Coverage Technology Reference Manual

I-2014.03
March 2014

SYNOPSYS®

Copyright Notice and Proprietary Information

Contents

1

Introduction

The Coverage Technology Reference Manual contains list of coverage commands that you can use for your reference while using VCS to collect your coverage data. This manual is not a comprehensive guide for coverage technology and should be used in conjunction with the Coverage Technology User Guide.

If you are using the VCS Online Documentation, click this link *Coverage Technology User Guide* to view in the HTML interface. If you are using the PDF interface, see cov_ug.pdf to view the PDF document.

The chapter contains the following sections:

- "Operations When You Compile or Simulate"

- "Merging VCS Results for Regressions"

- "SystemVerilog Support for Code Coverage"

Operations When You Compile or Simulate

There are many operations related to generating coverage metrics that you can perform while compiling or simulating your design. These include specifying coverage for libraries and cells, naming intermediate data files, using glitch suppression, and so on. You can also control the scope of the coverage compilation using either pragmas or a -cm_hier configuration file.

Using Coverage Metrics Files and Directories

When you use the -cm compile-time option, VCS creates the simv.vdb directory (the coverage metrics database) in the current directory. The directory structure within the simv.vdb directory is as follows:

```
<DB Directory>
    |___ snps
            |_____ fcov
            |                   ...
            |_____ coverage
                        |___ db
```
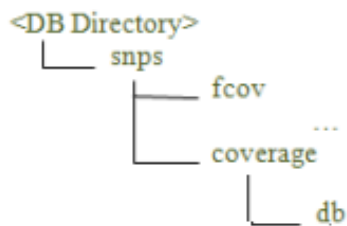
During compilation, VCS writes data files about the design in the snps/coverage/db/design directory (for Verilog and VHDL).

During simulation, VCS writes intermediate data files (also called test files) that record the coverage results from the simulation in the snps/coverage/db/testdata directory (for Verilog and for VHDL). Each directory under testdata contains all the coverage data for the test. The base name of the test is the name of the directory.

To see how to select names for the coverage data files, see "Naming Intermediate Data Files" . You might want to do so if you plan multiple simulations with different stimuli and you want to see the coverage results from the different simulations.

Also during simulation, VCS writes design information into the `simv.vdb/snps/coverage/db/` directory. These files are compatible with all the supported platforms. For example, if you have VCS compile and monitor for coverage using the Solaris platform, you can display coverage information using the RHEL32 platform. See the release notes (available through the online HTML documentation system) for a list of supported platforms.

The `simv.vdb` directory is named after the simv executable file. If you name the simulation executable file something else with the `-o` compile-time option, VCS names the coverage metrics database directory after the name you assigned to the simulation executable. For example, if you name the executable, `mysimv`, as follows:

```
% vcs source.v -cm fsm -o mysimv
```

VCS creates the `mysimv.vdb` directory instead of the `simv.vdb` directory.

You can also use the `-cm_dir` *directory_path_name* compile-time option and argument to specify a different name and location for the `simv.vdb` directory. For example:

```
% vcs -cm tgl -cm_dir /net/design1/mycm source.v
```

This command line instructs VCS to create the `mycm.vdb` directory in the `/net/design1` directory instead of the `simv.vdb` directory in the current directory. The `-cm_dir` compile-time option takes precedence over the `-o` compile-time option when specifying the new name for the `simv.vdb` directory.

When you use the `-cm_dir` option at compile time, the name and location of this directory is hard coded into the simv executable, so there is no need to use the option at runtime to specify this directory.

Using the `-cm_dir` option at runtime specifies a different location for the test files that VCS writes at runtime.

If you specify a different location for the intermediate data files at runtime, you must specify that directory's location using the `-dir` option to URG when generating reports, all of the directories can be listed as well with one `-dir` option. For example, `-dir db1.vdb db2.vdb`.

To summarize when to use `-cm_dir` and for what purpose:

- At compile-time, to specify a different name or location for the coverage database directory (`simv.vdb`) that is created during compilation.

- If you move the coverage database directory or the binary executable to a new location after VCS creates it during compilation, use the `-cm_dir` option at runtime, to specify the path for the coverage database directory .

- If you invoke your binary executable from a different location, then use the `-cm_dir` option at runtime to specify the path for the coverage database directory.

The following command lines illustrate these different possibilities:

```
% vcs -cm line -cm_dir /net/design1/my_cov_info source.v

% simv -cm line -cm_dir /net/design1/int_dat_files

% urg -dir /net/design1/my_cov_info -dir /net/design1/\
int_dat_files
```

Specifying Coverage for Libraries and Cells

By default, VCS does not compile the following for coverage:

- The source code in Verilog library directories

- Verilog library files

- Any module defined under the `celldefine` compiler directive

If you want this excluded code compiled for coverage, include the `-cm_libs` compile-time option along with one or both of the following arguments:

- `yv` — for compiling for coverage source code from Verilog libraries.

- `celldefine` — for compiling coverage modules defined under the `celldefine` compiler directive.

If you want to specify both arguments, include the plus delimiter (+) between the two arguments. For example:

```
% vcs source.v -v mylib.v -y /net/libs/teamlib -cm fsm \
-cm_libs yv+celldefine
```

Naming Intermediate Data Files

By default, when VCS monitors for any type of coverage, it records the results in intermediate data files named "`test`" with various extensions, as follows:

- `line.verilog.data.xml` for line coverage

- `fsm.verilog.data.xml` for FSM coverage

- `cond.verilog.data.xml` for condition coverage

- `tgl.verilog.data.xml` for toggle coverage

- `branch.verilog.data.xml` for branch coverage

By default, VCS writes these files in the `./simv.vdb/snps/coverage/db/testdata` directory. URG reads these files to show coverage results.

If you store the results of multiple simulations in the same directory, you will want to have a different name for these intermediate data files directory. For example, you might want to apply different stimuli in different simulations and have intermediate data files with different names so that you can compare coverage results using URG. To specify a different name for the intermediate data files, you can do either of the following:

- Include the `-cm_name` option at compile-time. For example:

  ```
  % vcs source.v -cm line -cm_name testm
  ```

  This command line compiles into the executable the name that the executable will use when it writes the intermediate data files. After simulation, the `simv.vdb/snps/coverage/db/testm` directory contains the `line.verilog.data.xml` file.

- Include the `-cm_name` option at runtime, for example:

  ```
  % simv -cm line -cm_name testm
  ```

  You can override the directory name (testm) that you have provided during compilation by providing a different directory name with the simv command line options (say testn).

  You cannot use the option `-cm_name` to specify a different location for these files. Use the `-cm_dir` option instead to have simv place the generated files in a different directory.

Merging VCS Results for Regressions

URG automatically merges the results from the intermediate data files that VCS writes during more than one simulation of your design. This section describes four methods for VCS for running the regressions to produce merged coverage results. In most of these methods, you only need to compile for coverage once and then run multiple simulations, either sequentially or in parallel. You can use either different names for intermediate data files in the same directory or create different directories for these intermediate data files and then inform URG where to look for them.

Method 1: Build the simv executable once, then simulate three times sequentially using the same testbench, but different inputs.

There are a number of ways to use the same testbench in different simulations with different input stimulus values. One common way is to use the `$readmemb` system task to read values from a file into the elements of a memory. Then for each simulation, you change the

contents of the data file specified in the system task. VCS reads this data at runtime and you make no changes to the system task. Therefore, you only need to compile for coverage one time.

The values in the memory elements control the execution of your design and exercise different parts of the design during the different simulations. When the simulations are over, you can see the total and merged coverage from all three simulations.

This procedure is as follows:

1. Compile for coverage:

   ```
   % vcs -cm coverage_arguments \
   additional_options_and_source_files
   ```

   This command line instructs VCS to compile for the selected types of coverage. VCS creates the `simv.vdb` directory in the current directory. In this directory, it creates the `snps/coverage/db` directory.

2. Start the first simulation, specifying a name for the intermediate data files that associates them with the first simulation:

   ```
   % simv -cm_name test1 -cm coverage_arguments \
   additional_runtime_options
   ```

   During simulation, VCS writes the `line.verilog.data.xml,` `fsm.verilog.data.xml` intermediate data files in the `./simv.vdb/snps/coverage/db/testdata/test1` directory.

3. Revise the contents of the data file in the `$readmemb` system task and then start the second simulation, specifying a name for the intermediate data files that associates them with the second simulation:

```
% simv -cm_name test2 -cm coverage_arguments \
additional_runtime_options
```

During simulation, VCS writes the `line.verilog.data.xml`, `fsm.verilog.data.xml` intermediate data files in the `./simv.vdb/snps/coverage/db/testdata/test2` directory.

4. Revise the contents of the data file in the `$readmemb` system task again and then start the third simulation, specifying a name for the intermediate data files that associates them with the third simulation:

```
% simv -cm_name test3 -cm coverage_arguments \
additional_runtime_options
```

During simulation, VCS writes the `line.verilog.data,xml`, `fsm.verilog.data.xml` intermediate data files in the `./simv.vdb/snps/coverage/db/testdata/test3` directory.

5. Run URG to generate report and a merged database:

URG reads all the test data files in the `./simv.vdb/snps/coverage/db` directory and writes the report files with the merged results in the `urgReport` directory.

Method 2: Build the simv executable once, then simulate three times sequentially as in the previous method, but write the intermediate data files to a common directory for your design group.

This procedure is as follows:

1. Compile for coverage:

```
% vcs -cm_dir /net/design1/oursimv.vdb \
```

```
            -cm coverage_arguments \
                    additional_options_and_source_files
```

This command line instructs VCS to compile for all types of
coverage. VCS creates the `oursimv.vdb` directory in the
`/net/design1` directory, and in the `oursimv.vdb` directory, it
creates the `snps/coverage/db` directory.

2. Start the first simulation, specifying both the directory that VCS
   created for coverage information, as done in the previous step,
   and a name for the intermediate data files that associates them
   with the first simulation:

   ```
   % simv -cm_name test1 -cm coverage_arguments \
   additional_runtime_options
   ```

   During simulation, VCS writes the `line.verilog.data.xml`,
   `fsm.verilog.data.xml` intermediate data files in the `/net/`
   `design1/oursimv.vdb/snps/coverage/db/testdata`
   directory.

3. Revise the contents of the data file in the `$readmemb` system
   task and then repeat the previous simv command line changing
   only the name of the intermediate data files:

   ```
   % simv -cm_name test2 -cm coverage_arguments \
   additional_runtime_options
   ```

4. Repeat the previous step, revising the contents of the data file
   and the name of the intermediate data files.

   ```
   % simv -cm_name test3 -cm coverage_arguments \
   additional_runtime_options
   ```

5. Now run URG, telling it the name and location of the directory for
   coverage data that you specified in step 1 and to write a merged
   database:

```
% urg -dir /net/design1/oursimv.vdb -dbname mergedir/\
merged
```

Method 3: Only for VCS, build the simv executable once, then simulate three times in parallel.

It is possible to compile for coverage once and simulate the design multiple times in parallel using different inputs. You do this by using the `$test$plusargs` VCS system function and the corresponding runtime options. You can use this system function to execute different `$readmemb` system tasks depending on the runtime option you use for each parallel simulation.

This procedure is as follows:

1. Compile for coverage:

   ```
   % vcs -cm_dir /net/design1/oursimv.vdb -cm \
   coverage_arguments additional_options_and_source_files
   ```

2. Start the three simulations with the following command lines specifying both the directory that VCS created for coverage information, as done in the previous step, and a name for the intermediate data files that associates them with the first simulation:

   ```
   % simv -cm_name test1 -cm coverage_arguments +one\
   additional_runtime_options
   ```

   ```
   % simv -cm_name test2 -cm coverage_arguments +two\
   additional_runtime_options
   ```

   ```
   % simv -cm_name test3 -cm coverage_arguments +three\
   additional_runtime_options
   ```

   The `+one`, `+two`, and `+three` options are used for the `$test$plusargs` system tasks in the testbench.

3. Run URG once again specifying the name of the directory and the name of the report files.

```
% urg -dir /net/design1/oursimv.vdb -dbname \
mergedir/merged
```

Method 4: Build three different simv executables and simulate sequentially or in parallel.

To simulate sequentially, the procedure is as follows:

1. Compile three times for coverage using different testbenches, specifying an alternative name for the simv executable. By doing this, you are also specifying an alternative name for the `simv.vdb` coverage metrics database:

```
vhdlan [additional_options] source_files //For VCS MX
% vcs -o simv1 test.v -cm coverage_arguments \
additional_options_and_source_files

% vcs -o simv2 test.v -cm coverage_arguments \
additional_options_and_source_files

% vcs -o simv3 test.v -cm coverage_arguments \
additional_options_and_source_files
```

When VCS does these compilations for coverage, it creates three coverage metrics directories in the current directory: `simv1.vdb`, `simv2.vdb`, and `simv3.vdb`.

Note:
In step 1, all three tests use different stimulus. However, they all use the same source files from the same directories:

- Identical hierarchy

- No differences with `+define` or `` `include``

- Source files are passed identically

If some source files are passed as a library in one design and not in the other, then use `cm_libs`.

2. Simulate the executables either sequentially or in parallel.

3. Run URG telling it the name and location of the three directories that you created and the name of the report files:

```
% urg -dir simv1.vdb simv2.vdb simv3.vdb -dbname \
mergedir/merged
```
The report files are generated in the urgReport directory.

Merging Multiple Coverage Databases

The preferred method to merge different coverage databases is to provide the coverage database, which contains code coverage, as the first one to URG.

For example, assuming that the design is not changed between the multiple runs:

1. First run (compile+simulation) contains the assert coverage alone

2. Second run (compile+simulation) contains the Code coverage alone

Use the following URG command to merge multiple coverage databases:

```
% urg –dir ./codeCov/simv.vdb  –dir ./assert/simv.vdb \
–dbname merged
```

SystemVerilog Support for Code Coverage

VCS support for code coverage metrics is now available for SystemVerilog language constructs. However, VCS does not support all the constructs in SystemVerilog, essentially the non-dynamic constructs. The following table illustrates the support matrix.

Name of the Construct	CoverageMetrics	Unsupported Coverage	Description
always[comb, ff, latch]	All metrics	NA	NA
initial @	All	NA	NA
case/casez/casex with arrays	line+tgl+cond+ branch	fsm	NA
case/casez/casex with part selects	line+tgl+cond+ branch	fsm	NA
case/casez/casex with struct/union members as case variables with don't cares and const expressions.	line+tgl+cond+ branch	fsm	NA
unique/priority	tgl+fsm	line+cond+branch	NA
fork / Join	Valid only for line	NA	NA
Loops			
generate	line+tgl	NA	NA
break / continue	line+tgl+cond+ branch	NA	NA

Name of the Construct	Coverage Metrics	Unsupported Coverage	Description
return / disable	All metrics	NA	NA
INTERFACES			
Interfaces instantiated in modules	All metrics	NA	interface modports, being passed as a port to any module is not reported.
Blocks **specify block	All metrics	NA	NA
Tasks and functions	line+cond+fsm+tgl	Branch coverage is reported with -cm_cond	Arguments in task and functions not reported in tgl coverage.
modports	Not supported	All metrics	NA
Parameterized interfaces	All metrics	NA	NA
COVERAGE			
Datatypes			
Enum	line+fsm+branch	enum methods in conditional expression doesn't extract a condition.	NA
event	line	not monitored in tgl	NA
byte, shortint, int, longint, time, shortreal	line	not monitored in tgl	NA

Name of the Construct	Coverage Metrics	Unsupported Coverage	Description
bit, logic,	tgl	NA	NA
packed, unpacked	All	NA	NA
struct / union [tagged]	line+tgl+cond+ branch	part select on struct variable doesn't generate fsm	NA
Nested struct and union	line+tgl+cond+ branch	NA	NA
Array of structs and unions (MDAs as members)	line+tgl+cond+ branch	NA	NA
Arrays			If it is used inside an expression, conditional and branch coverage are reported.
Dynamic arrays	line+cond+branch	tgl	
Associative array	line+cond+branch	tgl	
Queues	line+cond+branch	tgl	
Parameterized Type	Works for toggle coverage on signals of "parameterized type"	NA	NA

Name of the Construct	CoverageMetrics	Unsupported Coverage	Description
Blocks			
clocking	Not Supported	Not supported	NA
Program Block	Not Supported	Not supported	NA
packages	Not Supported	Not supported	NA
Classes	Not Supported	Not supported	NA

Introduction

1-18

2

Commands Reference

This chapter contains the following sections:

Compile Options for Coverage Metrics

`-cm line|cond|fsm|tgl|branch|assert`

Specifies compiling for the specified type or types of coverage. The arguments specifies the types of coverage:

line — Compile for line or statement coverage.

cond — Compile for condition coverage.

fsm — Compile for FSM coverage.

tgl — Compile for toggle coverage.

branch — Compile for branch coverage.

assert — Compile for SystemVerilog assertion coverage.

If you want VCS to compile for more than one type of coverage, use the plus (+) character as a delimiter between arguments, for example:

`-cm line+cond+fsm+tgl`

`-cg_coverage_control=value`

Enables or disables the coverage data collection for all the coverage groups in your NTB-OV or SystemVerilog testbench. The system task `$cg_coverage_control` has precedence over this compile-time option.

The valid values are 0 and 1. A value of 0 disables coverage collection and a value of 1 enables coverage collection.

`-cm_assert_hier` *filename*

> Limits assertion coverage to the module instances specified in *filename*. This option applies to assertion coverage only that is when you use `-cm assert` option. If this option is not used, coverage is implemented on the whole design.
>
> You should use the `-cm_dir` or `-cm_name` options for renaming the coverage database.

Note:

> The option `-cm_assert_hier` is not supported for mixed language simulations.

`-cm_cond` *arguments*

> Modifies condition coverage as specified by the argument or arguments:

`basic`

> Only logical conditions and no multiple conditions.

`std`

> The default: only logical, multiple, sensitized conditions.

`full`

> Logical, non-logical operator, and all the possible vectors not just the sensitized vectors.

`allops`

> Logical and non-logical conditions.

`event`

> Signals in event controls in the sensitivity list position are conditions.

`for`

> Enables conditions in for loops.

`tf`

> Enables conditions in user-defined tasks and functions.

`obs`

> Enables observability based condition coverage.

You can specify more than one argument. You do this by using the plus (+) character between arguments. For example:

```
-cm_cond basic+allops
```

`-cm_count`

> Enables coverage report tools (URG or Cov GUI) to do the following:
>
> - In toggle coverage, reports not just whether a signal toggled from 0 to 1 and 1 to 0, but also the number of times it toggled in either direction (0 to 1 and 1 to 0 toggles are not counted separately).
>
> - In FSM coverage, reports not just whether an FSM reached a state, and had such a transition, but also the number of times it did.
>
> - In condition coverage, reports not just whether a condition was met or not, but also the number of times the condition was met.

- In line coverage, reports not just whether a line was executed, but how many times.

`-cm_dir directory_path_name`

Specifies an alternative name and location for the `simv.vdb` directory. For testbench coverage, `-cm_dir` option is only a runtime option. For code coverage, it's a compile-time and runtime option.

If you specify `-cm_dir` during compile-time and runtime, you can use the `-cm_dir` option to specify a different location to store the coverage databases. Then while running URG, you should first list the coverage directory specified at compile-time, followed by the coverage database directories created at runtime as follows:

```
% vcs -cm_dir ./cov_dbs/cov.vdb -cm line
% simv -cm_dir ./coverage/test1.vdb -cm line
% simv -cm_dir ./coverage/test2.vdb -cm line

% urg -dir ./cov_dbs/cov.vdb -dir ./coverage/test1.vdb \
-dir ./coverage/test2.vdb -metric line
```

Note:

For code coverage, if runtime database is different from the compile-time database, it is required to specify both the databases at report generation time. For assertion coverage, only runtime database is sufficient for report generation.

If you have used the `-cm_dir` option at compile-time and have moved `simv.vdb`, you must use `-cm_dir` at runtime to point to the new location of `simv.vdb`.

```
-cm_exclude_macrofile filename
```

Specifies a file containing Verilog macro names (as specified by the `define compiler directive). so if your Verilog source contained the following:

```
`define DFF(q,i,clock)
    always_ff @(posedge clock)
        begin
          if (clock === 1'bX)
              q <= #1 'x;
          else
            q <= #1 i;
        end
```

If you wrote a file named `macros_excl.txt` and its content was the name of the macro:

```
DFF
```

The VCS excludes the macro from code coverage when you enter the following:

```
% vcs source.v -sverilog -cm code_coverage_options \
-cm_exclude_macrofile macros_excl.tx
```

URG marks `'DFF` as unreachable in urgReports, so, for example, `urgReport/modinfo.txt` contains the following:

```
 9            always_ff @(posedge clk)
10              begin
11        1/1    b    = b     + 1;
12              end
13        unreachable    `DFF (a[2:0],b[2:0],clk)
```

Note:
- The contents of the macro exclusion file are one or more macro names. You can separate the macro names with a space or a tab or put the names on separate lines.

- The compile-time options, `-cm_ignore_pragma` and `-cm_exclude_macrofile` are mutually exclusive options. If `-cm_ignore_pragma` is used, the `-cm_exclude_macrofile` *filename* option will not have an effect.

`-cm_fsmcfg` *filename*

Specifies the FSMs that VCS or VCS MX extracts from a module definition.

- Specifies which states and which transitions between states, VCS or VCS MX keeps track of in the FSMs.

- Specifies the maximum number of sequences that VCS or VCS MX keeps track of in any of the modules or design entities in your design, and specifies the maximum length of any sequence that VCS or VCS MX keeps track of.

For more information, see the section "Using an FSM Configuration File"

`-cm_fsmopt` *keyword_argument*

The keyword arguments are as follows:

`allowTmp`

By default, the variable that holds the current state of the FSM must be directly assigned a numerical constant or the value of a variable that holds the next state of the FSM. This keyword allows FSM extraction when there is indirect assignment to the variable that holds the current state.

`report2StateFsms`

By default, VCS does not extract two state FSMs. This keyword tells VCS to extract them.

`reportvalues`

Specifies reporting the value transitions of the reg that holds the current state of a One Hot or Hot Bit FSM where there are parameters for the bit numbers of the signals that hold the current and next state. The default behavior is to identify these parameters as the states of the FSM and report assignments to their bits as state transitions.

`reportWait`

Enables VCS to monitor transitions when the signal holding the current state is assigned the same state value.

`reportXassign`

Enables the extraction of FSMs in which a state contains the X (unknown) value.

`-cm_fsmopt excludeCalcFsms`

Rejects the FSMs with one of the following statements, when you use it with the `-cm fsm` switch, where cs = current state and ns = next state:

```
cs = Expr(cs) : Expr(cs) contains arithmetic;
    (following Expr is same)
cs = Expr(ns) ;
cs = Expr(foo);  : foo is a Net or Reg
ns = Expr(cs);
ns = Expr(ns);
ns = Expr(foo);
```

For example if the FSM contains the following line, then it will not be extracted.

```
ns = ns +1;
```

`-cm_fsmresetfilter` *filename*

> Filters out transitions in assignment statements controlled by `if` statements where the conditional expression (following the keyword `if`) is a signal you specify in the file. This filtering out can be for the specified signal in any module definition or in the module definition you specify in the file. You can also specify the FSM and whether the signal is true or false in the file.

`-cm_hier` *filename*

> When compiling for line, condition, toggle, branch, or FSM coverage, this option specifies a configuration file that lists the module definitions, instances and sub-hierarchies, and source files that you want VCS to either exclude from coverage or exclusively compile for coverage.

`-cm_ignorepragmas`

> Tells VCS to ignore pragmas for coverage metrics. When this flag is given, code inside coverage off/on pragma sections will be monitored for coverage.

`-cm_libs yv|celldefine`

> Specifies compiling for coverage source files in Verilog libraries when you include the `yv` argument. Specifies compiling for coverage module definitions that are under the `` `celldefine `` compiler directive when you include the `celldefine` argument. You can specify both arguments together using the plus (+) character.

`-cm_line contassign`

> Specifies enabling line coverage for Verilog continuous assignments.

`-cm_name` *name*

> As a compile-time or runtime option, specifies the name of the intermediate data files.

`-cm_report noinitial`

> Disables the monitoring of the contents of initial blocks for line, condition, branch, and path metrics. See section "Disabling the Monitoring of Initial Blocks"

`-cm_tgl mda`

> Enables toggle coverage for Verilog-2001 multi-dimensional arrays (MDAs) and SystemVerilog unpacked MDAs. Not required for SystemVerilog packed MDAs.

`-cm_tgl signalsort`

> Enables you to sort the signals of toggle coverage in alphabectical order.

`-power=coverage`

> Enables auto-creation of covergroups for low power objects based on the power intent. For more information, see the *MVSIM Native Mode User Guide*.

`-power=dump_hvp`

> When specified along with the `-power=coverage` compile option, dumps the necessary information to the coverage database that enables viewing of low power coverage in URG. Low power coverage view in URG is enabled with the `-lpcov` URG option. For more information, see the *MVSIM Native Mode User Guide*.

Constant Analysis Options

`-cm_constfile` *filename*

> Specifies signal/variable names with the values they are constantly at. The format is as shown in the following example:

```
top.t1.a 1
top.t1.b 4 b1010
```

> VCS reports for line and condition coverage as if these signals were permanently at the specified values. However, it does not really affect the values of signals during simulation.

`-cm_constfile_cont_on_error`

> Downgrades fatal errors encountered during the processing of a constant file, provided with the `-cm_constfile` switch. Normal behavior is to generate an error and stop compilation when non-existent signal names are provided. Should these errors occur during processing of the constant file with the `-cm_constfile_cont_on_error` switch, then the tool prints a warning message and continues. Care should be taken with this switch as unintended results may happen; always review the log file for errors.

`-cm_noconst`

>Enables constant analysis in combinational code, identifies objects for line, condition, and toggle coverage that cannot be hit because of those constant values, and removes them from coverage monitoring. Coverable objects removed this way are shown as "unreachable" in the coverage reports and are not used to compute the coverage scores. For example, a signal with a single continuous assignment of 1'b0 will be removed from monitoring for toggle coverage. For more information, see the *Coverage Technology User Guide*.

`-cm_seqnoconst`

>Enables a more sophisticated constant analysis compared to `-cm_noconst`. This includes analysis of non-blocking assignments and continuous assignments with delays, as well as handling multiple assignments to the same bits of a signal. As with `-cm_noconst`, coverable objects that VCS detects can never be hit are marked "unreachable" in coverage reports and removed from the computation of the coverage score. For more information, see the *Coverage Technology User Guide*.

`-diag noconst`

>When `-cm_noconst` or `-cm_seqnoconst` is enabled, the `-diag noconst` option also enables the creation of a `constfile.txt` file. This file contains, for each detected constant expression, diagnostics listing the location of its declaration, and the locations of all definitions of that expression. For each definition site, the list of other detected constant expressions that are inputs to the expression are listed. For more information, see the *Coverage Technology User Guide.*

Simulation Options for Coverage Metrics

Note:

If you compile VCS with the `-o` option, then the simv.vdb gets renamed to the executable name appended with ".vdb".

The `-cm <metrics>` option is also a runtime option.

`-cm_dir` *directory_path_name*

Specifies an alternative name and location for the `simv.vdb` directory. The `-cm_dir` option is also a compile-time option and a URG command-line option.

`-cm_glitch` *period*

Specifies a glitch period during which VCS does not monitor for coverage caused by value changes. The period is an interval of simulation time specified with a non-negative integer.

`-cm_log` *filename*

As a compile-time or runtime option, specifies a log file for monitoring coverage during compilation or simulation. For example,

```
% simv -cm fsm -cm_log run1.log
```

where, the `-cm_log run1.log` option specifies `run1.log` as the log file and directs the output to the `run1.log` file for monitoring coverage during simulation.

`-cm_name` *name*

>   As a compile-time or runtime option, specifies the name of the intermediate data files.

`-cm_start/-cm_stop N`

>   Specifies starting or stopping of code coverage collection, where N is the time you specify to start or stop the coverage collection. The time unit that you specify with the argument N takes the time unit of the current simulation, by default.

>   For example,

```
% vcs tb_top_mix_vlog -cm line+cond+tgl
% simv -cm line+cond+tgl -cm_start 10 -cm_stop 20
```

`-covg_cont_on_error`

>   If the simulation hits an illegal functional coverage bin it will stop. To advance the simulation bypassing this error, enter `-covg_cont_on_error` runtime option with the `simv` command.

`-covg_disable_cg`

>   Allows disabling all functional coverage items (covergroups). See section "Using `-covg_disable_cg` to Disable Functional Coverage Items" .

`-covg_dump_range`

>   Enables dumping of bins definition to database. You may pass the `-group show_bin_values` (see the URG "Command-Line Options") option to URG to get the bins definitions in the urgReport directory.

```
-covg_no_guard_in_shape
```

The guard conditions specified on illegal/ignore bins do not impact the shape of the covergroup. The bins are unconditionally retained irrespective of the value of the guard expressions. See section "Using `-covg_no_guard_in_shape` to Handle Guards on Illegal/ Ignore Bins"

More Options for Toggle Coverage

Toggle Coverage for MDAs

If your design contains memories, Verilog-2001 multi-dimensional arrays (MDAs), or unpacked SystemVerilog MDAs, and you want toggle coverage for these memories and MDAs, include the `-cm_tgl mda` compile-time option and keyword argument when you compile for toggle coverage. This option and argument are not required for packed SystemVerilog MDAs.

```
% vcs -cm tgl source.v -cm_tgl mda
% simv -cm tgl
% urg -dir simv.vdb -metric tgl
```

Toggle coverage supports the following VHDL data types:

```
bit            bit_vector          std_logic
std_ulogic     std_logic_vector    std_ulogic_vector
signed         unsigned
```

Toggle coverage supports SystemVerilog data types - bit, logic, reg, net etc.

Note:

Array of bit and array of `std_logic` are not supported in VHDL.

Realtime Control of Toggle Coverage

There is a realtime API for Verilog coverage. This API includes the `$cm_coverage` system function that enables you to disable and enable all types of coverage, including toggle coverage. See the section "The $cm_coverage System Function" .

For real-time control of VHDL toggle coverage, you can enter the following commands at the VCS interactive command prompt:

`coverage -tgl off`
    Disables monitoring for toggle coverage.

`coverage -tgl on`
    Enables monitoring for toggle coverage.

Limiting Toggle Coverage to Ports Only

You can instruct VCS to compile and monitor only the ports in your design for toggle coverage, so that the coverage reports that URG writes contain information only about the ports. These reports do not contain information about either the variables and nets that are not ports in your Verilog modules or the variables and signals declared in your VHDL architectures.

Consider the following design:

Example 2-1   source.v

```
module test;
reg r1;
reg [7:0] r2;
wire w1;

    dut dut1 (w1,r1);
```

```
initial
begin
    r1=0;
    r2=8'b00000000;
    #100 $finish;
end

    always
    #10 r1=~r1;

    always
    #25 r2=r2+1;

endmodule

module dut (out,in);
output out;
input in;
reg dutr1;

always @ in
dutr1=in;

assign out=dutr1;
endmodule
```

Use the `portsonly` keyword argument to the `-cm_tgl` compile-time option to monitor only ports, as follows:

```
vcs -cm tgl -cm_tgl portsonly source.v
or
vcs -cm tgl -cm_tgl portsonly cfg
```

The toggle coverage report for ports is as follows:

```
------------------------------------------------------------
Toggle Coverage for Module : dut
                Total Covered Percent
Totals           2      2       100.00
```

```
Total Bits       4    4        100.00
Total Bits 0->1 2    2        100.00
Total Bits 1->0 2    2        100.00


Ports          2 2 100.00
Port Bits      4 4 100.00
Port Bits 0->1 2 2 100.00
Port Bits 1->0 2 2 100.00

Port Details
    Toggle Toggle 1->0 Toggle 0->1 Direction
out Yes    Yes         Yes          OUTPUT
in  Yes    Yes         Yes          INPUT
---------------------------------------------------------
```

Excluding and Including Signals in Toggle Coverage

Compile time exclusion of toggle coverage is specified via the +/-
node option in the `-cm_hier` file.

