

VCS® / VCSi™ LCA Features

Version C-2009.06
June 2009

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

SYNOPSYS®

Contents

1. Using the DVE Coverage GUI	
Overview	2
Starting the DVE in Coverage Mode	4
Opening a Database	4
Loading and Saving Sessions	9
Saving a Session	9
Loading a Session	9
Using the Coverage GUI	11
The Navigation Pane	11
The Coverage Summary Map Window	15
The Coverage Detail Window	16
Navigating the Source Pane	17
DVE Coverage Source / Database File Relocation	20
Displaying Code Coverage	21
Displaying Assertion Coverage	33
SVA Naming Convention	35
OVA Naming Convention	35
The Covers Tab	37

Displaying Branch Coverage and Implied Branches	38
Displaying Testbench Coverage	39
Working with HVP Files	43
Working with Coverage Results	44
Running Scripts	44
Filtering the Hierarchy Display	45
Setting Display Preferences.	46
Coverage Menu and Toolbar Reference	48
Menu Bar	48
Edit Menu.	49
Toolbar Reference	53
2. VMM Planner MS Doc Annotation	
Introduction	58
Use Model	59
Process Flow	60
License Model	60
Supported Platforms	61
Compatibility with the VMM Planner Spreadsheet Annotation Flow	61
hvp annotate Command Arguments	61
Capturing a Verification Plan in Doc XML Format.	64
Built-In Styles	64
Table Keyword	68
Other Contents.	73
Debugging the Doc Plan	73
[Style Table] Keyword Indicator	74

Unique Error Code in Error Messages	76
How to Get Scores Back-Annotated in the Doc.	76
Plan/Feature Score Summary	77
Measure Score Table	78
Navigating Down to a Subplan	79
 3. SV Cover Property Path Coverage	
Using Cover Property Path Coverage	82
Summary table of properties with path coverage	85
Details for a Property	85
Details for built-in checkers	88
Other Changes	90
Computing the score in URG	91
 4. Using the SVAPP Utility	
Overview	93
Use Model	94
Example using SystemVerilog Testbench	96
Limitations	99
 5. Using the let Construct	
let Construct Overview	103
let Construct Examples	107
let Construct Syntax	116

6. New std::randomize() Function	
7. Support for IEEE Verilog Encryption	
Encryption Pragmas	127
8. Sequential Distance Coverage for Assertions	
Overview	132
Sequential Distance Analysis	133
Rules for Specifying Sequential Distance	136
NTL_COV_ASSERT01	136
NTL_COV_ASSERT02	138
NTL_COV_ASSERT03	139
NTL_COV_ASSERT04	140
NTL_COV_ASSERT05	140
NTL_COV_ASSERT06	141
Rule Output	142
NTL_COV_ASSERT01	142
NTL_COV_ASSERT02	142
NTL_COV_ASSERT03	143
NTL_COV_ASSERT04	143
NTL_COV_ASSERT05	143
NTL_COV_ASSERT06	144
Type of Assertions	144
9. Testbench Separate Compilation	
Overview	146
Use Model	148

Specifying Logical Libraries	149
Analyzing the Code	150
Generating Shared Object Library	154
Generating Shell File	156
Generating simv	157
Linking Partitions Dynamically at Runtime	157
Usage Notes	158
DesignWare VIP	158
XMR in Testbench-to-DUT or Top-Module Tasks	161
Parameterized Programs	163
Parallel Compilation	165
Random Stability	165
Testbench Separate Compile Flow Notes	166
NTB OpenVera/SystemVerilog Interoperability	169
Analyzing the Code	169
Generating Shared Object Library	170
Generating Shell File	171
Generating simv	171
Linking Partitions Dynamically at Runtime	171
Testbench Separate Compile Flow Notes	171
 10. Unified Exclusion from Coverage Analysis	
Using DVE Coverage with Unified Exclusion	174
Coverage Exclusion with DVE	174
Understanding and Excluding Half-Toggle Transitions	190
Excluding Covergroups	193
Using URG with Unified Exclusion	197
Generating URG Reports Using Exclude Files	197

Exclusion in Strict Mode	199
Covergroup Exclusion in URG	199
Editing Exclude Files	200
The Exclude File Format	200
Using UCAPI with Unified Exclusion	211
Loading/Saving Exclude File	211
 11. Back Tracing X Values for Gate-level Designs	
Setting the Back Trace Properties	217
Using the Back Trace Schematic Toolbar	219
 12. Constraints Features	
Using String Indexed Associative Arrays in Constraints	222
Using the array.exists() Function in Constraints	225
 13. Coverage Features	
Turning Coverage Collection Off for Covered Objects	228
General Use Model Flow of cg_coverage_control=2	228
Limitations	230
URG Difference Reports for Functional Coverage	231
Diff Results Shown in the Dashboard and Test Pages	232
What is Shown as Covered	233
Unsupported Flags	235
Exclusions	236
Correlation Report: Which Tests Covered Which Bins	237
Covered Objects	238
Tests Page	240

