URG, Unified Coverage Reporting User Guide

Version C-2009.06
June 2009

Comments?
E-mail your comments about this manual to:
vcs_support@synopsys.com.

Contents

1

Unified Coverage Reporting

The Unified Report Generator (URG) generates combined reports for all types of coverage information. You can view these reports organized by the design hierarchy, module lists, or coverage groups. You can also view the overall summary of the entire design/testbench on the dashboard. The reports consist of a set of HTML or text files.

The HTML version of the reports take the form of multiple interlinked HTML files. For example, a `hierarchy.html` file shows the design's hierarchy and contains links to individual pages for each module and its instances.

The HTML file that URG writes can be read by any web browser that supports CSS (Cascading Style Sheets) level 1, which includes Internet Explorer (IE) 5.0 and later versions, any version of Opera, and the later versions of Netscape Firefox 1.5.

Note:

Vera generates only group coverage data while VCS can generate group, assertion and code coverage data. You can generate assertion coverage data in Vera using OVASIM (in the `$VERA_HOME/ovasim` directory). You can use URG to generate coverage reports for all these types of coverage data.

This chapter contains the following sections:

- "Supported Metrics"
- "Invoking URG"
- "Using the Command-line Options"
- "Editing the Coverage Database"
- "Report Files"
- "Detailed Coverage Reports"
- "Parallel Merging"
- "Flexible Merging"
- "Grading and Coverage Analysis"
- "Reporting Element Holes"
- "Analyzing Trend Charts"

Supported Metrics

URG supports most of the existing coverage metrics. However, support for certain metrics is currently not available.

URG generates reports that include the following metrics:

- Code coverage
  - Line
  - Condition
  - Toggle
  - FSM
  - Branch
- Assertions
- Group coverage

The following reports are generated as `.html` or `.txt` files:

- dashboard
- hierarchy
- modlist
- groups
- `mod`*N*`.html` *or* `modinfo.txt`
- `grp`*N*`.html` *or* `grpinfo.txt`
- asserts

- assertdensities
- tests

URG currently does not support path and assign-toggle coverage.

Invoking URG

The usage model to invoke URG is as follows:

1. Compile the test file.

   `% vcs [`*`compile_options`*`]`

2. Simulate the test file.

   `% simv [`*`runtime_options`*`]`

3. Run URG command

   ```
   urg -dir simv.vdb
   ```

You need to specify the directories containing coverage data files. These directories can be:

- `*.vdb` directories containing covergroup or assertion/property coverage data
- `*.cm` directories containing code coverage data

You can specify these directories using the `-dir` *`directory`* option on the command line. Any number of directories can be given after the `-dir` option, and at least one directory argument must be given.

URG is invoked from the command line, and writes merged coverage data into designated directories. By default:

- Group and Assertion coverage data is written to a `*.vdb` directory
- Code coverage data is written to a `*.cm` directory

Data files are grouped into tests based on the names of the files. Therefore, if you have the following data files, URG considers all of them as data for '`test1`'.

```
./simv.vdb/snps/coverage/db/testdata/test1
./simv.cm/coverage/verilog/test1.*
./simv.cm/coverage/vhdl/test1.*
./simv.vdb/fcov/test1.db
```

For group, code, and assertion coverage, you invoke URG as follows:

```
urg -dir simv.cm simv.vdb
```

You can use the `-metric` argument as follows to select which types of coverage you want to report:

```
-metric [+]line+cond+fsm+tgl+assert+group
```

For example:

```
urg -dir simv.cm simv.vdb -metric line+cond+group
```

If no `-metric` argument is given, all types of coverage in the indicated coverage directories are reported. An initial plus sign is not required, but is allowed.

URG generates the reports and places them in a directory; by default, this is the `urgReport` directory in the working directory. Each time URG is run, the report directory and all of its contents are removed and replaced by the new report files. You can use `-report mydir` option to save the generated reports in `mydir`.

For example:

```
urg -dir simv.cm simv.vdb -metric line+group -report
covreport
```

Since `urg` is a UNIX command, the arguments may include shell variables, absolute, or relative paths, such as:

```
urg -dir $MYDIR/foo.cm
urg -dir $MYDIR
urg -dir ~username/covd ~username/covd/simv1.cm
```

Flow Examples

Code coverage and assertion coverage can be generated by VCS, VCS MX, or Magellan. Code coverage information is collected in `.cm` directories and assertion coverage information is collected in `.vdb` directories. VCS, VCS MX, or Vera can generate group coverage and this information is collected in `.vdb` directories.

Example 1

To produce a unified report showing collected code coverage from directories `simv1.cm`, `simv2.cm`, and `formal1.cm`, the URG command is:

```
urg -dir simv1.cm simv2.cm formal1.cm
```

This would report on all metrics collected into these three directories, since no `-metric` options were given. This command would generate a set of HTML pages into the default directory, `urgReport`.

Example 2

To generate a combined report of all code and assertions coverage data from the examples above, use the following command:

```
urg -dir simv.vdb simv.cm simv1.cm simv2.cm formal1.cm
```

Example 3

If the `simv.cm` and `simv.vdb` are as shown below:

```
simv.cm/coverage/verilog/test.line
simv.cm/coverage/verilog/test.cond
simv.vdb/snps/coverage/db/testdata/test/
```

If you use the following command:

```
urg -dir simv.vdb simv.cm -dbname foo/merged
```

it will create the following contents using `foo` as the base name:

```
foo.cm/coverage/verilog/merged.line
foo.cm/coverage/verilog/merged.cond
foo.vdb/snps/coverage/db/testdata/merged
```

Using the Command-line Options

URG supports the following command-line options:

`-cond exclude` *`file_name`*

Specifies excluded conditions.

`-dbname` *`name`*

Specifies the merged database name (see “Example 3”). Note that a single merged database file will be generated from the original tests. Grading for each original test cannot be done over this merged database.

`-dir` *`directory_name`*

Specifies coverage data directories.

`-f file_name`

Specifies multiple directories for source data in a file. You can also specify the `-f` option when there are multiple coverage directories.

For example, you can use the following command line options to generate the URG report:

```
% urg -dir ./simv1.cm ./simv2.cm ./simv3.cm ./simv1.vdb
./simv2.vdb ./simv3.vdb
```

The size of the command line might exceed the Linux limits while adding more and more coverage databases to the URG command line. In such cases, the `-f` option would suffice the requirement.

```
% urg -dir ./simv1.cm -f file_list
```

Here, `file_list` (./simv2.cm ./simv3.cm ./simv1.vdb ./simv2.vdb ./simv3.vdb) contains the databases other than the ./simv1.cm. These coverage databases can be mentioned either with the absolute or the relative path in `file_list`. You should at least pass one directory (./simv1.cm) if you are using the `-dir` option.

```
% urg -f file_list
```

Here, `file_list` contains all the databases (./simv1.cm ./simv2.cm ./simv3.cm ./simv1.vdb ./simv2.vdb ./simv3.vdb) mentioned either with the absolute or the relative path.

`-format text`

Generates text report files instead of HTML report files.

`-fsm disable_sequence`

Does not report FSM sequences.

`-fsm disable_loop`

Does not report FSM sequences containing loops.

`-full64`

Runs URG in 64-bit mode.

`-grade [quick|greedy|score] [goal` *`R`*`] [timelimit` *`N`*`]`

`[maxtests` *`N`*`] [minincr` *`R`*`] [reqtests` *`file_name`*`]`

For more information about grading tests, see “Grading Tests”.

`-group maxmissing` *`N`*

Shows at most *N* uncovered bins for any coverpoint or cross in group coverage reports. The default value is 256.

`-group ratio`

Instructs URG to compute covergroup scores and overall group scores as a simple ratio of the number of bins covered over the total number of coverable bins. The result is an average score of its variants and this option is shown in the `dashboard.html` page.

`-help` and `-h`

Shows command line and options supported by URG.

`-hier`

Specifies the module, definitions, instances, subhierarchies, and source files that you want URG either to exclude from reporting or exclusively compile for coverage reporting. This option is used with the configuration file.

If the cover dir is `simv1.cm`, the hier config file is `hfile1`. You can use the command:

```
urg -dir simv1.cm -hier hfile1
```

`-high` *N*

Shows any coverage number above `N` percent in green.

`-ID`

Displays the Host ID or dongle ID for your machine.

`-line nocasedef`

Excludes default cases in case statements from line coverage reports.

`-log` *`file_name`*

Sends diagnostics to *`file_name`* instead of to stdout/stderr.

`-low` *`N`*

Shows any coverage number below *`N`* percent in red.

`-mapfile`

Allows you to specify an instance in your “base design” for which you want to merge coverage data for two different designs.

```
urg -dir base.cm -dir input.cm -mapfile file_name
```

Where, `file_name` is the mapping configuration file.

`-map` *`module_name`*

Maps subhierarchy code coverage from one design to another. This option is not available for assert or group coverage. The full hierarchy is generated in `hierarchy.html` file.

`-metric [line+cond+fsm+tgl+branch+assert+group]`

Limits report to specified metrics.

`-noreport`

Generates only the merged results when used with `-dbname`; this option does not generate reports.

`-parallel [`*`machine_file`*`]`

Specifies merging the results from multiple tests in parallel, see “Additional Options for Parallel Merging” and “Specifying the Machines that Perform the Jobs”.

`-plan`

Annotates the user-defined HVP (Hierarchical Verification Plan) data.

For example,

```
urg -plan yourPlan.hvp -dir yourCoverageDB.vdb -annotate
bugRate.txt
```

For more information on how to generate the HVP using the VMM Planner Editor, see the *VMM Planner User Guide*.

`-report` *`mydir`*

Generates a report in `mydir` instead of default directory.

`-scorefile` *`file_name`*

Specifies a file containing different weights for each metric. The metrics that are not specified in the score file will have the default weight one.

`-show availabletests`

Lists the tests found in each in each of the specified `-dir` directories and exits without generating a report. You can edit the resulting list and use it with the `-tests` option.

`-show legalonly`

Shows only legal coverable objects and suppresses showing illegal coverable objects.

`-show maxtests` *`N`*

Specifies the maximum number of tests that are displayed with `-show tests`.

`-show tests`

Lists all the tests that covered a given object. Only supported for assertion and testbench coverage.

`-split metric`

Splits all module and instance reports by metric.

`-split` *`N`*

Controls the size of all files before being split. The argument is an integer specifying the maximum size in bytes for any generated file. This number is used as a guideline, not an absolute limit. The default value is 200KB.

`-tests` *`file_name`*

Specifies the file name containing the list of tests in the directories specified using `-dir` option, for which coverage data is reported. This is a text file with one test on each line. The test names used in this file must match the test names obtained with the "`-show availabletests`" switch.

You can use `urg -dir` *`directory_name`* `-show availabletests` to show all the tests listed in the correct format for the *`file_name`* file, then select the tests you want to report. If the file contains a test that does not appear in any of the specified directories, then URG displays an error message and exits.

`-trend [`*`trend_options`*`]`

Specifies the options to generate a trend chart. See the section “Analyzing Trend Charts”.

`+urg+lic+wait`

Waits for a network license if none is available when the job starts.

Note:

The `-tb maxmissing` *N* option has been deprecated. You can use the `-group maxmissing` *N* option which has the same function.

Additional Options for Parallel Merging

The options for parallel merging are as follows:

`-parallel [`*machine_file*`]`

Specifies merging the results from multiple tests in parallel. For more information, see "Specifying the Machines that Perform the Jobs".

`-grid ["`*GRID_arguments*`"] [-sub` *submit_command* | `-del` *delete_command*`]`

Specifies using a grid computing engine for parallel merging of the results and provides an optional means to pass arguments to the grid engine. For additional information, see "Using a GRID Computing Engine".

`-lsf ["`*LSF_arguments*`"] [-sub` *submit_command* | `-del` *delete_command*`]`

Specifies using a LSF (Load Sharing Facility) engine for parallel merging of the results and provides an optional means to pass arguments to the LSF engine. For additional information, see "Using LSF".

`-parallel_split` *integer*

Specifies the number of test results in a “merging” (or clump) of results that URG merges together at any one time on its way to merging all the results in parallel. For additional information, see “Specifying the Number of Tests in a Merging”.

Instance Coverage Score Option

By default, URG computes the overall score for a test from the cumulative coverage score for each of the cover groups in that test. This can be misleading in situations where a user has not enabled instance coverage for a particular covergroup. While the cumulative coverage for a covergroup that is instantiated more than once might be 100%, the coverage score for individual instances can be well below that. The final overall test score, which does not take into account the instance coverage, can different from the score for instance-based coverage.

The `urg` command `-group instcov_for_score` option invokes a coverage score computation that involves the instance coverage:

```
%urg -group instcov_for_score
```

With the `-group instcov_for_score` option, the overall score for a test takes into account cumulative coverage and instance coverage scores (for covergroups with instance coverage enabled), to provide a better picture of the coverage results:

Example 1-1 Sample Code for Cumulative vs. Instance-Only Coverage Score

```
class ex {
   bit a;
   coverage_group cov {
      cumulative = 0;
      sample_event = @(posedge CLOCK);
      sample a;
   }
}
coverage_group t_cov {
   …
}
```

```
coverage_group p_cov {
   …
}
program test {
   ex ex1 = new;
   ex ex2 = new;
   t_cov cov1 = new;
   p_cov cov2 = new;
   ….
   ex1.a = 0;
   ex2.a = 1;
   @(posedge CLOCK);
   …
}
```

In Example 1-1, assume that the cumulative coverage for `t_cov` and `p_cov` is 40% and 100% respectively. The instance coverage for both instances of `ex::cov` is 50% but the cumulative coverage for `ex::cov` is 100%: Each possible value for the `a` bit was hit once in each of the two instances of `ex::cov`. This means that, individually, `ex1` has 50% coverage on `ex:cov` and `ex2` has 50% coverage on `ex:cov`. However, the cumulative coverage of `ex:cov` is 100% because it has reached both possible values.

The overall (cumulative) score for the test is computed as:

> (cumulative_score(`t_cov`) + cumulative_score(`p_cov`) + cumulative_score(`ex::cov`))/3

or

> (40 + 100 + 100)/3

By this calculation, the overall score is 80%. This hides the fact that `ex1.a` was never 1 and `ex2.a` was never 0.

For a better indication of the overall coverage, use the `-group instcov_for_score` option to compute the overall score for `ex::cov`. For Example 1-1, the overall cumulative coverage score for the test, using the instance coverage of each instance of the `ex::cov` covergroup, is computed as:

(cumulative_score(`t_cov`) + cumulative_score(`p_cov`) + instance_coverage(`ex1`) + instance_coverage(`ex2`))/4

or

(40 + 50 + 50 + 100)/4

By this calculation, the overall score is 60%. This method assigns equal importance to each instance of the covergroup `ex::cov`.

The corresponding `groups.(txt|html)` file that appears in the `urgReport` directory is shown in Table 1-1:

Table 1-1 groups.html|txt file for the example

SCORE	INSTANCES	WEIGHT	GOAL	NAME
40	--	1	100	t_cov
50	50	1	100	ex::cov
100	--	1	100	p_cov

Note that the row corresponding to the covergroup `ex::cov` shows the average of the instance score for all the instances of `ex::cov` instead of the cumulative score.

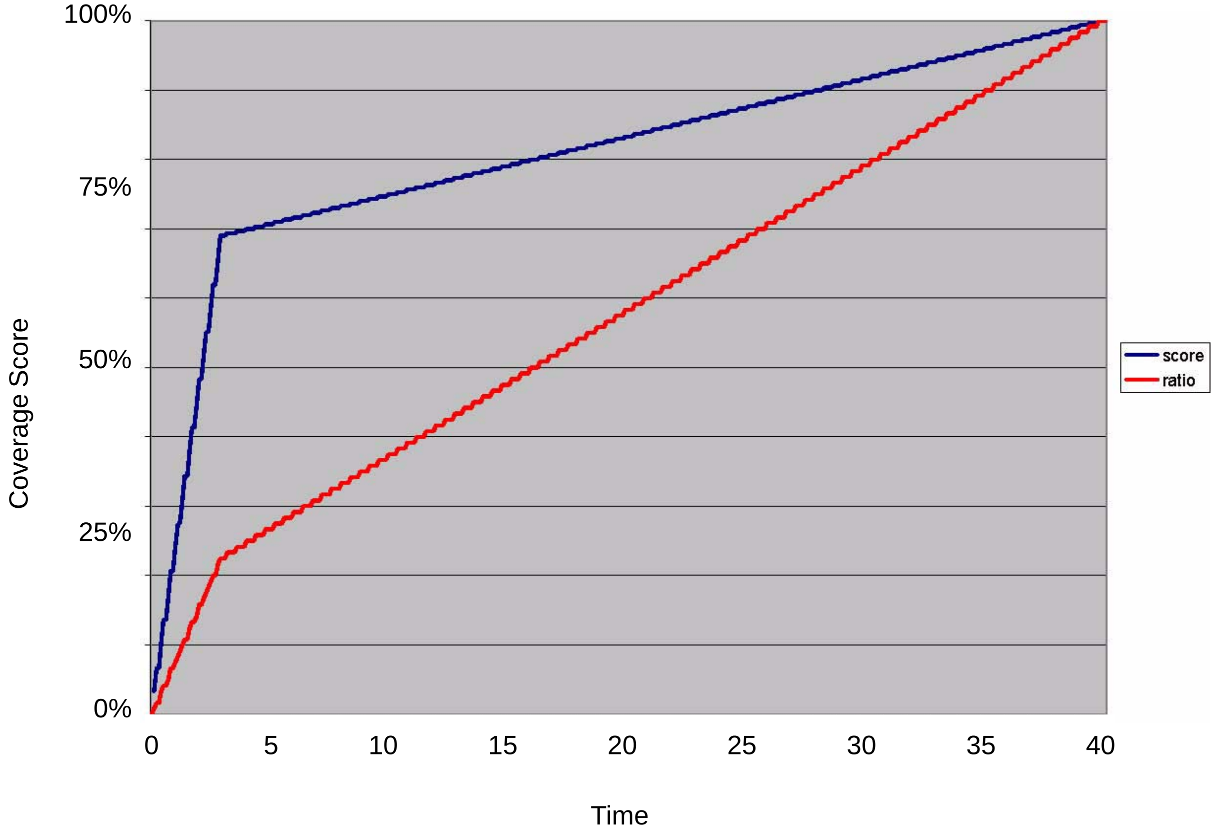
Covergroup Score Covered/Coverable Ratio Option

By default, URG computes covergroup scores as the average score of all the coverpoints and crosses. The overall covergroup score for a design is the average score of all of the covergroups.

This can lead to a nonintuitive increase or decrease in the score when new bins are covered or become uncovered. This is mainly because the number of bins in each cover group is not same, so their weight in the overall coverage score is also not same. Therefore, instead of the score linearly improving as each new bin is covered, the effect might be disproportionately high or low.

In Figure 1-1, the blue line shows the change in a covergroup score as new bins are covered, using the default coverage score computation. The red line shows the score computed as a simple ratio of covered bins over coverable bins, for a sample covergroup as its coverage improves day to day.

Figure 1-1   Typical coverage score changes over time



The reason the blue line has a sharp rise at first before flattening out is that the individual variable bins get covered quickly, but the cross between them has many more bins. Since the overall coverage score is computed as the average of the variables coverage score and the coverage score of the cross between them, the variables effectively have a higher weight in the score computation. In the ratio computation (the red line), each variable or cross is weighted by the number of bins it has, so the line is much smoother.