These entries work in conjunction with other entries in the
configuration file to exclude or include toggle and other types of
coverage, such as the following:

```
+file              -file
+filelist          -filelist
+library           -library
+module            -module
+tree              -tree
```

The `-node` and `+node` entries work for both Verilog and VHDL
signals, nets, and variables.

Excluding a Signal in Toggle Coverage

To exclude a signal, enter the `-node` entry in the `-cm_hier` file, followed by the signal's hierarchical name. For example:

```
+module dev
-node top.dev1.w2
```

In this example, all instances of module `dev` are compiled for coverage. In one such instance, `top.dev1`, signal `w2` is excluded from toggle coverage.

Including a Signal in Toggle Coverage

To include a signal, enter the `+node` entry in the `-cm_hier` file, followed by the signal's hierarchical name. For example:

```
-module dev
+node top.dev1.w2
```

In this example, all instances of module `dev` are excluded from various types of coverage, with one exception. Instance `top.dev1` is compiled for toggle coverage but only for signal `w2`.

Including Part-Selects and Bit-Selects

For Verilog nets and variables, you can include a part-select or a bit-select for a net or variable. For example:

```
-module test
+node test.w1[2:1]
+node test.w2[1]
+node test.l1[2:1]
+node test.l2[1]
```

Note:

> Make sure that there are no spaces between the signal name and the bit or bits of the bit-select or part-select.

Using Wildcard Characters

You can use the asterisk (`*`) and question mark (`?`) wildcard characters in these `-node` and `+node` entries. For example:

```
-module test
+node test.w*
+node test.logic?
```

The asterisk(`*`) can represent multiple characters, while the question mark (`?`) represents a single character.

Excluding and Including Half-Transitions of a Signal

To exclude the half-transitions of a signal, you can enter the `-node` entry with the corresponding half-transition "`0to1`" or "`1to0`" in the `-cm_hier` file followed by the signal's hierarchical name. The syntax is:

```
-node [0to1 or 1to0] Hierarchical_name_of_signal
```

To include the half-transitions of a signal, you can enter the `+node` entry with the corresponding half-transition "`0to1`" or "`1to0`" in the `-cm_hier` file followed by the signal's hierarchical name. The syntax is:

```
+node [0to1 or 1to0] Hierarchical_name_of_signal
```

For example, if the following configuration entries are provided in the `-cm_hier` file, Table 2-1 shows the corresponding coverage report.

```
-node 1to0 top.d2.q1[0] // Half toggle 1->0
-node 0to1 top.d2.q1[3:2] // Half toggle 0->1
-node top.d2.q1[4] // Both the half toggles are excluded.
```

Table 2-1   Signal Details of the top.d2 Instance

	Toggle	Toggle 1->0	Toggle 0->1
q1[0]	Covered	Unreachable	Covered
q1[1]	Not Covered	Covered	Not Covered
q1[2]	Covered	Not Covered	Unreachable
q1[3]	Covered	Covered	Unreachable
q1[4]	Unreachable	Unreachable	Unreachable

Specifying SystemVerilog Structures and Unions

You can specify an instance of a structure or a union, but not a member of the structure or union. For example, refer to the following source code:

```
module test;
typedef struct {
    logic log1;
    bit bit1;
    } control;
control ctl1;
endmodule
```

The following configuration file entry is valid:

```
-module test
+node test.ctl1
```

The following configuration file entry is *not* valid:

```
-module test
+node test.ctl1.log1
```

Sorting Signals in Toggle Coverage

The `-cm_tgl signalsort` compile-time option enables you to sort the signals of toggle coverage in the alphabetical order. The `signalsort` keyword argument can be used for compilation where the `` `define `` compiler directive can lead to different source order of the signals in different tests (based on the value of `` `define `` `MACRO`). In such a case, it is suggested to use `-cm_tgl signalsort` at compile-time as it generates the toggle coverage data for the signals in the same (alphabetical) order for different values of `` `define `` `MACRO`.

For example,

```
output reg [3:0] binary_out, // 4 bit binary Output
`ifdef CFG1
input wire [15:0] binary_in1, // 16-bit Input
input wire [15:0] binary_in2, // 16-bit Input
input wire [15:0] binary_in3, // 16-bit Input
`elsif CFG2
input wire [15:0] binary_in2, // 16-bit Input
input wire [15:0] binary_in3, // 16-bit Input
input wire [15:0] binary_in1, // 16-bit Input
`else
input wire [15:0] binary_in2, // 16-bit Input
input wire [15:0] binary_in1, // 16-bit Input
input wire [15:0] binary_in3, // 16-bit Input
`endif
input wire enable // Enable for the encoder
);
endmodule
```

Use the `signalsort` keyword argument to the `-cm_tgl` compile-time option as follows:

```
% vcs -cm tgl -cm_tgl signalsort +define+CFG1 test.v
% simv -cm tgl -cm_name test1
```

```
% vcs –cm tgl –cm_tgl signalsort +define+CFG2 test.v
% simv –cm tgl –cm_name test2
% vcs –cm tgl –cm_tgl signalsort test.v
% simv –cm tgl –cm_name test3
```

You can now merge `test1`, `test2`, and `test3` tests to generate the correct reports.

Note:

It is recommended not to merge tests generated from different compilations done with and without `–cm_tgl signalsort`.

More Options for Condition Coverage

Modifying Condition Coverage

The `–cm_cond` compile-time option accepts arguments that can add conditions to condition coverage and change the information in the report files. The arguments which you can specify are as follows:

```
full
```
Specifies the following:

- Logical and non-logical conditions — the subexpressions of all operators, not just logical AND `&&` and logical OR `||`, are conditions for condition coverage (see "Enabling Condition Coverage for More Operators").

- Multiple conditions — condition coverage reports show vectors containing multiple condition values. Each vector contains a value for each subexpression of the larger expression (see "Using the `–cm_cond allvectors` Option").

- Event control conditions (*Note: Does not apply to VHDL*) — the signals in the sensitivity list of an always are also conditions for condition coverage.

- Full vectors — the reports show all possible vectors of multiple condition values, not just the sensitized multiple condition value vectors (see "Using Sensitized Multiple Condition Coverage Vectors").

Note:
- In Verilog condition coverage, conditions with more than ten operands result in a warning message. VCS replace the `full` argument with the `allops` argument.

- In VHDL condition coverage, conditions with more than six operands are reported in `std` format of sensitized vectors. VCS displays a warning message when it switches to the `std` format.

`std`

Specifies the following:

- Logical conditions — the subexpressions of just the logical AND `&&` and logical OR `||` operator are conditions for condition coverage.

- Multiple conditions

- Sensitized vectors or sensitized conditions — the only vectors are those where only one subexpression makes the conditional expression true or false, and one more vector to indicate if the conditional expression was either true or false, depending on the operator (see "Using Sensitized Multiple Condition Coverage Vectors").

`basic`

Specifies the following:

- Logical conditions

- No multiple conditions — subexpression values are reported on separate lines (see "Disabling Vector Conditions").

for (*Note: Does not apply to VHDL*)

Enables compiling and monitoring conditional expressions in for loops (see "Enabling Condition Coverage in For Loops").

tf (*Note: Does not apply to VHDL*)

Enables compiling and monitoring conditional expressions in user-defined tasks and functions (see "Enabling Condition Coverage in Tasks and Functions").

event (*Note: Does not apply to VHDL*)

Specifies that the signals in event controls, in the sensitivity list position of an always block, are also conditions for condition coverage (see "Enabling Coverage for Event Controls").

allops

Specifies that the subexpressions of all operators, not just logical AND && and logical OR ||, in conditional expressions are conditions for condition coverage.

ports (*Note: Does not apply to VHDL*)

Enables monitoring conditional expressions in a Verilog module instance port connection list (see "Enabling Condition Coverage in Port Connection Lists").

`allvectors` (*Note: Does not apply to VHDL*)

> Reports all possible vectors for an expression. If this option is not used, then VCS prints only the sensitized vectors (see "Using the `-cm_cond allvectors` Option").

`scalarbitwise`

> Reports sensitized vectors for bitwise expressions, having bitwise operators, and single-bit or one-bit operands. Use `allvectors` argument along with `scalarbitwise` to report all possible vectors for a bitwise expression.

`obs`

> Enables observability based condition coverage. For more information, see the topic "Condition Coverage Observability" on page 43.

You can specify more than one argument. If you do, use the plus delimiter (+) between arguments. For example:

`-cm_cond basic+allops`

Since they each specify mutually exclusive behavior, you cannot use any of the `full`, `std`, `obs`, `allvectors` and `basic` arguments together.

The `std` argument specifies the default settings for condition coverage.

The following sections elaborate on the usage of the `-cm_cond` compile-time option and explain other options that you can use to modify condition coverage.

Enabling Coverage for Event Controls

Note:

This feature does not apply to VHDL.

Verilog event controls can have subexpressions. For example:

```
@ (r1 or r2)
```

The subexpressions in this event control are `r1` and `r2`.

You can use the `event` argument to the `-cm_cond` compile-time option to make the occurrence of the transition specified by these subexpressions into a condition in conditional coverage. For example:

```
vcs verilog/design.vc -cm cond -cm_cond event
```

Where `verilog/design.vc` is an example from the $VCS_HOME/ doc/examples directory.

The event control must meet the following requirements for its subexpressions to become conditions:

- The event control must be in the "sensitivity list" position, immediately following the `always` keyword for an always block.

- The event control expression list must contain more than one signal and also contain the `or` keyword or a comma.

- The event control must be explicit. This feature does not work for Verilog 2001 implicit event controls. For example:

  ```
  always @*
  ```

VCS monitors each of the signals in this list as conditions for condition coverage. The resulting report appears similar to the following:

```
LINE        45
 EXPRESSION (state or attention or full)
            --1--    ----2----    --3-

-1- -2- -3-  Status
 1   -   -   Covered
 -   1   -   Covered
 -   -   1   Covered
```

Enabling Condition Coverage for More Operators

The truth or falsity of subexpressions, that are the operands of the logical AND operator `&&` and the logical OR operator `||`, in conditional expressions and in assignment statements are, by default, conditions for condition coverage. To do this for subexpressions that are the operands of other operators with the `full` or `allops` arguments to the `-cm_cond` compile-time option, use one of the following command lines:

```
% vcs verilog/design.vc -cm cond -cm_cond full
```

or

```
% vcs verilog/design.vc -cm cond -cm_cond allops
```

Where `verilog/design.vc` is an example from the $VCS_HOME/ doc/examples directory.

The URG condition coverage reports are as follows:

With `-cm_cond full`

```
 LINE        52
 EXPRESSION (((!full)) && ((!x_not)) && y_tot)
```

```
                    ----1----    -----2----    --3--

      -1- -2- -3- Status
       0   0   0  Not Covered
       0   0   1  Not Covered
       0   1   0  Not Covered
       0   1   1  Not Covered
       1   0   0  Not Covered
       1   0   1  Not Covered
       1   1   0  Not Covered
       1   1   1  Covered
```

With **-cm_cond allops**

```
 LINE         53
 SUB-EXPRESSION (y_tot ^ ((!x_not)))
                --1--   -----2----

-1- -2- Status
 1   0  Not Covered
 1   1  Covered
```

The subexpressions of operators other that the logical AND `&&` and logical OR `||` are conditions for condition coverage.

Table 2-2 lists the Verilog operators whose subexpression operands you can include in this expanded condition coverage. Table 2-2 lists the VHDL operators whose subexpression operands you can include in this expanded condition coverage.

Operator	Description	Type
==	Logical equality	Binary
!=	Logical inequality	Binary
&	Bit-wise and	Binary
|	Bit-wise inclusive or	Binary
^	Bit-wise xor	Binary
^~ or ~^	Bit-wise xnor	Binary
&	Reduction and	Unary

Operator	Description	Type
~&	Reduction nand	Unary
|	Reduction or	Unary
~|	Reduction nor	Unary
^	Reduction xor	Unary
~^	Reduction xnor	Unary

Table 2-2   Additional VHDL Operators for Condition Coverage

Operator	Description	Type
=	Equality	Binary
/=	Non-equality	Binary
<	Less than	Binary
<=	Less than or equal to	Binary
>	Greater than	Binary
>=	Greater than or equal to	Binary

Enabling Condition Coverage for Allops

By default, VCS condition coverage covers only certain kinds of conditions such as boolean expressions (single bit) and expressions containing more than one term (for example, "if (a && b)" but not "if (a)") or a part of ternary operator ("a <= b ? c : d;");

VCS can monitor an expanded set of conditions using the allops flag. The VCS compile command for allops is as follows:

```
% vcs –cm cond –cm_cond allops mydesign.v
```

It is useful to track the logical combinations of every bitwise operation in your simulation. It can greatly increase the coverage runtime and memory usage overhead. The Table 2-3 shows some examples of the differences between default expression and allops condition coverage monitoring:

Table 2-3   Conditional Coverage Monitoring for allops

Example	Default	allops
if (a && b)	Yes	Yes
a <= b ? c : d;	Yes	Yes
a <= (b == 1'b0);	No	Yes
if (a ^ b)	No	Yes
if(a)	No	Yes
if ((a % 100) == 0)	No	Yes
if(b == 1'b0)	No	Yes
if(a && (b ^ c))	No	Yes
wire [3:0] z; a <= |z;	No	Yes
wire [3:0] y, z, w; assign y <= z & w;	No	Yes

Note that for vector operations, such as the last two rows in Table 2-3, each bit of the expression is monitored separately. For example, for the last continuous assignment of a 4-bit expression, the coverage report is as follows:

```
LINE 21

SUB-EXPRESSION BIT 3 of (z & w)

                          1   2

-1- -2- Status
 0   1   Not Covered
 1   0   Not Covered
 1   1   Not Covered


LINE 21

SUB-EXPRESSION BIT 2 of (z & w)
```

```
                                    1   2

-1- -2- Status
 0   1 Not Covered
 1   0 Not Covered
 1   1 Not Covered


LINE 21

SUB-EXPRESSION BIT 1 of (z & w)

                                    1   2

-1- -2- Status
 0   1 Not Covered
 1   0 Not Covered
 1   1 Not Covered


LINE 21

SUB-EXPRESSION BIT 0 of (z & w)

                                    1   2

-1- -2- Status
 0   1 Not Covered
 1   0 Not Covered
 1   1 Not Covered
```

Depending on the design, it can cause significant additional overhead in simulation time and memory use.

Enabling Condition Coverage in For Loops

By default, VCS does not monitor for conditional expressions in `for` loops. For example:

```
always @ r1
begin:named
integer i;
for (i=0;i<10;i=i+1)
    if(r1 && r2)
      r3=r1;
    else
      r3=r2;
end
```

In this example, VCS does not monitor the conditional expression
(r1 && r2) for condition coverage.

This default behavior occurs because for loops typically exist in
testbenches. You can instruct VCS to compile and monitor for
conditional expressions in for loops with the -cm_cond for
compile-time option. For example:

```
% vcs source.v -cm cond -cm_cond for
```

Note:
    This feature does not apply to VHDL.

Enabling Condition Coverage in Tasks and Functions

By default, VCS does not monitor conditions in user-defined tasks
and functions. For example:

```
module test;
reg r1,r2,r3,r4,r5,r6;
.
.
.
task mytask;
input in;
output out;
reg tr1,tr2,tr3;
```

```
begin
if (r1 && r2)
    out=in;
    else
      if(tr1 || tr2)
        tr3=~in;
end
endtask
.
.
.
endmodule
```

VCS does not monitor the `(r1 && r2)` conditions or the `(tr1 || tr2)` conditions in this task.

You can instruct VCS to compile and monitor these conditions with the `-cm_cond tf` compile-time option and keyword argument. For example:

```
vcs source.v -cm cond -cm_cond tf
```

VCS can monitor conditions in tasks (functions) independent of the operands in the task (function) being declared locally or in the module that contains the task (function).

Enabling Condition Coverage in Port Connection Lists

By default, VCS does not monitor conditional expressions in a port connection list in a Verilog module instantiation statement. For example:

```
module test;
.
.
.
dev d1 (.q(eIpu), .d(eIpurb), .en (rIpu || dIp || wIp),
.clk(cclk));
```

```
        .
        .
        .
endmodule
```

By default, VCS does not monitor the `rIpu`, `dIp`, or `wIp` conditions. You can monitor these expressions with the `ports` keyword argument to the `-cm_cond` compile-time option. For example:

```
% vcs exp1.v -cm cond -cm_cond ports
```

The URG condition coverage reports with `-cm_cond ports` option is as follows:

```
LINE        11
EXPRESSION (rIpu || dIp || wIp)
            --1-   -2-    -3-

-1- -2- -3- Status
 0   0   0  Covered
 0   0   1  Not Covered
 0   1   0  Not Covered
 1   0   0  Not Covered
```

Using Sensitized Multiple Condition Coverage Vectors

By default, VCS compile and monitor for sensitized multiple condition coverage vectors.

Sensitized multiple condition coverage vectors enable you to see if the execution of a block of code is sensitive to all the subexpressions in a conditional expression. Consider the following Verilog `if` statement:

```
   if (in1 && in2 && && in3 && in4)
     begin
      .
      .
```

```
        .
      end
```

VCS does not execute the begin-end block if any of the signals `in1`, `in2`, `in3` or `in4` are false. Because the operator in this expression is the logical AND operator `&&`, you would want to determine if:

- During simulation, each of these signals were the only instance that was false, and thus the sole cause of not executing the block.

- All of the signals were true, and therefore the block did execute.

```
if (in1 and in2 and in3 and in4) then
      .....
      end if;
```

The signals `in1`, `in2`, `in3` or `in4` are expected to be of type Boolean only.

To determine if the block was sensitive to all four signals, you can use sensitized condition coverage. With this coverage, VCS look for the following patterns or vectors of values for these signals.

in1	in2	in3	in4
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0
1	1	1	1

These patterns become conditions in condition coverage, therefore instead of looking for eight conditions, that is the truth or falsity of all four signals, VCS look for five conditions, one condition where all of the subexpression signals are true, and four conditions where only one of them is false.

Alternatively, if the operator in this expression were the logical OR operator | |, you would want to determine if:

- During simulation, each of these signals was the only one that was true, and the sole cause of executing the block.

- All of the signals were false and, therefore the block did not execute.

When URG reports condition coverage, it duplicates these vectors of values and informs you of whether or not these vectors were covered.

You can enable sensitized condition coverage with the std argument to the -cm_cond compile-time option:

```
% vcs source.v -cm cond -cm_cond std
```

or

```
% vcs cfg -cm cond -cm_cond std
```

However, you do not need to explicitly enable it. By default, VCS generate sensitized condition coverage. If you specify the full or basic arguments, you will not see sensitized condition coverage.

The URG report with the -cm_cond std option is as follows:

```
LINE        32
EXPRESSION (q && rIpu && dIp && wIp)
           -1-   --2-    -3-     -4-

-1- -2- -3- -4- Status
 0   1   1   1  Covered
 1   0   1   1  Not Covered
 1   1   0   1  Not Covered
 1   1   1   0  Not Covered
```

```
1   1   1   1  Covered
```

VHDL Data Type Difference

The VHDL LRM specifies that if the Boolean data type is the operand to the right of the AND, OR, NAND, and NOR operators, that operand is evaluated if the evaluation of the left operand does not determine the value of the expression.

Therefore, for the expression:

```
A and B and C and D
```

If A and B has the Boolean data type, the sensitized vectors are as follows:

0	-	-	-	The leftmost operand determines the value of the expression, which is 0. Therefore, VCS does not evaluate the operands to the right.
1	0	-	-	The leftmost operand enables the evaluation of the first right operand. The first right operand makes the expression 0. Therefore, VCS does not evaluate the other right operands.
1	1	1	0	The previous three left operands enable the evaluation of the last right operand. The last right operand makes the expression 0.
1	1	1	1	All the operands make the expression 1.

Using the `-cm_cond allvectors` Option

Note:

- This feature does not apply to VHDL.

- `-cm_cond full` is a subset of `-cm_cond allvectors` option.

By default, VCS monitors only the sensitized vectors for any condition expressions in your design.

source.v

```
module test;
    reg a, b;
    wire c;
    reg [3:0] e,f;
    wire [3:0] g;
    assign c = a || b;
    assign g = e |f;
endmodule
```

For example, if you compile the above example with the `-cm cond` option, VCS reports only the sensitized vectors for the expression "`assign c = a || b`", as follows:

```
LINE        6
EXPRESSION (a || b)
            1    2

-1- -2- Status
 0   0  Not Covered
 0   1  Not Covered
 1   0  Not Covered
```

To view all possible vectors for the expression "`assign c = a || b`", use the compile-time option `-cm_cond allvectors` with `-cm cond`, as follows:

```
% vcs source.v -cm cond -cm_cond allvectors
```

or

```
% vcs -cm cond -cm_cond allvectors cfg
```

The report generated with the `-cm_cond allvectors` option is as follows:

```
LINE        6
EXPRESSION (a || b)
            1    2

-1- -2- Status
 0   0  Not Covered
 0   1  Not Covered
 1   0  Not Covered
 1   1  Not Covered
```

Disabling Vector Conditions

If you do not want VCS to look for vectors of conditions, or URG to report them, you can specify the `basic` argument to the `-cm_cond` compile-time option:

```
% vcs source.v -cm cond -cm_cond basic
```

or

```
% vcs -cm cond -cm_cond basic cfg
```

The following example is the same example as shown in the sections, "Using Sensitized Multiple Condition Coverage Vectors" and "Using the `-cm_cond allvectors` Option" :

```
if (in1 && in2 && in3 && in4)
  begin
  .
  .
  .
  end
```

In this example, if you specify the `basic` argument to the `-cm_cond` compile-time option, the URG report appears as follows:

```
LINE        18
EXPRESSION (in1 && in2 && in3 && in4)
            -1-     -2-     -3-     -4-


-1- -2- -3- -4- Status
 0   -   -   -  Covered
 1   -   -   -  Covered
 -   0   -   -  Not Covered
 -   1   -   -  Covered
 -   -   0   -  Not Covered
 -   -   1   -  Covered
 -   -   -   0  Not Covered
 -   -   -   1  Not Covered
```

Note:

The `basic` argument also specifies that only the subexpressions of the logical AND operator `&&` and the logical OR operator `||`, are conditions, therefore, if you use this argument and want other conditions, also include the `allops` argument.

```
if (in1 AND in2 AND in3 AND in4)
   .
   .
   .
```

Note:

The `basic` argument also specifies that only the subexpressions of the logical AND operator `&&` and the logical OR operator `||`, are conditions, therefore, if you use this argument and want other conditions, also include the `allops` argument.

The URG report is different for the Boolean data type as follows:

```
LINE 6
```

```
EXPRESSION (in1 and in2 and in3 and in4)
           -1-       -2-       -3-       -4-

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

This is different because the VHDL language specifies that VCS cannot evaluate a Boolean operand to the right of the logical AND operator `&&` unless the evaluation of the Boolean operator to the left of the AND operator is evaluated first.

Excluded Subexpressions

The truth or falsity of subexpressions with explicit X or Z value operands cannot be conditions for condition coverage.

Note:
   This does not apply to VHDL.

For example, in Verilog:

```
if(a == 'bx) && (b == 'bz))
```

The truth or falsity of subexpression operands of the operators in Table 2-5 are not conditions in condition coverage even when you include the `full` or `allops` arguments to the `-cm_cond` compile-time option.

Operator	Description	Type
===	Case equality	Binary
!==	Case inequality	Binary
~	Bit-wise negation	Unary

Operator	Description	Type
!	Logical negation	Unary

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.

Condition Coverage Observability

The observability feature is used to find out the situations where a certain part of the combinational circuit is controlling (and hence observable) the primary output of the combinational circuit. A vector combination of a subexpression is marked as covered only if that combination is controlling the primary output in the context of the whole expression.

You can enable the observability based condition coverage using the `-cm cond -cm_cond obs` compile option.

Note:

Observability-based report is only for sensitized vectors. The obs sub-option cannot be used with `-cm_cond basic`, `full`, or `allvectors`.

Currently, observability supports pure Verilog/SystemVerilog. In case of MX, VCS generates a default condition coverage report for the VHDL side and an observability-based condition coverage report on the Verilog side.

Usage Model

Following is the syntax of the `-cm cond -cm_cond obs` switch:

```
% vcs -sverilog -cm cond -cm_cond obs observability_example.v
% simv -cm cond
% urg -format text -dir simv.vdb
```

where, `observability_example.v` contains the following code:

```
module cond_simple (a,b,c,d,out);
input a,b,c,d;
output out;
reg a,b,c,d;
reg out;
always@*
begin
 if((a && b)||(c && d))// Conditional expression
     begin
       $display ("AM at if loop \n");
     end
 else
    $display ("AM at else part of if loop \n");
end

initial
    begin
         #5 a=0; b=1; c=1; d=1;
    end
```

```
initial
    #10 $finish();

endmodule
```

The generated URG report in html format is as follows:

```
LINE        8
EXPRESSION ((a && b) || (c && d))
            ----1---     ----2---
```

-1-	-2-	Status
0	0	Not Covered
0	1	Covered
1	0	Not Covered

```
LINE        8
SUB-EXPRESSION (a && b)
                1    2
Dependencies:
(c && d) = 0
```

-1-	-2-	Status
0	1	Not Covered
1	0	Not Covered
1	1	Not Covered

```
LINE        8
SUB-EXPRESSION (c && d)
                1    2
Dependencies:
(a && b) = 0
```

-1-	-2-	Status
0	1	Not Covered
1	0	Not Covered
1	1	Covered

A list is added after the condition expression to show the dependencies.

In this example, for test vector combination {A, B, C, D} = {0, 1, 1, 1} (A && B) = 0 and (C && D) = 1. So, (C && D) is observable at the primary output, not (A && B).

"01" of table for "EXPRESSION ((a && b) || (c && d))" and "11" of table for "SUB-EXPRESSION (c && d)" are marked as covered. But, "01" of table for "SUB-EXPRESSION (a && b)" is not marked, as it represents the expression (a && b) which is not observable in this case.

Basic Rules of Determining Observability

The five gates AND, OR, NOT, XOR, and XNOR are treated as the basic gates for the observability analysis.

The observability rules for each of them are as follows:

- OR — If all the inputs are 1, then nothing is observable. This scenario is not considered. If any of the inputs is 0, it is not observable (unless the other input is also 0), but the other input will be observable. If both the inputs are 0, then both are observable.

- AND — If all the inputs are 0, then nothing is observable. This scenario is not considered. If any of the inputs is 1, it is not observable (unless the other input is also 1), but the other input will be observable. If both the inputs are 1, then both are observable.

    NOT, XOR, XNOR — Each of the inputs will always be observable.

Specifying Continuous Assignment Coverage

The conditional expressions in Verilog continuous assignments are conditions for condition coverage. For example, if a module definition contains the following continuous assignment:

```
assign w1 = r1 ? (r2 ? (r3 ? 2'b11 : 2'b01) : 2'b10) : 2'b00;
```

Each of the conditional expressions, in this case signals r1, r2, and r3, are conditional expressions for condition coverage, and URG reports them as follows:

```
LINE         24
EXPRESSION (in1 ? (in2 ? (in3 ? 2'b11 : 2'b1) : 2'b10) : 2'b0)
            -1-

-1- Status
 0  Covered
 1  Covered

LINE         24
SUB-EXPRESSION (in2 ? (in3 ? 2'b11 : 2'b1) : 2'b10)
                  -1-

-1- Status
 0  Not Covered
 1  Not Covered

LINE         24
SUB-EXPRESSION (in3 ? 2'b11 : 2'b1)
                  -1-

-1- Status
 0  Not Covered
 1  Covered
```

Displaying Condition IDs

You can view the condition IDs in the condition coverage report using
the -cond ids option.