Unsupported Arguments	241
Reporting Only Uncovered Objects	242
Command-Line Access	242
Report Changes	242
Detailed Coverage Reports	247
Hierarchical Covergroups	254
Constituent Covergroup Instances	255
Sampling Hierarchical Covergroups	257
Cross of Crosses	260
 14. Coverage Convergence Technology	
Compile-Time Options	264
Coverage Convergence Option	264
Coverage Model Autogeneration Options	264
Coverage Convergence Runtime Options	266
URG Options for Bias File Generation	267
Coding Guidelines	268
Constraints and Coverage Model on the Same Variables	268
No Procedural Overwriting of Values Generated by the Solver	268
Coverage Should Be Sampled Between Consecutive Calls to randomize	269
Use Open Constraints	269
Avoid Explicit or Implicit Partitioning in Constraints	270
Avoid In-line Constraints and the Use of “constraint_mode” and “rand_mode”	270
Automatic Generation of a Coverage Model from Constraints	270
Coverage Groups	271
Contribution to Coverage Scoring	278

Coverage Model Inference for In-line Constraints	278
Use Model	279
Understanding Bias Files and Coverage Convergence.	280
Motivation	280
What Is a “Test”?	280
Using Coverage Convergence Bias File to Target Coverage Holes	
281	
Automatic Generation of Coverage Convergence Bias Files	283
Repeatability of Test Results for Parallel Regression Runs	284
Usage Scenarios	284
Running a Single Test with Randomized Configurations	285
Running a Single Test with Randomized Transactions	285
Using a Bias File for a Parallel Regression	286
Autogenerating a Coverage Model	287
Methodology and Flow Issues.	288
Scenario: All Tests Have the Same Constraints and Coverage Space	
(Recommended)	288
Scenario: Tests Are Grouped Into Categories With Each Category	
Having Specific Test Constraints	289
Scenario: Coverage Database Being Loaded in the Beginning of a Test	
Run	290
15. VCS Multicore Technology	
Design Level Parallelism	
VCS Multicore Technology Options.	293
Use Model for Design Level Profiling and Simulation	295
Use Model for Assertion Simulation.	298
Use Model for Toggle and Functional Coverage	298
Use Model for VPD Dumping.	299

Running VCS Multicore Simulation	299
Design Level Simulation	299
Assertion Simulation	300
Toggle Coverage	300
Functional Coverage	302
VPD File.	304
Profiling a Simulation.	305
Profiling a Serial Simulation.	305
Specifying Partitions	306
Profiling a VCS Multicore Simulation	309
Running VCS Multicore Examples.	324
Supported Platforms	332
Current Limitations	332
16. SystemVerilog Assertions	
Use Model	333
IEEE Std. 1800-2005 Compliant Feature	334
IEEE P1800-2009 Draft 8 Compliant Features	334
Limitations	336
Debug Support for New Constructs	337
Note on Cross Features.	337
17. Using HDL and SystemC Sync Loops	
The Coarse-Grained Sync Loop	340
The Fine-Grained Sync Loop	340
Run Time	340
Alignment of Delta Cycles	341

Example Syntax	342
Restrictions	342
Restrictions That No Longer Apply	343

1

Using the DVE Coverage GUI

This chapter describes how to use DVE to view and examine coverage results. It includes the following sections:

- [“Overview”](#)
- [“Starting the DVE in Coverage Mode”](#)
- [“Opening a Database”](#)
- [“Loading and Saving Sessions”](#)
- [“Using the Coverage GUI”](#)
- [“DVE Coverage Source / Database File Relocation”](#)
- [“Displaying Assertion Coverage”](#)
- [“Displaying Branch Coverage and Implied Branches”](#)
- [“Displaying Testbench Coverage”](#)

- [“Working with HVP Files”](#)

Overview

Visualization for coverage data in the DVE (Discovery Visualization Environment) is a comprehensive visualization environment which integrates with the design hierarchy display and can be used to get a summary of the coverage results or details of various types of coverages. Besides features provided by URG (Unified Report Generator), DVE coverage also introduces more advanced feature enabling interactive operations, such as excluding parts of design. For more information about URG, see the *URG User Guide*.

DVE can display three kinds of coverage information:

- Code coverage information — DVE can display the following types of code coverage information:
 - Line coverage — what lines or statements were executed during simulation.
 - FSM coverage — VCS and VCS MX can identify blocks of code that make up finite state machines or FSMs. FSM coverage reports on the states, transitions, and sequences of states during simulation.
 - Toggle coverage — whether the signals in the design toggle from 0 to 1 and 1 to 0 during simulation.
 - Condition coverage — conditions are expressions and sub-expressions that control the execution of code or the assignment of values to signals. Condition coverage tests whether both true and false states of these conditions were covered during simulation.