The `-group ratio` option of `urg` can be used to specify that the covered/coverable ratio is to be used to compute covergroup scores (the red line in Figure 1-1) instead of the default method (the blue line). The usage is:

```
% urg -group ratio ...
```

When `-group ratio` flag is used, the `urgGroup::genCoverageData` function computes covergroup scores and the overall group score using a simple ratio of covered bins divided by coverable bins. This flag (like all other flags passed to URG) is shown in the dashboard.html page.

Note:

> The score of a group definition is still the average score of its variants, even when `-group ratio` is used.

Editing the Coverage Database

A URG command option, `-group db_edit_file`, opens the coverage database for editing. An example of the URG command with the `-group db_edit_file` option is follows.

```
urg -dir first.vdb -format text -report hier_rep
   -group db_edit_file db-edit-filename -dbname save/edited
```

The *`db_edit_filename`* is a user-written file that contains `funccov` and `assert` statements in the format described in "db_edit_file Syntax".

A `funccov` statement is used to reset or delete functional coverage data. An `assert` statement is used to reset or delete assertion coverage data. Examples of those statements are shown in "db_edit_file Syntax".

Note:
You cannot edit the code coverage database.

This section contains the following topics:

- "db_edit_file Syntax"
- "Editing the Function Coverage Database"
- "Editing the Assertion Coverage Database"

db_edit_file Syntax

The syntax for editing covergroups in the coverage database is:

```
begin funccov (reset|delete) (module|tree)
```

```
(module-name|instance-name)
covergroup-name [optional coverpoints or crosses]
end
```

The `module` parameter can be a module, interface, or program.

The `tree` parameter can be a module instance, interface instance, or program instance. Pathnames in trees must use periods (".") as instance delimiters.

The syntax for editing assertions in the coverage database is:

```
begin assert (reset|delete) (module|tree)
(module-name|instance-name) [optional assertions]
end
```

Editing the Function Coverage Database

This section contains the following topics:

- "Resetting Covergroups or Coverpoints"
- "Removing Covergroups from the Database"

Resetting Covergroups or Coverpoints

Example 1-2 resets the covergroup named `gc` in the `top.i1` module instance.

Example 1-2

```
begin funccov reset tree top.i1 gc end
```

Example 1-3 resets the covergroup named `gc` under all instances of the `my_mod` module definition.

Example 1-3

```
begin funccov reset module my_mod gc end
```

Example 1-4 resets the coverpoint or crosspoint named `ra` under the covergroup named `gc` under all instances of the `my_mod` module definition.

Example 1-4

```
begin funccov reset module my_mod gc ra end
```

Example 1-5 resets the coverpoint or crosspoint named `ra` under the covergroup named `gc` under all instances of `top.i1`.

Example 1-5

```
begin funccov reset tree top.i1 gc ra end
```

Example 1-6 deletes the covergroup named `gc` under all instances of the `my_mod` module definition.

Example 1-6

```
begin funccov delete module my_mod gc end
```

Removing Covergroups from the Database

Example 1-7 deletes the covergroup named `gc` under all instances of `top.i1`.

Example 1-7

```
begin funccov delete tree top.i1 gc end
```

Editing the Assertion Coverage Database

The syntax is similar assertions is similar to the syntax for functional covergroups:

```
begin assert (reset|delete) (module|tree)
(module_name|instance_name) [optional assertions] end
```

This section contains the following topics:

- "Resetting the Coverage Scores for Assertions"
- "Removing Assertions from the Coverage Database"

Resetting the Coverage Scores for Assertions

Example 1-8 resets the hit counts for the `mid_first` assertion in the `top.i1` instance.

Example 1-8

```
begin assert reset tree top.i1 mid_first end
```

Example 1-9 resets hit counts of all assertions under the `top.i1` instance.

Example 1-9

```
begin assert reset tree top.i1 end
```

Removing Assertions from the Coverage Database

Example 1-10 deletes the `mid_first` assertion from the `top.i1` instance.

Example 1-10

```
begin assert delete tree top.i1 mid_first end
```

Example 1-11 deletes all assertions under the `top.i1` instance.

Example 1-11

```
begin assert delete tree top.i1 end
```

Example 1-12 deletes the `mid_first` assertion from all instances of the mid module.

Example 1-12

```
begin assert delete module mid mid_first end
```

Example 1-13 deletes all assertions from all instances of the mid module.

Example 1-13

```
begin assert delete module mid end
```

Report Files

URG generates a number of report files and a common dashboard that you can use to access the report files. The report files can be either HTML or text files. The HTML files contain a navigation menu and also follow a color code that helps to visually assess the coverage metrics.

In addition, URG also generates a `session.xml` file which contains VCS basic coverage data and VMM Planner metrics data for use with trend analysis. You use the `-trend` option to parse and analyze `session.xml` file to produce a series of trend charts.

Common Report Elements

Coverage data boxes are used throughout the URG report files. These are tables containing one box for each type of coverage.

The HTML version includes color-coded boxes, or boxes left empty if no coverage data for the coverage type represented by the box was collected. For example:

Figure 1-2 Example of a Coverage Table

SCORE	LINE	COND	TOGGLE	FSM	ASSERT	GROUP
46.15	87.08	40.11	20.24		50.00	33.33

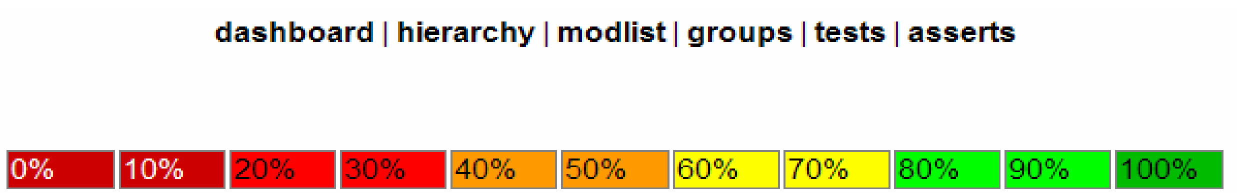
In the above example, the first box shows the overall score of all metrics. By default, this is the simple average of all the metric percentages. You can control the way the score is computed with the `-scorefile` option. For additional information, see “Grading and the -scorefile Option”.

In this example, URG displays line, condition, toggle, assertion, and group coverage collected. FSM coverage was turned on, but no FSM was found in this region.

The LINE box is green because it falls in the upper range of target values. As shown in Figure 1-3, values display in a range of 11 colors from red (low) to green (high). These colors are graduated every 10 percentage points (with 100 being the 11th class).

Each report file contains a legend showing the cutoff percentages for each color. Each file also contains a common navigation menu at the top. The menu is a simple list of the top-level pages that allow you to go directly to any of the main pages, including the hierarchy, modlist, groups, dashboard, asserts, or tests files.

Figure 1-3 Color Legend for Coverage Tables



Score tables have more than one row and are sortable by clicking any of the column headings: SCORE, LINE, COND, TOGGLE, TEST, and so on. Note that the hierarchy report does not support sorting, even for contiguous groups of instances under the same parent.

A text version of the report shown in Figure 1-2 is as follows:

SCORE	LINE	COND	TOGGLE	FSM	ASSERT	GROUP
46.15	87.08	40.11	20.24	--	50.00	33.33

The Dashboard File

The dashboard file (`dashboard.html` or `dashboard.txt`) describes the top-level view of all coverage data including coverage data boxes for the database as a whole. The following is an example of a dashboard report:

Figure 1-4 Example of a Dashboard Report

Dashboard

dashboard | **hierarchy** | **modlist** | **groups** | **tests** | **asserts**

Date: *day month date hour:minute:second year*
User: *username*
Version: Y-2006.06-SP1-7
Command line: urg -dir mysimv.cm mysimv.vdb

Total Coverage Summary

SCORE	LINE	COND	TOGGLE	FSM	ASSERT	GROUP
46.15	87.08	40.11	20.24		50.00	33.33

Hierarchical coverage data for top-level instances

SCORE	LINE	TOGGLE	FSM	ASSERT	NAME
38.46	100.00	15.38		0.00	grademe

Assertions density report

STATEMENTS	CHECKERS	ASSERTS	COVERS	CHECKERS/ STATEMENTS (%)	ASSERTS/ STATEMENTS (%)	COVERS/ STATEMENTS (%)
77	0	0	10	0	0	12.99

Note:

- In this example report, the boldface words: **dashboard**, **hierarchy**, **modlist**, **tests**, **groups**, **asserts**, and **Assertions density report**, are hyperlinked to the corresponding top-level files.

- If there is no group coverage information, the **groups** will not appear in boldface.

The Hierarchy File

The hierarchy file (`hierarchy.html` or `hierarchy.txt`) contains indented lists of all modules, interfaces, and component instances in the design. The indentation corresponds to the design hierarchy, that is, the child modules are indented underneath their parent modules.

The coverage data boxes for each instance in the `hierarchy.html` file shows the entire coverage information for the subtrees of the instantiation tree. The name of the instance is hyperlinked to its corresponding module in the `mod`*`N`*`.html` page. Each metric in the coverage data box is hyperlinked to the corresponding coverage metric section of the module instance in the `mod`*`N`*`.html` file.

Figure 1-5 displays a partial section of a `hierarchy.html` page. The data shown is the cumulative coverage information of the entire subtrees at each instance. For example, the coverage shown for `MMRk0` is the coverage for that instance plus the coverage for `DELAY_MOD` and `ovac_CHKR`.

To see the coverage of instance `MMRk0`, click `MMRk0` to open the coverage report. Notice that a mouseover will change the color of a hypertext link to red and hovering over a score turns the score red.

Figure 1-5   Example of a Hierarchy File

SCORE	LINE	COND	TOGGLE	ASSERT	
33.33	100.00	0.00	0.00		MOPS17
48.83	87.07	40.13	20.24	47.89	DUT

SCORE	LINE	COND	TOGGLE	ASSERT	
34.14	49.40	27.46	9.70	50.00	MMR00

SCORE	LINE	COND	TOGGLE	ASSERT	
42.67	59.88	38.68	22.11	50.00	MMRk0

SCORE	LINE	COND	TOGGLE	ASSERT	
67.95	100.00		35.90		DELAY_MOD
100.00		100.00	100.00	50.00	ovac_CHKR

To avoid overwhelming the browser with a single huge HTML file. a hierarchy tree may be broken into multiple pages if the design is very large. When this happens, you can click on '`subtree`' to see the elided part of the design.

Figure 1-6   Example of a Hierarchy Broken into Multiple Pages

SCORE	LINE	COND	TOGGLE	FSM	ASSERT	
63.27	49.93	92.01	95.84	45.24	33.33	HNWOOVISP_0

subtree...

If an entire subtree in the design has no coverage data, the instances in that subtree will not have links to `mod`*`N`*`.html` page. Although, they will still be shown in `hierarchy.html`, there will be empty coverage data boxes.

If a particular instance itself has no coverage data, but one of its children does, the instance has a link to a `mod`*`N`*`.html` page. Therefore, you can still traverse through the coverage data in the `mod`*`N`*`.html` page to the children or parents.

If there is no design coverage information, no hierarchy will be shown. The hierarchy page will not be generated and the hierarchy hypertext link will not appear in boldface.

The Modlist File

The modlist file (`midlist.html` or `modlist.txt`) contains a flat list of all modules, entity/architectures, and interfaces in the design. The module (or entity, or interface) names in the HTML file link to the corresponding `mod`*N*`.html` page. The entries, without indentation, are similar to those in `hierarchy.html`, but the labels are module names rather than instance names. The coverage data boxes show the accumulated coverage information for all instances of the module (or entity/architecture, or interface).

Score tables having more than one row are sortable by clicking any column heading: SCORE, LINE, COND, TOGGLE, FSM, ASSERT, and NAME.

Figure 1-7 Example of a Modlist File

SCORE	LINE	COND	TOGGLE	FSM ↑	ASSERT	NAME
89.65	98.77	85.71	63.79	100.00	100.00	JPRHFG
97.61	100.00	96.23	94.23	100.00		DstQueueAllocSco4Bit
92.12	100.00	95.00	82.26	83.33	100.00	TR
89.25	94.12	97.71	81.84	83.33		MOPS17
67.72	68.99	65.00	95.21	41.67		hik72bit
69.11	65.64	85.71	85.96	39.13		prwnDR0
65.90	70.24	59.26	97.27	36.84		HnOpvistidv_gtn

In this example, the FSM column has been sorted by FSM score in best-first order. If there is no assertion or code coverage information, URG does not generate the hierarchy or modlist files.

The Groups File

The groups file (`groups.html` or `groups.txt`) contains a flat list of coverage group definitions with coverage data boxes sortable by clicking any column heading. The data boxes show the coverage information of the coverage group.

The link from each group leads to a `grpN.html` file.

The following example shows three coverage groups as displayed in the `groups.html` page.

Figure 1-8 Example of a Groups File

SCORE	INSTANCES	WEIGHT	GOAL	NAME
8.00	8.00	1	100	m1::abc1
87.25	75.00	1	100	c::myg
100.00	100.00	1	100	scnr::z

If there is no covergroup coverage data, no groups will be shown and the groups page will not be generated.

The Tests File

The tests report file (`tests.html` or `tests.txt`) has several different formats depending on if the grading option is applied and what argument is used. Refer to “Examples Using the Grading Option” for details. The default file format of a `tests.html` is shown in Figure 1-9.

Figure 1-9   Example of a Tests File

Total Coverage Summary

SCORE	LINE	COND	TOGGLE	ASSERT
64.80	95.20	45.83	18.18	100.00

Data from the following tests was used to generate this report

mysimv/test0
mysimv/test1
mysimv/test2
mysimv/test3
mysimv/test4
mysimv/test5
mysimv/test6
mysimv/test7
mysimv/test8
mysimv/test9

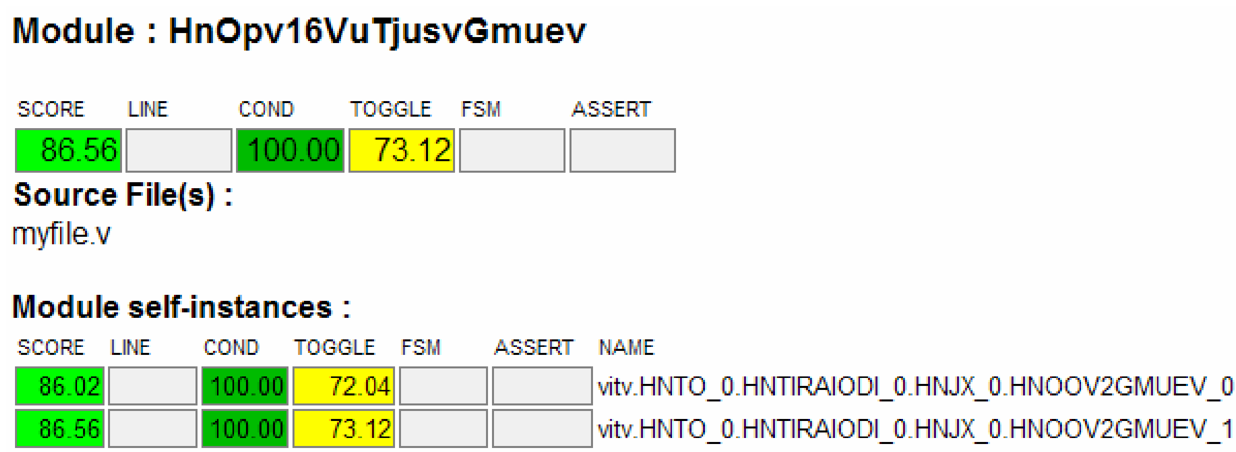
The mod*N* File

Each `mod`*`N`*`.html` file contains the summary coverage information for a module, entity/architecture, or interface. Unless the `mod`*`N`*`.html` file has been split for size, it also contains coverage information for each of the instances of the module. If the file is very large or has a large number of instances, the individual data for each instance and the module itself is put in a separate `mod`*`N`*`_`*`M`*`.html` file.

Each `mod`*`N`*`.html` table has a header section and a coverage data section. The header section contains either the name of a module or a list of self-instances of the module. The self-instances and the coverage metrics are all sortable by clicking any of the headings.

Coverage data boxes are shown for the module summary information and for each of its instances, so you can see the status of each. The coverage data boxes of the self-instances are smaller than that of the module.

Figure 1-10   Example of a Module File

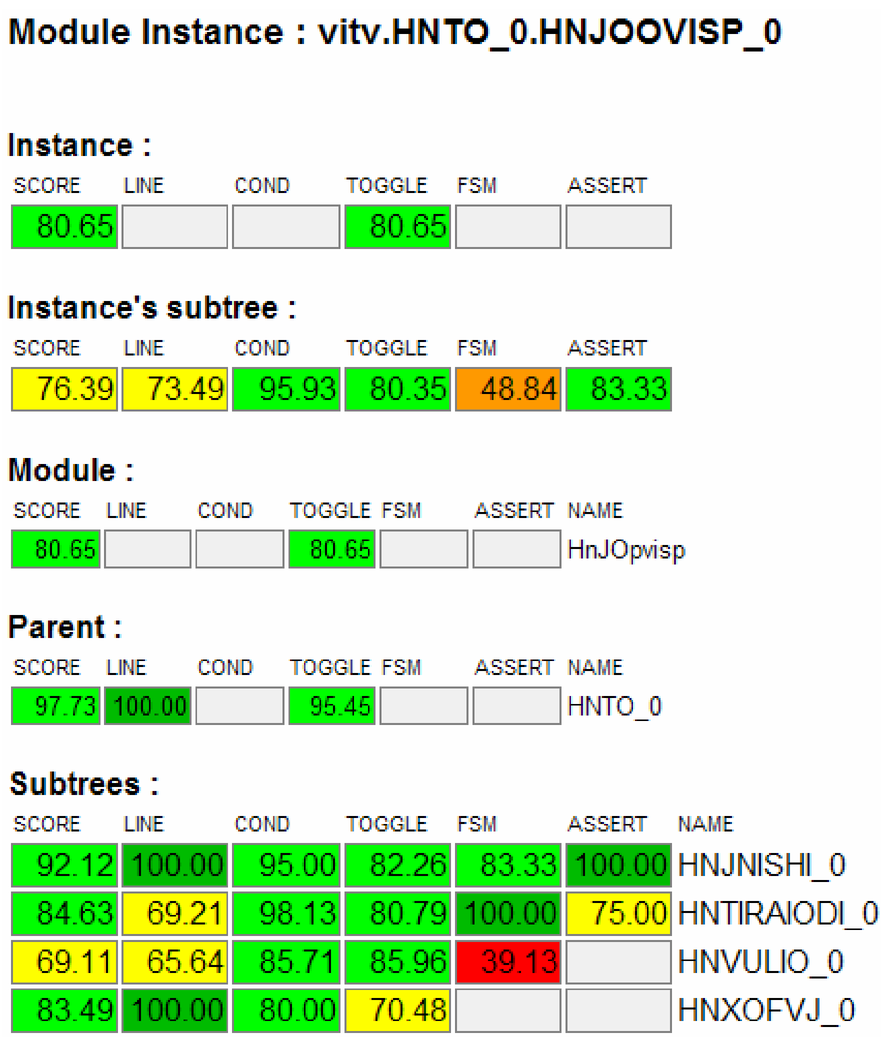


The self-instance hypertext links to the module instance information for each instance. These are the same links as from the `hierarchy.html` page for each of the instances.

The module instance sections also have header and coverage data sections. The header is similar to the module header, and it links to the parent instance and to child instances as shown.

Smaller coverage data boxes are used for the module summary information, the parent, and each child of the module instance. The names of each of these data boxes are hyperlinked to the respective module or module instance report.