```
% urg -dir <simv.vdb> -cond ids
```

The following URG report displays the condition IDs:

```
LINE        23
 CONDITION_ID   5
 EXPRESSION (((a && b) || (c && d)) && (e || f))
            ----------1----------    ----2---

    ID -1- -2- Status
     1  0   1  Not Covered
     2  1   0  Not Covered
     3  1   1  Covered

 LINE        23
 CONDITION_ID   4
 EXPRESSION ((a && b) || (c && d))
               ----1---     ----2---

    ID -1- -2- Status
     1  0   0  Not Covered
     2  0   1  Not Covered
     3  1   0  Not Covered

 LINE        23
 CONDITION_ID   3
 EXPRESSION (a && b)
             1     2

    ID -1- -2- Status
     1  0   1  Excluded
     2  1   0  Not Covered
     3  1   1  Covered
```

```
LINE          23
CONDITION_ID   2
EXPRESSION (c && d)
              1    2

    ID -1- -2- Status
     1  0   1  Not Covered
     2  1   0  Not Covered
     3  1   1  Covered

LINE          23
CONDITION_ID   1
EXPRESSION (e || f)
              1    2

    ID -1- -2- Status
     1  0   0  Not Covered
     2  0   1  Not Covered
     3  1   0  Not Covered
```

Using Multiple Condition Value Vectors With Constant Filtering

When you use constant filtering, condition coverage reports do not include condition value vectors in which one value in the vector cannot be covered. The vectors it does display are not sensitized, neither are all possible vectors, except those filtered out by the `-cm_noconst` compile-time option.

URG report without the `-cm_noconst <const_file>` option

```
LINE          52
EXPRESSION (((!full)) && ((!x_not)) && y_tot)
            ----1----    -----2----    --3--

-1- -2- -3- Status
0   1   1   Not Covered
```

```
1    0    1    Not Covered
1    1    0    Not Covered
1    1    1    Covered
```

In the following example, a certain condition, net `F` having a true value, cannot be covered. If you also use the `-cm_constfile` file to specify that signal `test.D` is permanently at 0, the report file would contain the following in the case of Verilog coverage:

```
module test;
 reg A,B,C,D,E;
 wire F;

 assign F = 1'b0;

 always @(A or B)
 begin
 C = (A && B && D);
 E = (C || F);

if (!D) C = 0;
 else C = (A || B);
 if (!F) E = 0;
 else E = 1;
 end

endmodule
```

URG report with the `-cm_constfile` option

```
LINE        9
EXPRESSION (A && B && D)
           1    2    3

-1- -2- -3- Status
 0   1   1  Unreachable
 1   0   1  Unreachable
 1   1   0  Not Covered
 1   1   1  Unreachable
```

For line 9, there are no vectors where signal D has a value of 1. Neither is there a vector where all the subexpressions have a value of 0, therefore, you are not seeing all possible vectors, only those that are filtered out. The three vectors for line 9 are the same vectors you would see if you replaced D in line 9 with its permanent value, 0.

Similarly for line 10, there are no vectors where F has a value of 1. The vectors are those you would see if you replaced F in line 10 with 0.

Omitting Coverage for Default Case Items

Typical usage of default case items in case statements is to tell VCS what to do if the simulation does something that you do not expect, that is, if it does not find a case item expression that matches the case expression. For example:

```
case (r1)
1'b0 : r2=0;
1'b1 : r2=1;
default : begin
            if (f1 || f2)
                $display("one flag true");
            else
                $display("no flag true");
            $stop;
          end
endcase
```

In this example, the default case item displays information about the current state of the design and then halts simulation for debugging. It is not part of how the design works.

If you use default case items in this way, you might not want URG to report line coverage information about default case statements when they are not executed. You do not want to see a lower percentage of coverage, just because VCS did not execute the default case statement.

In this case, you can specify URG not to report line coverage information when default case statements are not executed with the `-line nocasedef` option on the URG command line. For example:

```
% urg -line nocasedef
```

Note:

> This option does not stop URG from reporting toggle or FSM coverage information about default case items. It also does not stop URG from reporting line coverage information when VCS executes default case items. To completely exclude the code inside a default case item, use the `VCS coverage off and on` pragmas.

More Options for Branch Coverage

For Loops and User-defined Tasks and Functions

If the `if` and `case` statements and the ternary operator (`?:`) are in user-defined tasks or functions, or in code that executes as a result of a `for` loop, by default VCS does not monitor them for branch coverage. This is also true for condition coverage.

If you compile your Verilog design for condition coverage and enter the `-cm_cond` compile-time option with the `for` and `tf` keyword arguments, which tell VCS to compile the design for condition

coverage in `for` loops and user-defined tasks and functions, VCS will also compile the design for branch coverage in `for` loops and user-defined tasks and functions. For example:

```
% vcs -cm cond+branch -f verilog/design.vc -cm_cond tf \
-cm_cond for

% simv -cm cond+branch
```

Limitations

The following compile-time options for coverage metrics do not work with branch coverage:

`-cm_count` - Adds an execution count to reports.

More Options for FSM Coverage

Coding a Verilog FSM

There are a wide range of coding styles for FSMs. The FSM can consist of a continuous assignment of the next state value to a net using conditional operators along with procedural assignment statements in an always block to transfer (assign) the next state value of the net to the reg that holds the current state. More typically the FSM consists of procedural statements inside an always block including procedural assignments to one or more regs of current state and next state values controlled by `case,` `while,` or `if-else` statements.

This section includes various groups of statements which VCS/VCS MX automatically extracts as an FSM.

Using the Encoded FSM Style

The encoded FSM style does not require that:

- Only one bit, in the vector reg that contains the current state, be true

- You use the entire bit width of the reg for the state

In , the FSM has four possible states:

- NO_ONES

- ONE_ONE

- TWO_ONES

- AT_LEAST_THREE_ONES

Each state is delineated in a parameter declaration.

Example 2-2   FSM with States Delineated in a Parameter Declaration

```
module enum2_V(signal, clock, detect);
input signal, clock;
output detect;
reg detect;

parameter [1:0]
    NO_ONES = 2'h0,
    ONE_ONE = 2'h1,
    TWO_ONES = 2'h2,
    AT_LEAST_THREE_ONES = 2'h3;

// Declare current state and next state variables.
reg [1:0] cs, ns;

always @ (cs or signal)
  begin
     detect = 0;// default value
```

```
        if (signal == 0)
            ns = NO_ONES;
          else
           case (cs)
            NO_ONES: ns = ONE_ONE;
            ONE_ONE: ns = TWO_ONES;
            TWO_ONES,
            AT_LEAST_THREE_ONES:
               begin
                 ns = AT_LEAST_THREE_ONES;
                 detect = 1;
               end
            default: ns = NO_ONES;
           endcase
    end

always @ (posedge clock) begin
     cs = ns;
end

endmodule
```

In Example 2-2, the case expression is the reg that holds the current state of the FSM. The case item expressions are states of the FSM. When the current state of the FSM is that in the case item expression the case item statements specify the next state of the FSM.

You can use signed values for the states of an FSM, as shown in Example 2-3:

Example 2-3   FSM With Signed Values

```
module finite_state (clk, in, state);

input clk, in;
output state;

reg signed [1:0] state, next;

parameter signed idle=2'sb00,    // 0
```

```
                     first=2'sb01,  // +1
                     second=2'sb10, // -2
                     third=2'sb11;  // -1

    initial begin
     state=idle;
     next=idle;
     end

    always @ in
     begin
      next=state;
      case (state)
        idle  :  if(in) next=first;
        first :  if(in) next=second;
        second:  if(in) next=third;
        third :  if(in) next=idle;
      endcase
     end

    always @ (posedge clk)
      state=next;
    endmodule
```

In the FSM in Example 2-4, the four possible states are A, B, C and D. They are delineated in text macros specified in `define compiler directives. In these `define compiler directives, the macro text that VCS/VCS MX substitutes for the macro names A, B, C and D are numeric constants 2'b00, 2'b01, 2'b10 and 2'b11. The `define compiler directive text macros must substitute a numeric constant for the macro name.

Example 2-4   FSM With States Delineated with Text Macros

```
`define A 2'b00
`define B 2'b01
`define C 2'b10
`define D 2'b11
```

```verilog
module counter_top_fsm(clock, reset, count,
mode, countA, countB);

input clock, reset;
input [3:0] countA, countB;
output [3:0] count;
output [1:0] mode;

reg [3:0] count;
reg [1:0] mode,mode_next;
reg [1:0] top_state, top_state_next;

always @ top_state
begin
case (top_state)
`A :  begin
       top_state_next = `B ;
       mode_next = 1;
       end
`B :  begin
          top_state_next = `C;
          mode_next = 2;
      end
`C :  begin
          top_state_next = `D;
          mode_next = 3;
      end
`D :  begin
          top_state_next = `A;
          mode_next = 0;
      end
endcase
end



always @(posedge clock)
begin
    if (!reset)
      begin
        count = 4'b0000;
        mode =  0;
```

```
            top_state_next = `A;
            top_state = `A;
          end
      else
        begin
          mode = mode_next;
          top_state = top_state_next;
          count = (mode == 0)? countA : countB;
        end
    end
endmodule
```

Example 2-5 shows an FSM that has no reg or wire to hold the next state of the FSM. VCS/VCS MX can still identify and extract this FSM.

Example 2-5   FSM With No Next State Signal

```
module no_NS (en,clock,state);
input en,clock;
output [1:0] state;
reg [1:0]  state;

initial
state=0;

always @ (posedge clock)
case (state)
  0: if (en == 1)  state = 1;
  1: state = en == 0 ? 0: 2;
  2: state = en == 0 ? 1 : 0;
endcase

endmodule
```

In Example 2-5 the clock controls when the next state is assigned to the state reg. The group of statements uses an expression that uses the conditional operator to assign the next state to the state reg.

VCS/VCS MX can identify and extract an FSM even when the state value propagates from the next state signal to the current state signal through another module instance as shown in Example 2-6.

Note:
    This feature does not apply to VHDL.

Example 2-6   State Value Propagating Through Another Instance

```
module fsm_mod (in);
input [1:0] in;
parameter
start=2'b11,step1=2'b10,step2=2'b01,finish=2'b00;

wire [1:0] current;

reg [1:0] next;

always @ in
begin
case(current)
start  : next = step1;
step1  : next = step2;
step2  : next = finish;
finish : next = start;
default: next = step1;
endcase
end

connector ct1 (current,next);
endmodule

module connector (out,in);
output [1:0] out;
input [1:0] in;
reg [1:0] out;

always @ in
#5 out = in;

endmodule
```

VCS/VCS MX looks for a way for the state value to propagate from the next state signal to the current state signal. In this example, VCS/VCS MX determines that these two signals both connect to an instance of the connector module, and in this module exists a direct connection of the input and output ports. Therefore, VCS/VCS MX extracts the FSM in module fsm_mod.

VCS/VCS MX does not search for this propagation path through intermediate signals. If the connector module was defined as follows:

```
module connector (out,in);
output [1:0] out;
input [1:0] in;
reg [1:0] r1;

assign out = r1;

always @ in
#5 r1 = in;

endmodule
```

VCS/VCS MX does not search for the path in →r1 →out and, therefore, VCS/VCS MX does not automatically extract the FSM.

You can use a configuration file or pragmas to instruct VCS/VCS MX to extract the FSM (see"Using an FSM Configuration File" and "Coverage Pragmas").

Implementing Hot Bit or One Hot FSMs

Hot bit or One Hot FSMs are usually implemented with case statements. The case expression is `1'b1`, and the case item expression is a specific bit of the reg that holds the current state of the FSM.

Note:

Hot bit and One Hot FSMs are not applicable to VHDL.

Example 2-7   Hot Bit FSM

```
module prep3(clk, rst, in, out);
    input clk, rst;
    input [7:0] in;
    output [7:0] out;

    parameter [2:0]
    START = 0 ,
    SA = 1 ,
    SB = 2 ,
    SC = 3 ,
    SD = 4 ,
    SE = 5 ,
    SF = 6 ,
    SG = 7 ;

reg [7:0] state;
reg [7:0] next_state;
reg [7:0] out, next_out;
always @ (in or state) begin
    // default values
    next_state = 8'b0;
    next_out = 8'bx;

    case (1'b1)
    state[START]:
        if (in == 8'h3c) begin
            next_state[SA] = 1'b1 ;
```

```
                next_out = 8'h82 ;
                end
            else begin
                next_state[START] = 1'b1 ;
                next_out = 8'h00 ;
                end
        state[SA]:
            case (in)
            8'h2a:
                begin
                next_state[SC] = 1'b1 ;
                next_out = 8'h40 ;
                end
            8'h1f:
                begin
                next_state[SB] = 1'b1 ;
                next_out = 8'h20 ;
                end
            default:
                begin
                next_state[SA] = 1'b1 ;
                next_out = 8'h04 ;
                end
                endcase

        state[SB]:
            if (in == 8'haa) begin
                next_state[SE] = 1'b1 ;
                next_out = 8'h11 ;
                end
            else begin
                next_state[SF] = 1'b1 ;
                next_out = 8'h30 ;
                end
        state[SC]:
            begin
            next_state[SD] = 1'b1 ;
            next_out = 8'h08 ;
            end
        state[SD]:
            begin
            next_state[SG] = 1'b1 ;
```

```
            next_out = 8'h80 ;
            end
        state[SE]:
            begin
            next_state[START] = 1'b1 ;
            next_out = 8'h40 ;
            end
        state[SF]:
            begin
            next_state[SG] = 1'b1 ;
            next_out = 8'h02 ;
            end
        state[SG]:
            begin
            next_state[START] = 1'b1 ;
            next_out = 8'h01 ;
            end
        endcase
end

// build the state flip-flops
always @ (posedge clk or negedge rst)

begin
    if (!rst) begin
        state <= #1 8'b0 ;
        state[START] <= #2 1'b1 ;
    end
    else
        state <= #1 next_state ;
    end

// build the output flip-flops
always @ (posedge clk or negedge rst)
    begin
    if (!rst) out <= #1 8'b0 ;
    else out <= #1 next_out ;
end

endmodule
```

Example 2-7 contains case statements, one nested inside the other. The outer case statement is what makes this FSM a hot bit FSM. In the outer case statement:

- The case expression is `1'b1`.

- The case item expressions are `state[START]`, `state[SA]`, `state[SB]`, `state[SC]`, `state[SD]`, `state[SE]`, `state[SF]`, and `state[SG]`. These expressions specify specific bits of a reg named `state` that holds the current state of the FSM. The parameter declaration in Example 2-7 specifies parameters for bit numbers of that reg. More typically parameter declarations specify the states of the FSM.

- The case item statements are begin-end blocks of statements, including the nested case statement that control the assignment of the next state of the FSM to the reg named `next_state` that holds this next state.

The nature of the hot bit FSM is that only one bit of the reg that holds the current state of the FSM can be true, and in this example, that is also true of the reg that holds the next state.

Using Continuous Assignments for FSMs

Not all FSMs consist entirely of procedural statements in always blocks. In this example, the next state signal is a wire and the next state is assigned using a continuous assignment statement. An always block changes the current state contained in a reg.

Example 2-8   Continuous Assignment Statement FSM

```
module M (x,clock);

input x,clock;
reg [1:0] state;
```

```
wire [1:0] next;

always @ (posedge clock)
state = next;

assign next = state == 0 ?
                (x == 1 ? 1 : 0) :
                state == 1 ?
                    (x == 0 ? 0 : 2) :
                    state == 2 ?
                        (x == 0 ? 1 : 0) :
                        2'b00;
endmodule
```

In Example 2-8, the states are `0`, `1`, and `2`. The reg named `state` holds the current state and the wire named `next` holds the next state.

Avoiding Substituting the Same Numeric Constant

When coding an FSM, avoid substituting the same numeric constant for more than one macro name for a state in multiple `` `define `` compiler directives, otherwise, this can cause VCS to confuse one state for another. For example:

```
`define first_state 0
`define prime_state 0
```

Sequence Coverage

You can instruct VCS to compile for and monitor sequence coverage with the `-cm_fsmopt sequence` compile-time option and keyword argument.

With sequence coverage, you not only see the states that were covered, but the sequences of states that were covered during simulation.

Controlling How VCS Extract FSMs

When VCS compiles your design for FSM coverage, it extract FSMs from your source code. Extracting FSMs means identifying a group of statements to be an FSM, so that VCS is ready to keep track of the states and transitions that occur in the FSM during simulation.

By default, VCS automatically extract all the FSMs that it can identify in all the module definitions and VHDL architectures in your design.

When you compile the design with the `-cm fsm` switch, it extracts all the FSMs that adheres to FSM coding guidelines. When you give the `-cm_fsmopt excludeCalcFsms` switch with `-cm fsm`, then it rejects the FSMs with one of the following statements, where cs = current state and ns = next state:

```
cs = Expr(cs) : Expr(cs) contains arithmetic;
      (following Expr is same)
cs = Expr(ns) ;
cs = Expr(foo);  : foo is a Net or Reg
ns = Expr(cs);
ns = Expr(ns);
ns = Expr(foo);
```

For example if the FSM contains the following line, then it will not be extracted.

ns = ns +1;

However, you can limit the extraction of FSMs to a part or parts of the design hierarchy that you specify with the `-cm_hier` compile-time option.

You can also specify the FSMs in a module definition by using one of the following:

- An FSM configuration file

- Pragmas in your code (does not apply to VHDL)

Using an FSM Configuration File

A configuration file enables you to specify:

- The FSMs that VCS extracts from a module or entity definition

- Which states and which transitions between states that VCS keeps track of in the FSMs

- The maximum number of sequences that VCS keeps track of in any of the modules or design entities in your design, and specify the maximum length of any sequence that VCS keeps track of (see "Specifying the Maximum for Sequences")

When VCS compiles your Verilog design for coverage, it creates the `simv.vdb` directory, and the `snps/coverage/db` directory in the `simv.vdb` directory. If you compile your design for FSM coverage, VCS writes the `fsm.verilog.generated_config.txt` file in the `simv.vdb/snps/coverage/db/shape` directory. You can either use this file, editing it to suit your needs, or start a new file as your configuration file.

When VCS compiles your VHDL design, it creates the `simv.vdb/`
`snps/coverage/db/shape` directory and writes the
`fsm.vhdl.generated_config.txt` file in this directory.

You write a separate section in the configuration file for each module
definition or design entity definition for which you want to specify the
FSMs that VCS extracts.

VCS expects the entries in the FSM configuration file to be written in
a particular order. It expects the design level entries at the top,
followed by module level entries, and then the FSM level entries at
the bottom.

The syntax of a configuration file section is as follows:

```
SEQ_NUMBER_MAX = integer
SEQ_LENGTH_MAX = integer
MODULE = module_identifier
FSMS = RESTRICT | EXCLUDE
FSMS = START_STATE_DFLT
CURRENT  = reg_identifier
NEXT = net_or_reg_identifier
STATES = list_of_states
STATES_X = list_of_states
STATES_NEVER = list_of_states
START_STATE = state
TRANSITIONS = list_of_transitions
TRANSITIONS_X = list_of_transitions
TRANSITIONS_NEVER = list_of_transitions
SEQ_REQUIRE = pattern
SEQ_EXCLUDE = pattern
```

The following tables describe the FSM configuration file entries:

Design Level Entries	Description
`SEQ_NUMBER_MAX = integer`	Specifies the maximum number of sequences in any module or design entity that VCS maintains. The integer value must be non-negative.

Module Level Entries	Description
`SEQ_LENGTH_MAX = ` *`integer`*	Specifies the length of the longest sequence that VCS maintains. The integer value must be non-negative.

Module Level Entries	**Description**
Verilog: `MODULE = ` *`module_identifier`*	Specifies a module definition. VCS extracts FSMs from all instances of this module definition. This line is always required in a section.
VHDL: `MODULE = ` *`E`* or `MODULE = ` *`lib.E`*	For VHDL designs, specify the entity name as `MODULE = Ent_name,` or as `MODULE = Library_name.Ent_name.`
`FSMS = RESTRICT`	Specifies that VCS only extracts the FSMs specified in a line that begins with the keyword `CURRENT`. This line is optional and specifies the default condition.
`FSMS = EXCLUDE`	Specifies that VCS extracts all the FSMs in the module or design entity definition except those specified in a line that begins with the keyword `CURRENT`. This line is optional.
`FSMS = START_STATE_DFLT`	Specifies that VCS only maintains the sequences that begin with the state that has the lowest value. This line is also optional. You can enter an `FSMS = RESTRICT` or `FSMS = EXCLUDE` line along with an `FSMS = START_STATE_DFLT` line.

FSM Level Entries	**Description**
`CURRENT  = ` *`reg_identifier`*	Specifies the Verilog variable or VHDL signal that holds the current state of the FSM. This line is always required in a section. If you want to restrict extraction to, or exclude extraction from, more than one FSM in the module or design entity definition, enter this line for each FSM. (continued)

`NEXT = net_or_reg_identifier`	Specifies the wire or reg that holds the next state of the FSM. This line is required if the FSM has a reg or wire that holds the next state of the FSM. Like with the `CURRENT` line, if you want to restrict extraction to, or exclude extraction from, more than one FSM in the module or design entity definition, enter this line for each FSM.
`STATES = list_of_states`	Specifies a list of states, or the values of states, separated by commas. Specifying these states ensures that VCS keeps track of transitions to these states. This list is not restrictive; if the FSM transitions to other states VCS also keeps track of these transitions and reports these unlisted states by value. This line is required in a section.
`STATES_X = list_of_states`	Specifies a list of states, or the values of states, that you want VCS to ignore. Separate the states by commas. VCS does not keep track of these states, transitions to and from these states, or sequences that include these states. This line is optional.
`STATES_NEVER = list_of_states`	Specifies a list of states, or the values of states, that you want VCS to report as `ILLEGAL` in the report files if there is a transition or sequence involving these states or values. This line is also optional.
`START_STATE = state`	Specifies a start state for the FSM such as a reset state. This line is optional. When you include it VCS only keeps track of sequences that begin and end with this start state.
`TRANSITIONS = list_of_transitions`	Specifies a list of transitions, separated by commas. You can specify state names or state values. Like the `STATES` line, specifying these transitions ensures that VCS keeps track of them. This list is not restrictive; if the FSM makes other transitions, VCS also keeps track of them. This line is optional.
	(continued)
`TRANSITIONS_X = list_of_transitions`	Specifies a list of transitions that you want VCS to ignore. You can specify transitions by state names or state values. Separate the transitions by commas. This line is optional.

`TRANSITIONS_NEVER =` `list_of_transitions`	Specifies a list of transitions that you want VCS to report as `ILLEGAL` in the report files if there is a transition or sequence involving these states or values. You can specify transitions by state names or state values. Separate the transitions by commas. This line is also optional.
`SEQ_REQUIRE = pattern`	Specifies a pattern of transitions. URG reports only sequences that contain the pattern, see "Sequence Filtering in Reports" .
`SEQ_EXCLUDE = pattern`	Specifies a pattern of transitions. URG does not report sequences that contain the pattern, see "Sequence Filtering in Reports" .

As shown in Example 2-9 and Example 2-10, if you have more states or transitions than you want on a line, you can enter a line break and enter more states or transitions on the following line.

Example 2-9   Verilog Configuration File

```
MODULE = fsmmod

CURRENT = cs
NEXT = ns
STATES = ZERO,ONE,TWO,THREE,FOUR,FIVE
START_STATE = ZERO
TRANSITIONS = ZERO->ONE, ONE->TWO, TWO->THREE,
THREE->FOUR, FOUR->FIVE, FIVE->ZERO

CURRENT = current_state
NEXT = next_state
STATES = step1, step2, step3
```

Example 2-10   VHDL Configuration File

```
MODULE=DEFAULT.E A
CURRENT=CS
NEXT=NS
STATES=s0,s1,s3
TRANSITIONS=s0->s1,
s1->s0,
s1->s3,
```

```
s3->s0,
s3->s1
```

The TRANSITIONS Line

The `TRANSITIONS` line is optional. The syntax for the transition line is as follows:

```
TRANSITIONS = state -> state, ...
```

The `TRANSITIONS` line can list one or more transitions from a state, to a state, with these transitions separated by commas. Enter the characters "->" between states to specify a transition between these states.

You can add a second line to the `TRANSITIONS` line to list more transitions.

The syntax for a `TRANSITIONS_X` or `TRANSITIONS_NEVER` line is similar.

Specifying the Configuration File

VCS looks for a configuration file named `cm.fsm_config` in the current directory. You can specify a configuration file with a different name and location with an argument to the `-cm_fsmcfg` compile-time option. For example:

```
% vcs -cm fsm -cm_fsmcfg myconfig.txt source.v
```

or

```
% vcs -cm fsm -cm_fsmcfg myconfig.txt cfg
```

Sequence Filtering in Reports

You use the SEQ_REQUIRE and SEQ_EXCLUDE commands in the FSM configuration file for filtering sequences in report files. You can use either command separately or you can use them together to specify what sequences you want reported in the report file. The SEQ_REQUIRE command is used for specifying what must be included in a sequence for URG to report the sequence. The SEQ_EXCLUDE command is used for specifying what cannot be included in a sequence for URG to report it.

The arguments to these commands are patterns for transitions and can be any of the following:

- A state

- A transition between states

- A sequence of states of any length

You can use the wildcard character * in any transition or sequence to specify a transition from/to any state.

The following are examples of these commands and arguments:

```
SEQ_REQUIRE=state2->state3
```

In this example, URG only reports sequences that include the transition from `state2` to `state3`.

```
SEQ_EXCLUDE=*->state2
```

In this example, URG does not report any sequence that includes a transition from any other state to `state2`.

```
SEQ_REQUIRE=state4
```

```
SEQ_EXCLUDE=state4->state5
```

In this example, URG only reports sequences that include `state4`, but does not report sequences that include a transition from `state4` to `state5`.

Specifying the Maximum for Sequences

There are commands that you can enter at the beginning of the configuration file that specify how VCS compile and monitor the sequences in the FSMs in all the module or design entity definitions in your design. These commands must come before any section for a module or design entity definition. These commands are as follows:

`SEQ_NUMBER_MAX=integer`
    Specifies the maximum number of sequences in any module or design entity that VCS maintains. The integer value must be non-negative.

`SEQ_LENGTH_MAX=integer`
    Specifies the length of the longest sequence that VCS maintains. The integer value must be non-negative.

In a mixed HDL design, if you enter these commands in both the Verilog and VHDL FSM configuration files, and the integer values are not the same, VCS uses the higher value and displays a warning message.

Using the Configuration File for One Hot FSM

For one hot bit fsm, the states no longer represent values, but index of the bit that needs to be turned on. Hence, 0 actually represents the value 1 (if 0th bit is on, the value is 1), 1 represents 2, 2 represents 4, and so on.

For one hot bit FSM, you specify the indices instead of the values in the configuration file. To specify indices, you need to wrap it with square brackets. For values, you should not wrap it with square brackets. This applies to states and transitions.

For example, in the configuration file if you specify:

```
MODULE=simple_fsm
CURRENT=currentState
NEXT=nextState
STATES='h0,SA,SB
TRANSITIONS='h0->SA
SA->'h0,
SA->SB,
SB->'h0,
SB->SA
```

Now, SA=0 and SB=1.

Here VCS reads the states as 0,0,1, which is wrong, because you meant to specify 0 (as value), SA (as index, hence 1), SB (as index, hence 2)). So, the correct way to specify the value or index is to use square brackets as follows:

```
MODULE=simple_fsm
CURRENT=currentState
NEXT=nextState
STATES='h0,[SA],[SB]
TRANSITIONS='h0->[SA]
[SA]->'h0,
[SA]->[SB],
```

```
[SB]->'h0,
[SB]->[SA]
```

Reporting FSM State Values Instead of Named States

There are two ways to express an FSM:

- With procedural assignments to the entire reg (variable) that holds the current state and sometimes to the reg that holds the next state of the FSM. It could also be a continuous assignment to net that holds the next state.

- With procedural assignments to individual bits of the reg that holds the current or next state of the FSM.

VCS coverage metrics identifies the states of the FSM by analyzing assignment statements. If VCS finds parameter names for bit numbers in these assignment statements, and the FSM is expressed as assignments to bits, like the Hot Bit or One Hot FSM in Example 2-7, by default, it uses these parameter names as the states of the FSM. In the simulation of Example 2-7, by default, VCS monitors for, and URG displays or reports transitions of START to SA or SE to START instead of reporting the value transitions of the reg that holds the current state.

If you want VCS to monitor the value of the reg that holds the current state, and URG to display and report these values, instead of the parameters like START and SA, enter the `-cm_fsmopt reportvalues` compile-time option and keyword argument.

In Example 2-7, using this compile-time option with the START to SA transition would be reported as 'h1 to 'h2 and the SE to START transition would be reported as a 'h20 to 'h1.

This feature does not work for VHDL coverage metrics.

Enabling Indirect Assignment to State Variables

By default, the variable that holds the current state of the FSM must
be directly assigned a numerical constant or the value of a variable
that holds the next state of the FSM. You can allow FSM extraction
when there is indirect assignment to the variable that holds the
current state with the `-cm_fsmopt allowTmp` compile-time option
and keyword argument.

Without this option and argument, VCS does not extract the FSM in
:

Example 2-11   Indirect Assignment to State Variables

```
module fsm;
   reg [3:0] ns;
   wire [3:0] cs;
   Connect con(cs,ns);
   always @ cs
      casex(cs)
        4'b0000: ns=4'b0001; // 0?1
        4'b00x0: ns=4'b0011; // 2?3
        4'b001x: ns=4'b0010; // 3?2
        4'b0x11: ns=4'b0101; //none
        default: ns=4; // 1?4, 5?4
      endcase // casex(cs)
endmodule // fsm

module Connect(cs,ns);
   input [3:0] ns;
   output [3:0] cs;
   reg [3:0]    temp;    // this is the temp variable
   assign cs = temp;
   always @ ns
      begin #1 temp = ns;
      end
```

```
endmodule // Connect
```

In this FSM, the value of the signal that holds the next state of the FSM is assigned to signal `temp`, which is continuously assigned to the signal that holds the current state. You can instruct VCS to extract the FSM despite this indirect assignment by using the `-cm_fsmopt allowTmp` compile-time option and keyword argument.

This feature does not work in VHDL coverage metrics.

Enabling Two-state FSMs

By default, FSMs must have more than two states. You can tell VCS to extract two-state FSMs, scalar signals for the current and next state, with the `-cm_fsmopt report2StateFsms` compile-time option and keyword argument.

Without this option and argument, VCS does not extract the FSM in the following code:

```
module fsm(in);
   input in;
   reg   cs, ns;
   parameter zero=1'b0,
             one =1'b1;
   always @ in
      begin
        case(cs)
           zero : ns=one;
           one : ns=zero;
        endcase // case(cs)
      end
   always @ ns
      cs <= ns;
endmodule // fsm
```

This feature does not work in VHDL coverage metrics.

Enabling the Monitoring of Self Looping FSMs

VCS can extract an FSM from code in which the signal that holds the current state is assigned its current value. However, by default, VCS cannot monitor and have URG report "transitions" in which it is assigned its current value. You can enable this monitoring with the `-cm_fsmopt reportWait` compile-time option and keyword argument.

Without this option and argument, VCS cannot monitor certain transitions in Example 2-12:

Example 2-12   Monitoring Self Looping FSMs

```
module fsm(sig1,sig2);
   input sig1,sig2;
   reg [2:0] cs,ns;
   parameter zero = 3'b000,
             first = 3'b001,
             second= 3'b010,
             third = 3'b011,
             fourth = 3'b100;
   always @ (cs or sig1)
     begin
       case (cs)
         zero : begin
           if (~sig1)
              ns = zero;   // self loop
           else
              ns = first;
         end
         first :
           ns = second;
         second :
           ns = third;
         third :
```

```
              ns = fourth;
          fourth :
            if (~sig2)
               ns = zero;
            else
               ns = fourth; // self loop
        endcase // case(cs)
     end
  always @ sig2
     begin
        cs <= ns;
     end
endmodule // fsm
```

In this FSM, the current state signal is assigned the value of the next
state signal, and in some cases, the next state signal is assigned the
value of the current state signal, such as when the current state
signal value is zero or fourth.

Note:

There is a performance cost to using this feature. That is why it
is not the default behavior.

This feature does not work in VHDL coverage metrics.

Enabling X Value States

You may want to assign an X value to the signal that holds the
current state to indicate a problem in the logic of the FSM. By default,
VCS does not extract FSMs that have X values. However, you can
enable this extraction with the -cm_fsmopt reportXassign
compile-time option and keyword argument.

Without this option and argument, VCS does not extract the FSM in
Example 2-13:

Example 2-13   Enabling X Value States

```
module fsm (sig) ;
    input sig;
    reg [10:0] ns;
    reg [10:0] cs;
    parameter IDLE = 0,
              STATE1 = 1,
              STATE2 = 2,
              STATE3 = 3;

    always @ cs
       begin
          case(cs)
            IDLE :
                ns = STATE1;
            STATE1 :
                ns = STATE2;
            STATE2 :
                ns = STATE3;
            STATE3 :
                ns = IDLE;
            default:
                ns = 4'bxxxx;  // the whole value
          endcase // case(cs)
       end
    always @ sig
       begin
          cs <= ns;
       end
endmodule // fsm

module fsm1 (sig,reset) ;
    input sig,reset;
    reg [10:0] ns;
    reg [10:0] cs;
    parameter IDLE = 0,
              STATE1 = 1,
              STATE2 = 2,
              STATE3 = 3;

    always @ cs
```

```
        begin
           if(reset)
              cs=IDLE;
           else begin
              case(cs)
                 IDLE :
                    ns = STATE1;
                 STATE1 :
                    ns = STATE2;
                 STATE2 :
                    ns = STATE3;
                 STATE3 :
                    ns = IDLE;
                 default:
                    ns = 4'b000x;    // just 1 bit is x
              endcase // case(cs)
           end // else: !if(reset)
        end
   always @ sig
      begin
         cs <= ns;
      end
endmodule // fsm1
```

URG reports transitions to and from X as follows:

```
IDLE->X
X->STATE2
```

This feature does not work in VHDL coverage metrics.

Filtering Out Transitions Caused by Specified Signals

You can filter out transitions in assignment statements controlled by
if statements where the conditional expression (following the if
keyword) is a signal you specify. You might want to do this, for
example, to a reset signal. This filtering out can be used for the

specified signal in any module definition or in the module definition you specify. You can also specify the FSM and whether the signal is true or false.

To do this, use the `-cm_fsmresetfilter` compile-time option, and in the file that you specify, include what you want to filter. The following is an example of the contents of this file:

```
signal=reset case=TRUE
module=abc signal=rst case=FALSE
module=xyz fsm=state signal=more_rst CASE=TRUE
module=ABC fsm=STATE signal=reset case=NONE.
```

The first line does not specify a module, therefore, it applies to all signals named `reset` in any module definition. The filtering out applies when that signal is true.

The second line begins with a specified module, named `abc`. It applies to the signal named `rst` when that signal is false.

The third line is used for module `xyz`. It applies to assignments in FSM named `state`, to the signal named, `more_rst`, when that signal is true.

The fourth line applies to the module named `ABC`, the FSM named `STATE`, the signal named `reset`, and any type of transition on that signal. The statement `case=NONE` specifies not considering the value of the signal.

Note:

This applies only to assignments controlled by the `if` statement. If you are using an `if-else` statement, it does not filter out transitions controlled by the `else` part. Therefore, for example, if the file contains the following entry:

```
signal=reset case=TRUE
```

In the following FSM:

```
always@ns
cs <= ns;

always@cs
begin
   if (reset)
      cs = 1;
   else
      cs <= ns;
   case(cs)
    0: ns=1;
    1: ns=2;
    2: ns=3;
    default: ns=0;
   endcase
end
```

The explicit assignment of the 1 value to the signal that holds the current state is filtered out because it is controlled by the `if` part of the `if-else` statement, but neither of the assignments of the next state to the current state are filtered out, including the one controlled by the `else` part.

More Options for Functional Coverage

Options to Specify in the optconfigfile

Currently, the configuration file allows enabling/disabling of certain optimizations on a design object such as a module definition, a module instance, URG, or a module instance hierarchy.

You can enable instance coverage for all the covergroups at a global level, rather than having to set option.per_instance for each covergroup.

You can specify the following four options in the optconfigfile.

- CovgPerInstanceOn — This option is used to set option.per_instance to 1 for all the covergroups.

- CovgStrobeOn — This option is used to set type_option.strobe option to 1 for all the covergroups.

- CovgNoAutoCross — This option is used to disable auto-crosses, for crosses having user-defined bins.

- Auto_bin_max — This option is used to override the default value (64) of option.auto_bin_max for all the covergroups.

Usage

```
config_stmt := tree {$root} {list_of_covg_attribs}
        | module {mod_name} {list_of_covg_attribs}

list_of_covg_attribs := CovgPerInstanceOn
        | CovgStrobeOn
        | CovgNoAutoCross
        | auto_bin_max=<value>
        | list_of_covg_attribs(,list_of_covg_attribs)
```

For example, using this `optconfigfile`, you can set the option 'auto_bin_max' to some value for all the covergroups in a given module or in the entire design.

The "tree" keyword with "$root" as the (special) hierarchy name indicates that these coverage attributes, when specified, apply to all the covergroups in all module instances in the design.

Also note that the value of auto_bin_max mentioned in the config file applies only to those covergroups/coverpoints that do not explicitly override the option 'auto_bin_max'. That means, only the default value of auto_bin_max is changed to the value mentioned in the config file.

For more information about the auto_bin_max, strobe, and per_instance attribute, see the IEEE SystemVerilog LRM.

Example

Config file specification (assume the config file name as covg.config)

```
tree {$root} {auto_bin_max=100, CovgPerInstanceOn,
CovgStrobeOn, CovgNoAutoCross};
```

For this example, VCS command is as follows:

```
% vcs +optconfigfile+covg.config -sverilog design.v
```

Note:

"$root" used in any context other than the coverage attributes results in a compile-time error.

Unified Coverage Directory and Database Control

A coverage directory named `simv.vdb` contains all the testbench functional coverage data. For your reference, VCS associates a logical test name with the coverage data that is generated by a simulation. VCS assigns a default test name, which you can override, by using the following tasks:

```
$coverage_set_test_database_name
                     ("test_name","dir_name"); //For SV

task $coverage_set_test_database_name
                     ("test_name","dir_name"); //For NTB
```

In order to not save the coverage data to a database file (for example, if there is a verification error), use the following system task:

```
$coverage_save_database (flag);
```

The value of flag can be:

*   OFF for disabling database saving (not "backup")

*   ON for enabling database saving (not "backup")

Loading Coverage Data

Both cumulative coverage data and instance-specific coverage data can be loaded. The loading of coverage data from a previous VCS run implies that the bin hits from the previous VCS run to this run are to be added.

Loading Cumulative Coverage Data

The cumulative coverage data can be loaded either for all coverage groups, or for a specific coverage group. To load the cumulative coverage data for all coverage groups, use the following syntax:

```
$coverage_load_cumulative_data("test_name", "dir_name");
```

In this task, "dir_name" is optional. If you do not specify a "dir_name", by default, simv.vdb is taken as the directory containing the database.

The above tasks directs VCS to find the cumulative coverage data for all coverage groups found in the specified database file and to load this data if a coverage group with the appropriate name and definition exists in this VCS run.

To load the cumulative coverage data for just a single coverage group, use the following syntax:

```
$coverage_load_cumulative_cg_data("test_name",
                            "covergroup_name", "dir_name");
```

In this task, "dir_name" is optional. If you do not specify a "dir_name", by default, simv.vdb is taken as the directory containing the database.

In the following example, there is a SystemVerilog class MyClass with an embedded covergroup covType. VCS finds the cumulative coverage data for the coverage group MyClass::covType in the database file Run1 and loads it into the covType embedded coverage group in MyClass.

Example 2-14

```
class MyClass;
  integer m_e;
  covergroup covType @m_e;
```

```
      cp1 : coverpoint m_e;
   endgroup
endclass
...
$coverage_load_cumulative_cg_data("Run1",
                                          "MyClass::covType");
```

In the following example, a Vera class, `MyClass`, exists with an embedded coverage object `covType`. VCS finds the cumulative coverage data for the coverage group `MyClass::covType` in the database file `Run1` and loads it into the `covType` embedded `coverage_group` in `MyClass`.

Example 2-15

```
MyClass{
     integer m_e;
     coverage_group covType{
          sample_event = wait_var(m_e);
          sample m_e;
          }
     }
     ...
     ...
     coverage_load_cumulative_cg_data("Run1",
                                          "MyClass::covType");
```

Loading Instance Coverage Data

The coverage data can be loaded for a specific coverage instance. To load the coverage data for a standalone coverage instance, use the following syntax:

```
$covgLoadInstFromDbTest
            (coverage_instance,"test_name", "dir_name");
//For SV
coverage_instance.load("test_name", "dir_name");
//For NTB
```

In this task, "dir_name" is optional. If you do not specify a "dir_name", by default, simv.vdb is taken as the directory containing the database.

To load the coverage data for an embedded coverage instance, use the following syntax:

```
$covgLoadInstFromDbTest (class_object.cov_group_name,
                    "test_name", "dir_name"); //For SV
class_object.cov_group_name.load("test_name","dir_name");
//For NTB
```

In this task, "dir_name" is optional. If you do not specify a "dir_name", by default, simv.vdb is taken as the directory containing the database.

The above commands direct VCS to find the coverage data for the specified instance name in the database, and load it into the coverage instance.

In Example 2-16, there is a SystemVerilog class MyClass with an embedded covergroup covType. Two objects obj1 and obj2 are instantiated, each with the embedded covergroup covType. VCS will find the coverage information for the coverage instance obj1::covType from the database file Run1, and load this coverage data into the newly instantiated obj1 object. Note that the object obj2 will not be affected as part of this load operation.

Example 2-16

```
class MyClass;
  integer m_e;
  covergroup covType @m_e;
    cp1 : coverpoint m_e;
  endgroup
endclass
...
MyClass obj1 = new;
```

```
$covgLoadInstFromDbTest(obj1,"Run1");
MyClass obj2 = new;
```

Note:

The compile-time or runtime options `-cm_dir` and `-cm_name` will overwrite the calls to `coverage_set_test_database_name` and load coverage data tasks.

In Example 2-17, there is a Vera class `MyClass` with an embedded coverage object `covType`. Two objects `obj1` and `obj2` are instantiated, each with the embedded coverage group `covType`. VCS will find the coverage information for the coverage instance `obj1::covType` from the database file `Run1`, and load this coverage data into the newly-instantiated `obj1` object. Note that the object, `obj2`, will not be affected as part of this load operation.

Example 2-17

```
MyClass {
    integer m_e;
    coverage_group covType {
        sample_event = wait_var(m_e);
        sample m_e;
        }
    }
    ...
    ...
    MyClass obj1 = new;
    obj1.load("Run1");
    MyClass obj2 = new;
```

Using `-covg_disable_cg` to Disable Functional Coverage Items

The `-covg_disable_cg` option disables functional coverage (covergroups) from the verification environment. You can use this option at compile-time or runtime.

If you use this option at compile-time, all the covergroups are disabled and even statements using covergroup instances are removed. For example, if 'cg' is a covergroup, statements using covergroups such as `if(cg.get_coverage() < 80) begin ….end` gets eliminated.

If you use this option at runtime, all the covergroups are disabled. There is no affect on statements using covergroup instances. The `-covg_disable_cg` option is helpful in analyzing performance impact due to functional coverage.

Similar to the `-covg_disable_cg` option, the `-cg_coverage_control=0` option too turns off functional coverage collection. However, `-cg_coverage_control=0` turns off coverage collection at time 0, which you can turn on during runtime inside your design using the `$cg_coverage_control` system task.

Using `-covg_no_guard_in_shape` to Handle Guards on Illegal/Ignore Bins

For an illegal/ignore coverpoint/cross bin having a guard condition, the value of the condition evaluated when the covergroup to which the bin belongs is sampled for the first time determines the creation of the bin; it is dropped for a false value and is retained when the value is true. This dependency on the values of the guard conditions impacts the bin space of the covergroup. When there are multiple instantiations of the group inside a module, the difference in the guard values across the group instances will result in multiple variants (shapes) being created for the covergroup, with the guard values captured by the `GUARD_ON`/`GUARD_OFF` keywords in the covergroup shape names generated by VCS.

Such a shape altering behavior for illegal/ignore bins deviates from the way guard conditions on the regular bins are handled. Furthermore, SystemVerilog LRM does not indicate any special treatment for illegal/ignore bins.

Under the run time switch, `-covg_no_guard_in_shape`, the handling of guard conditions is unified across all bins. All illegal/ignore bins are unconditionally retained irrespective of the values of guard conditions specified on them, as done for regular bins. The direct consequence is the non-impact of the guard values on the binspace of the covergroup. The changes in the guard values across instances of the group do not result in new shapes for the covergroup. Therefore, the keywords `GUARD_ON` and `GUARD_OFF` will not appear in the covergroup shapes names under this behavior.

Example

The following example illustrates the difference between the default and under the switch behaviors:

The following example has a covergroup, `cg_def`. It has three arguments, where two arguments are used to guard ignore cross bins, `bc2` and `bd2`, respectively, while the third argument is used to set the option `auto_bin_max` for the coverpoints of the group. Since the value of `auto_bin_max` impacts the bin space of the covergroup, the respective argument `abm`, which is used to set the option becomes a shape defining parameter for the group. The group has six instances, namely, `cgd1`, `cgd2`, `cgd3`, `cgd4`, `cgd5` and `cgd6`, respectively, with all the four possible guard value combinations captured across them.

```
program test;
reg clk = 0;
bit  [1:0] dat1, dat2, dat3;
```

```
bit gc;

covergroup cg_def(bit grd1, bit grd2, int abm) @clk;
    option.auto_bin_max = abm;
    d1: coverpoint dat1;
    d2: coverpoint dat2;
    d3: coverpoint dat3;

    d1xd2: cross d1, d2 {
         bins bc1 = binsof(d1) intersect {0} && binsof(d2)
intersect {0};
          ignore_bins bc2 = binsof(d1) intersect {1} &&
binsof(d2) intersect {1} iff (gc==grd1);
    }
    d1xd3: cross d1, d3 {
         bins bd1 = binsof(d1) intersect {0} && binsof(d3)
intersect {0};
          ignore_bins bd2 = binsof(d1) intersect {1} &&
binsof(d3) intersect {1} iff (gc==grd2);
    }
endgroup

initial
forever #2 clk = ~clk;

initial begin
  cg_def cgd1 = new(0,1,2);
  cg_def cgd2 = new(1,0,4);
  cg_def cgd3 = new(0,0,2);
  cg_def cgd4 = new(1,1,4);
  cg_def cgd5 = new(0,1,2);
  cg_def cgd6 = new(0,1,4);

   #2 $finish;
end


endprogram
```

Under the default behavior, if in a particular group instance, a '0' passed to grd1 will turn off the bin bc2. Similarly, a '0' passed to grd2 will turn off bd2. If '1' is passed to grd1, it results in the respective bins getting retained. VCS generates five different shapes for `cg_def` across the six instances:

```
0.00 1      100
test::cg_def::SHAPE{abm=4,Guard_ON(d1xd2.bc2),Guard_OFF(d1
xd3.bd2)} (for cgd1 and cgd5)
  0.00 1      100
test::cg_def::SHAPE{abm=2,Guard_ON(d1xd2.bc2),Guard_OFF(d1
xd3.bd2)} (for cgd2)
  0.00 1      100
test::cg_def::SHAPE{abm=4,Guard_OFF(d1xd2.bc2,d1xd3.bd2)}
(for cgd3)
  0.00 1      100
test::cg_def::SHAPE{abm=2,Guard_ON(d1xd2.bc2,d1xd3.bd2)}
(for cgd4)
  0.00 1      100
test::cg_def::SHAPE{abm=4,Guard_ON(d1xd3.bd2),Guard_OFF(d1
xd2.bc2)} (for cgd6)
```

Under the switch, however, where only the value of 'abm' has an impact on the bin space, with guard values having no impact on the covergroup shape. Consequently, only two shapes are created for the group, as shown in the following code:

```
  0.00 1      100    test::cg_def::SHAPE{abm=4} (for cgd2,
cgd4 and cgd6)
  0.00 1      100    test::cg_def::SHAPE{abm=2} (for cgd1,
cgd3 and cgd5)
```

Note:It is not advisable to merge databases generated from run with and without the switch when there are covergroups having ignore or illegal bins specified with guard conditions. Since the names of the covergroups change under the switch, as illustrated in the example above, coverage reporting tools (URG, DVE and UCAPI) cannot merge the shapes and report all of them. For the example above, if a test is run twice, one each with and without the switch, and the two databases are passed to URG or DVE for merging, the tool will report seven shapes. It is strongly recommended to avoid mixing both behaviors, i.e., if the switch is to be used, make sure it is passed to all of the tests.

Functional Coverage System Tasks Summary Table

Table 2-4   FCOV System Tasks for NTB

System Tasks	Description
coverage_set_test_database_name (test_name, dir_name)	Sets the name of the toplevel directory in which coverage will be dumped and also the associated test name. dir_name is optional and the default is "simv.vdb".
coverage_set_test_database_name ("foo")	Creates a test named "foo" inside the directory hierarchy starting with "simv.vdb".
coverage_set_test_database_name ("foo", "top/cdir")	Saves the testdata inside a test named "foo" inside the toplevel coverage directory "top/cdir.vdb" Note that the tool will add the ".vdb" extension to the directory name (if the extension is not already specified).
coverage_save_database(0/1)	Passing the argument 1 will cause the database to be dumped on to the disk at that particular simulation time. Argument 0 will mean that the coverage database is not saved at all.
coverage_backup_database_test(0/1)	Assume a coverage database "test" is present inside "simv.vdb", During the next simulation run, the new database will generated as "test_gen1" thus in effect preserving the data from the previous run.
coverage_load_cumulative_data (test_name, dir_name)	Loads cumulative coverage data (all cover group variants) for the database identified by the test test_name and stored inside coverage hierarchy starting with dir_name. Note that dir_name is optional.
coverage_load_cumulative_data ("allTrans")	Loads all covergroup variants from the test "allTrans" in the coverage directory "simv.vdb".
coverage_load_cumulative_data ("allTrans", "moo/foo.vdb")	Loads the test "allTrans" from the toplevel directory "moo/foo.vdb".
coverage_load_cumulative_cg_data (test_name, cg_name, dir_name)	Allows the specified covergroup to be loaded. test_name and dir_name will work in the same manner as for the previous system call. cg_name is mandatory.

Table 2-4   FCOV System Tasks for NTB

System Tasks	Description
coverage_load_cumulative_cg_data ("foo", "covgrp1", "moo/foo.vdb")	Loads the cumulative data for covergroup "covgrp1" from the test "foo" inside the toplevel directory "moo/foo.vdb".
covg_inst.load(test_name, dir_name)	Loads the specified instance (covg_inst) from the coverage database identified by the test test_name and top level directory dir_name.
coverage_set_database_file_name (file_name)	Deprecated from VCS 2006.06.SP1 onwards. Internally gets treated as coverage_set_test_database_name.
coverage_set_database_file_name ("top/cdir/foo")	Will create the coverage database with test name "foo" inside the directory hierarchy "top/cdir.vdb"
coverage_backup_database_file(0/1)	Deprecated from VCS 2006.06.SP1 onwards. Works in the same manner as coverage_backup_database_test
coverage_load(file_name)	Deprecated from VCS 2006.06.SP1 onwards. Internally gets treated as coverage_load_cumulative_data.
coverage_load("moo/foo/allTrans"	will load the test named "allTrans" from the directory hierarchy "moo/foo.vdb"
coverage_control (0/1, covergroup_name)	Turns on/off coverage collection for the specified cover group.
coverage_control(0, "cov1")	will turn off coverage control for the covergroup named "cov1"

Table 2-5   FCOV System Tasks for SV

System Tasks	Description
$coverage_set_test_database_name(test_name, dir_name)	Same as coverage_set_test_database_name in NTB.
$coverage_backup_database_test(0/1)	Same as coverage_backup_database_test in NTB.
$coverage_load_cumulative_data(test_name, dir_name)	Same as coverage_load_cumulative_data in NTB.
$coverage_load_cumulative_cg_data(test_name, cg_name, dir_name)	Same as coverage_load_cumulative_cg_data in NTB.
$covgLoadInstFromDbTest(inst_name, test_name, dir_name)	Loads the instance specified by inst_name from the coverage database identified by the test test_name and top level directory dir_name. the last two arguments follow the same semantics as covg_inst.load in NTB.
$load_coverage_db("dir_name/test_name")	$load_coverage_db("moo/foo/allTrans") loads the test name "allTrans" from the directory hierarchy "moo/foo.vdb".
$cg_coverage_control(0/1, covergroup_name)	Same as coverage_control in NTB.

Note:

The test_name, dir_name, file_name, covergroup_name, and inst_name are string arguments. You can either use a variable of type string or a string constant itself.

Controlling the Scope of Coverage Compilation

You do not need to compile the entire design hierarchy for code coverage. Instead, you can limit the scope of coverage metrics by doing either of the following:

- Enter the -cm_hier compile-time option and its configuration file to specify the module definitions, instances and subhierarchies, and source files that you want VCS either to exclude from coverage or exclusively compile for coverage (see "Using a Configuration File").

- Enter the -cm_report noinitial compile-time option to disable the the monitoring of the contents of initial blocks for line, condition, branch, and path metrics (see "Disabling the Monitoring of Initial Blocks").

- Enter pragmas in your Verilog source code to exclude lines, source files, and module instances from coverage (see "Using Pragmas to Limit Line Coverage").

- Enter pragmas in your VHDL source code to exclude lines from coverage (see "Using Pragmas to Limit VHDL Lines From Coverage").

Using a Configuration File

Note:

URG has an option, -hier, that accepts a subset of the controls that can be used in the compile-time -cm_hier file. The files are the same other than the differences noted below.

Coverage metrics also has another type of configuration file "Using an FSM Configuration File" on page 67 for specifying FSMs in a design.

Statements in the configuration file begin with a plus (+) or minus (-) sign. If a statement begins with a plus sign (+), it specifies what VCS should compile for coverage. If a statement begins with a minus sign (-), it specifies what VCS should not compile for coverage.

You can use the following statements in the configuration file:

`+tree` *instance_name* [*level_number*]

VCS compile only the specified instance and the instances under it for coverage. These instances can be Verilog module or VHDL entity instances. VCS exclude all other instances from coverage.

A level number of 0 (or no level number) specifies the entire subhierarchy, 1 specifies only this instance, 2 specifies this instance and those instances directly under this instance, 3 specifies this instance and instances in the subhierarchies that are one and two levels below the specified instance. There is no limit to the integer you specify as the level number.

If the subhierarchy includes instances in the other HDL, VCS does not include these instances.

`-tree` *instance_name* [*level_number*]

VCS exclude this instance from coverage and other instances under it. These instances can be Verilog module or VHDL entity instances. VCS include all other instances in coverage.

A level number of 0 (or no level number) specifies the entire subhierarchy, 1 specifies only this instance, 2 specifies this instance and those instances directly under this instance, and so on.

If the subhierarchy includes instances in the other HDL, VCS does not exclude these instances.

`+module` *module_name* `|` *entity_name*
VCS compiles all instances of the specified Verilog module or VHDL entity definition, and excludes all other definitions under it, for coverage.

`-module` *module_name* `|` *entity_name*
VCS does not compile all instances of the specified Verilog module or VHDL entity definition, and includes all other definitions under it, for coverage.

`+file` *file_name*
VCS compile for coverage only the code in this file. If the file is not in the current directory, specify the path name of the file.

`-file` *file_name*
VCS exclude the code in this file from coverage. If the file is not in the current directory, specify the path name of the file.

`+filelist` *file_name*
VCS compile for coverage only the source files listed in the specified file.

`-filelist` *file_name*
VCS exclude from coverage the source files listed in the specified file.

The usage of environment variable in the path is allowed for the above `+/-file` or `filelist` statements. For example:

-file $PATH/common/abc.v

-file $PATH/common/xyz.vhd

The following statements (`+library` and `-library`) are not supported in the file you submit to URG with the `-hier` command-line option.

`+library` *library_name*
    VCS compiles for coverage all the Verilog module and VHDL entities in the specified library. VCS excludes all other Verilog modules and VHDL entities from coverage.

`-library` *library_name*
    VCS excludes from coverage all the Verilog modules and VHDL entities in the specified library. VCS includes all other Verilog modules and VHDL entities in coverage.

`+moduletree` *module_name* [*level_number*]
    VCS provides coverage metrics for all instances of the specified module and for all module instances in the hierarchy below the specified module. In other words, each hierarchy tree starting at each instance of the specified module will have coverage metrics provided. The coverage metrics are only provided for the number of levels of hierarchy specified by the optional level_number.

`-moduletree` *module_name* [*level_number*]
    VCS excludes coverage metrics for all instances of the specified module and for all module instances in the hierarchy below the specified module. In other words, each hierarchy tree starting at each instance of the specified module will have coverage metrics excluded. The coverage metrics are only excluded for the number of levels of hierarchy specified by the optional level_number.

`+/-node`

    Excludes or includes a signal in toggle coverage. For more information, see the topics "Excluding a Signal in Toggle Coverage" on page 19 and "Including a Signal in Toggle Coverage" on page 19.

This option is not supported with the `urg -hier` option.

You can use the `*`, `?` and `+` wildcard characters in the configuration file to specify definitions, instances, VHDL libraries, and file names.

You can add comments to this file using the convention for Verilog comments. Comments are supported both at compile-time and at report time with URG

Type	Used for...
`// comment`	The end of the line comment.
`code /* comment */ code`	The inside of a line comment.

`/* comment 1`
`comment 2`
`comment3 */`
	Multiple line comments.

Begin - End Blocks for Coverage Type Control

The keyword arguments to the `-cm` compile-time option (`line`, `cond`, `fsm`, and `tgl`) specify the type of coverage for which VCS compile. These arguments specify compiling the entire design for that type of coverage. With begin-end blocks in the `-cm_hier` file, you can specify what parts of the design VCS compile for each type of coverage.

Note:

When you use this feature, the argument to the `-cm` compile-time option enables the compilation of the type of coverage and the entries in the `-cm_hier` file begin-end blocks specify where VCS compiles for that type of coverage.

Lines in the `-cm_hier` file that are not begin-end blocks must come before any begin-end block lines.

The following are examples of the usage of begin-end blocks in the `-cm_hier` file:

Example 1
```
begin fsm+line +tree TOP.D1 end
```

This line specifies that VCS compile only the instances in the specified subhierarchy (`TOP.D1`) for FSM coverage. This line only specifies where VCS compile for FSM coverage. If other types of coverage are specified with other `-cm` compile-time option arguments, VCS compile the entire design for those types of coverage.

Note:

VHDL hierarchical names can only contain uppercase letters. This requirement is not true for Verilog.

Example 2
```
begin line+tgl+cond -module bdev end
```

This line specifies that VCS does not compile the instances of the module or design entity named `bdev` for line, toggle, and condition coverage. To specify more than one type of coverage, use the plus (+) delimiter.

Example 3
```
begin cond+line -library gate end
begin tgl -tree TOP.D1.L* end
```

The first begin-end block calls for excluding all instances of the entities in the VHDL library named `gate` from condition and line coverage. The second begin-end block calls for excluding all

subhierarchies where the top instance has a hierarchical name beginning with `TOP.D1.L` (some of these uppercase letters could be lowercase in the VHDL source code) from toggle coverage.

Example 4

```
+tree tb.dut1
begin line -module bdev end
```

The first line does not specify a type of coverage. It calls for VCS to compile the instances in the subhierarchy with the top instance `tb.dut` for the types of coverage specified with the `-cm` compile-time option (the hierarchical name has lowercase letters so it must be a Verilog instance).

The second line is a begin-end block that tells VCS not to compile all instances of the definition named `bdev` for line coverage.

Disabling the Monitoring of Initial Blocks

The `-cm_report noinitial` compile-time option can be used to disable the monitoring of the contents of initial blocks for line, condition, branch, and path metrics.

In most of the cases, the initial blocks are completely covered, which might increase the coverage percentage needlessly. Disabling the monitoring of the contents of these initial blocks allows you to focus on the actual design coverage.

Note:
  Using the `-cm_report noinitial` compile-time option does not affect toggle coverage and FSM coverage.

Use Model

To disable the monitoring of the contents of initial blocks, include the `-cm_report noinitial` compile-time option in the `vcs` command-line.

For example,

```
module dut(a,b,c);
    input a,b;
    output c;
    wire c;

    assign c = a&&b;
endmodule
module tb();
   reg y,x,w;
   reg clk;
   wire z;
   initial begin
     //The contents of this block are ignored for coverage
       $monitor($time,"x=%b,y=%b,z=%b",x,y,z);
       #1;
       clk = 1'b0;
       x = 1'b0
       y = 1'b0;
       #1;
       x = 1'b1;
       y = 1'b1;
       #1;
       x = 1'b0;
       y = 1'b0;
       #1;
       x = 1'b1;
       y = 1'b1;
       #1;
       x = 1'b0;
       y = 1'b0;
       #1;
       #50;
```

```
        $finish;
    end
    initial begin
       //The contents of this block are ignored for coverage
        if(x)
             $display("In the if loop\n");
        else
             $display("In the else loop\n");
     end

    always #5
        clk = !clk;
    always@(posedge clk) begin
        if(z)
            $display("z = 1'b1\n");
        else
            $display("z = 1'b0\n");
    end dut dut1(x,y,z);
endmodule
```

To disable the monitoring of the contents of initial blocks in the
example, execute the following command:

```
% vcs test.v -cm line+branch+cond -cm_report noinitial
```

URG Report

The following is the URG report for the example. Note that the
contents of initial blocks are ignored for coverage.

