile | -l | --resultlistfile 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 resultfile 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 5 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 1 in Table 5.  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*  *-l test\_suite\_input\_files.txt*  *--subrequirements 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.

**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.