Figure 1-11   Example of a Module Instance File



Module Instance : vitv.HNTO_0.HNJOOVISP_0

Instance :

SCORE	LINE	COND	TOGGLE	FSM	ASSERT
80.65			80.65		

Instance's subtree :

SCORE	LINE	COND	TOGGLE	FSM	ASSERT
76.39	73.49	95.93	80.35	48.84	83.33

Module :

SCORE	LINE	COND	TOGGLE	FSM	ASSERT	NAME
80.65			80.65			HnJOpvisp

Parent :

SCORE	LINE	COND	TOGGLE	FSM	ASSERT	NAME
97.73	100.00		95.45			HNTO_0

Subtrees :

SCORE	LINE	COND	TOGGLE	FSM	ASSERT	NAME
92.12	100.00	95.00	82.26	83.33	100.00	HNJNISHI_0
84.63	69.21	98.13	80.79	100.00	75.00	HNTIRAIODI_0
69.11	65.64	85.71	85.96	39.13		HNVULIO_0
83.49	100.00	80.00	70.48			HNXOFVJ_0

As for all report pages, the coverage data boxes column headings are linked to the corresponding coverage reports. In the above example, clicking on TOGGLE in the column headings opens the toggle coverage report for module instance `vitv.HNTO_0.HNJOOVISP_0`. These hypertext links provide a convenient way to navigate within a `mod`*`N`*`.html` page, and the only way to view coverage data reports if the `mod`*`N`*`.html` page has been split for size.

The grp*N* Files

Each `grp`*`N`*`.html` page has an outline similar to that of `mod`*`N`*`.html` pages. There is a header for each coverage group showing its name and a list of instances. Both sections have the appropriate data boxes linking to coverage data reports.

Each coverage group report contains a statistics table for both variables and crosses showing the overall coverage for each variable and cross. The header of a group shows the group's name, coverage (both covered and uncovered), goal, and weight, along with smaller data boxes for each of its child modules. For example:

Figure 1-12   Example of a grpN File

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Variables	112	78	34	51.95
Crosses	16385	16284	101	0.62

VARIABLES FOR GROUP **test::MyCov**

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT
i	16	5	11	68.75	100	1
j	16	5	11	68.75	100	1
k	16	5	11	68.75	100	1
mult	64	63	1	1.56	100	1

CROSSES FOR GROUP **test::MyCov**

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT
mycross0	16385	16284	101	0.62	100	1

The names of the variables and crosses in these tables are hyperlinked to their detailed reports.

Group instances have similar headers, with a link to the group summary information. Group instances have statistics tables and detailed reports in the same format as the group summary information shown in the previous example.

If there is a large number of covergroup bins, a `grpN.html` file may be split into multiple files. When this happens, the usual table will be replaced with an index table giving links to each of the sub-pages.

The Assertions File

The assertions file (`asserts.html` or `asserts.txt`) shows the scores of all assertions, cover properties, and cover sequences in separate tables sortable by clicking on the CATEGORY or SEVERITY heading. For example:

Figure 1-13 Example of an Assertions File:

ASSERTION	CATEGORY	SEVERITY	ATTEMPTS	REAL SUCCESSES	FAILURES	INCOMPLETE
test.ad2.ay	0	3	15	0	15	0
test.cm1.aoh1.assert_one_hot	1	1	10	0	9	1
test.cm1.aoh1.assert_example	0	2	18	0	18	0
test.cm1.aoh1.test_expr_x_or_z	2	0	18	0	18	0

By default, the assertions file will not show vacuous matches and the REAL SUCCESSES column indicates only the number of real matches. However, if you use the `-assert vacuous` option during runtime, URG will include a column named VACUOUS to indicate the number of vacuous matches and the column REAL SUCCESSES will have the total number of real and vacuous matches.

Figure 1-14   Cover Properties Table

PROPERTY	CATEGORY	SEVERITY	ATTEMPTS	MATCHES	VACUOUS MATCHES	INCOMPLETE
mymod.MC0	0	0	60	3	0	10
mymod.MC1	0	1	60	3	0	10
mymod.MC2	0	1	60	3	0	10
mymod.MC3	1	0	60	3	0	10
mymod.MC4	1	0	60	3	0	10
mymod.MC5	1	0	60	3	0	10
mymod.MC6	1	1	60	3	0	10
mymod.MC7	1	1	60	3	0	10
mymod.MC8	1	2	60	3	0	10
mymod.MC9	2	0	60	3	0	10

The `asserts.html` file is hyperlinked to the `assertdensities.html` file.

The Assert Densities File

The assertion densities file (`assertdensities.html` or `assertdensities.txt`) lists the number of statements, checkers, asserts, covers, and the individual ratios of checkers, asserts, and covers over the statements in the design and in each module instance. For example:

Figure 1-15 Example of an Assert Densities File

Assertions by Density

dashboard | **hierarchy** | **modlist** | **groups** | **asserts**

All modules

Statements	Checkers	Asserts	Covers	Checkers/ Statements	Asserts/ Statements	Covers/ Statements
61	1	7	5	1.64	11.48	8.20

Modules list

Statements	Checkers	Asserts	Covers	Checkers/ Statements	Asserts/ Statements	Covers/ Statements	
8	0	1	1	0	12.50	12.50	**assert_always**
10	0	1	0	0	10.00	0	**dev**
35	1	3	3	2.86	8.57	8.57	**test**
4	0	1	0	0	25.00	0	**intrctr**
4	0	1	1	0	25.00	25.00	**dcdr**

dashboard | **hierarchy** | **modlist** | **groups** | **asserts**

0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%

Detailed Coverage Reports

The following section discusses how each type of coverage is formatted in the reports.

Common Elements

URG introduces two basic types of format to display coverage results:

- Statistics table
- Table of coverable objects

Statistics tables are summaries of types of coverage elements. Each line in a statistics table reports the coverage for a class or category of object. Figure 1-16 shows an example of a statistics table for line coverage:

Figure 1-16 Example of a Statistics Table

	Total	Covered	Percent
Lines	16	12	75.00
Statements	16	12	75.00
Blocks	6	4	66.67
ALWAYS	5	3	60.00
FOR	1	1	100.00

Statistics tables are color-coded using the same color legend as for coverage tables shown in Figure 1-3.

The table of coverable objects shows the coverage results for individual coverable objects. Coverable objects do not have percentages; they are either covered or uncovered. Coverable object tables show covered (and observed) objects in green and uncovered objects in red. Figure 1-22 in the Condition Coverage Section shows a coverage data table for condition coverage.

For all types of coverage, the data section begins with a statistics table showing the basic categories (for example, lines, statements, and blocks, or logical and non-logical conditions). This is followed by a table of coverable objects.

Note that several metrics have options that change exactly what is covered and how to display it. For example, use the condition coverage option `-cm_cond allops` to control which vectors and conditions are monitored. For more information, refer to the *VCS/ VCS MX Coverage Metrics User Guide*.

The Line Coverage Report

The line coverage report starts with a table listing individual statistics of each always block, initial block, VHDL process, and continuous assignment. For example:

Figure 1-17 Example of a Line Coverage Section Report:

Line Coverage for Module : GmSequence

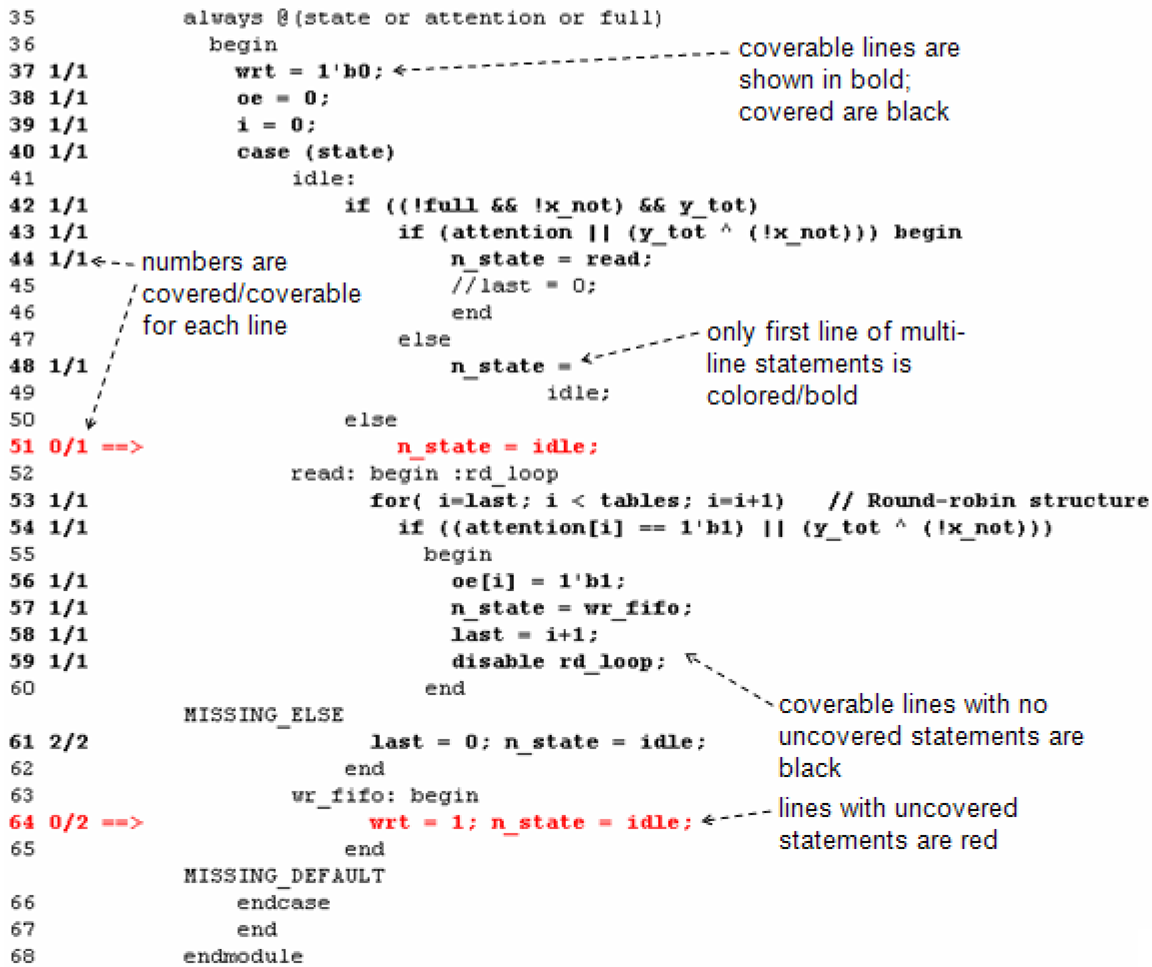
	Line No.	Total	Covered	Percent
TOTAL		71	64	90.14
always	12	16	14	87.50
contassn	50	1	0	0.00
initial	65	53	49	73.50
always	103	1	1	100.00

Note that each line in the table is identified by its type (always, initial, continuous assignment, and so on) and its starting line number. These entries do not represent the scores for individual lines or statements, but for the whole always block, initial block, VHDL process, or continuous assignment. You can then see which part(s) of the module require the most attention.

If the source code of your design is available, the second section in the report displays the annotated source code. The first column shows the line number in the source file. If a line contains a coverable statement, the second column shows the number that are covered and the total coverable statements that begin on that line. For example, on line 37 below, there is one coverable statement and it is covered (1/1). On line 51, there is one coverable statement and it is not covered (0/1). On line 64, there are two coverable statements and neither is covered (0/2).

Each coverable statement appears in boldface – black if covered, red if uncovered. For example:

Figure 1-18 Example of an Annotated Source Code File



Note:

When statements are spread across multiple lines, the covered/coverable numbers and the coloring/boldface will only be shown on the first line (as shown on line 48 in the example above).

If the source code of your design is not available (for example, source code files have been moved to a new location, the files are not read-accessible, or the files are not visible on the machine/

network on which you are running URG), then URG will generate a simplified report in place of the annotated source like Figure 1-18. An example of the simplified report is as follows:

Figure 1-19 Example of a Simplified Report

Line	Covered	Statements	
37	1	1	
38	1	1	
39	1	1	
40	1	1	
42	1	1	
43	1	1	
44	1	1	
48	1	1	
51	0	1	
53	1	1	
54	1	1	
56	1	1	
57	1	1	
58	1	1	
59	1	1	
60			MISSING_ELSE
61	2	2	
64	0	2	
65			MISSING_DEFAULT

Notice that URG only lists lines with statements on them in the simplified report.

The Toggle Coverage Report

The toggle coverage report starts with a table containing the number of nets, regs, and VHDL signals, the bits in each, and the summary coverage statistics for each type of signal. It then shows a table for each type of signal, listing each signal and indicating whether it was fully covered or not. The following figure is an example toggle summary table.

Figure 1-20 Example of a Toggle Coverage Summary Report

Toggle Coverage for Module : arbfGR

	Total	Covered	Percent
Totals	35	11	31.43
Total Bits	231	100	43.29
Total Bits 0->1	231	122	52.81
Total Bits 1->0	231	120	51.95
Nets	25	7	28.00
Net Bits	173	69	39.88
Net Bits 0->1	173	90	52.02
Net Bits 1->0	173	89	51.45
Regs	10	4	40.00
Reg Bits	58	31	53.45
Reg Bits 0->1	58	32	55.17
Reg Bits 1->0	58	31	53.45

The following figure shows a toggle coverage detailed table.

Figure 1-21   Example of a Toggle Coverage Detailed Report

Net Details

	Toggle	Toggle 1->0	Toggle 0->1
AlaMode	No	No	No
ChrByte1	No	No	No
ChrByte0	No	No	No
GsLinkOn[0:15]	No	No	Yes
SfLinkOn[0:1]	No	No	Yes
MtRawChkr[0:7]	Yes	Yes	Yes
MtRawChkr[8:15]	No	No	No

The Condition Coverage Report

Condition coverage information is shown as a table with each type of condition, followed by an enumeration of each condition showing the source code. For example:

Figure 1-22   Example of a Condition Coverage Report

Cond Coverage for Module : arbfGR

	Total	Covered	Percent
Conditions	152	125	82.24
Logical	52	43	82.69
Non-Logical	100	82	82.00
Non-Logical Event	0	0	

```
LINE      224
STATEMENT    Reset = (((~POW)) | (Enab & ((~RESET))))
                      ----1---    ---------2---------
EXPRESSION      -1-    -2-
                 0      0    | Covered
                 0      1    | Covered
                 1      0    | Not Covered

LINE      224
STATEMENT    Reset = (((~POW)) | (Enab & ((~RESET))))
                                  -1--    ----2-----
EXPRESSION      -1-    -2-
                 0      1    | Not Covered
                 1      0    | Covered
                 1      1    | Covered
```

When there are nested conditions, URG reports them hierarchically. For example, using the expression `((~POW)) | (Enab & ((~RESET)))` shown in Figure 1-22, the two terms of the bitwise or operator (`|`) are `((~POW))` and `(Enab & ((~RESET)))`. They are reported as a binary as shown in the first section.

The subexpression `(Enab & ((~RESET)))` is then broken down into its terms, `Enab` and `((~RESET))`. These are reported separately in the second section.

The Branch Coverage Report

URG branch coverage reports display the source code text along with annotations showing both covered and uncovered branches. For example, use the following source code:

Figure 1-23 Original Source Code

```
always@(posedge clk)
      if(rst)
            chg_cnt <=#2 3'h0;
      else
            begin
                  if((chg_cnt > 3'h0)&&(y_tot || x_not))
                        begin
                              chg_cnt <=#2 chg_cnt - 1;
                              nck_pulse <=#2 1'h1;
                        end
                  else
                        begin
                              chg_cnt <=#2 change;
                              nck_pulse <=#2 1'h0;
                        end
            end
```

Figure 1-24 URG Branch Coverage Report

```
always@(posedge clk)
     if(rst)
        -1-
        ==>
           chg_cnt <=#2 3'h0;
     else
           begin
                 if((chg_cnt > 3'h0)&&(y_tot || x_not))
                   -2-
                   ==>
                       begin
                             chg_cnt <=#2 chg_cnt - 1;
                             nck_pulse <=#2 1'h1;
                       end
                 else
                   ==>
                       begin
                             chg_cnt <=#2 change;
                             nck_pulse <=#2 1'h0;
                       end
           end

BRANCH        -1-   -2-
               1     -    | Not Covered
               0     1    | Not Covered
               0     0    | Not Covered
```

In Figure 1-24, the URG branch coverage report first shows the source code which contains the branch alternatives for a given branch, with each branch control highlighted and indexed with a number. Subsequently, the source code is followed by a table showing the different combinations and the coverage status of the control branches. One difference between URG and cmView reports is that URG displays an arrow (==>) for each branch.

The following is the same example with some of the branches covered:

Figure 1-25 The Same Example Showing Covered Branches

```
always@(posedge clk)
     if(rst)
        -1-
        ==>
           chg_cnt <=#2 3'h0;
     else
           begin
                 if((chg_cnt > 3'h0)&&(y_tot || x_not))
                   -2-
                   ==>
                       begin
                             chg_cnt <=#2 chg_cnt - 1;
                             nck_pulse <=#2 1'h1;
                       end
                 else
                   ==>
                       begin
                             chg_cnt <=#2 change;
                             nck_pulse <=#2 1'h0;
                       end
           end

BRANCH     -1-   -2-
            1     -    | Not Covered
            0     1    | Covered
            0     0    | Covered
```

Note that the source code and the index number for control branch `-2-` are both colored in green because it is fully covered. The arrow (==>) is at the same indentation as the corresponding index.

Figure 1-26 displays the same source code with different coverage. Note that the source code and index of control branch `-2-` are in red because it is not fully covered, that is, the else statement branch is not covered.

Figure 1-26   The Same Example Showing Different Coverage

```
always@(posedge clk)
     if(rst)
        -1-
        ==>
           chg_cnt <=#2 3'h0;
     else
           begin
                 if((chg_cnt > 3'h0)&&(y_tot || x_not))
                   -2-
                   ==>
                       begin
                             chg_cnt <=#2 chg_cnt - 1;
                             nck_pulse <=#2 1'h1;
                       end
                 else
                   ==>
                       begin
                             chg_cnt <=#2 change;
                             nck_pulse <=#2 1'h0;
                       end
           end

BRANCH         -1-    -2-
                1      -     | Covered
                0      1     | Covered
                0      0     | Not Covered
```

For ternary operators, the entire line of the source code is colored. It is only green if both branches are covered. There is no arrow (==>) for ternary operator branches. For example:

Figure 1-27 Coverage Example of a Ternary Operator

```
316      tri[7:0]dsko=oe_s ?dsko_n:8'hzz;
                            -1-

BRANCH          -1-
                 1     | Not Covered
                 0     | Covered
```

If there are multiple branches on a single line, each branch has its individual index and an arrow (==>) beneath each index. The entire line is colored in red unless all branches are fully covered. For example:

Figure 1-28   Example of Multiple Branches on a Single Line

```
15 if(a) x <= 3'h0; else if(y) x <= 3'h1; else x <= 3'h2;
    -1-                    -2-
    ==>
                           ==>
                           ==>

BRANCH          -1-   -2-
                 1     -     | Covered
                 0     1     | Covered
                 0     0     | Not Covered
```

Figure 1-29   Case Statements Example

```
case(state)
    -1-

idle:

if(go &&(press || x_not)&&(! oe_s))
  -2-
  ==>
 begin
    kp_hold=1'h1;
    n_state=hold;
 end


hold:
 if(oe_s &&(y_tot > 0))begin
  -3-
  ==>
    kp_hold=1'h0;
    n_state=idle;
  end
 else begin
  ==>
    n_state=hold;
    kp_hold=1'h1;
  end
service:begin
    ==>                    this is the 'service' branch of 'state'
end
endcase
end

BRANCH          -1-           -2-   -3-
                idle           1     -     | Covered
                idle           0     -     | Not Covered
                hold           -     1     | Not Covered
                hold           -     0     | Not Covered
                service        -     -     | Not Covered
            MISSING_DEFAULT    -     -     | Covered
```

Branch coverage for case statements follows the same reporting mechanism as described in the previous examples, except that there is no arrow (==>) indication for MISSING_DEFAULT in the `case` statement or MISSING_ELSE for control branch `-2-`.