```
---------------------------------------------------------------
Line Coverage for Module : tb

             Line No.   Total   Covered   Percent
TOTAL                       5         4     80.00
ALWAYS          42          2         2    100.00
ALWAYS          45          3         2     66.67

41                        end
```

```
42     1/1             always #5
43     1/1                 clk = !clk;
44                     always@(posedge clk) begin
45     1/1                 if(z)
46     0/1    ==>                 $display("z = 1'b1\n");
47                             else
48     1/1                     $display("z = 1'b0\n");
-------------------------------------------------------------
Branch Coverage for Module : tb
          Line No. Total Covered Percent
Branches          2    1       50.00
IF        45      2    1       50.00

45                     if(z)
                       -1-
46                         $display("z = 1'b1\n");
                        ==>
47                     else
48                         $display("z = 1'b0\n");
```

Coverage Pragmas

Pragmas are meta comments in your source code.

VCS have pragmas that allow you to specify that certain lines of Verilog code, Verilog source files, specified signal, and all instances of certain module definitions are not to be compiled for line, toggle, condition, FSM, or branch coverage. They work like compiler directives.

These pragmas are as follows:

Pragma	Description
`//VCS coverage off`	Specifies disabling coverage for the block of source code that follows.

`//VCS coverage on`	Specifies re-enabling coverage for the source code that follows when a previous block of source code is disabled by the `//VCS coverage off` pragma.
`//VCS coverage exclude_file`	Specifies disabling coverage for the code in the source file that contains this pragma. Synopsys recommends entering this pragma at the beginning of the source file, before any module definitions.
`//VCS coverage exclude_module`	Specifies disabling coverage for all instances of the module definition that contains this pragma. Synopsys recommends entering this pragma immediately after the port declarations. This pragma does not exclude from coverage the module instances hierarchically under these instances. To do this, see "Controlling the Scope of Coverage Compilation" .

The HDL Compiler and Behavioral Compiler users can use the `//synopsys translate_off` directive in place of the `//VCS coverage off` pragma and the `//synopsys translate_on` directive in place of the `//VCS coverage on` pragma.

The `//VCS coverage on` pragma enables line coverage after a `//synopsys translate_off` directive and a `//synopsys translate_off` directive disables line coverage after a `//VCS coverage on` pragma.

Similarly, the `//VCS coverage off` pragma disables line coverage after a `//synopsys translate_on` directive and a `//synopsys translate_on` directive enables line coverage after a `//VCS coverage off` pragma.

Using Pragmas to Limit Line Coverage

When you disable line coverage using pragmas, URG indicates this fact in its line coverage report.
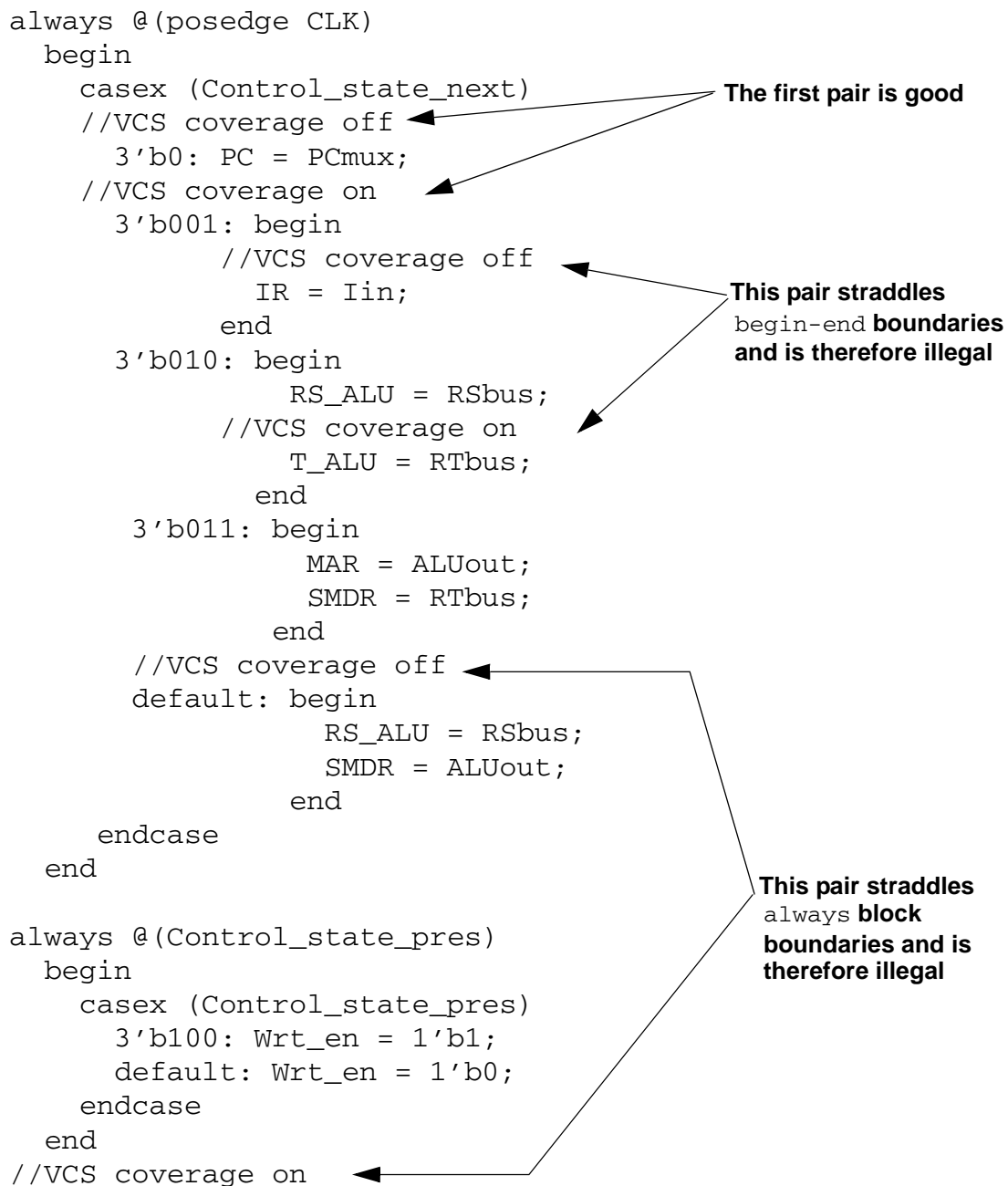
Pragmas do not exclude module instances. For example:

```
module test;
reg clk, a;
// synopsys translate_off
mod1 inst1(a,clk);
// synopsys translate_on
.
.
.
endmodule
```

This example does not exclude `test.inst1` from coverage.

Pragma pairs for disabling and re-enabling coverage cannot be used across module boundaries or across blocks of code (such as begin-end or always blocks). Figure 2-1 illustrates this point.

Figure 2-1   Using Pragmas

```
always @(posedge CLK)
  begin
    casex (Control_state_next)                          The first pair is good
    //VCS coverage off
      3'b0: PC = PCmux;
    //VCS coverage on
      3'b001: begin
            //VCS coverage off
              IR = Iin;                                 This pair straddles
            end                                         begin-end boundaries
      3'b010: begin                                     and is therefore illegal
              RS_ALU = RSbus;
            //VCS coverage on
              T_ALU = RTbus;
            end
      3'b011: begin
              MAR = ALUout;
              SMDR = RTbus;
            end
    //VCS coverage off
    default: begin
              RS_ALU = RSbus;
              SMDR = ALUout;
            end
    endcase
  end

always @(Control_state_pres)                            This pair straddles
  begin                                                 always block
    casex (Control_state_pres)                          boundaries and is
      3'b100: Wrt_en = 1'b1;                            therefore illegal
      default: Wrt_en = 1'b0;
    endcase
  end
//VCS coverage on
```

Using Pragmas to Limit VHDL Lines From Coverage

VCS has pragmas that instruct VCS not to compile certain lines of VHDL sequential statements or concurrent signal assignments for line and condition coverage. They also instruct VCS not to compile certain signal declarations for toggle coverage. These pragmas are as follows:

`--synopsys coverage_off`

> Specifies disabling line and condition coverage for the sequential statements or concurrent signal assignments that follow, for line and condition coverage. Specifies disabling toggle coverage for the signal declarations that follow.

`--synopsys coverage_on`

> Specifies re-enabling line and condition coverage for the sequential statements or concurrent signal assignments that follow and also specifies re-enabling toggle coverage for the signal declarations that follow, when a previous block of source code is disabled by the `--synopsys coverage_off` pragma.

The following examples use these pragmas to disable and re-enable line and condition coverage in a sequential code:

Example 1
```
architecture A of E is
.
.
.
begin
p : process
begin
--synopsys coverage_off
sig1 <= 0; -- not compiled for coverage
--synopsys coverage_on
```

```
if (x == '0') then
s <= y;
end if;
end process p;
end A;
```

VCS does not compile the signal assignment statement to signal `sig1` for line coverage.

Example 2

```
package body mypack is
procedure vec_assn(
signal out: bit_vector(3 downto 0);
x : bit_vector(3 downto 0)
) is
begin
--synopsys coverage_off
if (x == "000") then      -- not compiled for coverage
    sequential statements -- not compiled for coverage
.
.
.
end if;
--synopsys coverage_on
out <= x;
end vec_assn;
.
.
.
```

VCS does not compile the sequential statements controlled by the `if` statement in procedure `vec_assn` for line or condition coverage.

Example 3

You can also use these pragmas around signal declarations to disable and re-enable toggle coverage. For example:

```
architecture A of E is
signal clk : bit;
--synopsys coverage_off
signal flag1 : integer;   --not compiled for toggle coverage
--synopsys coverage_on
signal flag2 : integer;    --compiled for toggle coverage
begin
.
.
.
end A;
```

VCS does not compile signal `flag1` for toggle coverage; it does compile signal `flag2` for toggle coverage.

Pragmas to Limit Toggle Coverage

To disable toggle coverage for a variable or a net, enter the pragma before the variable or net declaration. The allowed pragmas are:

// VCS coverage off

// VCS coverage on

All code between the off and on is excluded for all code coverage metrics. In the case of toggle coverage, the pragmas should block out the reg / wire declaration and not the line that updates the reg or wire.

Example 2-18 shows how this works:

Example 2-18   Pragmas for Toggle Coverage

```
module test;

//VCS coverage off
reg r1;
```

```
//VCS coverage on
reg r2,r3;
//VCS coverage off
wire w1;
//VCS coverage on
wire w2,w3;

initial
begin
#1 r1=1; //pragmas
   r2=1;
   r3=1;
#1 r1=0;
   r2=0;
   r3=0;
#1 r1=1;
   r2=1;
   r3=1;
#1 r2=0;
#1 r2=1;
#100 $finish;
end

assign w1=r1; // pragmas
assign w2=r2;
assign w3=r3;

endmodule
```

Example 2-18 contains pragmas before the reg `r1` and wire `w1` declarations, in order to exclude them from the toggle coverage.

The toggle coverage report shows reg `r2` and `r3` and wire `w2` and `w3`. It does not report reg `r1` or wire `w1`.

Note:

This feature works only with Verilog code.

If you enter any of these pragmas in your source code, but some time later want VCS to ignore these pragmas, enter the -cm_ignorepragmas compile-time option.

Pragmas to Limit FSM Coverage

You can use pragmas to specify an FSM that VCS might not automatically extract.

Note:
   Pragmas do not apply to VHDL.

With pragmas, you can inform VCS about the following FSM matter:

- The vector signal, part-select of a vector signal, or concatenation of signals that hold the current state of the FSM.

- The vector signal that holds the next state of the FSM, unless the FSM does not use a signal for the next state (in which case VCS displays a warning and assumes that the current state and next state are in the same signal, part-select of a signal, or concatenation of signals).

- The possible states of the FSM that are specified in a parameter declaration. When using pragmas to specify an FSM, there must be a parameter declaration to specify the possible states of the FSM.

Specifying the Signal That Holds the Current State

You use the following pragma to identify the vector signal that holds the current state of the FSM:

```
/* VCS state_vector signal_name */
```

`VCS` and `state_vector` are required keywords. You must enter this pragma inside the module definition where the signal is declared.

You also must use a pragma to specify an enumeration name for the FSM. This enumeration name is also specified for the next state and the possible states, associating them with each other as part of the same FSM. There are two ways you can do this:

- Use the same pragma:

  ```
  /* VCS state_vector signal_name enum enumeration_name */
  ```

- Use a separate pragma in the signal's declaration:

  ```
  /* VCS state_vector signal_name */
  reg [7:0] /* VCS enum enumeration_name */ signal_name;
  ```

In either case, `enum` is a required keyword. If using the separate pragma, `VCS` is also a required keyword. Also, when using a separate pragma, enter the pragma immediately after the bit range of the signal.

Specifying the Part-Select that Holds the Current State

You can specify that a part-select of a vector signal holds the current state of the FSM. Normally, when URG displays or reports FSM coverage data, it names the FSM after the signal that holds the current state. In your FSM, if a part-select holds the current state, you must also specify a name for the FSM that URG can use. The FSM name is not the same as the enumeration name.

You can specify the part-select with the following pragma:

```
/* VCS state_vector signal_name[n:n] FSM_name enum
enumeration_name */
```

Specifying the Concatenation that Holds the Current State

Like specifying a part-select, you can specify a concatenation of signals to hold the current state. When you do this, you also need to specify an FSM name and an enumeration name:

```
/* VCS state_vector {signal_name, signal_name,...} FSM_name
enum enumeration_name */
```

The concatenation is of the entire signals. You cannot include bit-selects or part-selects of signals.

Specifying the Signal that Holds the Next State

You also specify the signal that holds the next state of the FSM with the pragma that specifies the enumeration name:

```
reg [7:0] /* VCS enum enumeration_name */ signal_name
```

If, and only if, the FSM does not have a signal for the next state, you can omit this pragma.

Specifying the Current and Next State Signals in the Same Declaration

If you use the pragma for specifying the enumeration name in a declaration of multiple signals, VCS assumes that the first signal following the pragma holds the current state and the next signal holds the next state. For example:

```
/* VCS state_vector cs */
reg [1:0] /* VCS enum myFSM */ cs, ns, nonstate;
```

In this example, VCS assumes that signal `cs` holds the current state
and signal `ns` holds the next state. It assumes nothing about signal
`nonstate`.

Specifying the Possible States of the FSM

You can also specify the possible states of the FSM with the pragma
that specifies the enumeration name:

```
parameter /* VCS enum enumeration_name */
    S0 = 0,
    s1 = 1,
    s2 = 2,
    s3 = 3;
```

Enter this pragma immediately after the keyword `parameter`,
unless you specify a bit width for the parameters. If you do specify a
bit width, enter the pragma immediately after the bit width:

```
parameter [1:0] /* VCS enum enumeration_name */
    S0 = 0,
    s1 = 1,
    s2 = 2,
    s3 = 3;
```

Pragmas in One Line Comments

These pragmas work in both block comments, between the `/*` and
`*/` character strings, and in one line comments, following the `//`
character string. For example:

```
parameter [1:0] // VCS enum enumeration_name
    S0 = 0,
```

```
       s1 = 1,
       s2 = 2,
       s3 = 3;
```

Specifying FSM With Pragmas - an Example

```
module m3;
                                    Signal ns holds the next state
   reg[31:0] cs;
   reg[31:0] /* VCS enum MY_FSM */ ns;
   reg[31:0] clk;
   reg[31:0] rst;           signal cs holds the current state

// VCS state_vector cs enum MY_FSM

   parameter // VCS enum MY_FSM
            p1=10,
            p2=11,
            p3=12;

  endmodule // m3              p1, p2, and p3 are possible
                              states of the FSM
```

Using Pragmas to Limit Branch Coverage

The //VCS coverage exclude_file and
//VCS coverage exclude_module pragmas exclude a source
file or module definition from branch coverage.

Additionally, you can use the //VCS coverage off and //VCS
coverage on pragmas to exclude certain code from branch
coverage.

When you use if-else statements as in Example 2-19:

Example 2-19   If Statements in a Design

```
always @ (r1 or r2 or r3)
begin
if (r1)
   begin
     $display("r1 is true");
     r4=r1;
   end
else
   begin
     $display("r1 not true");
     if (r2)
        begin
          $display("r2 is true");
          r5=r2;
        end
     else
        begin
          $display("r2 not true");
          if (r3)
             begin
               $display("r3 is true");
               r6=r3;
             end
          else
             $display("r3 not true");
        end
   end
$display("no op");
end
```

URG reports a column for each conditional expression as in
Example 2-20:

Example 2-20   Branch Coverage Report

```
23       if (r1)
             -1-
24          begin
25            $display("r1 is true");
26            r4=r1;
```

```
27              end
28          else
29              begin
30                  $display("r1 not true");
31                  if (r2)
                        -2-
32                      begin
33                          $display("r2 is true");
34                          r5=r2;
35                      end
36                  else
37                      begin
38                          $display("r2 not true");
39                          if (r3)
                                -3-
40                              begin
41                                  $display("r3 is true");
42                                  r6=r3;
43                              end
44                          else
45                              $display("r3 not true");
46                      end
47          end


BRANCH          -1-   -2-   -3-
                 1     -     -         | Covered
                 0     1     -         | Covered
                 0     0     1         | Covered
                 0     0     0         | Covered
```

This examples includes branches starting at `r1` and ending at `r1`, `r2`, or `r3`.

You can use these pragmas to exclude one of the conditional expressions for a `if-else` statement as shown in Example 2-21:

Example 2-21   Excluding a Conditional Expression for `if-else` *statement*

```
always @ (r1 or r2 or r3)
begin
```

```
        if (r1)
           begin
             $display("r1 is true");
             r4=r1;
           end
//VCS coverage off
        else
           begin
             $display("r1 not true");
             if (r2)
                begin
                  $display("r2 is true");
                  r5=r2;
                end
             else
                begin
                  $display("r2 not true");
//VCS coverage on
                  if (r3)
                     begin
                       $display("r3 is true");
                       r6=r3;
                     end
                  else
                     $display("r3 not true");
                end
           end
        $display("no op");
        end
```

In this case, URG reports on only the branch starting and ending at
r1. There is no new branch starting at r3 because the if statement
for r2 is excluded from coverage by the //VCS coverage off
pragma.

You can also use pragmas (see Example 2-22) to exclude code that
does not affect branch coverage, that is, the branches that VCS
identify as unchanged. For example:

Example 2-22   Using Pragmas Without Affecting Branch Coverage

```
always @ (r1 or r2 or r3)
begin
if (r1)
   begin
     $display("r1 is true");
     r4=r1;
   end
else
   begin
//VCS coverage off
     $display("r1 not true");
//VCS coverage on
     if (r2)
        begin
          $display("r2 is true");
          r5=r2;
        end
     else
        begin
          $display("r2 not true");
          if (r3)
             begin
               $display("r3 is true");
               r6=r3;
             end
          else
             $display("r3 not true");
        end
   end
$display("no op");
end
```

When you use case statements as shown in ,

Example 2-23   Using case Statements

```
always @ (r1 or r2 or r3)
case (r1)
    1      : case (r2)
               1     : r4=1;
```

```
                 0      :  r4=0;
              default :  $display("r4 not assigned");
           endcase
   0        : case (r3)
              1      :  r5=0;
              0      :  r5=1;
           default :  $display("r5 not assigned");
           endcase
default  : $display("no op");
endcase
URG includes a column for each case expression:

5                 case (r1)
                  -1-
6                 1 : case (r2)
                          -2-
7                     1 : r4=1;
                      ==>
8                     0 : r4=0;
                      ==>
9                   default : $display("r4 not assigned");
                      ==>
10                endcase
11                0 : case (r3)
                          -3-
12                    1 : r5=0;
                      ==>
13                    0 : r5=1;
                      ==>
14                  default : $display("r5 not assigned");
                      ==>
15                endcase
16                default : $display("no op");
                  ==>
```

Branches:

-1-	-2-	-3-	Status
1	1	-	Not Covered
1	0	-	Not Covered
1	default	-	Not Covered
0	-	1	Not Covered

```
0        -       0        Not Covered
0        -       default  Not Covered
default  -       -        Not Covered
```

However, when you use pragmas to exclude a case statement, as shown in Example 2-24, VCS identify no branches for that case expression (see Example 2-25):

Example 2-24   Using Pragmas to Exclude case Statements

```
case (r1)
    1        : case (r2)
                  1      : r4=1;
                  0      : r4=0;
                default : $display("r4 not assigned");
              endcase
//VCS coverage off
    0        : case (r3)
                  1      : r5=0;
                  0      : r5=1;
                default : $display("r5 not assigned");
              endcase
//VCS coverage on
  default  : $display("no op");
endcase
```

Example 2-25   URG Output When case Statements are Excluded With Pragmas

```
5                case (r1)
                 -1-
6                1 : case (r2)
                     -2-
7                    1 : r4=1;
                     ==>
8                    0 : r4=0;
                     ==>
9                default : $display("r4 not assigned");
                     ==>
10               endcase
11               //VCS coverage off
```

```
12                   0 : case (r3)
13                         1 : r5=0;
14                         0 : r5=1;
15                     default : $display("r5 not assigned");
16                   endcase
17                   //VCS coverage on
18                   default : $display("no op");
                     ==>
```

```
Branches:

-1-      -2-       Status
1        1         Not Covered
1        0         Not Covered
1        default   Not Covered
default  -         Not Covered
```

When you use the ternary operator as shown in Example 2-26:

Example 2-26   Using the Ternary Operator

```
assign w1 = (r1==1) ?
                 ((r2==1) ?
                       ((r4==1) ? r6 : r7)
                       : r5)
                 : r3;
```

URG reports a column for each conditional expression (see Example 2-27)

Example 2-27   URG Output for a Ternary Operator

```
5            assign w1 = (r1==1) ?
                             -1-
                             ==>
6            ((r2==1) ?
              -2-
               ==>
7            ((r4==1) ? r6 : r7)
               -3-
               ==>
```

```
Branches:

-1- -2- -3- Status
1   1   1   Not Covered
1   1   0   Not Covered
1   0   -   Not Covered
0   -   -   Not Covered
```

If you use these pragmas to exclude one of the conditional expressions for a ternary operator, as in Example 2-28,

Example 2-28   Excluding a Conditional Operator for the Ternary Operator

```
assign w1 = (r1==1) ?
//VCS coverage off
((r2==1) ?
//VCS coverage on
((r4==1) ? r6 : r7)
            : r5)
            : r3;
```

URG still reports the column for the conditional expression, as in Example 2-29:

Example 2-29   URG Output When a Ternary Operator is Excluded with Pragmas

```
5          assign w1 = (r1==1) ?
                        -1-
                        ==>
6          //VCS coverage off

7          ((r2==1) ?
             -2-
             ==>
8          //VCS coverage on

9          ((r4==1) ? r6 : r7)
             -3-
```

```
                ==>
                ==>

Branches:

-1- -2- -3- Status
1   1   1   Not Covered
1   1   0   Not Covered
1   0   -   Not Covered
0   -   -   Not Covered
```

However, if you use these pragmas to exclude all the ternary operators and conditional expressions, as shown in Example 2-30, VCS does not compile for or monitor, and URG does not report, these ternary operators and their conditional expression for branch coverage:

Example 2-30   Excluding All Conditional Operators for the Ternary Operator

```
//VCS coverage off
assign w1 = (r1==1) ?
                ((r2==1) ?
                    ((r4==1) ? r6 : r7)
                    : r5)
                : r3;
//VCS coverage on
```

If you enter any of these pragmas in your source code, and at later time, want VCS to ignore these pragmas, enter the `-cm_ignorepragmas` compile-time option.

Using Glitch Suppression

Signals often go through many oscillations before they settle down for an event. Such transitions on a signal to a value for a short period of time, or even transitioning to a value and then another value during the same simulation time step can cause transitions or toggles on other signals and cause some lines to execute, and perhaps execute more than once, during the same time step.

These oscillations, glitches, or narrow pulses can skew the coverage results in line, condition or toggle coverage, by showing lines, conditions, or toggles that were covered only when these glitches occur and not after these oscillating signals settle down.

To prevent this, you can use the `-cm_glitch` compile-time option as follows:

```
vcs -cm line+cond+tgl -cm_glitch period
```

```
Specify the period with a non-negative integer.
```
When you use this switch for Line coverage, VCS ignores the coverage in the first 'n' time units even though there are no glitches. This is to ignore any glitches that might be happening during the initial few timestamps due to the design setup.

You can use the `-cm_glitch 0` option to remove the delta cycle glitches. For example for

```
assign  na = !a;
assign   x = a ^ na;
```

The initial value of x is 1'b1. With change in signal 'a' at time T, x will temporarily become 1'b0 and will be back to 1'b1. All this occurs in the same time T. To remove such potential unnecessary glitch, `-cm_glitch 0` can be used. However, this will ensure that any glitch that has a non-0 period is not filtered out.

The simulation time that you specify is similar to the simulation time you specify in a delay specification (`#10` for example). It is scaled based on the `time_unit` and `time_precision` arguments to the `` `timescale `` compiler directive.

The `-cm_glitch` option is also a runtime option, but only works for toggle coverage.

Glitch suppression works slightly differently in line, condition, and toggle coverage, as explained in the following sections.

Line Coverage Glitch Suppression

In glitch suppression for line coverage, VCS looks for fast triggering of `always` blocks, not initial blocks, and the glitch period is an interval of simulation time in which there are one or more executions of the statements in the `always` block.

When using glitch suppression, VCS only monitors and records the results on the last execution of the `always` block during the glitch period.

The `-cm_glitch` *positive_integer* compile-time option specifies the simulation time period of glitch suppression.

Note:
   Glitch suppression does not work for VHDL code.

Figure 2-2 Line Coverage Glitch Period

```
-cm_glitch 5
```
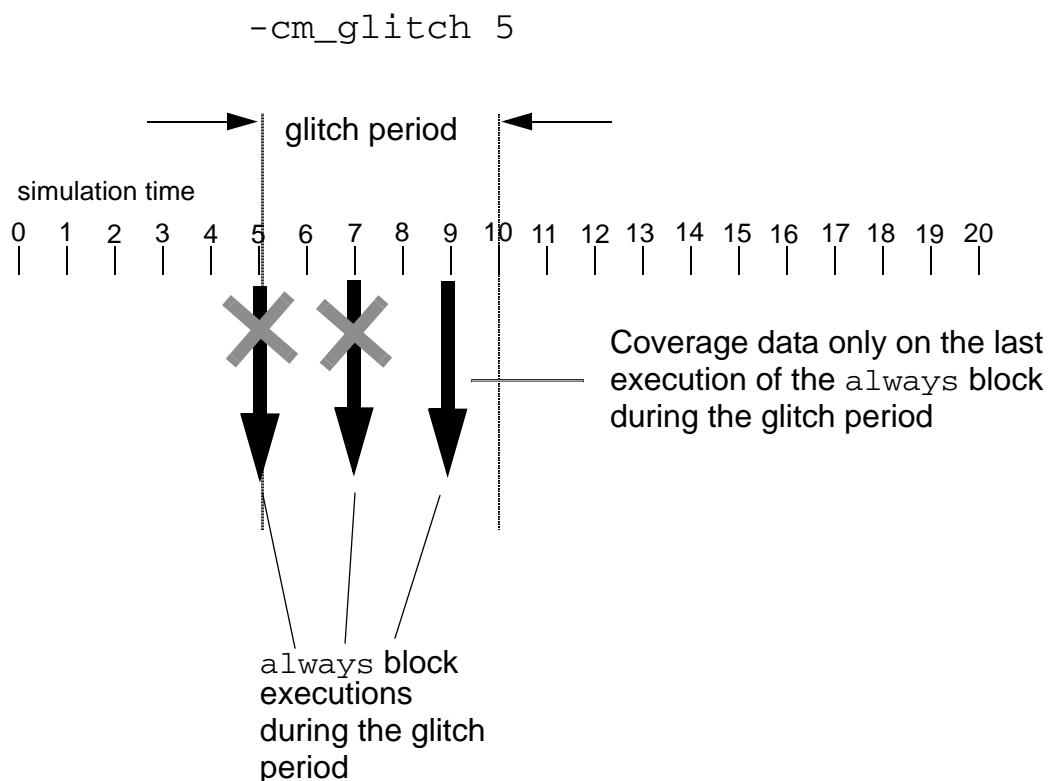

Consider the following example:

```
module dev (output1,input1,input2,input3);
output output1;
input input1,input2,input3;
reg output1;
.
.
.
always @ (input1 or input2 or input3)
if (input1 && input2)
    output1=input3;
else
    output1=~input3;
.
```

```
.
.
endmodule
```

Glitches, or narrow pulses, on the input ports of this module could cause this `always` block to trigger (or execute) several times, while little or no simulation time elapses, before the values on the input ports settle down and there is a final execution of the `always` block.

In this example, the conditional expression `(input1 && input2)` could be briefly true during one of these glitches or narrow pulses, but after settling down the expression is finally not true. If this happens, when VCS monitors instances of this module for line coverage, it sees that the lines controlled by the `if` part and the `else` part of this conditional expression, both assignment statements, are executed.

With glitch suppression, VCS does not monitor this code for line coverage during the entire length of these glitches or narrow pulses, and in this example only, VCS sees the execution of the second assignment statement in the final execution of the `always` block.

Limitation on Clocks

Line coverage interprets an event control with the `posedge` or `negedge` keywords as specifying a clock and does not suppress glitches, or narrow pulses, when you include the `-cm_glitch` compile-time option. Consider the following line numbered example:

```
1 module test;
2 reg r1;
3 initial
4 begin
5   r1=0;
6   #5 r1=1;
7   #1 r1=0;
```

```
8   #10 $finish;
9 end
10
11 dev dev1 (w1,r1);
12
13 endmodule
14
15 module dev (out,in1);
16 output out;
17 input in1;
18 reg out;
19 always @ (posedge in1 )
20 if (in1 )
21     out=in1; //Is this line covered?
22 else
23     out=~in1;
24 endmodule
```

Line coverage interprets signal `in1` in line 19 as a clock because the event control that controls the execution of the `always` block (sometimes called the sensitivity list of the `always` block) includes the keyword `posedge`.

If you entered the following command line:

```
vcs example.v -cm line -cm_glitch 1
```

You might expect line 21 not to be covered, because VCS executes it when `in1` is true, and as you can see from lines 6 and 7 that `in1` will be true for only one time unit, which is the glitch period specified on the command line.

However, line coverage does not suppress glitches on clocks, therefore, line 21 will be reported as covered even when you specify glitch suppression with the `-cm_glitch` compile-time option.

Toggle Coverage Glitch Suppression

The `-cm_glitch` *positive_integer* compile-time option specifies the simulation time period of glitch suppression.

Unlike glitch suppression in line and condition coverage, glitch suppression in toggle coverage is not focused on rapidly triggering always blocks. Glitch suppression in toggle coverage indicates that VCS does not monitor or record any value change on a signal that lasts fewer simulation time units than the simulation time specified with the `-cm_glitch` compile-time option. It does not matter whether these short interval changes, or glitches, are caused by statements in an always or initial block, continuous assignments, or connections to a gate, primitive, or user-defined primitive (UDP).

Another difference between glitch suppression for toggle coverage and glitch suppression for line or condition coverage, is that there is a `-cm_glitch` runtime option for toggle coverage. This runtime option overrides the glitch period for toggle coverage specified at compile-time. This runtime option does not work for either line or condition coverage.

Glitch suppression in toggle coverage is not necessary for zero time glitches, that is transitions from $0 \rightarrow 1$ and then $1 \rightarrow 0$, or from $1 \rightarrow 0$ and then $0 \rightarrow 1$, during the same simulation time step. VCS never monitors or records these transitions for toggle coverage.

Note:
> This feature works only with Verilog code.

Using Condition Coverage Glitch Suppression

The `-cm_glitch` *positive_integer* compile-time option specifies the simulation time period of glitch suppression.

In glitch suppression for condition coverage, like glitch suppression for line coverage, VCS looks for fast triggering of always blocks, not initial blocks, and the glitch period is an interval of simulation time in which there are one or more executions of the statements in the always block.

Note:
    Glitch suppression does not work for VHDL code.

When you use glitch suppression, VCS only monitors and records the results on the last execution of the always block during the glitch period.

Consider the following example:

```
module dev (out,in1,in2,in3,in4);
output out;
input in1,in2,in3,in4;
reg out;
.
.
.
always @ (in1 or in2 or in3)
if (in4)
    out=in1 && in2 && in3;
else
    out=~in4;
.
.
.
endmodule
```

Glitches or narrow pulses on the input ports could cause the always block to execute several times, while little or no simulation time elapses, before the values on the input ports settle and there is a final execution of the always block.

In this example, the default condition coverage is on the assignment statement that assigns an expression with the logical AND `&&` operator:

```
out=in1 && in2 && in3;
```

These glitches, although momentary, by default would show a high amount of coverage. For example:

```
LINE 29

 EXPRESSION (in1 && in2 && in3)
             -1-     -2-     -3-
1 0 1 | Covered
1 1 0 | Covered
1 1 1 | Covered
```

Including the `-cm_glitch` compile-time option, specifying a short simulation time interval, but longer than the glitches or narrow pulses, changes the coverage to the following:

```
LINE 29

 EXPRESSION (in1 && in2 && in3)
             -1-     -2-     -3-
-1- -2- -3-
0 1 1 | Not Covered
1 0 1 | Not Covered
1 1 0 | Not Covered
1 1 1 | Not Covered
```

3

User-defined Coverage System Functions

This chapter includes the following sections:

- "Coverage System Functions"

- "The $cm_coverage System Function"

- "The $cm_get_coverage System Function"

- "The $cm_get_limit System Function"

- "Examples"

- "Accessing Coverage Data During Simulation Using UCAPI"

Coverage System Functions

VCS has system functions for a real time API that enable you to write test fixture modules that can do the following:

- Find out what type of coverage VCS is running for part of a design

- Start or stop a type of coverage for part of a design

- Determine how close you are to recording all the coverage data that you can in a simulation run for a part of the design.

The parts of the design for which you can determine this information or start or stop collecting coverage information can be any of the following:

Note:

   You can specify multiple parts of the design.

- All instances of a specific module definition

- All instances of a specific module definition and all module instances hierarchically under these module instances

- A specific module instance

- A specific module instance and all module instances hierarchically under this module instance

The user-defined system functions that enable you to do these things are as follows:

`$cm_coverage`

> Starts or stops monitoring for a particular type of coverage or returns a value telling you what type of coverage data VCS is gathering for a particular part of the design.
> See "The $cm_coverage System Function" .

`$cm_get_coverage`

> Used with `$cm_get_limit`, its return value represents how much of a specific type of coverage data VCS has gathered so far. See "The $cm_get_coverage System Function" .

`$cm_get_limit`

> Used with `$cm_get_coverage`, its return value represents the maximum amount of a specific type of coverage data VCS can gather for part of a design. See "The $cm_get_limit System Function" .

You use the `$cm_coverage` system function to both determine what type of coverage information VCS is gathering and to enable or disable gathering that information.

You can use the `$cm_get_coverage` and `$cm_get_limit` system functions together, comparing their return values, to see how close you are to gathering all the coverage information you can. There are arguments for specifying the type of coverage and for looking at how close you are for only part of the design.

The arguments to these system functions are for the most part integer values. Synopsys provides a source file that defines text macros for these integer values so that you can more easily see what you are specifying in these system functions. For example:

```
$cm_coverage(3,4,11,"top.dev1");
```

and

```
$cm_coverage('CM_CHECK,'CM_FSM,'CM_HIER,"top.dev1");
```

Both specify checking to see if VCS is gathering FSM coverage for the part of the design that is hierarchically under module instance top.dev1. The second example is a lot more obvious because it uses the macros defined in the $VCS_HOME/include/CoverMeter.vh file.

To use the text macros in the CoverMeter.vh file, do the following:

- Specify including the file with the `include compiler directive:

  ```
  `include "CoverMeter.vh"
  ```

- Tell VCS where to look for this file with the +incdir compile-time option:

  ```
  +incdir+$VCS_HOME/include
  ```

Note:
  These system functions do not work if you do not enable monitoring for coverage at runtime with the -cm runtime option.

The $cm_coverage System Function

The `$cm_coverage` system function starts or stops VCS from monitoring for a particular type of coverage, or returns a value telling you what type of coverage data VCS is gathering, for a particular part of the design. Its syntax is as follows:

```
$cm_coverage(mode,type,include_hierarchy,
"module_or_instance",...)
```

Here:

mode

Specifies starting, stopping, or checking for a certain type of coverage. These modes, their text macros, and their integer equivalents are as follows:

`CM_START`	1	Specifies starting the type of coverage specified in the *type* argument for the part or the design specified in the *include_hierarchy* and *module_or_instance* argument.
`CM_STOP`	2	Specifies stopping the type of coverage specified in the *type* argument for the part or the design specified in the *include_hierarchy* and *module_or_instance* argument.
`CM_CHECK`	3	Specifies looking to see if VCS is gathering coverage information for the type of coverage specified in the *type* argument for the part or the design specified in the *include_hierarchy* and *module_or_instance* argument. The return value from this system function, when you specify this mode, tells you whether VCS is gathering this information.

type

Specifies the type of coverage you want to start, stop, or check for. These types, their text macros, and their integer equivalents are as follows:

`CM_SOURCE`	1	Line coverage
`CM_CONDITION`	2	Condition coverage
`CM_TOGGLE`	3	Toggle coverage
`CM_FSM`	4	FSM coverage

include_hierarchy

Specifies whether or not you want to also start, stop, or check for the type of coverage in the design hierarchy under all instances of the module or modules you specify in the `module_or_instance` argument.

The text macros and integer equivalents that you can enter for this argument are as follows:

`CM_MODULE`	10	Specifies *not* also starting, stopping, or checking for coverage in the design hierarchy under the instances you specify in the `module_or_instance` argument.
`CM_HIER`	11	Specifies also starting, stopping, or checking for coverage in the design hierarchy under the instances you specify in the `module_or_instance` argument.

module_or_instance

The module identifier (or name) of a module or the hierarchical name of a module instance. When you specify a module identifier, you specify all instances of this module.

Always enclose this argument in quotation marks.
You can specify more than one module identifier or module instance. If you do, separate these arguments with commas and enclose each in quotation marks, for example:

`"module_identifier","module_instance_hierarchcal_name"`

Consider the following matrix for the *include_hierarchy* and *module_or_instance* arguments:

	Specify a module identifier in the *module_or_instance* argument	Specify a hierarchical name for a module identifier in the *module_or_instance* argument
Specify `CM_MODULE or its equivalent integer for the *include_hierarchy* argument	Applies to all instances of the specified module but none of the module instances hierarchically under these instances.	Applies only to the specified instance
Specify `CM_HIER or its equivalent integer for the *include_hierarchy* argument	Applies to all instances of the specified module and all the module instances hierarchically under these instances.	Applies to the specified instance and all module instances hierarchically under the specified instance.

Return Values

The $cm_coverage system function returns positive or negative integer values. The meaning of these values is determined by the *mode* argument. The $VCS_HOME/include/CoverMeter.vh file also contains text macros for these return values.

When the *mode* argument is `CM_CHECK the text macros for, integer values of, and meaning of the return values are as follows:

`CM_NOERROR	0	VCS is gathering the type of coverage information specified with the *type* argument for the module instances specified with the *include_hierarchy* and *module_or_instance* arguments.
`CM_ERROR	-1	There is an invalid argument.
`CM_NOCOV	-2	There is no coverage data of the type specified for the instances specified.
`CM_PARTIAL	-3	There is coverage data of the type specified for some, but not all, of the instances specified

When the *mode* argument is `CM_START the text macros for, integer values of, and meaning of the return values are as follows:

`CM_NOERROR	0	VCS has started gathering the type of coverage information specified with the *type* argument for the module instances specified with the *include_hierarchy* and *module_or_instance* arguments.
`CM_ERROR	-1	There is an invalid argument.
`CM_NOCOV	-2	VCS cannot gather the type of coverage information specified with the *type* argument. Usually this is because the source code is not compiled for the type of coverage specified.
`CM_PARTIAL	-3	VCS cannot gather the type of coverage information specified with the *type* argument for all instances specified, but it can for some. Usually this means only some of the specified instances are not compiled for this type of coverage, while other specified instances are. The coverage data collected will be no different than if you specified only the instances compiled for the specified coverage type.

When the *mode* argument is `CM_STOP the text macros for, integer values of, and meaning of the return values are as follows:

`CM_NOERROR	0	Coverage successfully stopped
`CM_ERROR	-1	There is an invalid argument.

The $cm_get_coverage System Function

You can use the $cm_get_coverage system function to return a value that you compare with the return value from the $cm_get_limit system function to determine how close you are to gathering all the coverage data you can gather in the current simulation, of a specified coverage type, and from a specified part of the design.

The return value from this system task is an integer representing how much of the specified type of coverage information VCS has gathered so far for the specified part of the design.

The syntax of the $cm_get_coverage system function is as follows:

```
$cm_get_coverage(type, include_hierarchy,
"module_or_instance",...)
```

Here:

type

Specifies the type of coverage. These types, their text macros, and their integer equivalents are as follows:

`CM_SOURCE`	1	Line coverage
`CM_CONDITION`	2	Condition coverage
`CM_TOGGLE`	3	Toggle coverage
`CM_FSM`	4	FSM transition coverage
`CM_FSM_TRANS`	4	FSM transition coverage (same as `CM_FSM)
`CM_FSM_STATE`	5	FSM state coverage

include_hierarchy

> Specifies whether you want to see how close you are to gathering all the coverage information you can in the design hierarchy under the module instances you specify in the *module_or_instance* argument.

> The text macros and their integer equivalents for specifying whether to include the subhierarchies under these instances are as follows:

`CM_MODULE	10	Specifies *not* including in the design hierarchy under the instances or instance of the module or instance you specify in the *module_or_instance* argument.
`CM_HIER	11	Specifies including the design hierarchy under the instances or instance of the module or instance you specify in the *module_or_instance* argument.

module_or_instance

> The module identifier (or name) of a module or the hierarchical name of a module instance.

Note:

> Less detail is provided here for the *include_hierarchy* and *module_or_instance* arguments than is provided for the $cm_coverage system function. This is to avoid repeating a lot of details because these arguments work the same way for this system function as for the $cm_covergage system function. You use these arguments to specify, for which part of the design, the operation (in this case determining how much coverage data VCS has gathered so far) applies.

Return Values

If there are no error conditions, the `$cm_get_coverage` system function returns either a zero or a positive integer value. You compare this return value with the return value of the `$cm_get_limit` system function. When they return matching positive integer values for a specific type of coverage and for a matching part of the design, then the current simulation run yields no more additional coverage data. Without this comparison, which positive integer is returned from this system function has no particular significance.

When there is an error this system function returns negative integer values. The text macros for, integer values of, and meaning of the return values are as follows:

`CM_ERROR`	-1	There is an invalid argument.
`CM_NOCOV`	-2	The specified type of coverage data is not available in this simulation run.

The $cm_get_limit System Function

You use the `$cm_get_limit` system function to return a value that you compare with the return value from the `$cm_get_coverage` system function to determine how close you are to gathering all the coverage data you can gather in the current simulation of a specified coverage type and from a specified part of the design.

The return value from this system task is a integer representing how much of the specified type of coverage information, for the specified part of the design, VCS could gather during the simulation. In contrast, the return value of the `$cm_get_coverage` system function is an integer representing how much coverage information of the specified type VCS has gathered so far.

The syntax of the `$cm_get_limit` system function is, with the exception of the system function name, identical to the syntax of the `$cm_get_coverage` system function.

```
$cm_get_limit(type, include_hierarchy,
"module_or_instance",...)
```

The arguments, their text macros, and integer values are identical to those of the `$cm_get_coverage` system function. When you use these two system functions as intended, you use the same arguments to specify the type of coverage, whether to include the design hierarchy under the specified module definitions and instance hierarchical names and matching module definitions and instance hierarchical names.

Examples

Example 3-1 shows how to use this API to stop simulation when you have sufficient coverage data. Example 3-2 show how to monitor how close you are to full coverage data.

Example 3-1   Stopping Simulation When You Have Sufficient Coverage Data

```
`timescale 1 ns / 1 ns

module top;
reg r1,r2;
wire w1, w2;
integer coverage_on_or_off,check_coverage,get_coverage,get_limit;

design design1 (w1,r1);

initial
begin
   #0 get_limit=$cm_get_limit(`CM_SOURCE,`CM_MODULE,"top.design1");
   #5 coverage_on_or_off= $cm_coverage(`CM_STOP,`CM_SOURCE,`CM_HIER,"top");
   #5 coverage_on_or_off=
                $cm_coverage(`CM_START,`CM_SOURCE,`CM_MODULE,"top.design1");
      r1=0;
   #1000 $finish;
end

always
begin
   #25 r1=~r1;
   get_coverage=$cm_get_coverage(`CM_SOURCE,`CM_MODULE,"top.design1");
   if (get_coverage>=get_limit*0.9)
      $finish;
end

endmodule
```

In Example 3-1, the initial block does the following:

1. Assigns to integer `get_limit` the return value of
    `$cm_get_limit`, which represents how much line coverage
    data for module instance top.design1 VCS could gather during
    the simulation.

2. Turns off line coverage for the entire design after 5 time units. The
    delay is specified so that turning off line coverage happens after
    it is started by the CoverMeter.tasks file.

3. Turns on line coverage 5 time units after that but only for module instance top.design1.

4. Initializes a stimulus reg.

5. Schedules simulation to end at simulation time 1010.

Also in , the always block does the following:

1. Toggles the stimulus reg.

2. Assigns to integer `get_coverage` the return value of `$cm_get_coverage`, which represents how much line coverage data for module instance top.design1 VCS has gathered so far.

3. Looks to see if the integer assigned to `get_coverage` is greater than or equal to 90% of the integer assigned to `get_limit`, and if it is, terminates the simulation.

Example 3-2   Displaying How Close You Are To Full Coverage Data

.

.

.

```
initial
begin
        covdata = $cm_coverage('CM_CHECK, 'CM_SOURCE, 'CM_HIER, "i1");
        $display($time,,"Source coverage check: %d", covdata);
    #1; // wait until after the coverage stuff is initialized
        covdata = $cm_coverage('CM_CHECK, 'CM_SOURCE, 'CM_HIER, "i1");
        $display($time,,"Source coverage check: %d", covdata);
        covdata = $cm_get_limit('CM_SOURCE, 'CM_HIER, "i1");
        $display($time,,"Source limit : %d", covdata);
    #999;
        forever
          begin
            covdata = $cm_get_coverage('CM_SOURCE, 'CM_HIER, "i1");
```

In Example 3-2, the initial block does the following:

1. At time 0, assigns to integer `covdata` the return value of the `$cm_coverage` system function that is checking to see if VCS is gathering line coverage information about module instance `i1` and the hierarchy under `i1`.

2. Displays the return value.

3. Waits 1 time unit and then once again assigns to integer `covdata` the return value of the `$cm_coverage` system function that is checking for the same thing.

4. Displays the return value again. This value should be different now that the CoverMeter.tasks file has started VCS for line coverage.

5. Assigns to integer `covdata` the return value of the `$cm_get_limit` system function for the same type of coverage and the same part of the design.

6. Displays the return value from the `$cm_get_limit` system function.

7. After 999 time units begins a forever loop that does the following:

   Assigns to integer `covdata` the return value of the `$cm_get_coverage` system function for the same type of coverage and the same part of the design.

   Displays this return value.

   Waits 100,000 time units before beginning the loop again.

Accessing Coverage Data During Simulation Using UCAPI

You can access covergroup coverage data during simulation using C code. You need to add `$coverage_dump` and `$coverage_reset` system tasks to the Verilog code to perform the following tasks:

- Monitor the coverage status periodically and modify external stimuli.

- Clear the coverage data from the simulation state so that any further data collected starts from scratch.

Monitoring the Coverage Data

You can monitor the coverage data during simulation as follows:

• Invoke a system task inside Verilog to dump the current coverage status to disk.

• Open the dumped database using UCAPI functions and examine the coverage status.

The system task $coverage_dump atomically dumps all coverage data to the specified test name. You can give a simple test name, mytest, or a composite name mydir/mytest, and specify where the data will be dumped. For example, a Verilog program could periodically dump the coverage status as follows:

```
always@(dump_me) begin
      $coverage_dump("mydir/myfile");
      $check_my_coverage(…);
end
```

Using this syntax, you can toggle the signal dump_me to cause the current coverage status to be dumped to the mydir directory. The snapshot of data that UCAPI calls will be given the name, myfile. In this example, mydir/mytest is overwritten whenever dump_me is toggled, but you can write a code to give a different name each time based on a counter.

You can also call $coverage_dump directly using DPI if it is wrapped in a Verilog task.

```
export "DPI" task dump_cov_data(string s);
task dump_cov_data(string s);
$coverage_dump(s);
endtask
```

After the dump is complete, the PLI function (in this example, `check_my_coverage`) will be invoked, and you can use UCAPI to open the dumped database, check the coverage status, and take appropriate action. You can use UCAPI to load the data as follows:

```
covdbHandle design = covdb_load(covdbDesign, NULL,
"mydir"); covdbHandle test = covdb_load(design, design,
"mydir/myfile");
```

From here, the application can iterate through all covergroups, or walk over the design hierarchy. The `$coverage_dump` operation is automatic, and there is no opening or closing of a database. When `$coverage_dump` is called, the specified database is opened, written, and closed, in one automatic operation.

Note:

> As soon as a database is dumped using `$coverage_dump`, it is accessible. Do not attempt to access the database until the `$coverage_dump` operation is complete.

Resetting the Coverage Data

You can reset the accumulated coverage data using the system task, `$coverage_reset`. The system task, `$coverage_reset`, clears all coverage data for all covergroups in the current simulation. For example:

```
always
begin
#10000
   $coverage_dump("mydir/myfile");
   if    ($check_my_coverage(…))
   $coverage_reset();
end
```

If the PLI function, `$check_my_coverage`, returns `true`, all functional coverage data is reset, as if nothing had been covered up to this point. To dump an empty coverage database, where no bins are marked covered, you can write as follows:

```
$coverage_reset(); $coverage_dump("mydir/myfile");
```

The `$coverage_reset` system task does not remove any covergroups from the current simulation run. It just resets all the bins' hit counts to `0`. You can only overwrite existing databases. However, if you want additional data to be merged with an existing database, you can write it with a new name as follows:

```
$coverage_dump.

$coverage_dump("mydir/mytest1"); … $coverage_reset(); …
$coverage_dump("mydir/mytest2");
```

This code writes two different databases to disk – mydir/mytest1 and mydir/mytest2. You can use UCAPI to load the first one with covdb_load, and merge the second with covdb_loadmerge, merging the data from the two separate databases. This way, you can save and merge distinct databases.

Ignoring Coverage Collected during Parts of Simulation

You can ignore any collected coverage data. Consider the following example:

```
At time N: $coverage_dump("mydir/beforeN");
At time M: $coverage_reset();
```

In this example, if there is a period between time `N` and `M` for which you wish to ignore any collected coverage data, turn coverage off during this time by writing the coverage status before time `N` to one database, then clearing coverage data at time `M` before allowing simulation to proceed.

The data collected after time `M` will be written out when simulation exits (or at the next call to `$coverage_dump`) and will include only the data collected since time `M`. For example, if `test` is called "`mydir/test.db`", when merged with the data stored in `mydir/beforeN`, the effect is as if coverage had been disabled between time `N` and `M`.

```
design = covdb_load(covdbDesign, NULL, "mydir");
test = covdb_load(covdbTest, design, "mydir/beforeN");
test = covdb_loadmerge(covdbTest, test, "mydir/test");
```

Now the handle "`test`" contains the coverage data from simulation from time `1` through `N`, and from time `M` through the end of simulation. However, any covergroups instantiated between time `N` and `M` will still be in the final written database (or any database written after time `M`). You cannot disable or delete these groups through this interface.

How the Coverage Data Is Accessed

The coverage data is accessed during simulation as follows:

1. DPI function is defined and coverage data is dumped on demand.

```
export "DPI" task dump_cov_data(string s);
export "DPI" task clear_cov_data;
task dump_cov_data(string s);
$coverage_dump(s);
endtask task clear_cov_data;
$coverage_reset();
endtask
```

2. After simulation begins, coverage is dumped by PLI code calling the DPI-exported.

```
task dump_cov_data:
… dump_cov_data("mydir/test1"); …
```

3. The PLI code, which has been linked with UCAPI, then opens the database and looks for the covergroup of interest (assume `tbMetric` has already been found).

```
covdbHandle design = covdb_load(covdbDesign, NULL,
"mydir");
covdbHandle test = covdb_load(design, design, "mydir/
test1");
covdbHandle group, groups =
covdb_qualified_iterate(test, tbMetric,
covdbDefinitions);
while((group = covdb_scan(groups))) { if (this is the
group I want …) { … check coverage status …
} } covdb_release_handle(groups);
```

4. Control then returns to the simulation, which continues collecting coverage. At a later point, the PLI code again causes coverage results to be dumped and checks the covergroup of interest once more. The new test is loaded and the covergroup handle is acquired from scratch.

```
… dump_cov_data("mydir/test2");
design = covdb_load(covdbDesign, NULL, "mydir");
test = covdb_load(design, design, "mydir/test2");
groups = covdb_qualified_iterate(test, tbMetric,
covdbDefinitions);
while((group = covdb_scan(groups))) { if (this is the
group I want …) { … check coverage status …
} } covdb_release_handle(groups);
```

5. The coverage data is reset and simulation continues.

```
clear_cov_data(); return;
```

6. This iteration is repeated as many times as required.

At any given point, the PLI code can dump coverage, analyze it, reset it, or allow it to continue without resetting. When simulation exits, coverage data is dumped as normal2 – the `$coverage_dump` and `$coverage_reset` system tasks have no effect on the final database name.

After simulation exits, a standalone UCAPI program can be used to load any of the databases dumped during simulation (or the name of the final db), using the names specified when they were dumped, as shown in the example code.

The name specified explicitly at compile time or the name given during simulation using the SystemVerilog `$set_coverage_db_name` system task. Because covergroups can be dynamically instantiated during simulation, the database written at time N may have fewer groups in it than the database written at a later time $N+C$. This is because VCS only collects coverage data for covergroups that are instantiated during simulation.

Thus, there is no guarantee that all databases dumped from a given simulation run will have the same groups in them. However, these databases can still be merged to create a single result, just as coverage results from multiple distinct simulation runs that may instantiate different groups can be merged. Note that there is an existing mechanism that allows individual covergroups to be disabled and that is using the `$cg_coverage_control` system task. The same system task can be used to disable all the covergroups if there is no covergroup name specified as an argument.

The use model for this system task is,

```
$cg_coverage_control(0/1 <,covergroup_name>)
```

where,

0 - coverage off

1 - coverage on

<,covergroup name> - (Optional) Overall covergroups are affected if nothing is supplied here.

There is no additional way to explicitly disable functional coverage. If `$coverage_dump` is called from multiple blocks at the same time with the same argument, both calls will be processed, although the order does not matter since they both will write exactly the same data to disk. If `$coverage_dump` is called from multiple blocks at the same time with different arguments, two identical databases will be written.

If `$coverage_dump` is called multiple times during simulation with no `$coverage_reset`, the databases dumped will have "overlapping" data. For example, if you perform the following:

```
$coverage_dump("mydir/test1");
…
more simulation
…
$coverage_dump("mydir/test2");
```

Then the `mydir/test2` directory contains all the coverage data (hit counts, etc.) from `mydir/test1` and the data collected after `mydir/test1` is dumped. Thus, if you use `covdb_loadmerge` to merge `mydir/test1` and `mydir/test2`, some of the coverage data in effect is duplicated.

For example, say that when `mydir/test1` is dumped, bin `B1` had 15 hits. When `mydir/test2` is dumped, `B1` will have 25 hits. If you merge `mydir/test1` and `mydir/test2`, `B1` will have 40 hits. It is the responsibility of your application to manage the relationship of these databases, since there is no way for UCAPI to know that `mydir/test` is really a subset of `mydir/test2`.

User-defined Coverage System Functions

4

URG Options

This chapter contains the following sections:

- "Command-Line Options"

- "Instance Coverage Score Option"

- "Covergroup Score Covered/Coverable Ratio Option"

- "Trend Chart Command-Line Options"

- "Reporting Element Holes"

Command-Line Options

Important:

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

URG supports the following command-line options:

```
urg
[-assert minimal]
[-dbname dirname][-dbname dirname/testname]
[-diff]
[-dir dir1 [dir2 ...] [-dir dir3 ...]
[-dump full_exclusions metric1[+metric2..]]
[-elfile <file>]
[-elfilelist <filelist>]
[-excl_bypass_checks]
[-excl_strict]
[-f file]
[-format text]
[-format both]
[-fsm disable_sequence]
[-fsm disable_loop]
[-full64 | -mod64]
[-grade [help] [... other grading options ...]]
[-group db_edit_file file]
[-group flex_merge_drop]
[-group instcov_for_score]
[-group merge_across_scope]
[-group maxmissing N]
[-group ratio]
[-group show_bin_values]
[-h | -help]
[-hier <file>]
[-high N]
[-hvp_disable_wildcards_in_cgshapeparam]
[-ID]
```

```
[-lca]
[-line nocasedef]
[-log file]
[-low N]
[-lpcov]
[-mapfile <file>]
[-map <mod>]
[-metric[+]line+fsm+cond+tgl+branch+assert+group]
[-mod filter.file mod.file]
[-noreport]
[-parallel <machine_file>]
[-parallel -maxjobs <Number>]
[-parallel_temproot]
[-plan file]
[-pathmap <file>]
[-report dir] [-parallel ...]
[-scorefile file]
[-show availabletests]
[-show brief]
[-show fullhier]
[-show hvpfullhier]
[-show hvpprob]
[-show legalonly]
[-show maxtests N]
[-show ratios]
[-show tests]
[-srcmap <from> <to>]
[-split N] [-split metric]
[-tests file]
[-tgl portsonly]
[-trend [help] [... other trend options ...]]
[+urg+lic+wait]
[-userdata file]
[-userdatafile file]
[-warn no<ID>,...,no<ID>] [-warn none]
[-warn none,<ID>,...,<ID>]
```

```
-assert minimal
```

Reports assertion coverage without loading the design information. This is to improve performance and memory cost, for example, for sparse assertions inside a big design, you needn't load the whole design information, but only load some modules/ instances which have assertions. Only report modules and instances which have assertions in assertion coverage. Code coverage database can not be loaded with this option.

With this option, the difference in the hierarchy page is as follows:

- If all instances have assertions, the hierarchy page is the same as usual.

- If some of the instances have assertions, the nodes for the instances with no assertions will not be existing in the hierarchy. An instance will be taken as top level instance if its parent has no assertion.

For example A.B.C.D.E.F, the 'A', 'B', 'C', and 'F' instances have assertion. The hierarchy page looks like:

```
A

|----->B

|     |----->C

|

A.B.C.D.E.F
```

```
-cond ids
```

Displays the condition IDs in the condition coverage detail report. See the section "Displaying Condition IDs" for more information.

`-dbname dirname`

Creates a merged database in the directory 'dirname.vdb'. The name of the merged test will be 'test' by default. See "Using -cm_dir and -dbname Options with the Unified Coverage Database" .

`-dbname dirname/testname`

Creates a merged database in the directory 'dirname.vdb'. The name of the merged test will be 'testname'.

`-diff`

Compares two databases and generates a difference report. Only supported for assertion and testbench coverage.

`-dir directory_name`

Specifies coverage data directories. You can specify multiple directories also and you need not specify `-dir` option with each directory name as shown:

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

`-dump full_exclusions`

Dumps the metric specific exclusion files at instance-level and module level. Each metric specific exclusion file includes the exclusion entries for all the corresponding coverable objects for each instance and module of a design.

`-echo`

Generates Echo bias file based on the holes in the given coverage database. The suboptions are:

`help` — shows this description.

`gen_bias N` — splits uncovered holes into N bias files (a positive integer must be given to enable Echo).

`max_hole_size N` — specifies the maximum number of bins that can be grouped in single hole (optional - the default is 10).

`gen_bias_dir path` — generates bias files in the given path (optional - the default path is echoBiasConfigs/).

Echo is an LCA feature. For more information about Echo, see the LCA category in the *VCS Online Documentation*.

`-elfile <file>`

Excludes coverable objects specified in <file> for code/assertion/ group coverage. For example:

`urg -dir … -elfile filename.el`

For more information about elfile, see the chapter "Exclusion" in the *Coverage Technology User Guide*.

`-elfilelist <filelist>`

Specifies a file containing a list of exclude files to be loaded. For example, a file 'foo' with these lines in it:

```
elfile1.el
elfile2.el
```

could be passed to URG as follows:

`% urg -elfilelist foo …`

Both elfile1.el and elfile2.el would be loaded as if they had been passed to the `-elfilelist` option directly. The exclude files are loaded in the order in which they are specified in the argument to `-elfilelist`:

```
% urg -elfilelist <elfilelist_filename>
```

Note:
- You should create the file list of the exclusion files manually in a text editor.

- You can specify an elfile at each line and add comments which can begin with '#'.

- The elfile provided can be supplied with either a full path name or relative path name and any environment variable substitutions are not supported.

For example,

```
foo
-----------------------
elfile1.el
elfile2.el
dir1/dir2/file1.el
dir1/dir2/file2.el
/home/dir1/dir3/file3.el # for line coverage exclusion
File4.el                 # current directory
-----------------------
```

`-excl_bypass_checks`

By default, when an exclude file is specified to URG, the checksums stored in the exclude file are compared to the checksums in the design, and if there are mismatches, the exclusions are not applied.

If the `-excl_bypass_checks` option is given, those comparisons are not done, and all the exclusions are applied.

```
% urg -elfile <elfilename> -excl_bypass_checks
```

You must specify the `-elfile` option with the `-excel_bypass_checks` option, else an error is generated.

`-excl_strict`

Does not allows covered objects to be excluded. '-elfile <file>' must be given when this option is used.

`-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.vdb ./simv2.vdb ./simv3.vdb
```

The size of the command line might exceed the RHEL32 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.vdb -f file_list
```

Here, `file_list` (./simv1.vdb ./simv2.vdb ./simv3.vdb) contains the databases. 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.vdb) if you are using the `-dir` option.

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

`-format text`

Generates text report. By default html report is generated unless
`-format` option is specified with `text/both` argument.

`-format both`

Generates both text and html report.

`-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. However, URG auto invokes the 64-
bit URG binary if the following conditions are met:

- The `-full64` option is not specified in the `urg` command-line.

- $VCS_HOME includes the 64-bit URG installation.

- At least one of the coverage databases passed to the `urg`
command-line is generated in the 64-bit VCS mode.

Note:

If you do not pass the `-full64` option for $VCS_HOME that includes only 64-bit installation and if at least one of the coverage databases is generated in the 64-bit VCS mode, 64-bit URG is auto invoked.

If you do not pass the `-full64` option for $VCS_HOME that includes only 32-bit installation and if at least one of the coverage databases is in the 64-bit VCS mode, 32-bit URG is invoked.

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

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.

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.

`testfile`

> Use this option with `-grade` to generate the `gradedtests.txt` text file, the `gradedtests.txt` file contains a simple list of graded tests.

`-group db_edit_file file`

> Specifies the filename for editing database.

`-group flex_merge_drop`

> Enables flexible merging for covergroups. For more information, see the section "Flexible Merging" in the *Coverage Technology User Guide*.

`-group instcov_for_score`

> Compute scores using coverage of each instance for covergroups with instance coverage enabled. For more information, see the section "Instance Coverage Score Option" .

`-group maxmissing` *N*

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

`-group merge_across_scopes`

Merges the functional coverage data from different coverage databases having same covergroup but with different hierarchy. For more information, see the section "Merge across-shape" .

`-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.

`-group show_bin_values`

Displays bin definitions of coverpoints.
Requires `-covg_dump_range` at runtime.

`-help` and `-h`

Shows command line and options supported by URG.

`-hier filename`

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.

Note:

> The `urg -hier filename` option has the same format as that of the file used in the `-cm_hier filename` option. For more information, see "Controlling the Scope of Coverage Compilation" section.

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

```
% urg -dir simv1.vdb -hier hfile1
```

`-high N`

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

`-hvp_disable_wildcards_in_cgshapeparam`

This switch disables "`*`", "`/`" and "`?`" wildcards in an expression of a covergroup shape.

For example,

```
Group : test::cg::SHAPE{(this.arg11_0_cg_init *
this.arg12_0_cg_init)=3,(this.arg13_0_cg_init **
this.arg14_0_cg_init)=1,(this.arg1_0_cg_init **
this.arg2_0_cg_init)=1,(this.arg3_0_cg_init *
this.arg4_0_cg_init)=2,(this.arg5_0_cg_init **
this.arg6_0_cg_init)=1,(this.arg7_0_cg_init *
this.arg8_0_cg_init)=2,(this.arg9_0_cg_init **
this.arg10_0_cg_init)=2}.*
```

Inside the expression of `test::cg::SHAPE` none of the "*" are treated as wildcards. However, the "*" that is provided after the covergroup shape expression is treated as wildcard.

`-ID`

Displays the Host ID or dongle ID for your machine.

`-lca`

Enables limited customer availability features and print warning message.

`-line nocasedef`

Excludes default cases in case statements from line coverage reports.

`-log` *file_name*

Sends diagnostics to *file_name* as well as stdout/stderr.

`-low` *N*

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

`-lpcov`

Enables you to view the LP coverage report.

`-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.vdb -dir input.vdb -mapfile file_name
```
Where, `file_name` is the mapping configuration file. For more information, see the section "Mapping Coverage" in the *Coverage Technology User Guide*.

`-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]`

> Limits report to specified metrics. For more information about coverage metrics, see the *Coverage Technology User Guide*.

`-mod file`

> Reads filtered or overrided HVP data from the specified 'file'. The `-plan` options must also be given.

`-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. For more information, see the section "Parallel Merging" in the *Coverage Technology User Guide*.

`-parallel -maxjobs <Number>`

> Sets the maximum number of jobs that run at the same time. It works in both single and multi-machine mode, but not in LSF or Grid mode. It also works for both level1 and other high level jobs.
>
> In the single machine mode, the default value is 10. Only 10 jobs will be running at the same time in a local machine. In the multi-machine mode, the default value is 1. Each machine in machine list file will be running 1 URG job.

Following is the algorithm for `-maxjobs` in multi-machine:

```
create queue for available machine, such as for 2 machines
and '-maxjobs 3' 'machine1, machine2, machine1, machine2,
machine1, machine2'.
```

Remove one from this queue to launch a job and push back the machine name when one job is finished.

Limitation

The jobs across levels do not run at the same time. You need to finish all level1 jobs, then start to launch level2 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 Verification Planner Editor, see the *Verification Planner Editor under the LCA Category in the VCS Online Documentation*.