- Branch coverage — analyzes how if and case statements and the ternary operator (?) establish branches of execution in your Verilog design. It shows you vectors of signal or expression values that enable or prevent simulation events.
- Testbench coverage — coverage of SystemVerilog and OpenVera testbench coverage groups.
- Assertion coverage — coverage of OpenVera and SystemVerilog cover directives.

For more information about generating coverage databases, see the *VCS / VCS MX Coverage Metrics User Guide*.

Starting the DVE in Coverage Mode

To start the DVE in coverage mode, enter the `dve` command with a coverage command line options as follows:

- `-cov`
Opens the DVE Coverage GUI.
- `-covdir`
Opens the coverage database in the given directory.
- `-covf`
Opens the coverage directories listed in the given file.
- `-covtests`
Opens coverage tests listed in the given file.
- `-covavailabletests`
Lists the available tests for the given design.

The next step is to open a coverage database.

Opening a Database

In DVE, you can open one of the three types of coverage databases to display coverage information.

You can select one of the following kinds of databases:

- A code coverage directory (by default named `simv.cm` by VCS (or VCS MX with a Verilog top design).

- An OpenVera or SystemVerilog assertions database directory (by default named simv.vdb by VCS).
- A testbench (by default named simv.vdb) directory containing testbench coverage files.

When you select a code coverage database (.cm), DVE looks for a .vdb database with the same name in the same location and selects it for opening too. And similarly, if you select a .vdb database, DVE looks for a .cm database with the same name and same location to open.

Note:

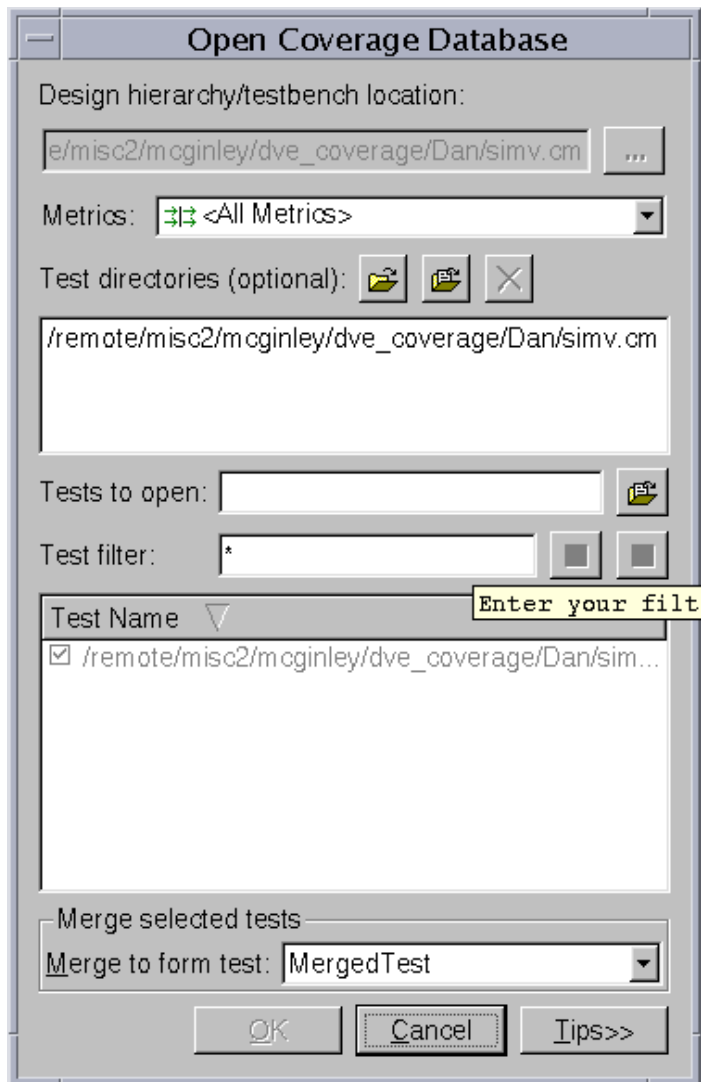
DVE can load only one .cm/db directory at a time.

To open a coverage database

1. Click **File > Open Database**.

The Open Coverage Database dialog box appears.