The summary table for branch coverage is organized by top-level branch statements (these correspond directly to the tables in the detailed reports). The summary table shows the line number on which the branch statement starts, the number of branch alternatives the branch contains, and the number of branches covered. For example:

Line Number	Number of Branches	Covered	Percentage
1	3	0	0.00
15	3	2	66.67
316	2	1	50.00

The FSM Coverage Report

The FSM coverage report begins with a summary table for states, transitions, and sequences for all FSMs in the module/instance/entity. Subsequently, it shows individual state, transition, and sequence tables for each FSM.

Figure 1-30 Example of an FSM Coverage Summary Report

FSM Coverage Summary

	Total	Covered	Percent
States	3	2	66.67
Transitions	5	2	40.00
Sequences	16	2	8.25

State	Covered
'h0	Covered
'h1	Covered
'h3	Not Covered

Transition	Covered
'h0->'h1	Covered
'h1->'h0	Covered
'h1->'h3	Not Covered
'h3->'h0	Not Covered
'h3->'h1	Not Covered

Sequence	Covered
'h0->'h1	Covered
'h1->'h0	Covered
'h1->'h3	Not Covered
'h3->'h0	Not Covered
'h3->'h1	Not Covered
'h0->'h1->'h3	Not Covered
'h1->'h3->'h0	Not Covered
'h3->'h0->'h1	Not Covered
'h3->'h1->'h0	Not Covered
'h0->'h1->'h0	Not Covered Loop
'h1->'h0->'h1	Not Covered Loop
'h1->'h3->'h1	Not Covered Loop
'h3->'h1->'h3	Not Covered Loop
'h0->'h1->'h3->'h0	Not Covered Loop
'h1->'h3->'h0->'h1	Not Covered Loop
'h3->'h0->'h1->'h3	Not Covered Loop

The Assertions Coverage Report

The assertion coverage report displays a table showing the statistics of assertions.

Figure 1-31 Example of an Assertions Coverage Report

Assert Coverage for Module : cntrlr_0000

	Total	Attempted	Percent	Succeeded/ Matched	Percent
Assertions	15	15	100.00	13	86.67
Cover properties	0	0		0	
Cover sequences	0	0		0	
Events	0	0		0	
Total	15	15	100.00	13	86.67

Detail Report for Cover Properties

COVER PROPERTIES	CATEGORY	SEVERITY	ATTEMPTS	MATCHES ↑	VACUOUS MATCHES	INCOMPLETE
dut.chkr_0.HnVisp_0.HNGUSNEVMOSP_0.gfl0	0	0	117038	41417	0	1
dut.chkr_0.MoSys3_0.gmh1	0	0	117038	3784	0	1
dut.chkr_0.MoSys3_0.gmh0	0	0	117038	1707	0	1
dut.chkr_0.LmComp0.gmh2	0	0	117038	1645	0	0
dut.chkr_0.HnDipv_0.HNFIDUAPMI_0.gmd0	0	0	117038	1076	0	0
dut.chkr_0.LmComp0.gmh1	0	0	117038	803	0	0
dut.chkr_0.LmComp0.gmh0	0	0	117038	717	0	0
dut.chkr_0.HnDipv_0.HNFIDUAPMI_0.gmd1	0	0	117038	351	0	0
dut.chkr_0.LmComp0.HNAMOSP16Y9D_0.gml0	0	0	117038	0	0	0
dut.chkr_0.HnVisp_0.HNGUSNEVMOSP_0.gfl4b	0	0	117038	0	0	0
dut.chkr_0.HnVisp_0.HNGUSNEVMOSP_0.gfl4a	0	0	117038	0	0	0
dut.chkr_0.HnDipv_0.HNGOGU_0.gmf1	0	0	117038	0	0	0
dut.chkr_0.HnDipv_0.HNGOGU_0.gmf0	0	0	117038	0	0	0

The first column lists the names of the block identifier for `assert` or `cover statements`.

The ATTEMPTS column lists the number of times VCS or VCS MX began to see if the `assert` statement or directive succeeded or the `cover` statement or directive matched.

The MATCHES column lists the total number of times the `assert statement` succeeded and the `cover` statement matched. A real success is when the entire expression succeeds or matches without the vacuous successes.

The MATCHES column is color-coded according to its content. A cell with a 0 value is considered not covered and is displayed in red, while a cell with a non-zero value is considered covered and is displayed in green.

The FAILURES column lists the number of times the `assert` statement does not succeed. Because `cover` statements do not have failures, the entry for `cover` statements in this column remains `0`.

The INCOMPLETES column lists the number of times VCS or VCS MX started keeping track of the statement or directive, but simulation ended before the statement could succeed or match.

The Covergroup Report

Each covergroup report lists all points and crosses and their coverage scores at the top.

Figure 1-32   Example of a Covergroup Report

SUMMARY FOR VARIABLE i

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Automatically Generated Bins	16	5	11	68.75

AUTOMATICALLY GENERATED BINS FOR i

UNCOVERED BINS

NAME	COUNT	AT LEAST	NUMBER
[auto[11] - auto[15]]	--	--	5 bins

COVERED BINS

NAME	COUNT	AT LEAST
auto[0]	99	1
auto[1]	100	1
auto[2]	100	1
auto[3]	100	1
auto[4]	100	1
auto[5]	100	1
auto[6]	100	1
auto[7]	100	1
auto[8]	100	1
auto[9]	100	1
auto[10]	1	1

In Figure 1-32, the VCS/VCS MX coverage feature hole analysis compresses 5 bins into a single row for the UCOVERED BINS table. The following is an example of cross details:

Figure 1-33   Example of a Detailed Cross Table

SUMMARY FOR CROSS mycross0

SAMPLES CROSSED: i j mult

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT	MISSING
Automatically Generated Cross Bins	16384	16283	101	0.62	16283
User Defined Cross Bins	1	1	0	0.00	

AUTOMATICALLY GENERATED CROSS BINS FOR mycross0

UNCOVERED BINS

i	j	mult
[auto[0] - auto[9]]	[auto[0] - auto[9]]	[auto[67108864:134217727] - auto[4227858432:42949672
[auto[0] - auto[9]]	[auto[10] - auto[15]]	[auto[0:67108863] - auto[4227858432:4294967295]]
[auto[10]]	[auto[0] - auto[9]]	[auto[0:67108863] - auto[4227858432:4294967295]]
[auto[10]]	[auto[10]]	[auto[67108864:134217727] - auto[4227858432:42949672
[auto[10]]	[auto[11] - auto[15]]	[auto[0:67108863] - auto[4227858432:4294967295]]
[auto[11] - auto[15]]	[auto[0] - auto[15]]	[auto[0:67108863] - auto[4227858432:4294967295]]

COVERED BINS

i	j	mult	COUNT	AT LEAST
auto[10]	auto[10]	auto[0:67108863]	1	1
auto[0]	auto[0]	auto[0:67108863]	9	1
auto[0]	auto[5]	auto[0:67108863]	10	1
auto[0]	auto[9]	auto[0:67108863]	10	1
auto[0]	auto[3]	auto[0:67108863]	10	1
auto[0]	auto[6]	auto[0:67108863]	10	1
auto[0]	auto[8]	auto[0:67108863]	10	1
auto[0]	auto[2]	auto[0:67108863]	10	1

There is also a section for each point and cross showing the individual coverage percentage, information about the point or cross, and so on.

Viewing Results for Coverage Group Variants

A shape designation in a coverage group name indicates coverage results for variants of a coverage group. Because parameter values can affect the number or size of bins to be monitored, coverage group instances can have different shapes.

In the following Vera example, the program has two instances of W (`w1` and `w2`). Variants of the coverage group, `cov0`, were instantiated with different parameters in `w1` and `w2`.

```
#define UPPER 4'h7
#define LOWER 4'h0

class W {
    rand bit [3:0] addr;
    rand bit [3:0] resp;

    coverage_group cov0(bit [3:0] lower, bit [3:0] upper)
    {
        sample_event = @(posedge CLOCK);
        sample resp;
        sample addr;
        cross cc1 (resp, addr) {
            state cross_low_range( addr >= lower && addr
                    <= upper );
        }
            cumulative = 1;
    }

task new(bit [3:0] lower, bit [3:0] upper) {
       cov0 = new(lower,upper);
    }
    task display(integer id = -1) {
       printf("%d -> \t%h \t%s \n", id, addr, resp);
    }
}

program prog {
```

```
    W w1, w2;
    w1 = new(LOWER, UPPER);
    w2 = new(LOWER+8, UPPER+8);
    @(posedge CLOCK);

    void = w1.randomize() with {addr == 4'h6;};
    w1.display(1);

    void = w2.randomize() with {addr == 4'h6;};
    w2.display(2);

    @(posedge CLOCK);

}
```

The coverage results for `w1` and `w2` are found in `W::cov0_SHAPE_0` and `W::cov0_SHAPE_1`, respectively.

```
Crosses for Group : W::cov0_SHAPE_0
Automatically Generated Bins for resp
name      count at least
auto[3:3] 1     1

Samples for Group : W::cov0_SHAPE_1
Automatically Generated Bins for resp
name      count at least
auto[6:6] 1     1
```

Understanding Covergroup Page Splitting

The HTML version of the detailed covergroup report can become quite large, which can cause difficulties when you load and view this report.

Important:

The page-splitting functionality applies only to HTML reports, not text reports.

Note the following page-splitting guidelines for covergroup reports:

- Instance splitting: URG splitting behavior for covergroup instances resembles code coverage splitting for module instances. When URG generates a report for any group instance, URG checks the size of the current page and creates a new page if the report exceeds the value you defined with `-split` *N*.

  URG never splits a group report.

- Bin table splitting: If the bin table is so large that the previous splitting strategy is not enough to make the page fit in the page size limit, URG splits the bin table across several pages. In this situation, the covergroup page displays a note alerting you to the multiple-page splitting that URG has performed. The covergroup page also contains links to the various pages.

Each page that is split contains a summary table of coverage information.

The following examples show how a report for a hypothetical `covergroup a` is split:

Page for `covergroup a`, before splitting:

Group a

Group instance a1

Group instance a2

Group instance a3

Group instance a4

Pages for `covergroup a`, after splitting:

Page for `covergroup a`:

Group a

Group instance a1

Subpage 1:

Group instance a2

Group instance a3

Subpage 2:

Group instance a4

Mapping Coverage

You can instantiate a subhierarchy (a module instance in your design and all the module instances hierarchically under this instance) in two different designs and see the combined coverage for the subhierarchy from the simulation of both designs.

You can do this by mapping the coverage information for that subhierarchy from one design to another. This is still possible even though the hierarchy above this subhierarchy in the two designs is completely different.

Use the `-map` option to map subhierarchy coverage from one design to another. Full hierarchy should be generated in the `hierachy.html` file. This option is available in code coverage, but not supported in assert/group coverage.

The `-map` option syntax is as follows:

```
urg -map <module name>
```

Where:

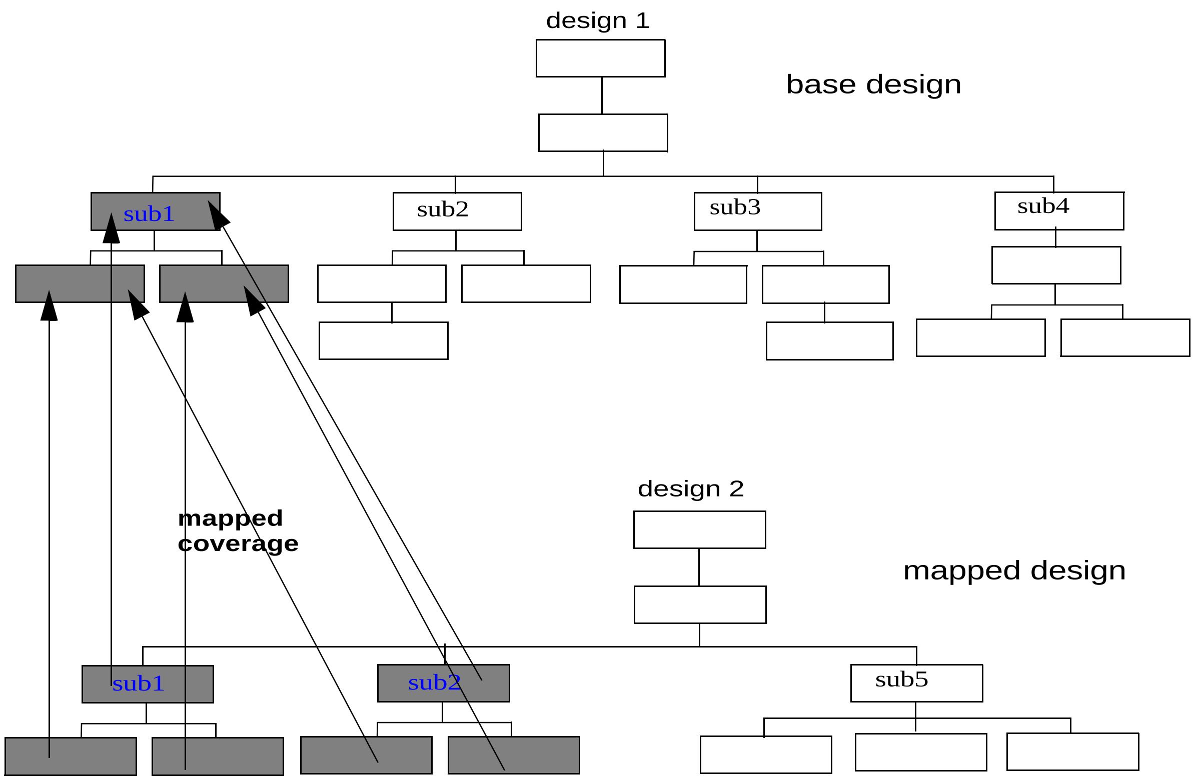
`<module name>`

Defines the top-level module name (identifier) of the subhierarchy. Do not put the hierarchical name of the top-level module instance in the subhierarchy.

Note:

When you map coverage from one design to another, the source file names must be identical. For example, consider the following illustration:

Figure 1-2 Mapping Coverage



The illustration shown in Figure 1-2 on page 67, two designs instantiate a common subhierarchy, labeled `sub1`. The code for the subhierarchy, in both designs, is in the source file named `sub1.v`. The module name (identifier) of the top-level module in the subhierarchy is `sub1`. This illustration shows mapping coverage information for that subhierarchy from the simulation of `design 2` to the coverage information for that subhierarchy from the simulation of `design 1`. There can be multiple instances of the subhierarchy in the design from which coverage information is mapped (mapped design). However, there can only be one instance of the subhierarchy in the design *to* which the coverage information is mapped (base design).

For more information about mapping subhierarchy coverage between designs in Verilog, see the section "Mapping Subhierarchy Coverage Between Designs in Verilog" in the *VCS Coverage Metrics User Guide*.

Instance-Based Mapping

You use the `-mapfile` option for instance-based mapping in URG.

The syntax is as follows:

```
urg -dir base.cm -dir input.cm -mapfile file_name
```

Where, `file_name` is the mapping configuration file.

Note the following guidelines:

- If instance name from input design matches to the pattern given in `-mapfile file_name` file, but if it doesn't corresponds to the module for which the pattern is given, the instance from the input design is ignored.
- Rules applied for mapping are applied on all directories given by the `-dir` option.
- You cannot use `-mapfile` and `-map` options together in URG.

For more information about instance based mapping, see the chapter "Common Operations" in the *VCS Coverage Metrics User Guide*.

Parallel Merging

The default mechanism URG uses for merging test results is a serial technique where it merges the results from the first test with those from the second test. It then merges the merged results with the results from the third test, then merges the new merged results with the results from the fourth test, and so on. Continuing on, by adding the results from each test, one test at a time. In many cases, particularly with a large amount of coverage data in a large amount of tests, merging the results can take a considerable amount of time.

If you find that merging the results is time consuming, URG has a parallel merging technology to accelerate the merging of the results. This technology simultaneously merges the results from different tests.

You specify this technology by using parallel merging with the `-parallel` option on the `urg` command line.

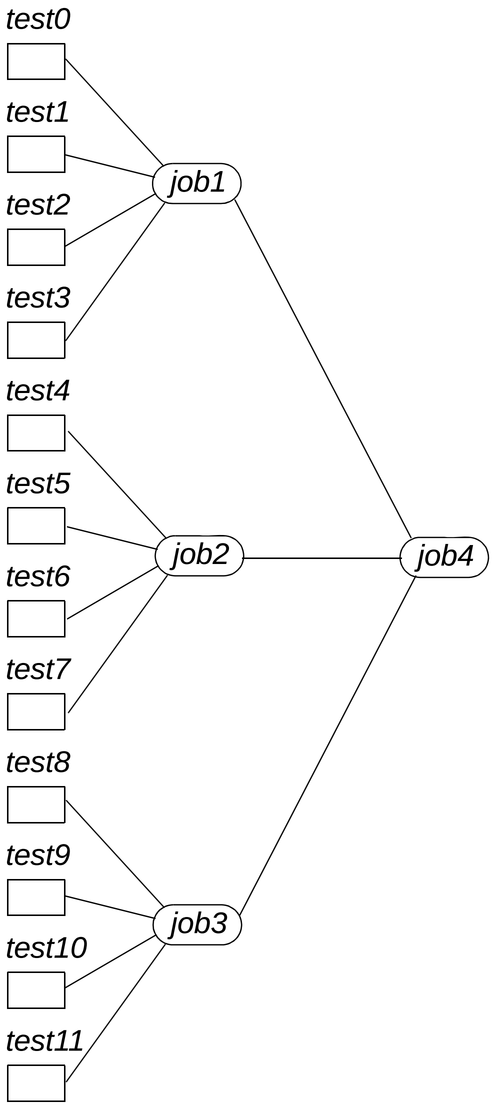
In this technology, the results from different tests are merged together into different “mergings” or clumps. URG simultaneously creates more than one merging or clump. After URG creates the mergings, it then combines them together. Sometimes, this process may take more than one pass to combine all the mergings together.

Note:

In this technology, this underlying mechanism is associating coverage data with each other, not creating new or different coverage data. Therefore, the coverage results in parallel merging are always the same as the default mechanism.

Figure 1-34 shows the parallel merging of 12 tests. URG merges `test0`, `test1`, `test2`, and `test3` into one merging; `test4`, `test5`, `test6`, and `test7` into a second merging; and `test8`, `test9`, `test10`, and `test11` into a third merging.

Figure 1-34   Parallel Merging of 12 Tests



In parallel, combining the creation of each of these mergings is called a job, therefore, URG runs three jobs to create the three mergings. URG then combines the mergings together in a fourth job to merge all the results.

Notice that there is a hierarchy of levels to the jobs. The first three jobs were on the first level, the fourth job was on the second level.

You can control the number of tests that URG merges into a merging, and the number of lower level mergings that go into higher level mergings. The smaller the number of tests (and mergings) the more jobs and levels in the parallel merging. For additional information, see the section entitled, “Specifying the Number of Tests in a Merging”.