`-report` *mydir*

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

```
-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 of the specified `-dir` directories and exits without generating a report. By default, URG reads all the test files in the directories specified by the `-dir` option. You can edit the resulting list and use it with the `-tests` option.
>
> For example:
>
> ```
> % urg -dir simv.vdb -show availabletests
> ```
>
> The output looks like this:
>
> ```
> Available tests names:
> simv/test1
> simv/test2
> ```
>
> You can control which tests from a directory (or set of directories) are loaded by URG by passing a text file to the `-tests` option. The entries in the `-tests` file are in the same format as the output of `urg -show availabletests`. In fact, you can edit the availabletests output to create your `-tests` input file.
>
> ```
> % urg -tests file_name
> ```

```
-show brief
```

> Shows uncovered data only.

`-show fullhier`

> Shows full hierarchy, including instances that have a hierarchical coverable count of zero.

`-show hvpfullhier`

> Shows the full hvp hierarchy, including the plans and all the features which are filtered out.

`-show hvpprob`

> Shows problem hvp hierarchy only. When you use this option, the default hvp*.html reports are reduced in size. This is used in cases where you would like a concise report of only the problems.

`-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`. The default number of test is three.

`-show ratios`

> Displays ratio score in the URG report. See "Displaying Ratio Score in URG Report" .

`-show tests`

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

When you have multiple test files (intermediate results files) from multiple simulations, URG merges the coverage results so that if something is covered in one, but not all the test files, URG reports it as covered.

In covergroup and assertion coverage, you can have URG indicate in the modinfo.txt/mod*.html pages which test covered each assertion or bin. This feature is enabled with the `-show tests` option.

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

`-pathmap file`

Relocates source files with mapping rules in <file>. For more information, see the section "Mapping Coverage' in the *Coverage Technology User Guide*.

Note:

The `-srcmap` option is replaced with the `-pathmap` option

`-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 kilobytes (KB) 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.

`-tgl portsonly`

> Generates toggle coverage report only for module ports. Ports in SV Interfaces not yet supported.

`-trend [`*trend_options*`]`

> Specifies the options to generate a trend chart. For more information on -trend, see "Trend Chart Command-Line Options" . For more information on generating trend charts, see the section "Analyzing Trend Charts" in the *Coverage Technology User Guide*.

`+urg+lic+wait`

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

`-userdata file`

> Reads HVP data for annotation from the specified 'file'. The `-plan` option must also be given.

```
-userdatafile file
```

Specifies a file containing HVP data file names for annotation.
The `-plan` option must also be given.

```
-version
```

Displays the URG version and version of the coverage database.

-novercheck

Disables version checking.

```
-warn [no]ID|none|all
```

Controls the display of warning messages.

Where,

`no` — Specifies disabling warning messages with the ID that
follows. There is no space between the keyword number and the
ID.

`none` — Specifies disabling all warning messages. IDs that follow
in a comma-separated list, specify exceptions. There is no space
around the comma.

`all` — Specifies enabling all warning messages, IDs that follow
preceded by the keyword number in a comma separated list,
specify exceptions.

The following are examples that show how to use these options:

```
-warn all
```

Enables all warnings. This is the default.

```
-warn noCMR-VCINF
```

Suppresses CMR-VCINF, other warnings are enabled.

```
-warn noCMR-VCINF,noCMR-VCE
```

Suppresses CMR-VCINF and CMR-VCE.

```
-warn none
```

Suppresses all warnings.

```
-warn none,CMR-VCINF
```

Enables CMR-VCINF, other warnings are disabled.

Using `-cm_dir` and `-dbname` Options with the Unified Coverage Database

With the `-cm_dir` option,

- a `.vdb` extension is appended to the name given in `-cm_dir` if it is not present already. Therefore, the default name of the directory will be `simv.vdb`.

- while running URG, the argument to `-dir` option can either be the name given during `-cm_dir` or the actual directory created.

- wildcard characters can be given in the URG command-line. Some examples of valid usage are:

```
urg -dir *.vdb
urg -dir abc*
```

Using `-tests` feature in the URG command-line

- The test names which are given in a testList file (`urg -dir abc -tests <testList>`), should contain `abc/test` or `abc.vdb/test` if `-cm_dir abc` was given in the `vcs/simv` command line.

- Old db behavior remains unchanged. Use `urg -show availabletests` to get list of valid test names.

With the `-dbname` option,

- a `.vdb` is appended to `<dirName>` if not already present, and the merged data is dumped there.

- combination of the old database and unified database directories in URG command-line results in an error.

Redirecting All the Temporary Files Generated By urg -parallel -debug Option

By default, `urg -parallel` operation in debug mode generates the temporary files in the present working directory.

```
Temporary_URG_db_{process
id}_{hostname}_{timestamp} : temporary db

directoryTemporary_URG_scripts_{process
id}_{hostname}_{timestamp}: script directory
```

For example,

```
Temporary_URG_db_20817_peemt503_Thu_Oct_11_00:15:
10_2012

Temporary_URG_scripts_20817_peemt503_Thu_Oct_11_0
0:15:10_2012
```

With this enhancement, you can redirect temporary files generated by `urg -parallel -debug` to a different location by providing `-parallel_temproot` option.

Use Model

```
% urg -parallel -debug -parallel_temproot
temprootdiradditional_options
```

You can use this additional option along with `urg -parallel -debug` to redirect temporary files under specified directory location.

Examples

- `urg -parallel -debug -parallel_temproot userlocation -lca -format both`

  It creates a directory with name userlocation in the present working directory and redirects all the temporary files inside it.

- `urg -parallel -debug -parallel_temproot ~/userlocation -lca -format both`

  It creates a directory with name userlocation in your home directory and redirects all the temporary files inside it.

- `urg -parallel -debug -parallel_temproot ~/mydir1/mydir2/userlocation -lca -format both`

  It creates a directory with name userlocation under `~/mydir1/mydir2` located in your home directory and redirects all the temporary files inside it.

Note:Make sure that `~/mydir1/mydir2` directory hierarchy is already present else you will see unexpected behavior during "URG -parallel" operation in this case.

Limitations

1. The special characters such as #, @, &, _etc. cannot be used while providing directory name where all the temporary directories are going to redirect.

2. You cannot provide hierarchies of directories name (while providing directory name where all temporary directories are going to redirect) which do not exist in that path.

Displaying Ratio Score in URG Report

You can view the ratio of the coverage score in the URG report using the `-show ratios` option. For example,

```
% urg -dir simv.vdb -show ratios
```

When you use the `-show ratios` option, the `-group ratio` option is automatically enabled and you get the following message in the command-line:

```
Note-[URG-RATIO] URG show ratios
URG '-show ratios' option will turn on '-group ratio'
automatically. Please check the scores in group pages.
```

The URG report is as follows:

Figure 4-1   Dashboard Page

Total Coverage Summary

SCORE	LINE		COND		TOGGLE		FSM		BRANCH		PATH		ASSERT		GROUP	
42.11	72.66	513/706	33.98	157/462	50.61	989/1954	34.83	31/89	57.77	171/296	50.00	111/222	20.00	2/10	16.98	2015/11865

Hierarchical coverage data for top-level instances

SCORE	LINE		COND		TOGGLE		FSM		BRANCH		PATH		ASSERT		NAME
45.69	72.66	513/706	33.98	157/462	50.61	989/1954	34.83	31/89	57.77	171/296	50.00	111/222	20.00	2/10	test_jukebox

Figure 4-2   Hierarchy Page

SCORE	LINE		COND		TOGGLE		FSM		BRANCH		PATH		ASSERT		NAME
45.69	72.66	513/706	33.98	157/462	50.61	989/1954	34.83	31/89	57.77	171/296	50.00	111/222	20.00	2/10	test_jukebox
91.23	100.00	9/9			73.68	28/38			100.00	2/2					cd1
74.28	88.24	15/17			75.56	68/90			66.67	4/6	66.67	4/6			fifo1
77.14	96.00	24/25	42.86	3/7	53.12	34/64	100.00	4/4	87.50	7/8	83.33	5/6			jb1
45.61	70.91	78/110	37.36	34/91	62.59	184/294	35.29	6/17	60.71	34/56	52.38	22/42	0.00	0/2	st0
39.20	64.20	52/81	25.93	14/54	54.84	34/62	35.29	6/17	48.72	19/39	45.45	15/33	0.00	0/2	coin1
74.00	89.66	26/29	54.05	20/37	60.27	88/146			88.24	15/17	77.78	7/9			kp1
73.61	89.09	98/110	43.96	40/91	55.10	162/294	70.59	12/17	80.36	45/56	76.19	32/42	100.00	2/2	st1

Figure 4-3   Groups Page

Total Groups Coverage Summary

COVERED	EXPECTED	SCORE	COVERED	EXPECTED	INST SCORE	WEIGHT
2015	11865	16.98	2015	11865	16.98	1

Total groups in report: 10

SCORE	INSTANCES	WEIGHT	GOAL	NAME
0.82	0.82	1	100	test_jukebox.st4.coin1::Cvr
1.81	1.81	1	100	test_jukebox.st2.coin1::Cvr
2.96	2.96	1	100	test_jukebox.st0.coin1::Cvr
2.96	2.96	1	100	test_jukebox.st1.coin1::Cvr
2.96	2.96	1	100	test_jukebox.st3.coin1::Cvr
22.05		1	100	test_jukebox.st0.coin1::atm::packet
22.05		1	100	test_jukebox.st1.coin1::atm::packet
22.05		1	100	test_jukebox.st2.coin1::atm::packet
22.05		1	100	test_jukebox.st3.coin1::atm::packet
22.05		1	100	test_jukebox.st4.coin1::atm::packet

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.

-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.

-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.

-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.

Unsupported Options in Parallel Merging

The following options are not supported when you use parallel
merging:

```
[-diff] [-ID]
[-grade [help] [... other grading options ...]]
[-trend [help] [... other trend options ...]]
```

Merge Covergroups Across Scopes

If a module containing a covergroup instance is instantiated multiple times, then for each elaborated module instances, a covergroup shape is created. The results are not automatically accumulated into a single shape or score. If you want to know the overall score for a covergroup definition, you can use the URG option `-group merge_across_scopes`.

This technique is useful in cases where you have defined similar covergroups in different tests, but the program-name is different. So even though two tests may have exactly the same covergroup (both name and definition), since enclosing program is different they are not merged.

Using the `-group merge_across_scopes` option, you can merge across shape and across the program scope. It allows the reduction of shape creation for groups that are otherwise the same but at a different scope.

Merge across-shape

Suppose there is a covergroup "cg", in a module "M" as follows:

```
module M;
    covergroup cg; ... endgroup
    cg cg1 = new;
```

```
endmodule
```

Suppose module "M" is instantiated twice in the top-module "top" as follows:

```
module top;
   M I1();
   M I2();
endmodule
```

Then, two covergroup instances are created at runtime, one through "top.I1" and the other through "top.I2". Two covergroup shapes are created for "cg", with shape-scope as "top.I1" and "top.I2".

When "across-shape-scope" merging is enabled, the scope-name of these two shapes is removed, and these two shapes are merged into one.

The prerequisite for this kind of merging is that both these shapes should have the same bin-space. If bin-spaces of these two shapes are not same, then these two shapes are not merged (though scope-name would still be removed).

Merge across-program-scope

If two programs or modules have a covergroup with the same name, and same set of coverpoints/crosses, then they are merged into one covergroup and their shapes are also merged.

Suppose there is a covergroup "cg", in a module "M" as follows:

```
module M1;
    covergroup cg; ... endgroup
    cg cg1 = new;
endmodule

module M2;
```

```
    covergroup cg; ... endgroup
    cg cg2 = new;
endmodule
```

In this case, both M1::cg and M2::cg are renamed to unified_prog::cg. If both of them have the same coverpoints and crosses, then they are merged into one covergroup. In that case, the shapes corresponding to M1::cg1 and M2::cg2 are also merged if their bin-spaces are same.

Suppose the covergroup has multiple shapes; say M1::cg has three shapes namely S1, S2, S3; M2::cg has two shapes namely S4 and S5. Suppose S1 and S5 have the same bin-space, and S2 and S4 also have the same bin-space. So, after using the `merge_across_scopes` option, you have a covergroup unified_prog::cg with three shapes, S1_5, S2_4 and S3, where S1_5 is a merged copy of S1 and S5, and so on.

For example, consider that a unit level testbench has a covergroup named "sb_usbspeed_cov" (coverage database is usbvip.vdb):

```
Group : test_top.TestProgram::Scoreboard::sb_usbspeed_cov
```

While the system level also has the same covergroup, but different hierarchy (coverage database is usbvmm.vdb):

```
Group : test_usb_top.tb::Scoreboard::sb_usbspeed_cov
```

To merge these two coverage results, use the `-group merge_across_scopes` URG command as follows:

```
urg -dir usbvmm.vdb usbvip.vdb -group merge_across_scopes
```

The merged result would be:

```
unified_prog::Scoreboard::sb_usbspeed_cov
```

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 you have 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 4-1   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 4-1, 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 4-1, 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 4-1:

Table 4-1   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 non-intuitive 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 4-4, 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 4-4   Typical coverage score changes over time
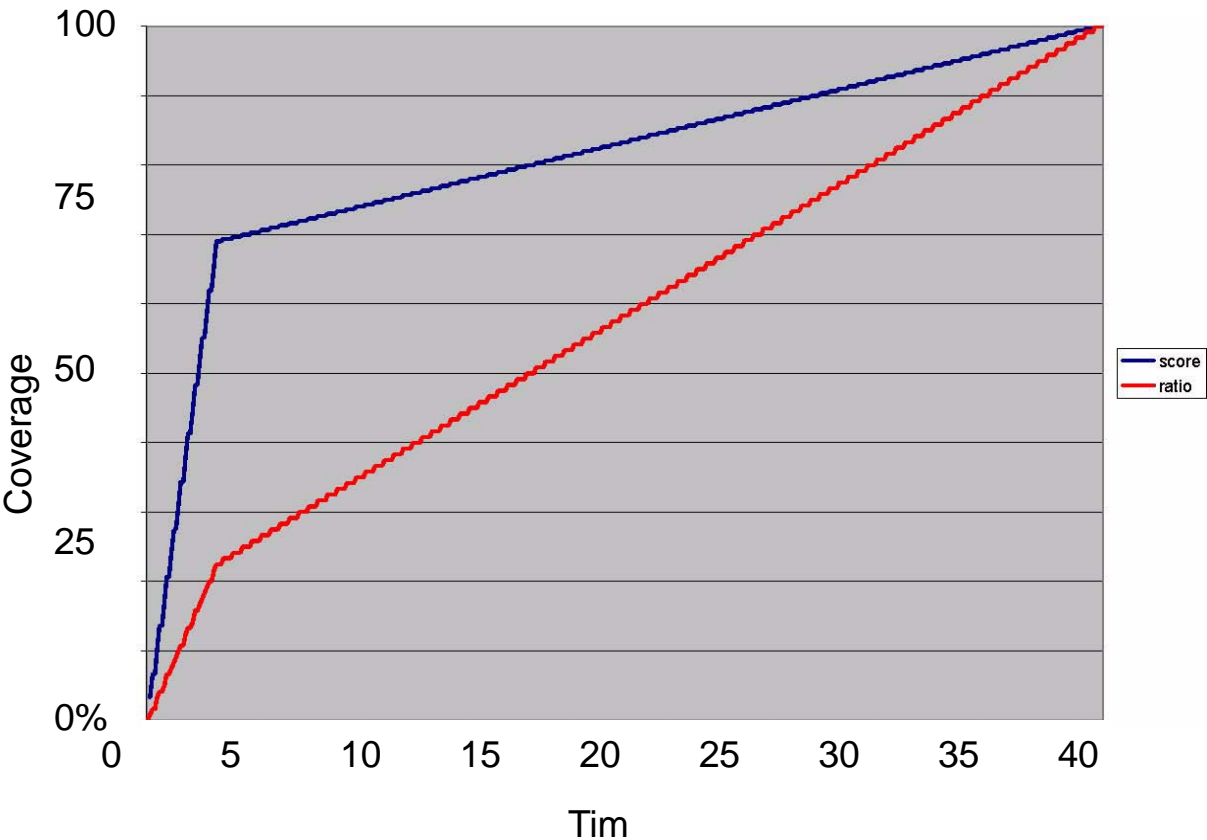
The reason the blue line has a sharp rise at first before flattening out is that the individual variable bins get covered quickly, but the cross between them has many more bins. Since the overall coverage score is computed as the average of the variables coverage score and the coverage score of the cross 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 4-4) 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.

Trend Chart Command-Line Options

Trend Chart is described in the URG chapter of the Coverage Technology User Guide. In this section, only command-line options for `-trend` have been described.

```
%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 also use the `-trend` option with other URG options.

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 4-5   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

URG Options

4-42

5

Unified Coverage API Functions

This chapter describes the functions available in UCAPI and contains the following sections:

- "Coverage Data Load/Unload"

- "Coverage Database Version Check"

- "Coverage Data Model Traversal"

- "Bypass Checksum Validation"

- "Memory and Pointer Management"

- "Reading Properties"

- "Reading Annotations"

- "Setting Properties"

- "Error Handling and Recovery"

- "APIs for Exclusion"

- "Types, Properties, and Relations"

Coverage Data Load/Unload

The following functions are used to load and unload designs and tests.

Function	Description
covdb_load	load a design or test
covdb_loadmerge	load and merge addition design or test
covdb_unload	unload or unload merge a design or test
covdb_save	save merged test data

covdb_load

The covdb_load function is used to load a design or test:

```
covdbHandle covdb_load(covdbObjTypesT type,
                       covdbHandle design,
                       char *name);
```

The *type* argument may be **covdbDesign** or **covdbTest**. If *type* is **covdbDesign**, *name* is the pathname to a coverage design directory, and *design* should be NULL. The handle returned is of type **covdbDesign**.

If *type* is **covdbTest**, *name* is the logical test name, and *design* is the design for which to load the test. The handle returned is type **covdbTest**.

For example,

```
covdbHandle design = covdb_load(covdbDesign, NULL,
"simv.vdb");
```

Note:

> Only one design may be loaded in a single UCAPI session. If the application wants to load another design, covdb_unload must first be called on the first design.

covdb_loadmerge

The covdb_loadmerge function loads and merges a design (or test) with an already-loaded design (or test).

```
covdbHandle covdb_loadmerge(covdbObjTypesT type,
                            covdbHandle design_or_test,
                            char *name);
```

After a successful call of covdb_loadmerge, the covdbName property of the destination design or test will be the string "merged". Applications can set a new name for a test using the covdb_set_str function (see "covdb_set_str" on page 23).

The primary use of covdb_loadmerge is to load and merge a list of tests from a design. See the UCAPI user guide for an example.

The other use of covdb_loadmerge is to combine multiple database directories into a single design handle. To do this, call covdb_load (covdbDesign, ".....") to load the first directory. This must be a directory containing compile time data and not just test data. Then for each additional directory, use covdb_loadmerge(covdbDesign, designHdl, "....") to add the new directory to the previously-loaded design handle.

covdb_unload

The covdb_unload function unloads all of the data corresponding to the given design or test.

```
int covdb_unload(covdbHandle design_or_test);
```

After a design is unloaded, no associated information, such as related tests, can be used. Applications should unload all tests associated with a design before unloading the design to avoid memory leaks. After unloading, all handles related to this design will be invalid and attempts to use such handles will have unpredictable effects.

If a test is unloaded, only the coverage information related to that test will be removed. After this call, handles to this test or coverage information belonging to this test will be invalid and further attempts to use such handles will have unpredictable effects.

Returns non-zero if successful, 0 if it fails.

covdb_save

The covdb_save function saves all coverage data in a given test handle to disk.

```
int covdb_save((covdbHandle test,
               const char *testname);
```

The coverage data will be stored in the directory named 'test', and the name of the test will be 'testname'.

covdb_save_exclude_file

The covdb_save_exclude_file function is used to save exclusions set on objects using the covdb_set function:

```
int covdb_save_exclude_file(covdbHandle design,
                            const char *filename,
                            const char *mode);
```

A test handle must be given as the first argument. The mode string should be "w" to overwrite any existing file, or "a" to append the exclusions to the end of a file. If the file does not exist, "w" and "a" have the same effect.

The return value is 0 on success and non-zero on failure.

covdb_load_exclude_file

The covdb_load_exclude_file function loads an exclusion file from the disk and applies its exclusions to objects in the currently-loaded design.

```
int covdb_load_exclude_file(covdbHandle design_or_test,
                            const char* filename);
```

A test handle must be given as the first argument. The return value is 0 on success and non-zero on failure.

covdb_save_attempted_file

The covdb_save_attempted_file function is used to print the list of objects that were covered, but that the application tried to exclude. In Strict mode, excluding individual covered objects is not allowed –

see the User Guide for more information. This function is typically used for application debugging or to give information to the user of the application:

```
int covdb_save_attempted_file(covdbHandle testHdl,
                              const char *filename);
```

Note that a test handle must be given since objects can't be covered without knowing which test (or merged test handle) is involved.

---

covdb_load_mapfile

The covdb_load_mapfile function is used to specify how UCAPI should merge data from differing designs as they are loaded.

```
void covdb_load_mapfile(covdbHandle designHdl,
                        const char *mapfilename);
```

The function returns 0 on success and non-zero if an error occurs.The format of the mapfile is any number of entries of the following form:

```
MODULE: modname
INSTANCE:
SRC: instance_list
DST: instance_list
SRC: instance_list
DST: instance_list
…
```

Each instance_list is a comma-separated list of full pathnames in the base design (SRC) or an input design (DST), or the wildcard character *.

Coverage Database Version Check

To detect version mismatch between UCAPI and design/shape/test, version information needs to be added to the newDB XML files. Version information must be hard-coded into the UCAPI library.

Version Check rules apply to all UCAPI applications as well as URG and DVE Coverage GUI.

Version format

```
<?xml version="1.0"?>
<!DOCTYPE cov SYSTEM "ucdb_cond.dtd">
  <cond type="verilog"
chksum="4E133C48A1C639D0C8DD2CCE906B7D48CFACFE53"  >
    <ucapi_version  major_ver="10"  minor_ver="1"
patch_str="G-2012.09"  />
      <conddef id="3"  chksum="954450407"  >
        <condshape >
```

Files that are versioned:

- sourceinfo.xml

- design.xml

- shape.xml

- data.xml

- testbench.cumulative.xml

- testbench.inst.xml

Files that are not versioned:

- exclude.xml

*   non-XML files

Version Check

For compatible reasons, XML files without version are accepted. Note that coverage databases generated by Vera are not versioned.

Before loading any XML file, UCAPI will check its version if available. If a file is identified as incompatible, UCAPI will print an error and skip loading the design/shape/test, depending on the file type.

```
Error-[UCAPI-VM] CovDB Version Mismatch
The file 'simv.vdb/snps/coverage/db/shape/
line.verilog.shape.xml' is generated by an incompatible VCS
(G-2012.09 with CovDB version 11.1).  Please regenerate the
coverage database or use a compatible VCS.
```

A file is considered compatible if,

*   its major version is equal to UCAPI's own major version (which means a compatible stream),

*   its minor version is less than or equal to UCAPI's own minor version (which means patch over patch compatibility).

Note that patch_str is not used for version check, it is meant for error reporting.

Once an incompatible change is checked in, the major version must be increased. This can only happen on a new stream.

A new VCS patch release may increase the minor version if there is an extension to the coverage database, and it is required to be backward compatible.

UCAPI version check can be disabled with:

```
covdb_configure(covdbVersionCheck, (char *)"false");
```

In URG, this can be done with an option `-novercheck`.

Coverage Data Model Traversal

The following functions are defined for traversing coverage data.

Function	Description
covdb_get_handle	get the handle for a 1-to-1 relation
covdb_get_qualified_handle	get a qualified handle for a 1-to-1 relation
covdb_iterate	get an iterator for a 1-to-many relation
covdb_qualified_iterate	get a qualified iterator for a 1-to-many relation
covdb_scan	get the next handle from an iterator
covdb_release_handle	release reference to a UCAPI handle
covdb_qualified_object_iterate	get qualified information about coverable objects

Functions returning handles will generally return NULL if an error occurs or if the relationship is empty. If an error occurs, any registered error callback function will also be invoked.

covdb_get_handle

The covdb_get_handle function may be called with any covdbHandle and any covdb1To1RelationT relation. If the relation `rel` is not defined for the `handle` type it will return NULL. NULL will also be returned if the relation is empty for *handle*, even if it applies to its type.

```
covdbHandle covdb_get_handle(covdbHandle handle,
```

```
                                    covdb1To1RelationT rel);
```

covdb_get_qualified_handle

Similar to covdb_get_handle, but requires a qualifier handle, which must be of type covdbMetric or covdbTest. If the relation `rel` is not defined for the `handle` type it will return NULL. NULL will also be returned if the relation is empty for `handle`, even if it applies to the *handle* type.

```
covdbHandle covdb_get_qualified_handle(covdbHandle handle,
                                       covdbHandle qual,
                                       covdb1To1RelationT rel);
```

covdb_iterate

Returns a handle to an iterator over the objects for the specified 1-to-many relation for `handle`. If the relation `rel` is empty, or does not apply to `handle`, a NULL iterator handle will be returned.

Handles returned by covdb_iterate may only be scanned through one time - they cannot be reset to the beginning. To start over and iterate from the beginning again, acquire a new handle using covdb_iterate:

```
covdbHandle covdb_iterate(covdbHandle handle,
                          covdb1ToManyRelationT rel);
```

covdb_qualified_iterate

Returns a handle to an iterator over the objects for the qualified 1-to-many relation for `handle`. If the relation `rel` is empty, or does not apply to `handle`, a NULL iterator handle will be returned.

Handles returned by covdb_qualified_iterate may only be scanned through one time - they cannot be reset to the beginning. To start over and iterate from the beginning again, acquire a new handle using covdb_qualified_iterate.

```
covdbHandle covdb_qualified_iterate(covdbHandle handle,
                              covdbHandle qual,
                              covdb1ToManyRelationT rel);
```

covdb_scan

The covdb_scan function returns the next object from an iterator handle and advances the iterator.

Only one handle returned by covdb_scan for a given iterator is valid at any time – once covdb_scan is called again, the handles returned by previous calls are invalid.

If called on an object that is not an iterator, it returns NULL.

```
covdbHandle covdb_scan(covdbHandle iter_handle);
```

The handles returned by covdb_scan are volatile and may be overwritten by the next call to a UCAPI function. You can make a handle persistent by calling covdb_make_persistent_handle.

covdb_qualified_object_iterate

The covdb_qualified_object_iterate function is used when iterating the qualified contents of an object inside a region.

```
covdbHandle covdb_qualified_object_iterate(
                     covdbHandle objHdl,
                      covdbHandle regionHdl,
```

```
                    covdbHandle qualHdl,
                    covdb1ToManyRelationsT rel);
```

For example, the covdbTests relation from an object handle is such a 1-to-many relation. The object is the coverable object whose test coverage you want to iterate. The region handle is the region that contains the object (e.g., a covdbSourceInstance handle). The qualified is the merged test handle containing tests t1, t2, …, tN.

The iterator for a given object will contain the subset of { t1, t2, …, tN } where any ti in the set is a test that covers the object.

Bypass Checksum Validation

covdb_qualified_configure

When loading an exclude file via UCAPI, you can suppress checksum comparison by configuring UCAPI using the covdbExclBypassChecks configuration item:

```
covdb_qualified_configure(urgCovered::getDesignHdl(),
covdbExclBypassChecks, "true");
```

Memory and Pointer Management

A covdbHandle is a pointer that points to an object. If the object is made persistent, it gets copied into a safe region in the memory. When the handle is later passed to covdb_release_handle, the memory is cleaned up. The covdbHandle will still be pointing to the same (corrupted) memory location, but the object itself will be gone.

Example 1:

```
covdbHandle H = covdb_get(obj, covdbObjects);
H = covdb_make_persistent_handle(H);
covdbHandle C = H;
covdb_release_handle(H)
```

Here, C is a persistent handle.

However, releasing handle H invalidates the safe memory created for the object, thus C and H both point to memory that is now corrupt.

Example 2:

```
covdbHandle K = covdb_get(obj, covdbObjects);
covdbHandle J = covdb_make_persistent_handle(K);
covdb_release_handle(J);
```

Here, K is not persistent and covdb_release_handle(J) does not affect the status of K. However, since K was not made persistent, it may become invalid at the next UCAPI function call.

UCAPI handles returned by covdb_scan, covdb_get_handle, covdb_get_qualified_handle are not persistent. That is, they are only guaranteed to be valid until the next call to a UCAPI function.

Handles returned by covdb_iterate and covdb_qualified_iterate are persistent and must be explicitly released by the application after it exits the loop, or the memory associated with them will leak.

The following functions are provided for managing handles:

Function	Description
covdb_make_persistent_handle	make a UCAPI handle persistent
covdb_release_handle	release a persistent handle or iterator

covdb_make_persistent_handle

The covdb_make_persistent_handle function returns a persistent handle for an object.

```
covdbHandle covdb_make_persistent_handle(covdbHandle
                                                   handle);
```

The handle it returns is guaranteed to remain valid until the application releases it with covdb_release_handle.

covdb_release_handle

The covdb_release_handle function releases the given handle.

```
void covdb_release_handle(covdbHandle handle);
```

When an application is done using a persistent handle, it should call covdb_release_handle. If applications do not call covdb_release_handle before discarding a persistent handle, memory may be lost (leaked).

Reading Properties

These functions are defined to read properties from UCAPI object handles.

Function	Description
covdb_version()	returns the version string
covdb_get	read an integer-valued property
covdb_get_str	read a string-valued property
covdb_get_real	read a real-valued property

covdb_version

The char *covdb_version() returns the version string. For example:

```
char *version = covdb_version();
  if (!is_supported_version(version))  {
   printf("Error: this version of UCAPI (%s) is not in the
supported list for my application.\n", version);
  }
```

covdb_get

The covdb_get function returns the value of the specified integer property *prop* for the given object *handle*. If the property is not an integer-valued property, or the value is not defined for the given object, it returns -1. Note that the value returned may be an integer, covdbObjTypeT, or covdbScalarValueT, depending on which property is read.

An object handle must always be specified. If the object is itself a region, no region handle is required. If the object is not a region, then a handle to its enclosing region must be given as well.

A test handle must be given if the property is test-qualified (such as covdbCovered or covdbCovCount).

The integer properties are defined in.

```
int covdb_get(covdbHandle object,
              covdbHandle region,
              covdbHandle test,
              covdbPropertiesT prop);
```

For example, to get the UCAPI type of an instance handle:

```
type = covdb_get(instHdl, NULL, NULL, covdbType);
```

To get the number of coverable objects for a given metric for a metric-qualified instance handle:

```
instTot = covdb_get(metricInstHdl, NULL, NULL,
                        covdbCoverable);
```

To get the number of covered objects for a given metric for a metric-qualified instance handle, you have to add a test handle:

```
instCov = covdb_get (metricInstHdl, NULL, testHdl,
                        covdbCovered);
```

To get the number of coverable objects inside a metric-qualified object (such as a covdbContainer object), specify the instance, the object, and the covdbCoverable property, but no test handle is required:

```
objTot = covdb_get(objHdl, metricInstHdl, NULL,
                        covdbCoverable);
```

To get the number of covered objects inside such an object, you must use a test handle as well. This is the same form you would use to query whether a given coverable object is covered or not with respect to a given test:

```
objCov = covdb_get(objHdl, metricInstHdl, testHdl,
                    covdbCovered);
```

covdb_get_str

The covdb_get_str function returns the value of the specified string property `prop` for the given object `handle`. If the property is not a string-valued property, or the value is not defined for the given object, it returns NULL.

```
char *covdb_get_str(covdbHandle handle,
                    covdbPropertiesT prop);
```

The string properties are defined in section "Object Properties" on page 32.