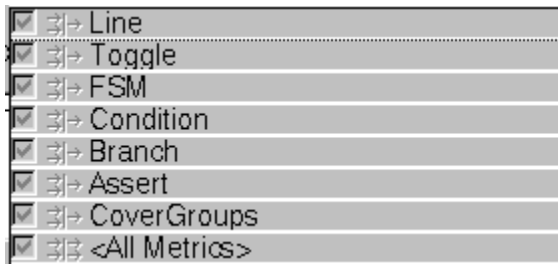
Figure 1-1



2. Click the **Browse** button to locate and specify the design hierarchy/testbench location.


The Browse for the design hierarchy/testbench location dialog box appears.

3. Select the metrics you want to include.



4. Select the code coverage directory (by default named simv.cm) or the OpenVera or SystemVerilog assertions database directory, then click the **Select** button.

5. (Optional) Click the following buttons above the test directories list box to add and remove the directory:


-  Adds a test directory.


-  Adds the test files.


-  Deletes a selected item from the list.

6. Select tests to manage one or more test files from the **Available tests** list box.

-  Displays a dialog box for selecting a filter file.

-  Toggles enabling and disabling of a filter file.

-  Selects all tests in the list. If you select more than one test file, the results of the tests are merged.

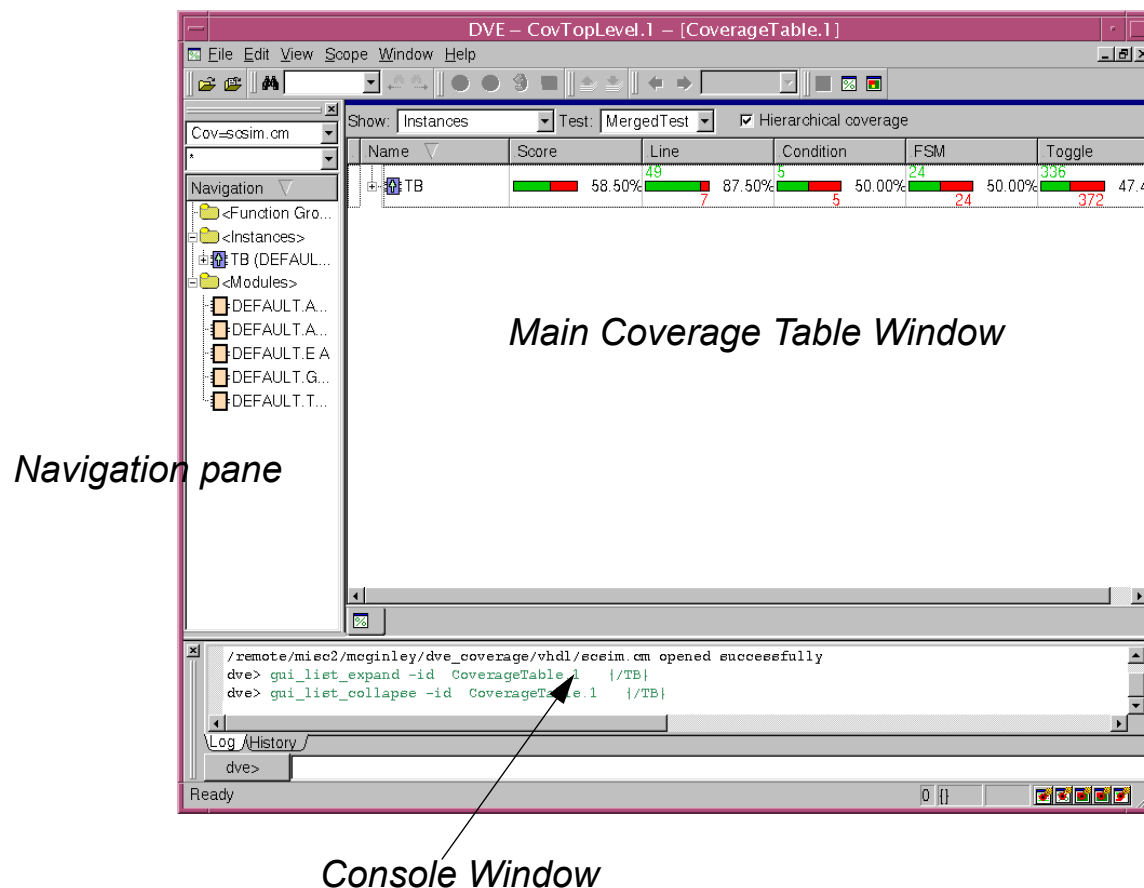
-  Clears all files in the list.

If you select more than one test file, the results of the tests are merged. Note that you can also check and clear tests in the list.

7. Accept the default name or enter a new name for the test results.
8. Click **OK**.

The DVE coverage windows are populated with coverage information.

Figure 1-2



Loading and Saving Sessions

This section describes how to save a session and load a previously saved session.

Saving a Session

You can save all session data or arrangement of windows, views and panes for later reuse.