When using parallel merging, you can perform one of the following:

- Specify the machines that perform the jobs
- Use a GRID computing Engine
- Use LSF

Specifying the Machines that Perform the Jobs

You can specify which machines on your network perform the jobs in the parallel merging. To do so, enter the machines on separate lines in a text file and include the file name as an argument to the `-parallel` option.

The following is an example of a machine file:

```
linux_machine1
linux_machine2
linux_machine3
linux_machine4
```

The following displays an example using this argument:

```
urg -dir test0 test1 test2 test3 test4 test5 test6 test7
test8 test9 test10 test11 -parallel machine_file
```

Using a GRID Computing Engine

You specify using a GRID computing engine and pass optional arguments to that engine with the `-grid` command-line option. You pass the optional arguments for the engine in quotation marks that follow the option.

Following the `-grid` option, you can also enter the secondary options `-sub` and -`del` to enter the GRID engine commands to run and clear the GRID engine. The following is an example of entering the `-grid` option:

```
urg -dir test0 test1 test2 test3 test4 test5 test6 test7
test8 test9 test10 test11 -parallel -grid "-l arch=1x24-amd64
-P bnormal" -sub qsub -del qdel
```

In this example, URG starts the GRID engine with the job throwing command, `qsub`. URG appends `-l arch=1x24-amd64 -P bnormal` to the `qsub` command line. After parallel merging, URG passes the `qdel` command to the engine to clear the engine.

Using LSF

Use the `-lsf` command-line option to specify the use of an LSF engine and pass optional arguments to that engine. Similar to the `-grid` option, you pass the optional arguments for the engine in quotation marks that follow the option.

With the `-lsf` option, you must enter the secondary options, `-sub` and -`del`, to enter the commands to throw and control jobs in the engine. The following is an example of entering the `-lsf` option:

```
urg -dir test0 test1 test2 test3 test4 test5 test6 test7
test8 test9 test10 test11 -parallel -lsf "-q queuename -R
```

```
res_req" -sub bsub -del bkill
...
```

Specifying the Number of Tests in a Merging

You specify the number of tests in a merging, and the number of lower-level mergings in higher-level mergings, with the `-parallel_split` *`integer`* command-line option and argument.

If for 12 test, by default URG divides the process into four jobs on two levels (see Figure 1-34).

If you specify a lower number of tests and mergings, the number of jobs, and perhaps levels, increases. Therefore, for the 12 tests, specifying a value of 3 creates in six jobs on three levels and specifying a value of 6 creates three jobs on two levels.

Flexible Merging

URG facilitates flexible merging using a new merge database option. It follows a set of rules to merge databases depending on the functional coverage model. This feature enables you to get a more accurate coverage report when the coverage model is still evolving and you are running tests repeatedly with minor changes in the coverage model between the test runs.

To enable flexible merging, use the `-group flex_merge_drop` option on the URG command line, as follows:

```
urg -dir simv1.vdb -dir simv2.vdb -group flex_merge_drop
```

URG assumes the first specified coverage database as a reference for flexible merging.

Example

Consider two databases, `first.vdb` and `second.vdb`. Using the `-group flex_merge_drop` option and flexible merging database rules, URG generates a merged report. For example:

```
urg -dir first.vdb -dir second.vdb -group flex_merge_drop
```

In this example, URG considers the `first.vdb` coverage database directory as a reference to generate the flexible merged report.

Merge Equivalence

To merge two coverpoints or crosspoints, you should merge them equivalent to each other. The following section lists the requirements for merge equivalence.

Merge Equivalence Requirements for Autobinned Coverpoints

The `coverpoint P1` is said to be merge equivalent to a `coverpoint P2` only if the name, `auto_bin_max` and the width of the coverpoints are the same, where `P1` and `P2` are autobinned coverpoints.

Merge Equivalence Requirements for User-defined Coverpoints

The `coverpoint P1` is said to be merge equivalent to a `coverpoint P2` only if the coverpoint names and width are the same.

Merge Equivalence Requirements for Crosspoints

The `crosspoint C1` is said to be merge equivalent to a `crosspoint C2`, if the crosspoints have the same number of coverpoints and their corresponding coverpoints are merge equivalent.

Rules for Flexible Merging Databases

The following sections list the rules to merge the database with flexible merge dropping semantics. With the dropping semantics, you can take advantage of the information available from the newer database to eliminate the redundant information from the older databases.

Rules for Merging Coverpoints

The `coverpoints P(T1)` in first test run `T1` and `P(T2)` in the second test run `T2` are merged according to the following rules:

- If the coverpoints are merge equivalent. The merged coverpoints will contain a union of all the coverpoint bins in `P(T1)` and `P(T2)`, but URG will drop the coverpoint bins that are defined only in the earlier coverage model.
- If the coverpoints are not merge equivalent. The merged coverpoint will contain all the coverpoint bins in the most recent test run and the older test run data is not considered and dropped.

Rules for Merging Crosspoints

The `crosspoint C(T1)` in test `T1` and `C(T2)` in test `T2` are merged according to the following rules:

- If the crosspoints are merge equivalent. The merged crosspoints will contain a union of all the crosspoint bins in `C(T1)` and `C(T2)`, but URG will drop the crosspoint bins that are defined only in the earlier coverage model.

- If the crosspoints are not merge equivalent. The merged crosspoint will contain all the crosspoint bins in the most recent test run and the older test run data is not considered and dropped.

Example

The following example shows two tests with minor changes in the functional and assertion coverage models. The changes are marked in red.

Example 1-35 Test01

```
cp1: coverpoint firstsig;
     option.auto_bin_max = 64;
cp2: coverpoint secondsig {
        bin first = [0:63];
        bin mid = [71:82];
}
cp3: coverpoint thirdsig;
bit[7:0]signal;
cp4:coverpoint signal;
cc1: cross cp1, cp2;
cc2: cross cp2, cp3 {
        bins mybin = binsof(cp2) intersect  [0:255];
}
cc3: cross cp2, cp3 {
        bins my_st = binsof(cp2) intersect  [0:255];
}
```

Example 1-36 Test02

```
cp1: coverpoint firstsig;
      option.auto_bin_max = 32;
cp2: coverpoint secondsig {
```

```
        bin first = [0:63];
        bin second = [65:128];
}
cp3: coverpoint thirdsig;
bit[15:0]signal;
cp4:coverpoint signal;
cc1: cross cp1, cp2;
cc2: cross cp2, cp3 {
        bins mybin = binsof(cp2) intersect  [0:255];
        bins yourbin = binsof(cp2) intersect  [256:511];
}
cc3: cross cp2, cp3 {
        bins my_st = binsof(cp2) intersect  [0:8191];
}
cc4: cross cp1, cp2, cp3
```

Using the two coverage model examples, let's analyze the flexible merged database. In this example, test `02` is the latest test that is run and is the reference coverage database. URG considers the first database specified as the reference coverage database directory.

```
urg -dir test02.vdb -dir test01.vdb -group flex_merge_drop
```

Coverpoint Analysis

- `cp1`, the `auto_bin_max` is changed to `32`. Therefore, they are not merge equivalent and only `cp1` data from `test 02` is included in the generated report.
- The `coverpoint cp2` is merge equivalent and the data from both tests are merged. The new bin, `second`, added in `test02` is included in the generated report, but the bin, `mid`, from the previous test, `test01`, is dropped from the generated report since it is removed from latest test, `test02` coverage model.
- The `coverpoint cp3` is unchanged. The data from both the tests are merged to be included in the generated report.

- `cp4`, the signal width is changed. Therefore, they are not merge equivalent and only `test02` data is included in the generated report.

Crosspoint Analysis

- `cc1`, the component pair of `coverpoint cp1` is not merge equivalent, and therefore, only data from the `test02` is included in the generated report.
- `cc2`, a new bin, `yourbin`, is added, but the component pair of `coverpoint cp2` and `cp3` are merge equivalent, and therefore, the data from both the tests are merged to be included in the generated report.
- `cc3`, the component `coverpoint cp2` intersect range is changed. They are not merge equivalent and user-defined `my_st` will be considered only from the `test02`, but the autocrosses in `test01` will be merged with the autocrosses in `test02`.
- The `crosspoint cc4` is a new introduction in `test02` and is included in the generated report.

Flexible Merge Database

Coverage model for test01	Coverage model for test02	Flexible merge Database Profile for test01 and test02

cp1: coverpoint firstsig;
option.auto_bin_max = 64;
	cp1: coverpoint firstsig;
	option.auto_bin_max = 32;
		//auto_bin_max differs and is not
		merge equivalent.
		
		cp1: coverpoint firstsig;
		auto_bin_max = 32;
		// (From test02)

cp2: coverpoint secondsig{
bin first = [0:63];
bin mid = [71:82];}
	cp2: coverpoint secondsig{
	bin first = [0:63];
	bin second = [65:128];}
		// (cp2 is merged across test01 and test02)
		cp2: coverpoint secondsig {
		bin first = [0:63];
		bin second = [65:128];}

cp3: coverpoint thirdsig;	cp3: coverpoint thirdsig;	//cp3 is merged across test01 and test02 cp3: coverpoint thirdsig;
bit[7:0]signal; cp4: coverpoint signal;	bit[15:0]signal; cp4: coverpoint signal;	//width of signal has changed and not merge equivalent cp4: coverpoint signal; // (From test02)
cc1: cross cp1, cp2;	cc1: cross cp1, cp2;	// cc1 is not merged equivalent because cp1 is not merge equivalent cc1: cross cp1, cp2; // (From test02)

cc2: cross cp2, cp3 {
bins mybin = binsof(cp2)
intersect [0:255]; }
	cc2: cross cp2, cp3 {
	bins mybin = binsof(cp2)
	intersect [0:255];
	bins yourbin=binsof(cp2)
	intersect [256:511];}
		// cc2 is merged across test01 and
		test02)
		cc2: cross cp2, cp3 {
		bins mybin = binsof(cp2)
		
		intersect [0:255];
		bins yourbin = binsof(cp2) intersect [256:511];}

cc3: cross cp2, cp3 { bins my_st = binsof(cp2) intersect [0:255];}	cc3: cross cp2, cp3 { bins my_st = binsof(cp2) intersect [0:8191];}	cc3: cross cp2, cp3 { bins my_st = binsof(cp2) intersect [0:8191];} The autocrosses in test01 will be merged with the autocrosses of test 02.
	cc4: cross cp1, cp2, cp3;	cc4: cross cp1, cp2, cp3; // (From test02)

Grading and Coverage Analysis

Use the `-grade` option to specify and run test grading and generate test grading reports.

Grading Tests

To invoke grading in URG, use the following syntax:

```
urg -grade [quick|greedy|score] [goal R] [timelimit N]
          [maxtests N] [minincr R] [reqtests file_name]
```

`quick`

Generates a grading report, displaying cumulative and incremental values of each metric for each test. The quick grading algorithm is linear in the number of tests.

Cumulative value is the coverage score after that test is merged with all previous tests in the graded list. For each metric, incremental value is the score improvement contributed to the cumulative value by that test after merging.

`greedy`

Produces a report where the tests have been put in best-first order based on usefulness of the tests. The greedy result shows the cumulative, stand-alone, and incremental scores for each test in the graded list. The greedy grading algorithm is quadratic in the number of tests.

Cumulative value is the coverage score after that test is merged with all previous tests in the graded list. Stand-alone value represents the individual score of that test by itself. For each metric, incremental value is the score improvement contributed to the cumulative value by that test after merging. The greedy argument is the default for the `-grade` option.

`score`

Shows the tests in default order and gives their stand-alone scores only. The score grading algorithm is linear in the number of tests.

Note:

The simulation time/random seed for testbench coverage in URG is shown with the `-grade score` option only. The `-grade score` gives information for seed/time and score for each test, and also simply lists the tests in the best first order, which is not expensive.

`goal` *`R`*

Displays the cumulative coverage goal. If not specified, the program will process all specified tests.

`timelimit` *N*

Specifies the time limit for the report generator to run before exiting. Only those tests that are graded before the time limit is hit are included in the graded list.

`maxtests` *N*

Specifies the maximum number of tests to include in the report.

`minincr`

The score improvement for each metric that the test contributed to the cumulative value when it was merged. This value is specified as a real number between 0.00 and 100.00.

`reqtests` *`file_name`*

Use this option with `greedy` to specify reading a list of test names from *`file_name`* for inclusion in the grading report. Those tests are included at the top of the graded list, regardless of their scores or effectiveness for coverage.

Note:

The `-show alltests` option has been deprecated. Use the `-grade score` option which has the same function.

Examples Using the Grading Option

Scoring

Use the `-grade score` option to generate a report which contains the individual absolute score of each test. Each column of the table is sortable by clicking any of the column headings: SCORE, LINE, COND, TOGGLE, and so on.

Note:

A scoring report takes significant more time for URG to compute than the preceding default report.

Figure 1-37 Example of a Scoring Report

Total Coverage Summary

SCORE	LINE	COND	TOGGLE	ASSERT
64.80	95.20	45.83	18.18	100.00

Tests are in the order found in the database (the same as the order shown by urg -show availabletests).

Scores are the standalone scores for each test.

SCORE	LINE	COND	TOGGLE	ASSERT	NAME
14.21	36.80	8.33	4.55	7.14	**mysimv/test8**
14.01	36.00	8.33	4.55	7.14	**mysimv/test7**
14.01	36.00	8.33	4.55	7.14	**mysimv/test9**
16.78	32.80	8.33	4.55	21.43	**mysimv/test3**
13.61	34.40	8.33	4.55	7.14	**mysimv/test6**
18.16	31.20	8.33	4.55	28.57	**mysimv/test2**
16.38	31.20	8.33	4.55	21.43	**mysimv/test1**
14.79	32.00	8.33	4.55	14.29	**mysimv/test4**
13.21	32.80	8.33	4.55	7.14	**mysimv/test5**
14.59	31.20	8.33	4.55	14.29	**mysimv/test0**

Quick Grading

Use the `-grade quick` option to generate a quick-grading report which describes the cumulative and incremental contribution of each metric for each test. Since the table displays the incremental value of each test in alphanumeric order, it is not sortable. The TOTAL boxes are color-coded according to the color legend of Synopsys Coverage Metrics, while INCR boxes are either green or white (white denotes no incremental value). In the following figure, `mysimv/test1` and `mysimv/test2` do not contribute additional condition coverage score to what is already scored by `mysimv/test0`, therefore, they are color-coded in white.

Figure 1-38 Example of a Quick-grading Report

Total Coverage Summary

SCORE	LINE	COND	TOGGLE	ASSERT
64.80	95.20	45.83	18.18	100.00

Tests are in the order found in the database (the same as the order shown by urg -show availabletests).

Scores are accumulated (Total) and incremental (Incr) for each test.

SCORE		LINE		COND		TOGGLE		ASSERT		
TOTAL	INCR	TOTAL	INCR	TOTAL	INCR	TOTAL	INCR	TOTAL	INCR	NAME
14.59	14.59	31.20	31.20	8.33	8.33	4.55	4.55	14.29	14.29	**mysimv/test0**
18.94	4.35	32.80	1.60	8.33	0.00	6.06	1.52	28.57	14.29	**mysimv/test1**
25.08	6.14	34.40	1.60	8.33	0.00	7.58	1.52	50.00	21.43	**mysimv/test2**
30.60	5.53	41.60	7.20	14.58	6.25	9.09	1.52	57.14	7.14	**mysimv/test3**
35.93	5.33	48.00	6.40	20.83	6.25	10.61	1.52	64.29	7.14	**mysimv/test4**
41.46	5.53	55.20	7.20	27.08	6.25	12.12	1.52	71.43	7.14	**mysimv/test5**
47.39	5.93	64.00	8.80	33.33	6.25	13.64	1.52	78.57	7.14	**mysimv/test6**
52.99	5.61	73.60	9.60	37.50	4.17	15.15	1.52	85.71	7.14	**mysimv/test7**
59.00	6.01	84.80	11.20	41.67	4.17	16.67	1.52	92.86	7.14	**mysimv/test8**
64.80	5.81	95.20	10.40	45.83	4.17	18.18	1.52	100.00	7.14	**mysimv/test9**

Greedy Grading

Use the `-grade greedy` option to generate a full-grading report which includes cumulative (TOTAL), stand-alone (TEST), and incremental (INCR) scores of each metric for each test. This report is not sortable on any column heading. TOTAL is the cumulative coverage score after each test is merged with all previous tests in the graded list. TEST is the individual coverage score of each test. INCR is the improvement of coverage score for each test contributing to the cumulative value. The TEST boxes are color-coded like the TOTAL boxes. Applying the `greedy` argument provides a thorough report, but it is more time-consuming than using the `score` or `quick` argument.

Figure 1-39 Example of a Full-grading Report

Total Coverage Summary

SCORE	LINE	COND	TOGGLE	ASSERT
64.80	95.20	45.83	18.18	100.00

Tests are in graded order

Shows the accumulated (Total), standalone (Test), and incremental (Incr) score overall and for each metric

SCORE			LINE			COND			TOGGLE			ASSERT			
TOTAL	TEST	INCR	TOTAL	TEST	INCR	TOTAL	TEST	INCR	TOTAL	TEST	INCR	TOTAL	TEST	INCR	NAME
18.16	18.16	18.16	31.20	31.20	31.20	8.33	8.33	8.33	4.55	4.55	4.55	28.57	28.57	28.57	**mysimv/test2**
25.01	14.01	6.85	41.60	36.00	10.40	16.67	8.33	8.33	6.06	4.55	1.52	35.71	7.14	7.14	**mysimv/test7**
31.86	14.01	6.85	52.00	36.00	10.40	25.00	8.33	8.33	7.58	4.55	1.52	42.86	7.14	7.14	**mysimv/test9**
37.79	13.61	5.93	60.80	34.40	8.80	31.25	8.33	6.25	9.09	4.55	1.52	50.00	7.14	7.14	**mysimv/test6**
43.27	14.21	5.49	72.00	36.80	11.20	33.33	8.33	2.08	10.61	4.55	1.52	57.14	7.14	7.14	**mysimv/test8**
48.28	13.21	5.01	79.20	32.80	7.20	37.50	8.33	4.17	12.12	4.55	1.52	64.29	7.14	7.14	**mysimv/test5**
53.08	16.78	4.81	85.60	32.80	6.40	41.67	8.33	4.17	13.64	4.55	1.52	71.43	21.43	7.14	**mysimv/test3**
57.89	14.79	4.81	92.00	32.00	6.40	45.83	8.33	4.17	15.15	4.55	1.52	78.57	14.29	7.14	**mysimv/test4**
62.24	16.38	4.35	93.60	31.20	1.60	45.83	8.33	0.00	16.67	4.55	1.52	92.86	21.43	14.29	**mysimv/test1**
64.80	14.59	2.56	95.20	31.20	1.60	45.83	8.33	0.00	18.18	4.55	1.52	100.00	14.29	7.14	**mysimv/test0**

Grading and the -scorefile Option

You can use the `-scorefile` option to modify how the overall grading scores are computed for tests. By default, each metric is weighted the same. The `-scorefile` option enables you to specify a separate "score file" that allows you to give a different weight to each metric.

To instruct URG to use the score file, use the following syntax:

```
%urg -grade [grading options] -scorefile file_name
```

Note:

> You can use either the `-scorefile` or `-metric` option, but not both.

The score file has the following simple format:

```
metric1 weight1
metric2 weight2
…
metricN weightN
```

In this file, each metric may only be specified one time. The metric names are the same as those used for the `-metric` option on the command line (for example, `line`, `cond`, `assert`). Each weight must be a non-negative integer.