String values returned by covdb_get_str are volatile and are not guaranteed to persist beyond the next call to covdb_get_str. Applications must make a copy if they want a persistent string.

covdb_get_real

The covdb_get_real function returns the value of the specified real property `prop` for the given object `handle`. If the property is not a real-valued property, or the value is not defined for the given object, it returns -1.0.

```
double covdb_get_real(covdbHandle handle,
                      covdbPropertiesT prop);
```

Reading Annotations

Annotations may be unqualified or qualified. Annotations may be read either by the name (key) of the annotation, or through iteration of covdbAnnotation objects. To iterate over all annotations for an object, the iteration functions described in this section are used. The following functions are used to read annotations by name.

Function	Description
covdb_get_annotation	read an annotation by name
covdb_get_qualified_annotation	read a qualified annotation by name
covdb_get_integer_annotation	read an integer type annotation
covdb_set_annotation	Sets exclusion annotation

covdb_get_annotation

The covdb_get_annotation function returns the value of the specified annotation `key` for the given object `handle`. If the annotation is not defined for `handle`, it returns NULL.

```
char *covdb_get_annotation(covdbHandle handle, char *key);
```

You can read the category and severity failures for a given assertion handle using the covdb_get_annotation function:

```
char *category = covdb_get_annotation(assertionHdl,
                          "FCOV_ASSERT_CATEGORY");

char *severity = covdb_get_annotation(assertionHdl,
                          "FCOV_ASSERT_SEVERITY");
```

Example

Here, the return value is the annotation stored for the coverage object corresponding to the handle `obj`.

string cmt = covdb_get_annotation(objHandle, "ExclComment");

covdb_get_qualified_annotation

The covdb_get_qualified_annotation function returns the value of the specified annotation `key` for the given object `handle` qualified by covdbTest or covdbMetric handle `qual`. If the annotation is not defined for this handle and qualifier, it returns NULL.

```
char *covdb_get_qualified_annotation(covdbHandle handle,
                                     covdbHandle qual,
                                     char *key);
```

covdb_get_integer_annotation

The covdb_get_integer_annotation function returns the value of the specified integer-type annotation key for the given object handle. If the annotation is not defined for this handle, an error will be flagged and -1 will be returned. The only way to distinguish between the value not being set and it being a valid value of -1 is to check the error status, either by calling covdb_get_error or by using a error callback function.

```
int covdb_get_integer_annotation(covdbHandle obj, const
char* key);
```

You can also use this API to get the ID of a given conditional expression or a condition vector handle as follows:

```
covdb_get_integer_annotation(expressionHandle/
vectorHandle, "id");
```

The option of viewing the condition ID with this API is only supported with the new unified coverage database.

covdb_set_annotation

Sets exclusion annotation for coverable objects.

Syntax

```
int covdb_set_annotation(covdbHandle obj, const char* key,
const char* value);
```

To set an exclusion annotation, the argument key should be "ExclComment" and the value is the annotation to be set for the coverage object corresponding to the handle obj.

Note:

"ExclComment" is case-sensitive.

Example

```
covdb_set_annotation(objHandle, "ExclComment", "This is
unreachable");
```

Currently only "ExclComment" is supported for setting annotations on UCAPI objects.

Setting Properties

Function	Description
covdb_set	set an integer property of an object
covdb_set_str	set a string property of an object

covdb_set

This API is supported for all metrics. UCAPI allows applications to change some properties on coverable objects, as described in this section.

The covdb_set function is used to set the value of an integer-type property on a UCAPI handle.

```
void covdbStatusT covdb_set(covdbHandle handle,
                            covdbHandle region,
                            covdbHandle test,
                            covdbPropertiesT property,
                            int value);
```

This function is used to set integer properties of UCAPI handles. The properties which can be set are:

* `covdbCovered`

* `covdbCovCount`

Applications may also set one of the following flags in the covdbCovStatus property:

* `covdbStatusExcludedReportTime`

* `covdbStatusUnreachable`

- `covdbStatusCovered`

These properties can only be set on coverable objects (value sets, blocks, sequences and crosses), and not on coverable objects used only for annotation purposes (such as on the value sets inside sequences). covdbCovered and covdbCovCount are set directly:

```
covdb_set(objHdl, regHdl, testHdl, covdbCovered, 1);
covdb_set(objHdl, regHdl, testHdl, covdbCovCount, 19);
```

To set a value in the covdbCovStatus property, applications should read the existing value and modify it:

```
curval = covdb_get(objHdl, regHdl, testHdl, covdbCovStatus);
covdb_set(objHdl, regHdl, testHdl, covdbCovStatus,
                        (curval | covdbStatusCovered));
```

Clearing a flag in the covdbCovStatus property is similar:

```
curval = covdb_get(objHdl, regHdl, testHdl, covdbCovStatus);
    covdb_set(objHdl, regHdl, testHdl, covdbCovStatus,
                        (curval & (~covdbStatusCovered)));
```

Setting covdbStatusCovered on a supported handle type will automatically set the covdbCovered value of that handle equal to its covdbCoverable value.

Setting the covdbCovered property equal to the covdbCoverable property for a handle will automatically set the covdbStatusCovered flag on the handle.

Setting covdbCovered to any value less than the covdbCoverable for a handle will clear the covdbStatusCovered flag on that object.

The test handle is always a required argument to covdb_set, because properties that can be set are always test-specific. When a property is changed on an object for a given test, if that test is saved, the new value of the property will be saved as well.

covdb_set_str

UCAPI allows applications to change the name of a covdbTestHandle using the covdb_set_str function. Changing the value of other string properties is not supported for any other handle types of string properties.

The covdb_set_str function is used to set the value of a string-type *property* on a UCAPI *handle*.

```
char *covdb_set_str(covdbHandle handle,
                    covdbPropertiesT property,
                    char *value);
```

Error Handling and Recovery

Function	Description
covdb_set_error_callback	register a function to be called back
covdb_get_error	check error status
covdb_configure	set error reporting status

covdb_set_error_callback

The covdb_set_error_callback function may be used by an application to register a function to be called if an error is detected by UCAPI.

```
int covdb_set_error_callback(void (*cbfn)(covdbHandle
                                          errHdl, void* data),
                             void *data);
```

When an error is detected, UCAPI calls the `cbfn` function with a UCAPI error handle and the data pointer that was given when the callback function was registered. The error handle can be queried for the error code using the covdbValue property and a string error message using the covdbName property, for example:

```
covdb_set_error_callback(mycallback, "phase0");
…
perform some UCAPI operations
…
covdb_set_error_callback(mycallback, "phase1");
…

void mycallback(covdbHandle errHdl, void *data)
{
    char *phase = (char*)data;
    int errcode = covdb_get(errHdl, NULL, NULL, covdbValue);
    char *errmsg = covdb_get_str(errHdl, covdbName);

    printf("error #%d occurred in phase %s: %s\n", errcode,
            errmsg, phase);
}
```

The handle passed to the application's callback function is of type covdbError. It should be released using covdb_release_handle when the application is finished with it.

The covdb_set_error_callback function returns -1 if the callback function could not be registered, and 0 if the registration was successful.

covdb_get_error

The covdb_get_error function returns a covdbStatusT code (as defined in covdb_user.h). A value of covdbNoError indicates no error has occurred. Other values give the type of error (such as covdbOutOfMemoryError). If an error occurred, the value of *msg will be set to a descriptive string.

```
covdbStatusT *covdb_get_error(char **msg);
```

covdb_configure

The covdb_configure function allows applications to configure the error reporting of UCAPI. By default, UCAPI will not print error messages to the display – applications must use the covdb_get_error function to check when an error has occurred.

If covdb_configure is called to set covdbDisplayErrors to true, then error messages will be printed. For example:

```
covdb_configure(covdbDisplayErrors, "true");
```

The legal configuration items for covdb_configure include:

- covdbDisplayErrors. Default value false. If true, errors will be printed to the standard output when they occur.

The default value of each of these configuration options is true. If set to "false", data for the given metric will not be loaded even if it is present in the coverage directories:

• covdbLoadLine.

• covdbLoadCond

- covdbLoadTgl

- covdbLoadFsm

- covdbLoadBranch

- covdbLoadAssert

- covdbLoadPath

- covdbLoadGroup

The configuration option covdbLimitedDesign has the default value "false". If set to "true", then the design hierarchy information will not be loaded, and only assertion and covergroup data will be loaded. In "limited design" mode, to access assertion or covergroup data applications must iterate from the test handle. For example, to get the list of assertions:

```
assts = covdb_qualified_iterate(testHdl, assertMetricHdl,
                                        covdbObjects);
```

Applications can also control simple mapping (merging of data from different designs) using covdb_configure. The configuration item covdbMappedModule is set to the name of the module to be mapped. This has the same effect as the -map option to URG. Finer control of mapping is available using the mapfile - see "covdb_load_mapfile" .

You can opt for saving elfiles in the old database format using the covdb_configure API as follows:

```
covdb_configure(covdbExcludeFormat, "olddb");
```

The other value for the covdbExcludeFormat configuration is "newdb", which is the default.

covdb_qualified_configure

The covdb_qualified_configure is used to set configuration options. For implementation reasons, some configuration options require that the design handle be passed.

```
int covdb_qualified_configure(covdbHandle designH,
covdbConfigItemT item, const char* value)
```

For example,

```
covdb_qualified_configure(designHdl,
covdbMaxTestsPerCoverable, "5");
```

The configuration items for covdb_qualified_configure include:

- covdbKeepTestInfo — Default value "false". If set to true, the list of tests that covered each object will be preserved on that object when multiple tests are loadmerged. This data can be read using covdb_qualified_object_iterate.

- covdbMaxTestsPerCoverable —Default value "3". If set to non-zero, the value that you set for this API becomes the maximum number of tests preserved for each object when multiple tests are loadmerged.

APIs for Exclusion

UCAPI can load and modify exclude files whether they were created through DVE Coverage or manually. It can also be used to create exclude files itself, from scratch. This section describes the functions you can use to load an exclude file through UCAPI and save results.

Loading/Saving Exclude File

The following functions are used to load and unload exclude files for a design.

Function	Description
covdb_load_exclude_file	loads an exclude file
covdb_save_exclude_file	saves all the excluded coverage data into a .el file
covdb_unload_exclusion	unloads previously loaded .el file

covdb_load_exclude_file

The covdb_load_exclude_file function is used to load an exclude file:

```
covdb_load_exclude_file(covdbHandle design,
                        const char *filelocation);
```

After a successful call of covdb_load_exclude_file, all objects specified in the loaded file will be marked covdbExcludedAtReportTime, and these objects will no longer contribute to covdbCoverable or covdbCovered counts for themselves or any containing objects or regions.

The function covdb_load_exclude_file returns 1 on success and -1 on failure.

covdb_save_exclude_file

The covdb_save_exclude_file function is used to save exclusion data to a file:

```
covdb_save_exclude_file(covdbHandle design,
                              const char *filename,
                              const char *mode);
```

After a successful call of covdb_save_exclude_file, any objects that were marked covdbExcludedAtReportTime will be saved to the specified exclude file. If "w" is given as the mode, the file will be overwritten. If "a" is given as the mode, the excluded objects will be appended to the end of the file if the file already exists.

The function covdb_save_exclude_file returns 1 on success and -1 on failure.

covdb_unload_exclusion

The covdb_unload_exclusion function is used to clear all exclusions done by loading any previously-loaded exclude file.

```
covdb_unload_exclude(covdbHandle design);
```

All exclusions previously loaded are cleared. The function covdb_unload_exclusion returns 1 on success and -1 on failure.

covdb_save_attempted_file

In Strict mode, the covdb_save_attempted_file function saves the list of covered objects that the application attempted to exclude.

```
int covdb_save_attempted_file (covdbHandle design,
                       const char *filename,
                       const char *mode);
```

The attempted object details are overwritten into the file if the mode "w" is given, or appended if the mode is "a."

Types, Properties, and Relations

This section contains the complete list of all properties and relations in the UCAPI model.

Object Types

UCAPI has a small number of different object types. The type of an object may be retrieved from any handle, for example:

```
covdbObjTypesT objty = covdb_get(objHandle, covdbType);
```

The type of a UCAPI handle will always be one of:

```
typedef enum {
            covdbSourceDefinition,
            covdbSourceInstance,
            covdbDesign,
            covdbTest,
            covdbTestName,
            covdbMetric,
            covdbContainer,
            covdbSequence,
            covdbCross,
            covdbBlock,
            covdbAnnotation,
            covdbIterator,
            covdbIntervalValue,
            covdbBDDValue,
            covdbIntegerValue,
            covdbScalarValue,
            covdbVectorValue,
```

```
            covdbInterval,
            covdbVector,
            covdbBDD
        }
        covdbObjTypesT;
```

1-To-1 Relations

You can use these with the covdb_get_handle or
covdb_get_qualified_handle functions.

```
typedef enum {
            covdbIdentity,
            covdbParent,
            covdbDefinition,
            covdbBDDTrue,
            covdbBDDFalse,
            covdbVecValue,
            covdbFromValue,
            covdbToValue
        }
        covdb1To1RelationsT;
```

1-To-Many Relations

You can use these with the covdb_iterate or covdb_qualified_iterate
functions.

```
typedef enum {
            covdbObjects,
            covdbMetrics,
            covdbInstances,
            covdbDefinitions,
            covdbLoadedTests,
            covdbAvailableTests,
            covdbTests,
            covdbAnnotations,
```

```
                covdbComponents
            }
        covdb1ToManyRelationsT;
```

Object Properties

These properties can be used with the covdb_get, covdb_get_str,
and covdb_get_real functions, as noted:

```
typedef enum {
  /* integer properties for covdb_get */
                covdbLineNo,
                covdbWeight,
                covdbCoverable,
                covdbDeepCoverable,
                covdbValue,
                covdbType,
                covdbCovCount,
                covdbCovCountGoal,
                covdbCovered,
                covdbDeepCovered,
                covdbNumObjects,
                covdbIsVerilog,
                covdbIsVhdl,
                covdbAutomatic,
                covdbCovStatus,
                covdbWidth,
                covdbSigned,
                covdbTwoState,

  /* string properties for covdb_get_str */
                covdbName,
                covdbValueName,
                covdbFullName,
                covdbFileName,
                covdbSamplingEvent,
                covdbGuardCondition,
                covdbTypeStr,
                covdbParameters,
                covdbMessages,
```

```
            covdbTool,

  /* properties for covdb_get_real */
            covdbCovGoal
            }
        covdbPropertiesT;
```

Limitations

covdbNumObjects, covdbSamplingEvent, covdbParameters, and covdbGuardCondition are not yet supported.

Values

Objects of type covdbScalarValue return one of these enumerated values for the property covdbValue (objects of type covdbIntValue return an integer).

```
typedef enum {
        covdbValue0,
        covdbValue1,
        covdbValueX
    }
            covdbScalarValueT;
```

The figures below show how diagrams are used to indicate properties and relations in the user guide:
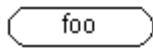
foo — Object of type foo
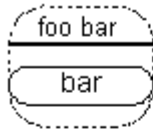
foo bar / bar — An abstract class foobar of which bar is a subtype
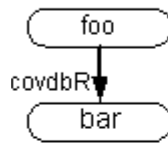
foo / covdbR / bar — One-to-one relation *covdbR* from an object *foo* to an object of type *bar*

bar = covdb_get_handle(foo, covdbR)
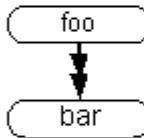
foo / bar — One-to-many relation *covdbR* from an object *foo* to an object of type *bar*

iter = covdb_iterate(foo, covdbR);
while((bar = covdb_scan(iter))) { ... }
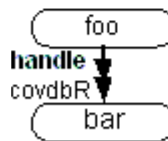
foo / handle / covdbR / bar — Qualified one-to-many relation *covdbR* from an object *foo* to an object of type *bar*, qualified by *handle*

iter = covdb_qualified_iterate(foo, handle, covdbR);
while((bar = covdb_scan(iter))) { ... }
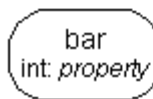
bar int: *property* — Object of type *bar* has integer property *prop*

6

Coverage GUI, Menu, and Toolbar Reference

This chapter describes the Menu and Toolbar commands and options in the DVE Coverage GUI and contains the following sections:

- "Coverage GUI Command-Line Options"

- "Menu Bar Options"

- "Editing Preferences"

- "Toolbar Options"

Coverage GUI Command-Line Options

Following are the DVE Coverage GUI command-line options and their descriptions:

```
dve [-assert minimal] [-cond ids] [-cov][-dir <dir>*] [-
elfile <file>*] [-excl_strict] [-f <file>]
[-fsm disable_loop] [-fsm disable_sequence] [-hier <file>]
[-line nocasedef] [-map <module>*] [-mapfile <file>*]
[-metric line+fsm+cond+tgl+branch+assert+group]
[-show availabletests] [-tests <file>] [-tgl portsonly]
[-elfilelist] [-excl_bypass_checks][-excl_resolve on]
[-excl_resolve off] [-excl_strict]
```

`-assert minimal`

> It is a database dependent option and you must use it with the `-covdir` or `-covf` options. When you invoke DVE using `dve -dir <x_dirs> -assert minimal`, DVE enters a special coverage mode. In this mode, DVE reports only functional coverage data and partial design information (without loading cm.decl_info). The Open Coverage Database dialog box disables code coverage metrics before populating the available tests, and does not allow you to select code coverage metrics (line, toggle, etc.). It also adds `-metric {assert group}` in the `gui_open_cov` command.

`-cond ids`

> Shows the expression and vector IDs in the Coverage Detail table.

`-cov`

> Starts up in coverage mode.

`-dir <dir>*`

> Open the coverage database in <dir>*.

`-elfile <file>*`

Loads exclusion files.

`-excl_strict`

Does not allow covered objects to be excluded.

`-f <file>`

Opens coverage directories listed in <file>.

`-fsm disable_loop`

Does not report loops in FSM coverage.

`-fsm disable_sequence`

Does not report sequences in FSM coverage.

`-hier <file>`

Specifies the module definitions, instances, hierarchies, and source files that you want to exclude or include for the report in <file>.

`-line nocasedef`

Disables uncovered case defaults report in line coverage.

`-map <module>*`

Reports on merging mapped modules coverage given by <module>*.

`-mapfile <file>*`

Reports on merging mapped modules coverage given in <file>*.

`-metric line`

Reports line fsm, cond, tgl, branch, cover directives, events and assertions, testbench coverage (Vera or NTB coverage groups).

`-show availabletests`

Lists the available tests for the given design.

`-tests <file>`

Opens coverage tests listed in <file>.

`-tgl portsonly`

Reports only ports in toggle coverage.

`-elfilelist`

Provide a list containing the names of the exclusion files to be loaded.

`-excl_bypass_checks`

Bypass checks when loading exclusion files.

`-excl_resolve on`

Enables adaptive exclusion.

`-excl_resolve off`

Disables adaptive exclusion.

`-excl_strict`

Does not allow covered objects to be excluded.

Menu Bar Options

The Menu bar contains the following menus and options:

File Menu

The following items comprise the **File** menu:

Open/Add Database	Displays the Open/Add Database dialog box, which enables you to select and open a coverage database.
Close Database	Displays the Close Database dialog box, which enables you to close an open coverage database .
Reload Database	Reloads the database.
Save Test	Saves the current test.
Open File	Displays the Open Source File dialog box, which enables you to select and display a source file in the Source Window.
Close File	Closes the source file displayed in the active Source Window or Window.
New HVP File	Creates a new HVP file.
Load HVP File	Loads an existing HVP file.
Save HVP File	Saves the current HVP file.
Save As HVP File	Saves the current HVP file with a new file name.
Close HVP File	Closes the current HVP file.
Manage User Data	Manages user data files.
Generate URG Report	Opens the URG window for you to select the URG command-line options for generating the URG report.
Import User Defined Groups	Opens previously defined cover groups from a file.
Export User Defined Groups	Saves to a file the user defined groups you used in your session.
Tests List	Lists all the tests that you have loaded in the GUI.
Load Session	Displays the Load Session Dialog which enables you to Load a saved session from a previously saved session file..
Save Session	Displays the Save Session Dialog which enables you to Save the current session to a session file.
Load Exclusions	Loads previously save exclusions from an exclusion file.

Save Exclusions		Saves current session's exclusions. from an exclusion file.
Recent Databases		Loads the recent database.
Recent Tcl Scripts		Loads the recent Tcl script.
Recent Sessions		Loads the recent session.
Close View/Pane		Closes the current focussed view or pane.
Close Window		Closes the currently active pane in the Top Level Window.
Exit		Exits DVE.

Edit Menu

The following items comprise the **Edit** menu:

Expand By Levels		Allows expansion by multiple levels with a single action.
Expand All		Expands the entire hierarchy at once. There may be a delay getting the hierarchy from the simulation when working interactively.
Collapse Parent		Collapses the parent of the selected scope.
Collapse All		Collapses all expanded scopes.
Select By Levels		Allows selection of more than 1 level at a time.
Select All		Selects all that are visible (does not implicitly expand)
Find		Finds specified text in a DVE pane or window. Field options vary depending on headers, if any, in the selected pane or window. Multiple Find dialog boxes can be open at any time with each identified by in the dialog box name.
Find Next		Finds the next occurrence of the search text.
Find Previous		Finds the previous occurrence of the search text.
Filters	Add	Adds filters.
	Remove All	Deletes previously defined filters.
Exclusion	Exclude	Excludes the selected item.
	Exclude Tree	Excludes the parent item.
	Unexclude	Unexcludes the selected item.
	Unexclude Tree	Unexcludes the parent item.

	Edit Exclude Annotation	Adds or edits an annotation to the selected items.
	Delete Exclude Annotation	Deletes the annotation from the selected items.
	Recalculate	Displays result with item excluded or unexcluded.
	Clear Marks	
Clear Exclusions		
Create/ Edit User Defined Groups		Manages user defined groups.
Preferences		Opens the Applications Preferences dialog box to allow customization of the display settings on a global or window basis.

View Menu

Apart from the Zoom options, following are the main options in the View menu:

Show Values		Annotates the display with coverage totals.
Show Condition IDs		Displays the condition IDs/values in the Condition Coverage Summary table.
Toolbars	Edit	Toggles the display of the Edit toolbar buttons.
	File	Toggles the display of the File toolbar buttons.
	Window	Toggles the display of the Window toolbar buttons.
	Exclusion	Toggles display of the Exclusion toolbar button.
	Navigation	Toggles the display of the Navigation toolbar buttons.
	Treemap	Toggles the display of the Map view navigation toolbar buttons.

Scope Menu

The following items comprise the Scope menu::

Show Detail		Displays source code and coverage results for the selected scope in the Source Window.
Show Coverage Map		Shows you the coverage map.
Show Coverage Table		Shows coverage table for the selected object.
Show	Current Scope	Displays the current scope.
	Parent	Displays the parent of the currently selected scope.
Edit Source		Opens an editor with the current source file.
Edit Parent		Opens an editor with the parent source of the current source file.
Navigation Criteria	Covered	Shows covered scopes.
	Uncovered	Shows uncovered scopes.
	Excluded	Shows excluded scopes.
	Any	Shows all scopes.
Backward		Goes to the previous scope.
Forward		Goes to the next scope.
Path up		Traces path up.
Path down		Traces path down.
Move Up to Parent		Displays the parent.of selected scope.
Move Down to Children		Displays children of selected scope.

Window Menu

The following items comprise the **Window** menu:

New	Coverage Detail View	Opens a Coverage Detail Window, if one is not already open.
	Coverage Table View	Opens a Coverage Table Window, if one is not already open.

	Coverage Map View	Opens a Coverage Map Window, if one is not already open.
Set the Frame Target For	Coverage Detail View	Opens a Coverage Detail Window, if one is not already open.
	Coverage Table View	Opens a Coverage Table Window, if one is not already open.
	Coverage Map View	Opens a Coverage Map Window, if one is not already open.
	Grading View	If selected, new coverage grading views will be created in this frame.
Panes	Console	Displays new console pane.
	Navigation	Displays new Navigation pane.
New Top Level Frame	Coverage	Displays new frame.
Load Default Layout		Returns display to default layout.
Load Layout	Reset Layout	Returns display to default layout.
	From File	Load layout from the file.
Save Current Layout	To Default	Saves the current layout as the default layout.
	To File	Saves the current layout in a file.
Cascade		Arranges all open workspace windows so they are displayed in a cascade pattern.
Tile		Arranges all open workspace windows so they are displayed in a horizontal tile pattern.
Dock in New Row	Left	Docks the selection to the left of the selected row.
	Right	Docks the selection to the right of the selected row.
	Top	Docks the selection to the top of the selected row.
	Bottom	Docks the selection to the bottom of the selected row.
Dock in New Column	Left	Docks the selection to the left of the selected row.
	Right	Docks the selection to the right of the selected row.
	Top	Docks the selection to the top of the selected row.

	Bottom	Docks the selection to the bottom of the selected row.
Undock		Undocks the selected window from the Top Level Window.

Help Menu
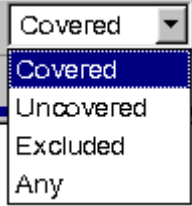
The following items comprise the Help menu:
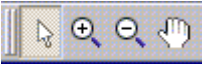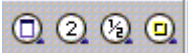
Help Contents	Displays HTML help for DVE Coverage.
Help Search	Displays the Search tab of the VCS HTML Help.
A Quick Start Example	Loads a DVE example coverage database.
About DVE	Displays DVE Coverage version and copyright information.

Toolbar Options

This section describes all Toolbar text fields, menus, and icons

Icon	Description
![icon] **Open/Add Database or File**	Displays the Open/Add Database or Open File dialog box, depending on the DVE window displayed, and enables you to select and open a VPD file.
![icon] **Search**	Searches for selection.
![icon] **Search Forward/Back**	Searches forward or back.
![icon] **Exclude**	Excludes selected item.
![icon] **Unexclude**	Unexcludes selected item.
![icon] **Recalculate**	Recalculates results with excluded/Unexcluded items.
Clear Exclusion ![icon]	Clears all exclusion.
Load Exclusions ![icon]	Loads exclusion from a file.
Save Exclusions ![icon]	Saves exclusion to a file.
![icon] **Move Up to Parent**	Displays parent of selected item.

Move Down to Child	Displays child of selected item.
Previous/Next	Goes to previous/next instance or scope.
Navigation Criteria	Displays a list of options to view scopes, such as covered, uncovered, excluded, or any scope.
Path UP/Path Down	Traces path up and down.
Coverage Detail Window	Opens a new Coverage Detail Window.
Coverage Table Window	Opens a new Coverage Table Window.
Coverage Map Window	Opens a new Coverage Map Window.

	From left to right:
	Selection tool – Uses to select objects. Zoom In – Makes objects in current view twice as big so fewer objects will be viewable. Zoom Out – Makes objects in current view twice as small so more objects will be viewable. Pan tool – Moves the view so the selected object is centered and viewable in the current pane. It does not change the zoom.
	From left to right: Zoom Full – Fits all viewable objects into the current view. Zoom in 2x – Doubles current view scale. Zoom out 2x – Halves current view scale. Zoom to selection– Moves the view so the selected object is centered and viewable in the current pane. It does not change the zoom.
	From left to right: Backward in Zoom and Pan History – Provides an easy way to go to the previous view. Forward in Zoom and Pan History – Provides an easy way to go to the next view. Named Zoom and Pan Settings – Chooses from any views that you saved with a name.

Editing Preferences

To set your display preferences

1. Select **Edit > Preferences.**

   The Application Preferences dialog box appears.

2. Select the **Global** category to set the font for menus or dialog boxes and other general settings.
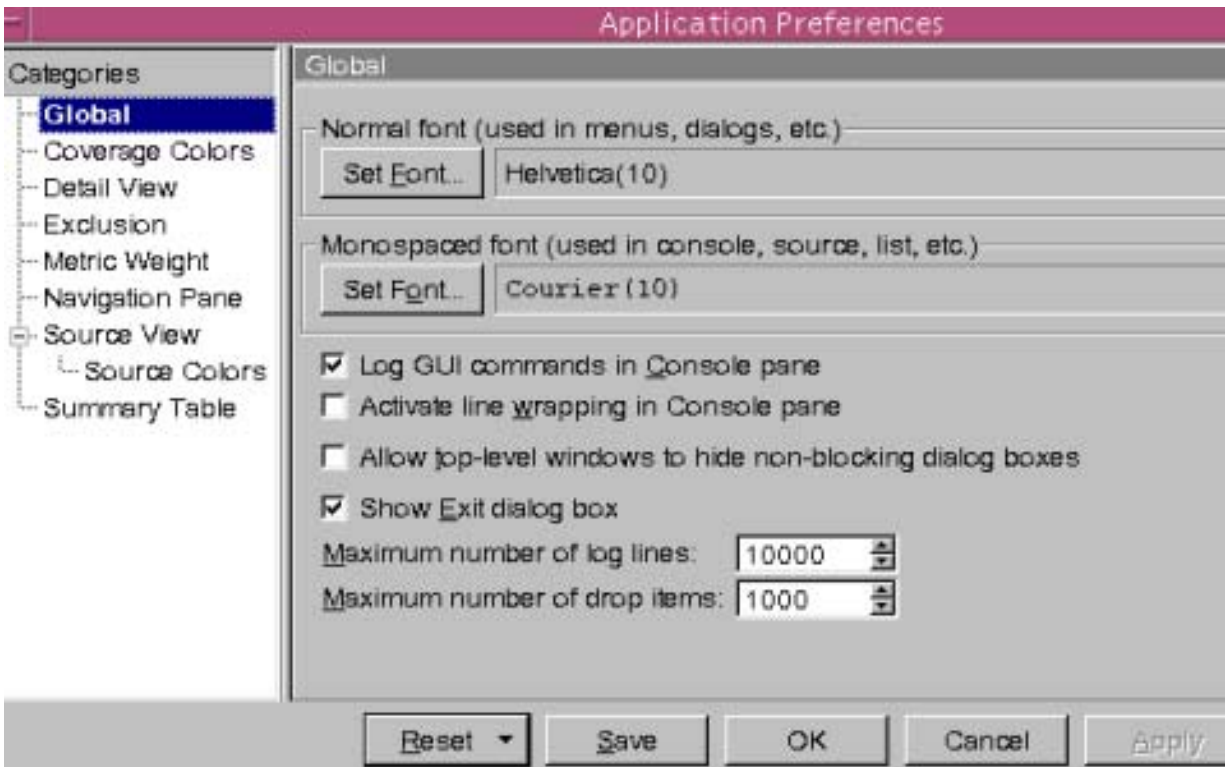
3. Select the **Coverage Colors** category to customize the color display of Source Window cover states, Coverage Map window, and the number of coverage ranges and their associated colors.

4. Click the **Exclusion** category and select any of the following options:

- **Save exclusion data when saving session** – Select this option when you want the exclusion data to get saved automatically while saving the session.

- **Remind to save exclusion data when reloading database or exiting** – Select this option when you want DVE to remind you while reloading your database or you are exiting DVE.

- **Do not allow covered objects to be excluded** – Select this option when you do not want to exclude the covered objects. This option is to enable the Strict exclusion mode. You will need to reload the database after changing the exclusion mode.

5. Select the **Metric Weight** category to specify weight for each metric listed to be used for computing the score of each object in the Summary table or Map window.

   The score is a weighted average of the coverage percentage of each metric. You can remove a metric from score computation by setting its weight to 0.

6. Select the **Navigation Pane** category to set display options for the Navigation pane.

7. Select the **Summary Table** category to display or hide the detail values in the table.

8. Click **Apply** to save the settings or **OK** to close the Applications Preferences window.

Index