To save a session

1. Select **File > Save Session**

The Save Session dialog box appears.

2. Select one of the following:

- Save all session data including window layout, the current view and its contents, and coverage databases.
- Save the arrangement of windows, views, and panes.

3. Enter a file name.

4. Click **Save**.

The session is saved as a Tcl file for future use.

Loading a Session

To load a previously saved session

1. Select **File > Load Session**.

The Load Session dialog box appears.

2. Browse to a previously saved DVE session file (.tcl), then select the file.

When you select a session file, the right pane displays information on the session file.

3. Click **Load**.

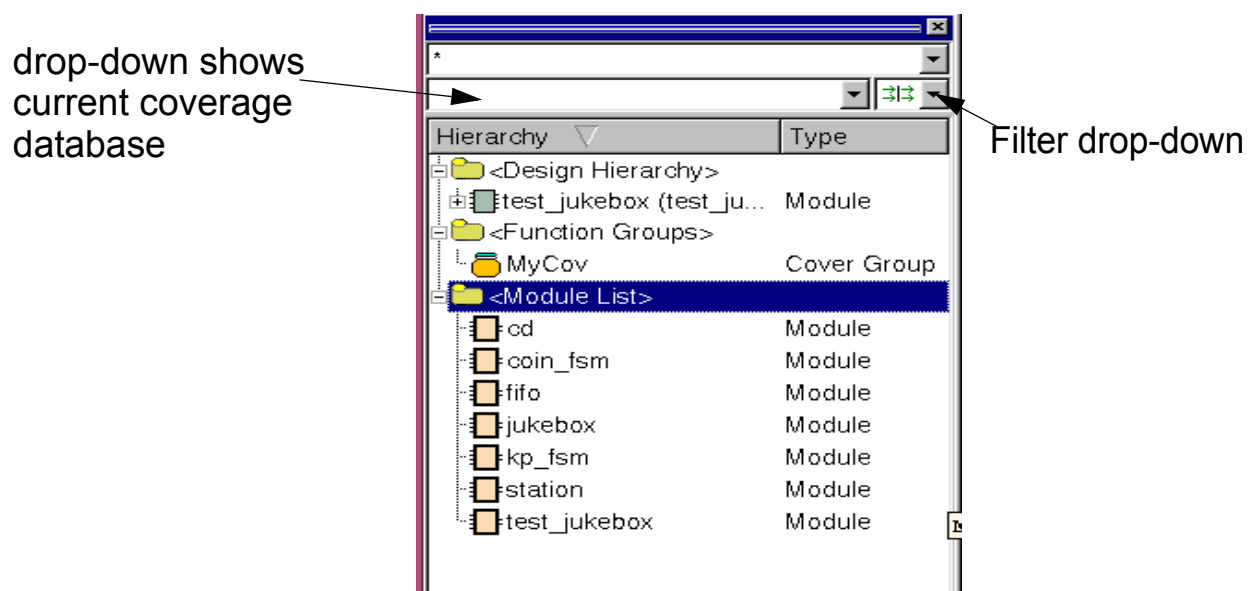
The session is loaded.

Using the Coverage GUI

This section describes how to use the DVE Coverage GUI.

The Navigation Pane

Figure 1-3



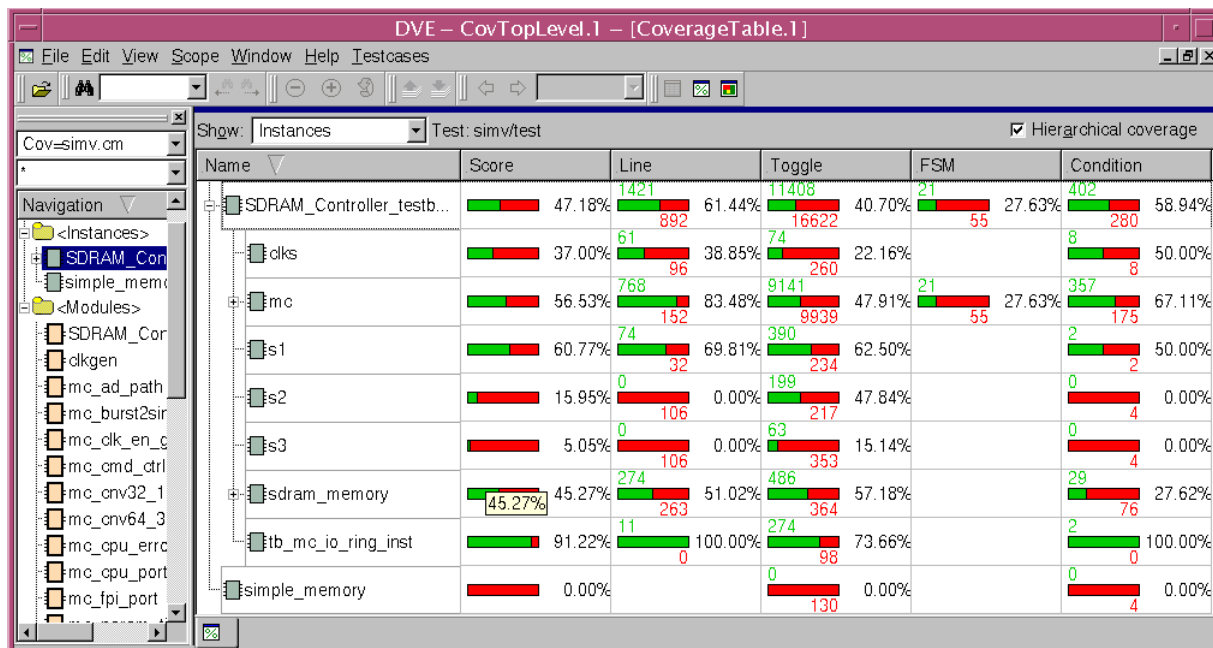
The Navigation pane is to navigate to the required database. It consists of the following items:

- Design Hierarchy - contains the module instances in the hierarchy. Click the plus (+) next to the top-level to see the names of the module instances in the top-level module. They are listed by instance name with their corresponding module name in parentheses.
- Function Groups - contains the testbench coverage groups in your testbench database and the OpenVera (OVA) and SystemVerilog (SVA) assertions and cover sequences and properties.

- Module Lists - lists all the module definitions in the design
- Text drop-down - shows the list of the coverage databases that you have loaded recently. If you want to load another database, you must close the database using the "Close Database" option in File menu.
- Filter drop-down - maintains a history of previously applied filters. You can see the history and select a filter to apply.

The Coverage Summary Table Window

Figure 1-4

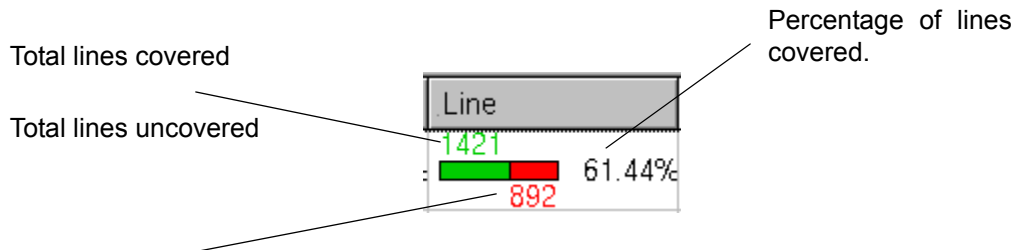


The Main Coverage Table window gives you summary information for each type of coverage. For example, [Figure 1-5](#) shows the details for lines covered.

To view the coverage details, select a cell and click **View > Show Values**.

The coverage detail appears as follows;

Figure 1-5



The Coverage Table window contains the following fields and options:

Show

Specifies the following options;

Functional Groups

Displays results by user defined groups including total score and scores for cover directive and covergroup coverage.

Instances

Displays results by module instances for including total coverage score and individual scores for line, condition, FSM, and toggle coverage. There is a row for each module instances (click the plus (+) to the left of the top-level module to see the instances under it). A plus down the design hierarchy indicates new instances to be revealed.

Modules

Contains module names instead of instance names. There is a row for each module definition. The coverage information displayed for each module definition is a union of the information that would be displayed for each instance of the module.

Test

Provides the name of the merged test that you specified in the Open Coverage Database dialog box, see [Figure 1-1 on page 6](#).

Hierarchical Coverage

Specifies whether to show hierarchical or local coverage numbers in the table. If it is checked, for each instance, functional group, or module, the numbers are aggregated for all objects hierarchically under the selected item. When you clear this check box, the coverage numbers are for the particular item and do not include the objects under them. When the Show field is set to Modules, there is no hierarchical display possibility and this check box disappears.