When the `-scorefile` option is given along with the `-grade` option, grading is done only for the metrics as specified in the `-scorefile` file. You cannot give a `-metric` option with the `-scorefile` option, since the score file spells out which metrics are being used.

The following is an example score file. It indicates that line coverage is weighted normally, but that each group coverable object should be weighted double:

```
line 1
group 2
```

In this case, the overall score and the score for each test will be computed as follows:

```
score = (linescore + (groupscore * 2)) / 3
```

The score file weights only affect the computation of the overall score and the overall score for each test; it does not affect the score reported for any individual metric. For example, the group score will be reported the same regardless of the weight put in the score file.

Reporting Element Holes

This section describes how URG reports covergroup cross bins in the special case where large chunks of cross space are uncovered.

Definition

For a cross of n variables v_0, v_1, . . ., v_{n-1}, an m-element hole is a set of uncovered bins, such that m number of the variables have all possible values in the set, and each of the remaining n-m variables has only one fixed value for all bins in the set.

For example, consider a three-way cross of variables v_0, v_1, and v_2 with possible values ranging from 1 to 3. Then the following set of uncovered bins, which can be called (`1`, `1`, `*`), is a one-element hole because all possible values of v_2 are uncovered for $v_0 = 1$ and $v_1 = 1$:

```
{ (1, 1, 1), (1, 1, 2), (1, 1, 3) }
```

Finding Element Holes

You can find all the holes of size ranging from `1` to *n*, but you would have to exhaustively search the space. URG looks for holes by fixing values from the left to right of the cross. Thus, you can find a hole `(x = 2, y = 1, z = *)`, but you would not find a hole such as `(x = *, y = 2, z = 5)`.

Since element holes can be found by modifying the range package, URG will not search for element holes. They already exist in the list of (compressed) bins in the UCAPI interface.

URG will detect element holes by looking for any cross bins with the `"*"` character as the `covdbName` of any of the cross components.

Displaying Range Values

To display the full range of a variable, URG uses the `"*"` character. For variables which do not consist of the full range of values, URG uses the auto `[…]` format.

Note:

Any cross bin that has a value bin with a full range is an element hole.

Showing Element Holes

You do not need to use any command-line option to turn on element holes detection because URG reports them automatically.

In URG reports, element holes are part of the uncovered bins table. They are obvious in the reports since one or more variable columns will have the `"*"` character in place of any real value.

URG displays element holes in a separate table before the list of uncovered bins. For example:

Figure 1-40   Example of a Report Showing Element Holes

Automatically Generated Cross Bins for lmno

Element holes

i	j	k	COUNT	AT LEAST	NUMBER
auto[0:3]	auto[0:3] - auto[8:11]	*	--	--	192
auto[4:7] - auto[8:11]	*	*	--	--	8192

Uncovered bins

Analyzing Trend Charts

URG uses the raw data from VCS basic coverage data and VMM Planner metrics data to plot trend charts. These trend charts allow you to graphically analyze the data from a number of previous URG sessions. You can combine a set of URG reports created over a span of time and plot their metrics as a time series chart to observe their trend.

Quick Overview

Each URG report contains coverage data and metrics data which constitute a snapshot of the verification metrics for a single session at a point in time. URG provides trend analysis capability to combine and analyze multiple URG reports. To invoke URG trend analysis, you specify the `-trend` option with arguments to generate various trend chart reports.

```
%urg -trend [trend_options]
```

You can use URG to generate a set of trend charts to track the progress of your projects. Click on different elements of a URG trend chart to expand the metric to its sub-components, to drill down to the DUT or the verification plan hierarchical structure, and to retrieve previous URG reports to view the details of high-level metrics.

This overview covers the following three topics:

- "Generating Trend Charts"
- "Customizing Trend Charts"
- "Navigating Trend Charts"

Generating Trend Charts

Trend charts are most useful if you periodically sample the same set of data over a period of time. We recommend that you execute a URG run on a standard set of coverage and test result data after each complete regression. For example, you could run a nightly cron job that first runs your regressions and then runs URG with the trend option turned on.

There are various options that you can provide to the `-trend` argument as follows:

`-report` *`someUniqueName`*

Makes sure that each snapshot of URG reports has a unique name. If you do not assign unique names, then the URG results will be overwritten and you will not be able to generate a meaningful trend report. One way to generate a unique name is to base the name on the current date/time.

`-trend root` *`path`*

Simplifies specifying which historical URG reports to use for trend charting. With the `root` option, URG recursively explores the path that you supply to locate all URG reports. If used with `-report` and unique report names, you need not change the URG command line arguments as new reports are added.

`-trend root` *`path`* `rootdepth` *`N`*

Scan the given root path and subdirectory recursively to find URG reports. If you do not specify `rootdepth`, the default depth for *`N`* is `1`.

`-trend rootfile` *`txtfile`*

Permits enumeration (in a text file) of multiple root paths for scanning.

`-trend src` *`report1`* `[`*`report2`* `... ]`

Permits enumeration (with the `src` keyword) all the URG reports. This option is useful to locate reports stored at different directories.

`-trend srcfile` *`txtfile`*

Permits enumeration (in a text file) of all URG report paths.

Trend Plot

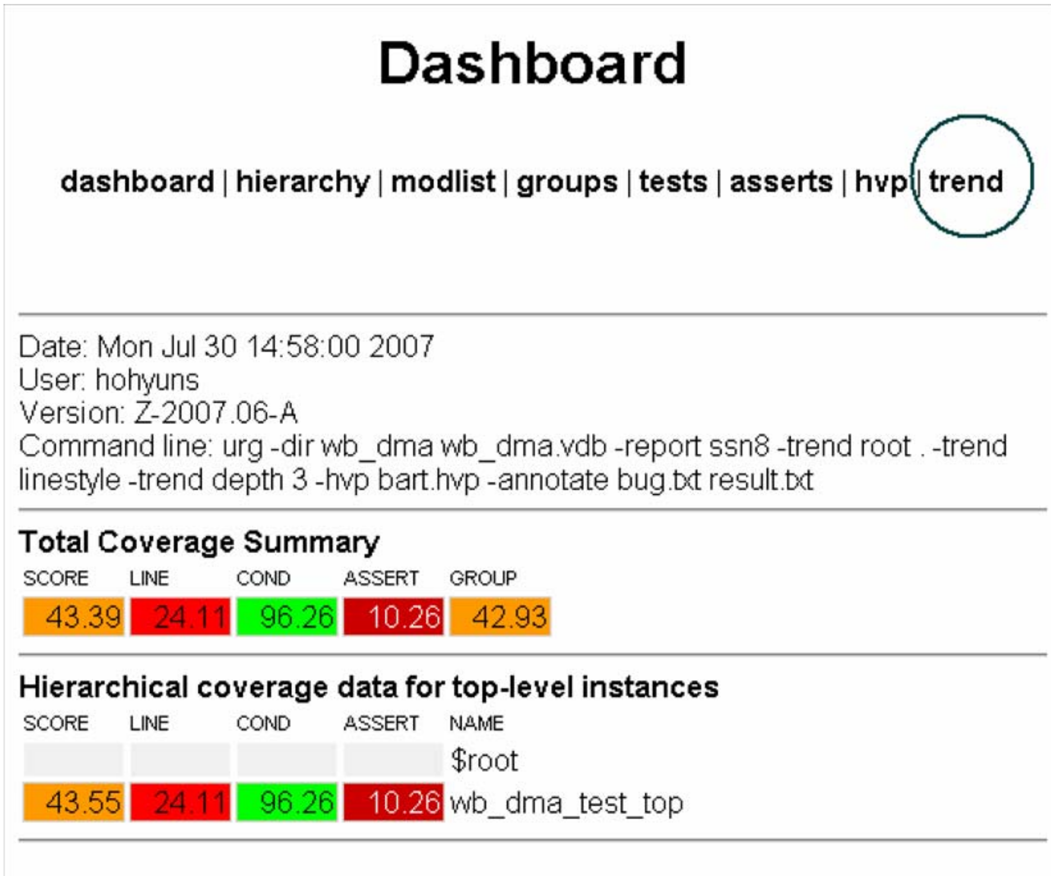
URG automatically generates trend charts by scanning previously saved URG reports to extract metrics data for trend plot. You must specify the path where URG can locate the previous saved reports. Once URG finds the reports, it will extract all the metrics to create a set of trend charts. There are three ways to instruct URG where to find the prior reports:

- Use the `root` keyword to specify the root directory where all the snapshots of URG reports are stored. Ideally, this is all you need to do to retrieve default reports. For example:

```
%urg -trend root urg_report_root_path
```

When you specify the `-trend` option on the command line, URG reports will include a **trend** tab in the menu bar of trend chart dashboard. See Figure 1-41.

Figure 1-41   URG Dashboard with the -trend Option



Customizing Trend Charts

At the time URG reports are generated, you can select the maximum level of data details for trend charts.You select the level of chart details using the `-trend depth` option. It is recommended that you choose the same level of `depth` to create all URG reports during the project duration. You can customize trend charts using the following command options:

`-trend depth` *`number`*

Sets the depth *number* of the DUT instance/HVP feature hierarchy for which URG generates trend charts. The default depth is 1, that is, only the top-level chart is generated. The more levels that you specify, the more you can drill down into the trend reports to find finer-grained details of information. The larger values of *number* also result in larger URG report sizes and longer runtimes. The report sizes and runtimes grow exponentially with respect to the depth *number*. See “Trend Chart Linkage” for more details. A command-line example follows:

```
%urg -dir wishbone.vdb wishbone.cm -plan wishbone.hvp
-trend root path depth 3
```

URG can also display curves with different line styles on the chart, for example, solid, dash, and so on. Different line styles are useful for printing on a monochrome printer. See the `-trend linestyle` option for more details. You should use the same value of `depth` for all URG reports to create a trend series, otherwise there may be gaps in the data on the trend charts.

`-trend linestyle`

Displays a different line style (solid line, dashed line, and so on) for each curve on the trend chart, particularly useful for black-and-white printing. Refer to Figure 1-43 for a chart with various line styles as compared to Figure 1-42 without different line styles. In Figure 1-43, not only has each line a distinct color, but also a unique line style. The line styles are automatically selected by internal line style palette. A command-line example follows:

```
%urg -dir wishbone.vdb wishbone.cm -plan myplan.hvp
-trend root path linestyle
```

`-trend offbasicavg`

Turns off basic coverage curves and displays only VMM Plan related score curves. For example:

```
%urg -dir wishbone.vdb wishbone.cm -plan myplan.hvp
-trend root path offbasicavg
```

With the `-offbasicavg` option, URG displays only the Plan Average curves without the Basic Average curves on the chart. This option is useful to avoid cluttering trend charts.

Note:

The Basic Average consists of the raw Synopsys built-in coverage metrics (Line, Cond, FSM, Toggle, and Branch) and the two functional coverage metrics (Assert and Group). The Plan Average consists of the Synopsys seven built-in coverage metrics score captured by the VMM Verification Plan.

To manipulate the timestamp of the current URG report session, set the `URG_FAKE_TIME` environment variable. You must set this variable before using the `urg` command. The format of the variable is `mm-dd-yyyy hh:mm:ss`.

For example, in a C shell, you set the timestamp variable as follows, providing the “fake” value:

```
setenv URG_FAKE_TIME "11-16-2007 10:20:30"
```

To suppress the generation of the `session.xml` in the URG report, set the `URG_NO_SESSION_XML` environment variable. You should suppress `session.xml` if you want to prevent the current URG report from affecting future trend analysis.

For example, in a C shell, you suppress `session.xml` as follows:

```
setenv URG_NO_SESSION_XML
```

Figure 1-42   Trend Chart Without the linestyle Option

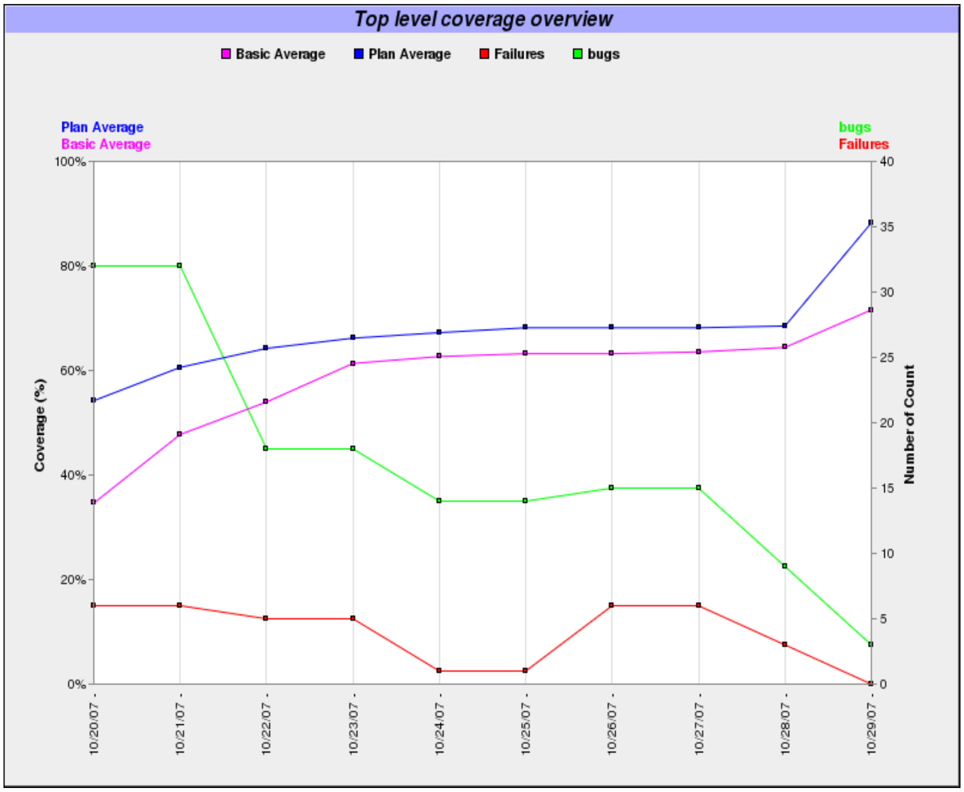


Figure 1-43   Trend Chart with the linestyle Option

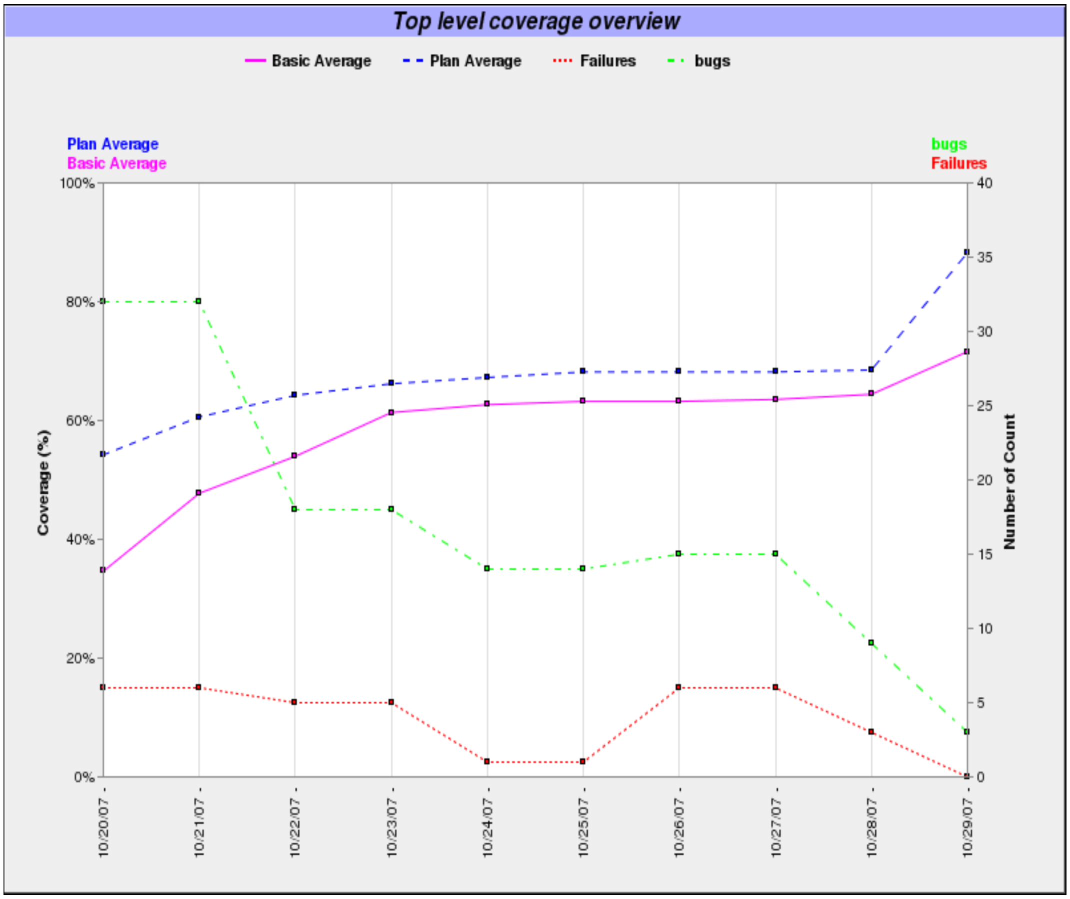
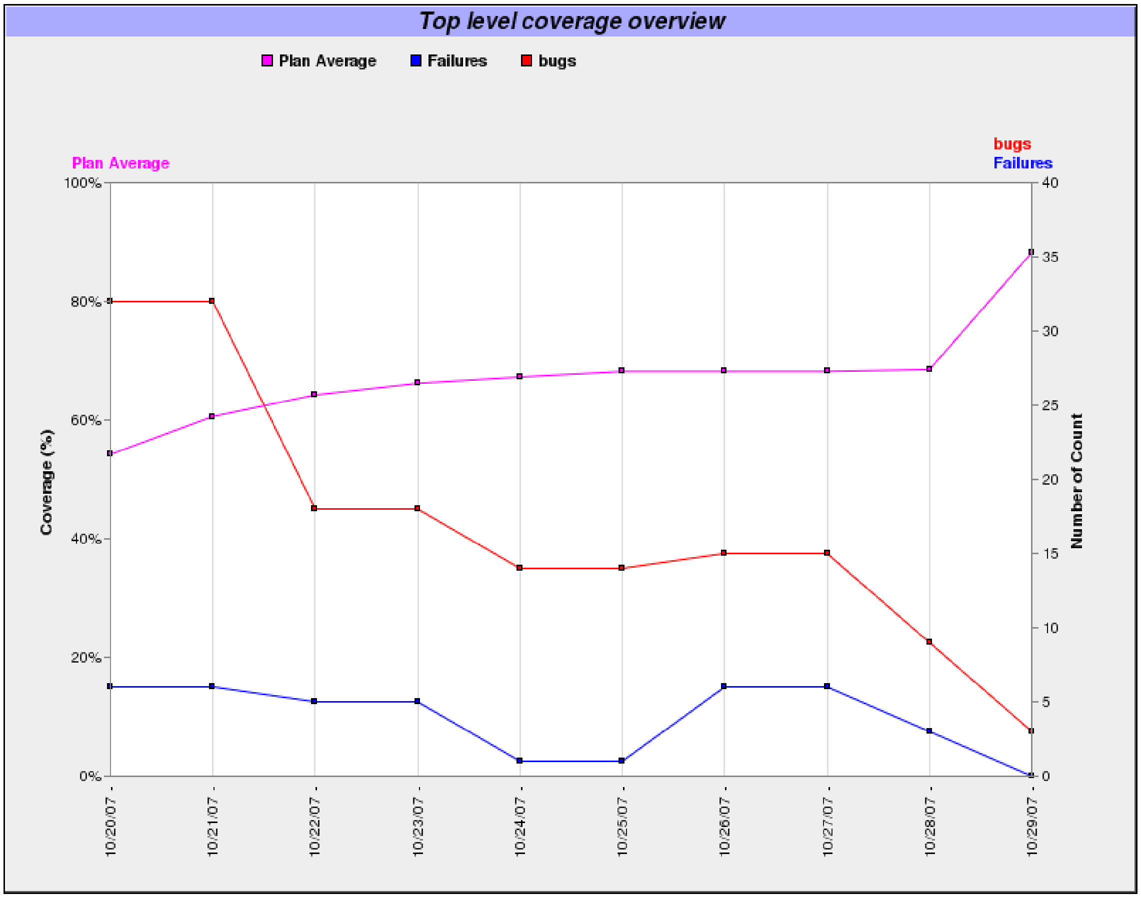


Figure 1-44   Top-level Trend Chart Using the offbasicavg Option



Navigating Trend Charts

URG includes the capability to navigate from one trend chart to another trend chart or to a URG report. You use a web browser to view trend charts. URG uses hyperlinks to link one graphical user interface, such as an element in trend charts to some section of information in URG reports.