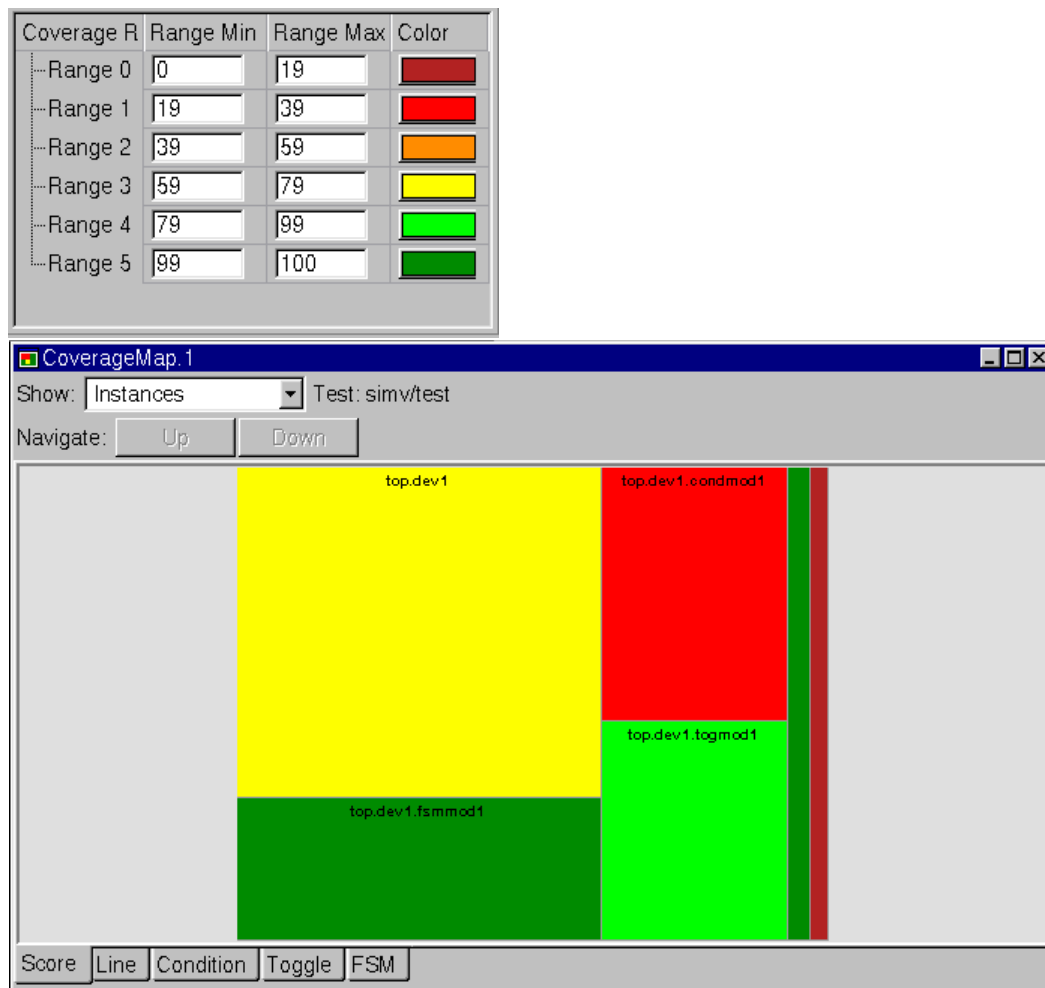
Detail Window

Displays coverage for each coverage type available. By default, the Detail Window shows the coverage for the metric corresponding to the clicked column for the object in the clicked row. Click a cell to view the Detail window.

The Coverage Summary Map Window

Use the Map window for a quick view of the overall coverage results. The map window tabs displays results in customizable color-coded ranges to help identify coverage areas of interest and overall progress. [Figure 1-6](#) shows a coverage map and the default color key and range.

Figure 1-6



You can change the root of the map by dragging and dropping a functional group, instance, or module from any other window.