URG trend charts provide the following three types of hyperlinks:

1. Legend label hyperlinks above the y-axes link to metric-wise detailed charts.

2. Curve hyperlinks on a trend chart link to instance-wise or feature-wise hierarchical sub-level charts.

3. Session date hyperlinks of the x-axis link to older URG reports used to generate the data points on a trend charts.

The next section describes in more details the behavior of these various links.

In addition to hyperlinks, URG trend charts show graphical user interface elements like tooltips where appropriate. You hover the cursor over an element on a trend chart, without clicking it, and a small box appears with supplementary information regarding the element. Tooltips allow supplementary information to be annotated to the chart without cluttering it since only one tooltip can be displayed at a time. URG provides the following tooltips usage:

- The tooltips for curves display the metric name and score of the point for each curve. In addition, if the curve is clickable, the tooltip shows the next chart that you can see by clicking the label. For example:

  ```
  78.1% - Basic raw coverage score curve
  90.5% - Click to see feature-wise subhierarchy breakdown
  for Plan Average Score
  ```

- The tooltips for the legends above y-axis show the next chart that you can see by clicking the label. For example:

  ```
  Click to see metric-wise basic coverage breakdown chart
  ```

- The tooltips for the x-axis date labels show the exact date and time in `mm/dd/yyyy hh:mm:ss` format and the name of the URG report for each session.

Notice that each x-axis date label displays the date of each session in MM/DD/YY format without a time. If the number of sessions increases, URG shows only the “-” character instead of the date for each session.

Trend Chart Linkage

The approach to view coverage and VMM Verification Plan information is to start from high-level aggregated results down to low-level more specific information and data. If the `-trend` option is given, URG generates a top-level chart that makes a natural starting point for exploring trend information.

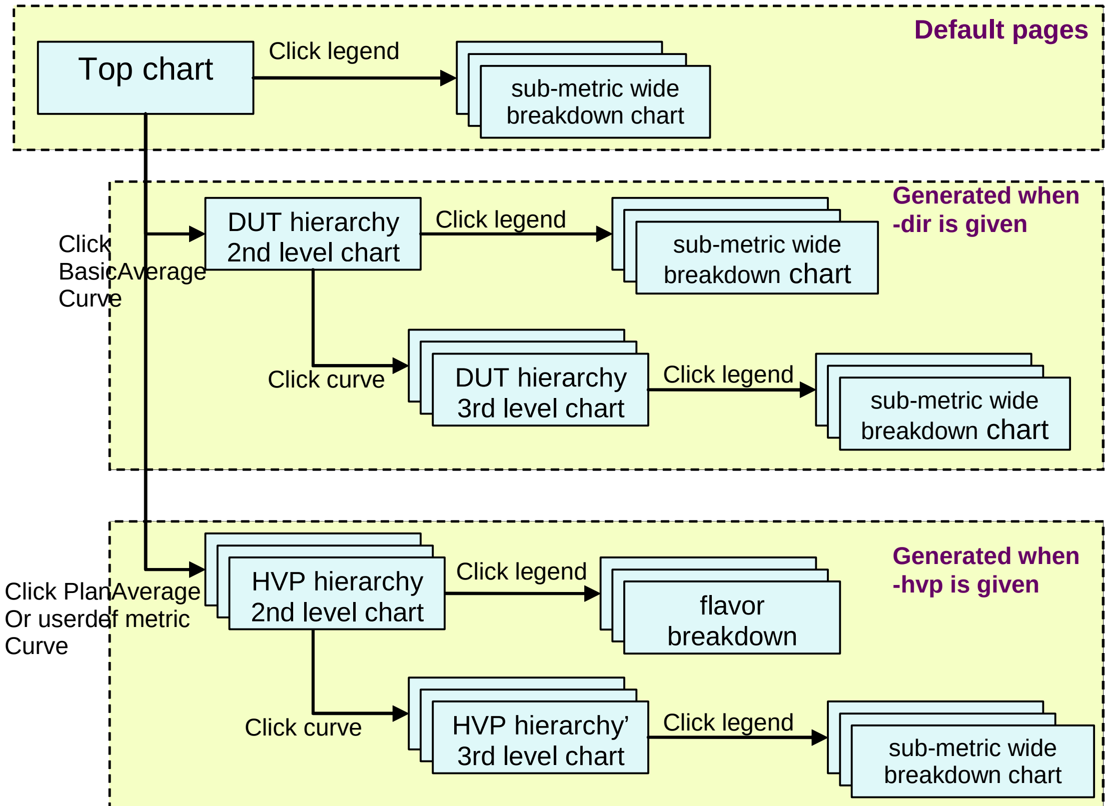
Many of the linkage paths are hierarchical, although the chart linkage does not need to be strictly hierarchical. There are two main hierarchies to consider: metric-wide and design-wide. The metric-wide hierarchy breakdown expands metrics into their sub-metrics. The design-wide hierarchy is simply the DUT hierarchy for Basic Average or the verification plan feature hierarchy for all other VMM Plan metrics. This section covers the following five topics:

- “Organization of Trend Charts”
- “Top-Level Chart”
- “Metric-Wide Breakdown Linkage”
- “Hierarchical Linkage”
- “Links to Previous Sessions”

Organization of Trend Charts

One HTML page displays one or more chart images. All other charts can be accessed from the top-level chart by clicking on curves or legend labels to drill down on the metric-wide or design-wide hierarchy. Figure 1-45 shows a diagram that illustrates the URG trend chart organization:

Figure 1-45 Trend Chart Hierarchy



Top-Level Chart

URG displays a top-level chart when you select the **`trend`** tab in the menu bar from a URG report (see Figure 1-41). It shows the raw coverage metric score which is called the Basic Average score by default. If you use the `-plan` option, URG will display the top-level VMM Plan Average score, the Failures for `tests.fail` metric, and other user-defined metrics such as bug count.

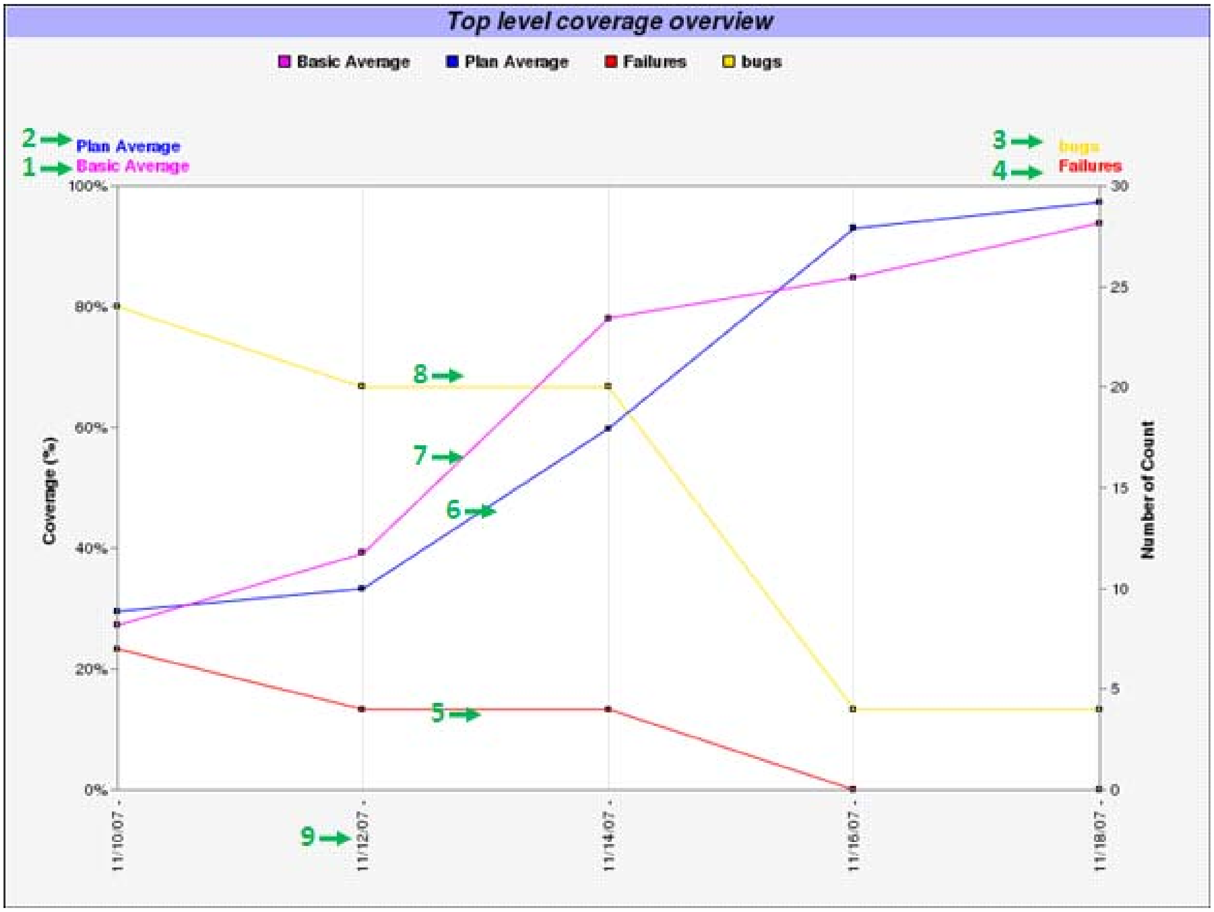
If you do not use the `-plan` option, the top-level chart will simply display a single line charting of the basic raw coverage average.

Although it seems redundant to include both the top-level raw coverage and the top-level VMM Plan coverage score on the same chart, yet you can use both information to perform basic sanity checks to ensure that your verification plan correlates sensibly with the raw metrics. You can turn off the Basic Average coverage display by specifying the `-trend offbasicavg` option.

In Figure 1-46, the top-level trend chart contains four metric legends:

- the Plan Average legend for VMM Plan coverage score.
- the Basic Average legend for raw coverage.
- the bugs legend for user-defined metric bug count.
- the Failures legend for test failure count.

Figure 1-46   Top-Level Chart with Linkage Description



The Failures metric is the same as the VMM Plan built-in metric `test.fail`. Note that the bugs metric is a simple user-defined integer metric for bug count in this particular chart and is not displayed by default in other trend charts.

The y-axis on the left-hand side represents the percentage of coverage, another y-axis on the right-hand side represents integer count. The legend labels above each y-axis identify the relation between curves and their axes.

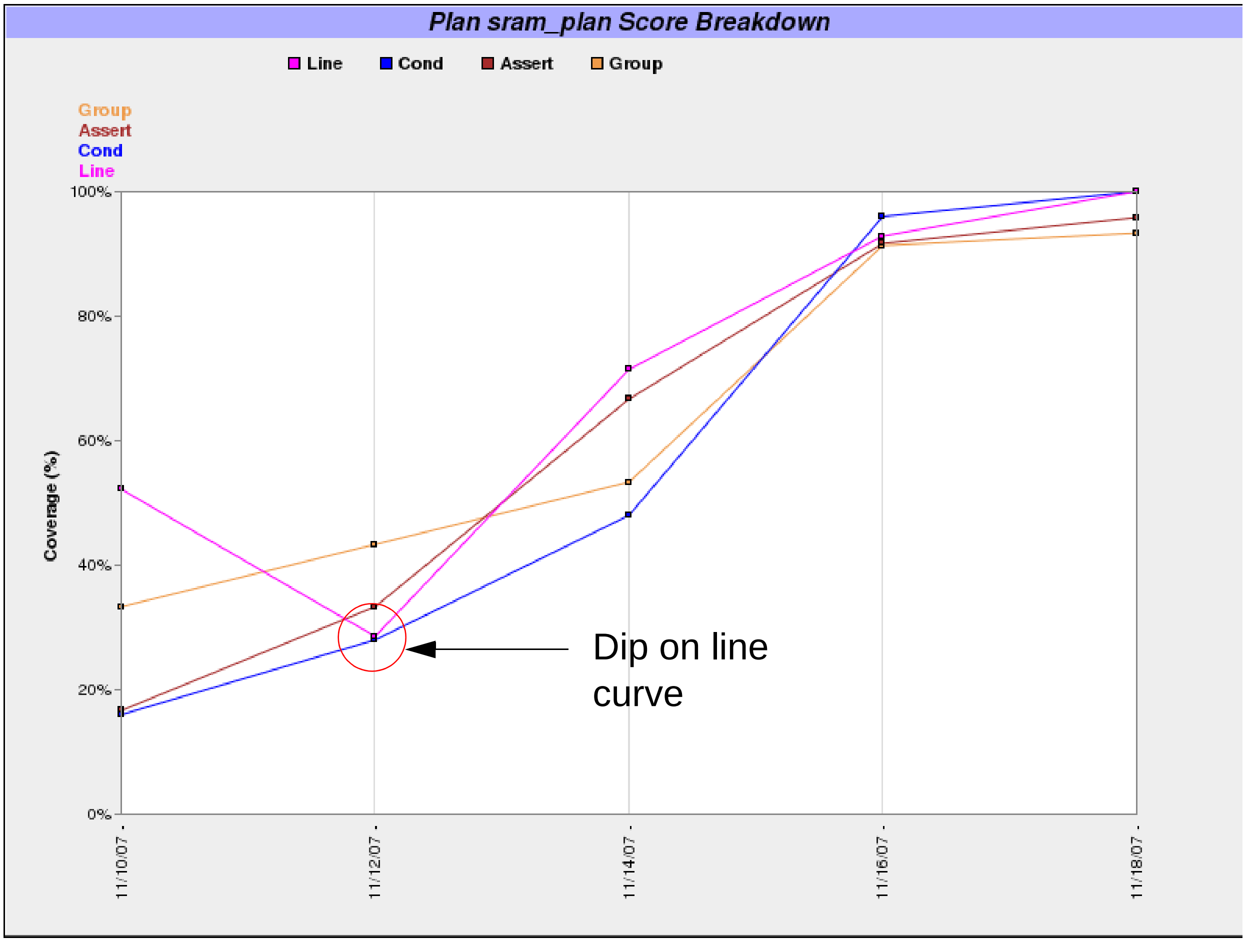
The Failures legend in Figure 1-46 represents the `test.fail` count. The Failures legend links to other enumerated scores in the test metric, such as total test count, `test.pass`, `test.warn`, and so on.

The metrics that have the same type of values (for instance, `test.fail` and the bugs metric integer values) share the same axis. If the score values of metrics vary over a wide range, the curves might not reflect the trend well because the metric with a lower range will be compressed due to the need for a higher range of y-axis.

Metric-Wide Breakdown Linkage

To view the complete breakdown chart of a metric, click a metric legend label above either of the y-axes. In Figure 1-46, the legend labels at the top of the y-axis are metric-wide breakdown chart links (Group, Assert, Cond, and Line). The Basic Average and Plan Average breakdown charts show the components that comprise the average score.

Figure 1-47 Plan Average Score Breakdown Page

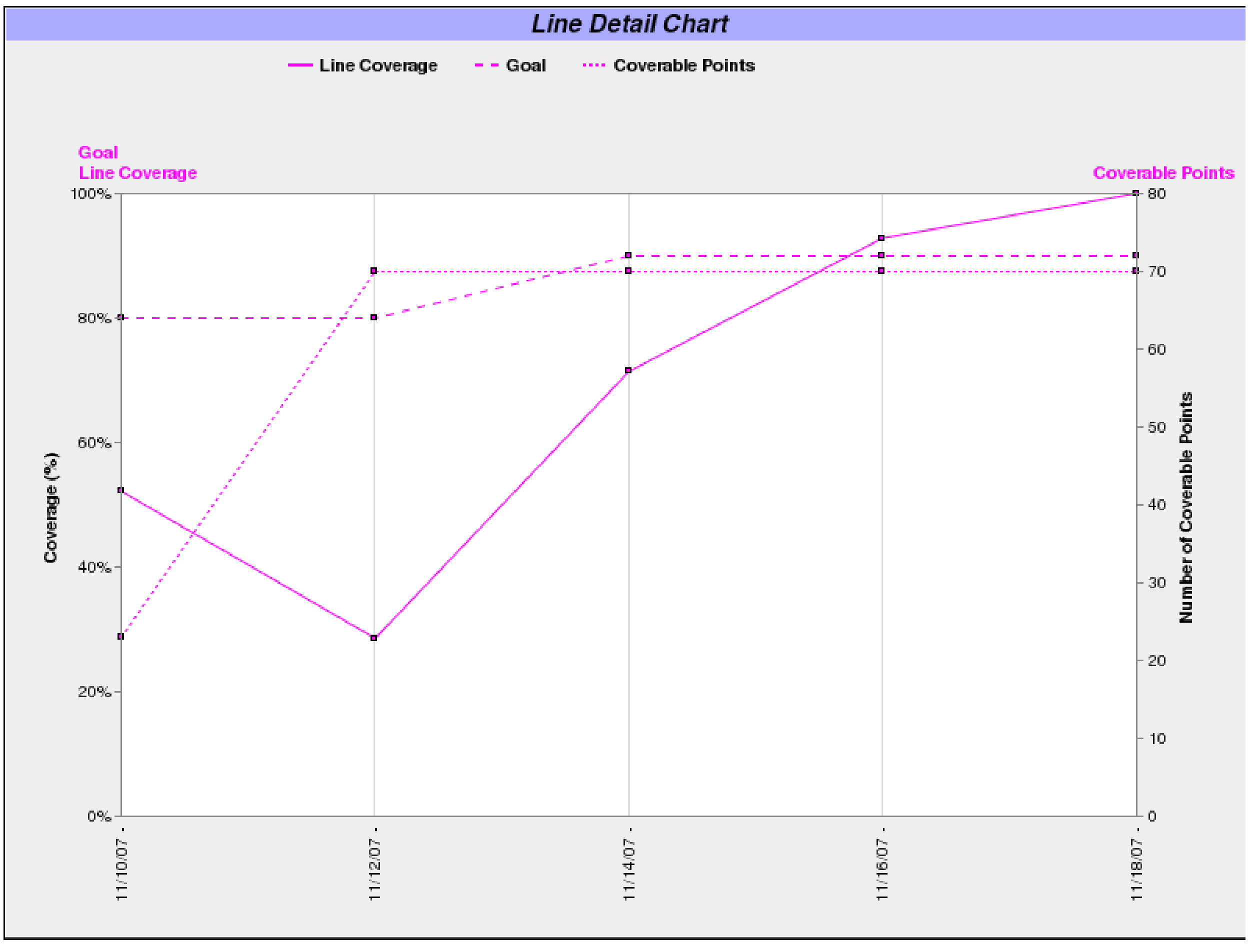


When you click the green arrow in Figure 1-46, you see the Plan Average score breakdown page. This page displays multiple charts. Figure 1-47 shows coverage metric curves that comprise the Plan Average score.

The metric detail curve is useful in many cases. For instance, In Figure 1-47, there is a dip on the Line curve for the 11/12/2007 session. This dip could be caused the introduction of new code. If you inspect the Line detail chart, you see that total number of coverable objects suddenly increased on 11/12/2007. This increase in coverable objects caused the drop of Line coverage score. The metric detail curve allows you to compare the goal and the real score in one chart. According to the chart, the Line score started meeting the goal on 11/16/2007.

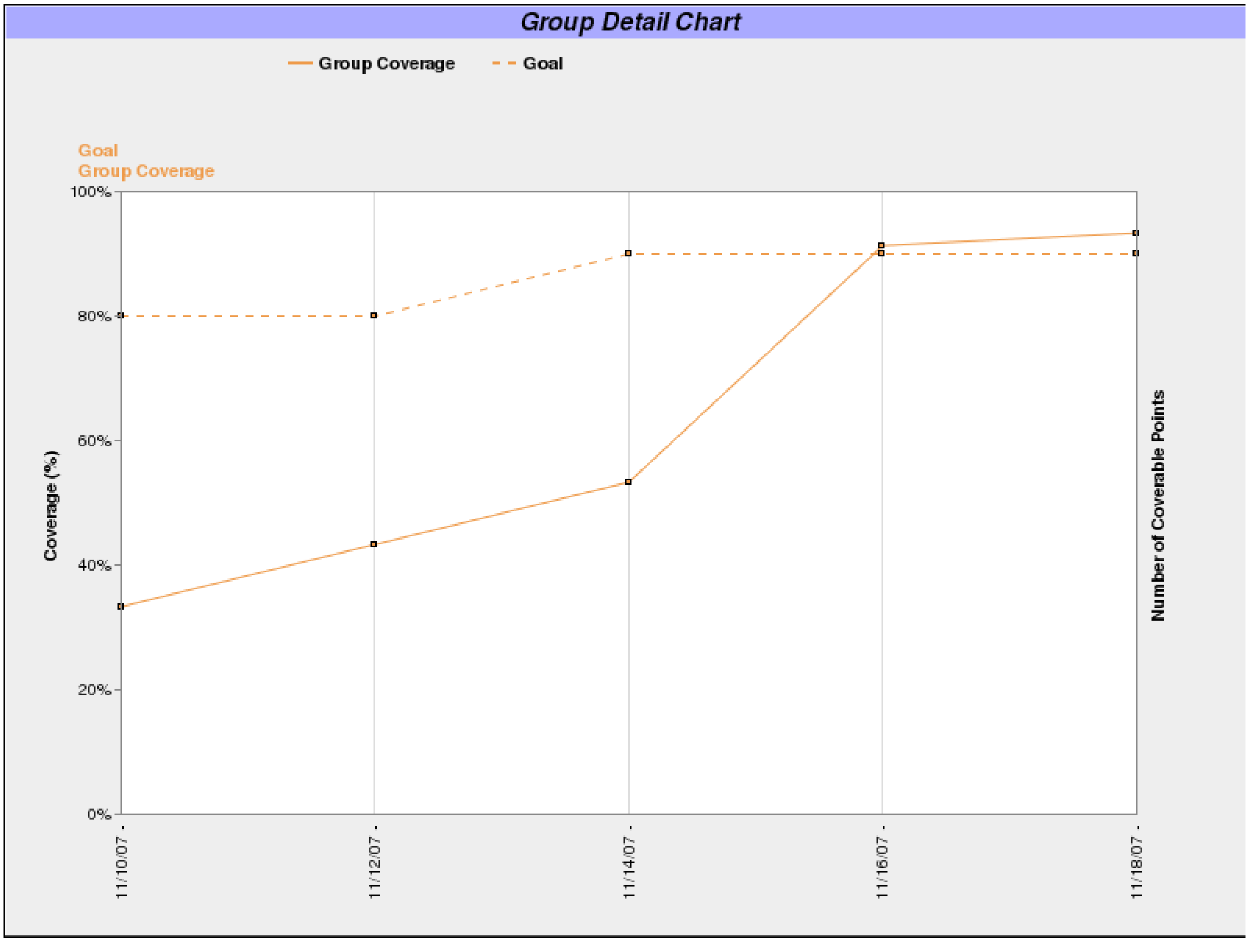
Figure 1-48 and Figure 1-49 show each metric's details, including the total coverable points curve and goal curve (if they exist).

Figure 1-48 Line Metric Detail for Plan Average Breakdown



In Figure 1-48, the three curves represent the Line Coverage score, the Goal of the line metric, and total number of Coverable Points, respectively.

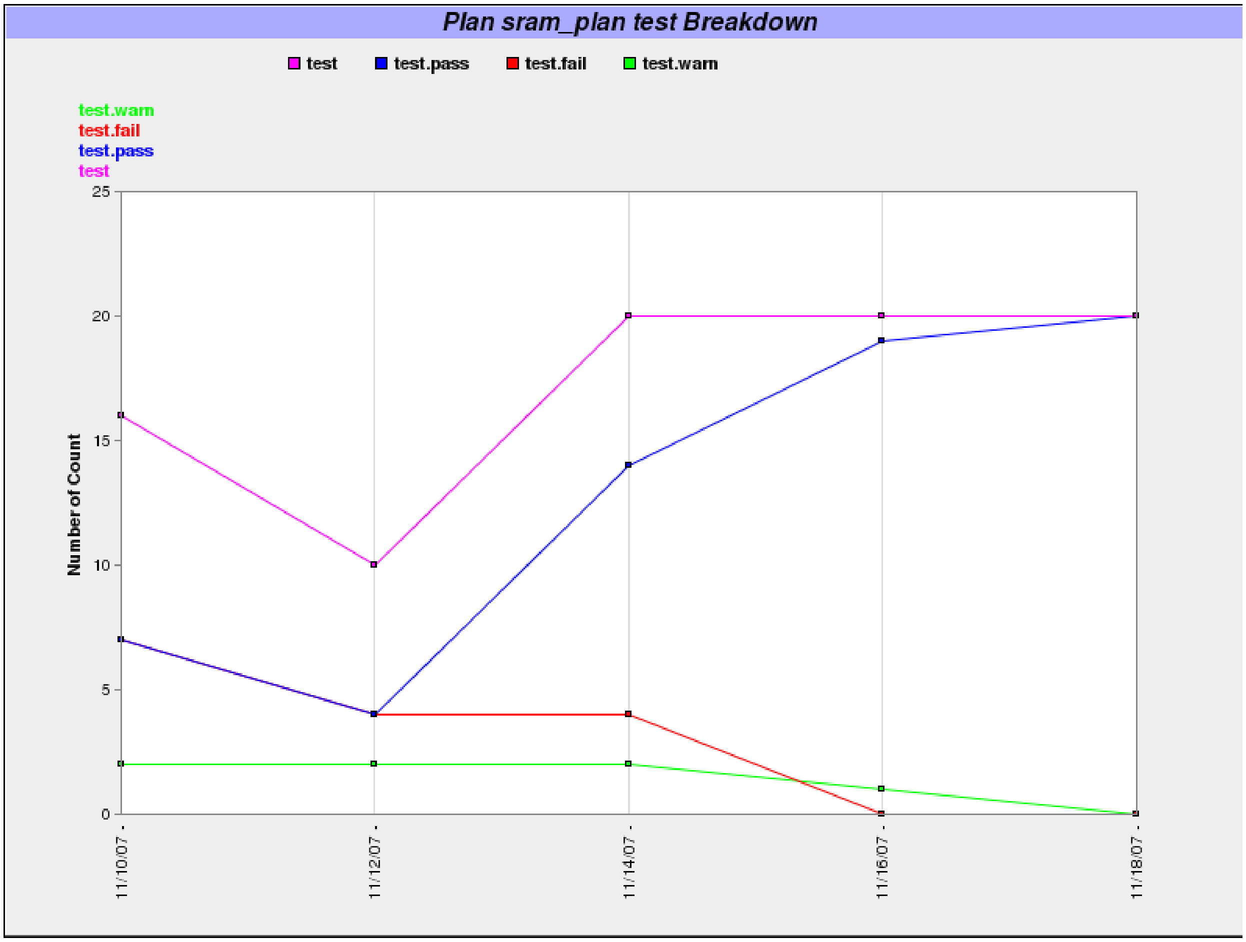
Figure 1-49   Group Metric Detail for Plan Average Breakdown



In Figure 1-49, there is no curve for the total number of coverable objects because the group coverage metric is not a ratio type, but rather is a percentage type. If you have not specified a goal for the metric, URG does not create goal curve.

If there are other enumeration types of user-defined metrics in the verification plan, their legend label hyperlinks operate the same way as described in this section.

Figure 1-50 Failure Breakdown Chart



Notice that curves for instances such as `test.unknown` and `test.assert` may not be visible on a trend chart. It is because they have zero values throughout the time span of trend analysis and are therefore not interesting.

If a verification plan has several user-defined metrics, all of the metric curves are plotted on the top-level chart. Each hyperlink of the user-define metric curves links to feature-wide sub-level chart. Also, if any of the user-defined metrics is an enumeration type, the hyperlink on legend label links to the enumeration breakdown chart the same way as the hyperlink does for `test.fail` label.

Each breakdown chart such as the Basic Average, the Plan Average, or the Failures, is a leaf node chart. Therefore, there is no more linkage either on legend labels or curves to additional charts except for the session timestamp labels on the x-axis.

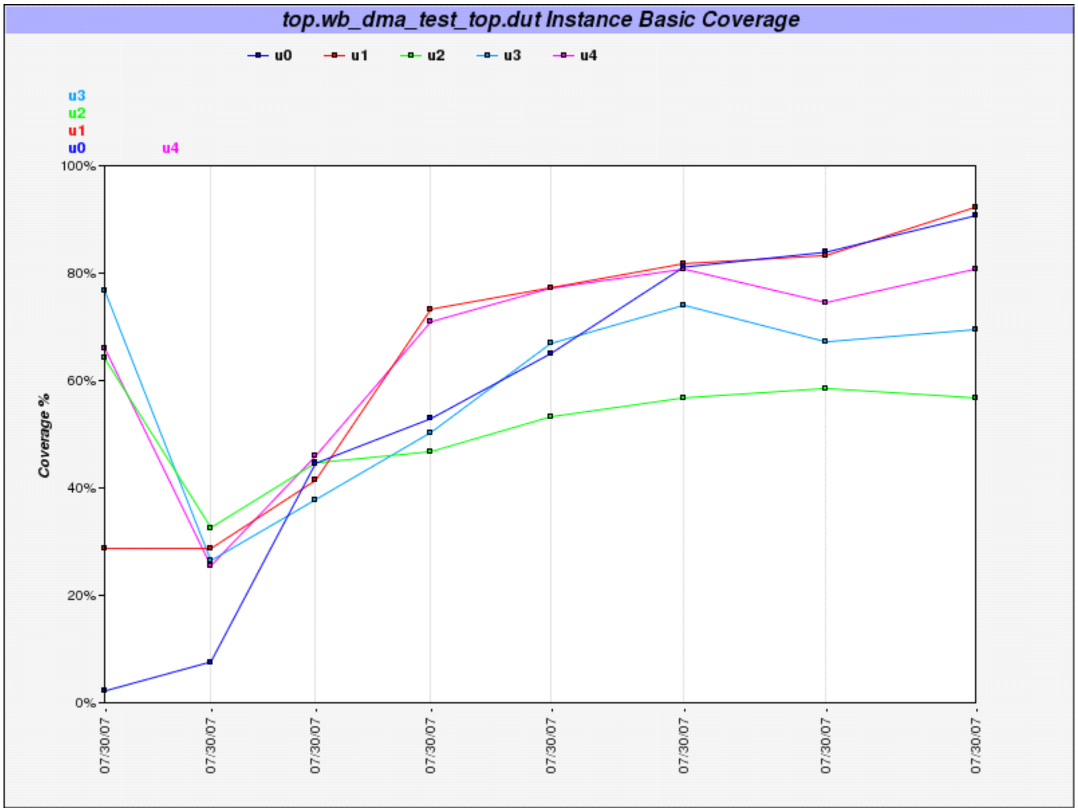
Hierarchical Linkage

You can view subhierarchy charts by clicking individual curve on a top-level chart. Take Figure 1-46 for an example, the green arrows 5, 6, 7, and 8 are subhierarchy chart hyperlinks. Each curve displays the trend of a metric and is color coded to match the metric legend it represents. Figure 1-46 displays:

- the Basic Average curve of raw coverage DUT instance hierarchy. It is clickable if you specify the `-dir covdb` option to generate the URG reports and the `depth` value is at least two or higher.
- the Plan Average curve of VMM Plan feature subhierarchy. It is clickable if the `depth` value is at least two or higher.
- other metric curves of VMM Plan feature subhierarchies. They are clickable if the `depth` value is at least two or higher.

For example:

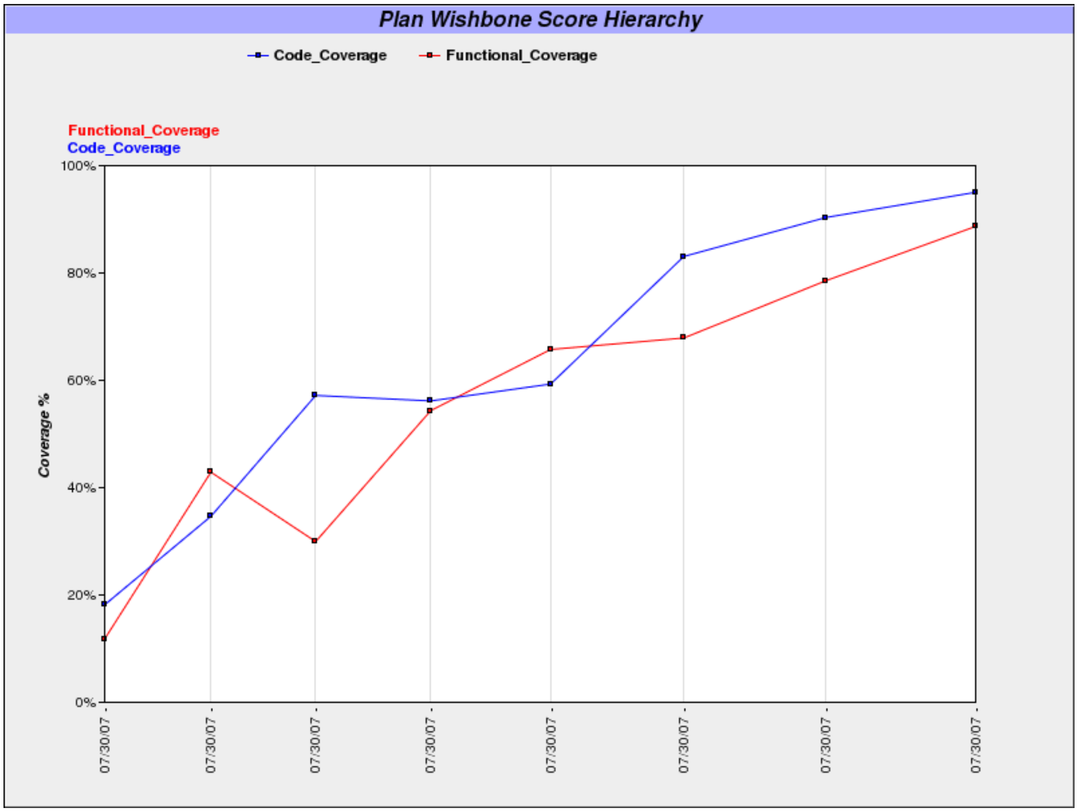
Figure 1-51   Plan Average Feature Hierarchy Chart



Click the Basic Average legend (the green arrow 1 in Figure 1-46), which links to the chart of raw coverage score as shown in Figure 1-51. The `top.wb_dma_test_top.dut` instance contains five sub-instances. You can drill down one more level by clicking any individual curve to view the sub-instance chart. Further, the legend labels are also clickable to view the flavor breakdown charts of this average coverage score.

The hierarchy levels of charts are determined by the value of `depth` you specify with the `-trend depth` option. As you increase the `depth` value, the exponential increase in chart number lengthens URG runtime and takes up disk space. Therefore, it is recommended that you use a `depth` value of 4 or less.

Figure 1-52   VMM Plan Hierarchy Chart for Plan Average Score



Click the Plan Average legend (the green arrow 2 in Figure 1-46), which links to the chart of coverage score curves for each sub-feature as shown in Figure 1-52. Each curve is clickable if you specify the `-trend depth <number>` argument with a value of 2 or higher and there are indeed VMM Plan subhierarchies available. The legend labels above the y-axis are also clickable for flavor breakdown charts.

Other VMM Plan metric curves on the top chart are also clickable if their sub-features exist and the `-trend depth <number>` argument is large enough.

If you do not invoke URG with the `-plan` option, you will not see VMM Plan subhierarchy charts. Instead, you will only see the top-level chart which contains raw coverage data.

Links to Previous Sessions

The session date labels on the x-axis of a trend chart are clickable. You can use these links to view the URG report dashboard pages for previous sessions. If the number of sessions is too large to display clearly, the date labels for some sessions might be replaced with the "|" character to avoid text overlay. The tooltip of the "|" label shows the timestamp, and is clickable.

If the current URG report is located under the same trend root directory as other URG session reports, then the URL to the other session is a relative path, such as `../../urgReport_session_N/dashboard.html`. Therefore, this hyperlink works even if you move the whole trend root directory to another path or you access the URG report from a Windows disk mount.

On the other hand, the hyperlink URL is based on the absolute UNIX path. In this situation, the hyperlink cannot be resolved.

URG Trend Chart Command-Line Options

Following are the command-line options for `-trend`.

```
%urg -trend (root path [rootdepth N] | src dir1 [dir2 ... ] ) [other_options]
```

`root` *path*

Refers to the base path that contains URG reports from previous sessions. URG uses reports in this `root` directory to facilitate trend analysis. If you run your current session in the same directory, the resulting reports from the current run will also be included in the trend analysis.

`src` *`dir1`* `[`*`dir2 dir3`*`...]`

Specifies the `urgReport` directories if the URG reports are saved in various locations. You can use both `root` and multiple `src` options to locate the `urgReport` directories.

`rootfile` *`txtfile`*

Permits enumeration (in a text file) of multiple root paths for scanning.

`srcfile` *`txtfile`*

Permits enumeration (in a text file) of all URG report paths.

`linestyle`

Displays each curve of a trend chart with a different line style (solid line, dashed line, and so on) and color. If the linestyle option is not given, all curves are shown in solid lines with different colors. This option is particularly useful for black-and-white chart printing.

`offbasicavg`

Turns off basic coverage curves and displays only VMM Plan related score curves.

`depth` *`N`*

Specifies the number of hierarchy scope levels for which to generate instance and feature charts. The default level number is 1 (that is, only the top-level chart is generated).

`rootdepth` *N*

Defines the depth of the `root` path hierarchy through which URG recursively scans to find URG reports. If you do not specify `rootdepth`, the default depth *N* is `1`.

Note:

You can use the `-trend` option with other URG options.