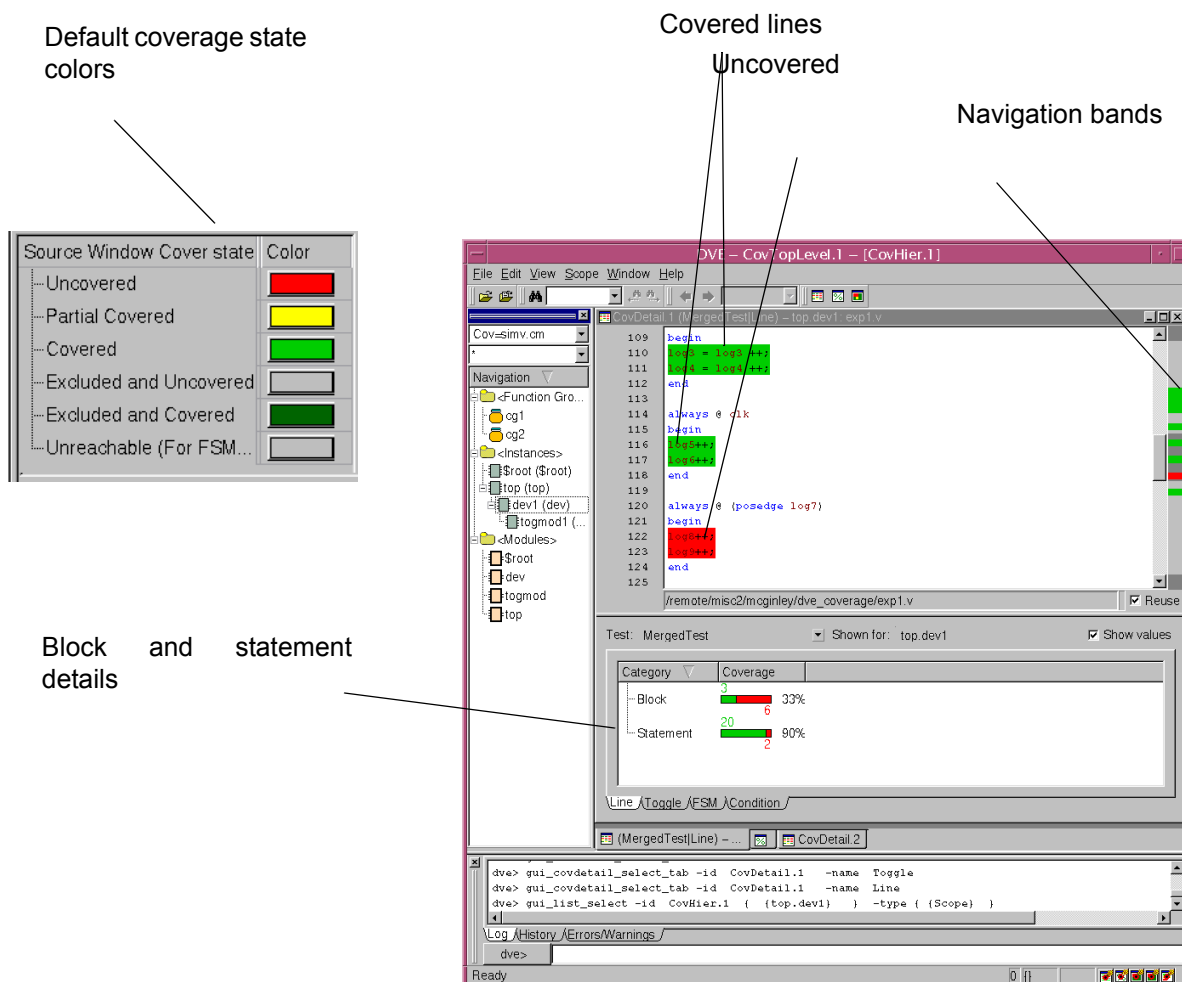
Drill down into the view by selecting a rectangle. Double-clicking the selected rectangle changes the view to display its children.

The Coverage Detail Window

The Coverage Detail Window displays:

- The source file with annotations indicating covered and uncovered lines or objects.
- The **Coverage Detail** tabs where you select the coverage type to view.

Figure 1-7



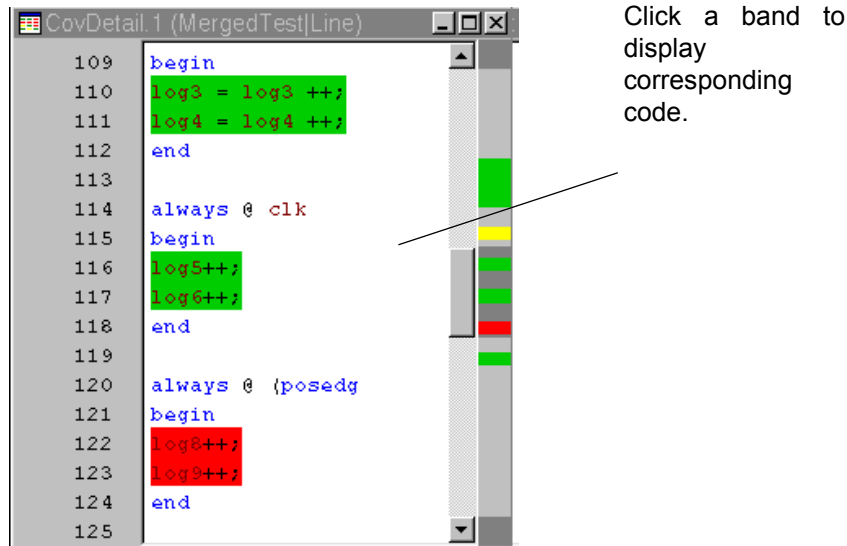
Navigating the Source Pane

Use the navigation bands in the source pane to view coverage results for the source code. The bands are color coded to display the coverage range. To navigate to an area of interest, click on a band to display the color coded source line.

Note:

When viewing annotated data, DVE requires the original source files to be unaltered since compilation.

Figure 1-8



Creating User-Defined Groups

You can create and modify user-defined groups to organize and navigate to your functional coverage elements such as cover groups and assert/cover statements.

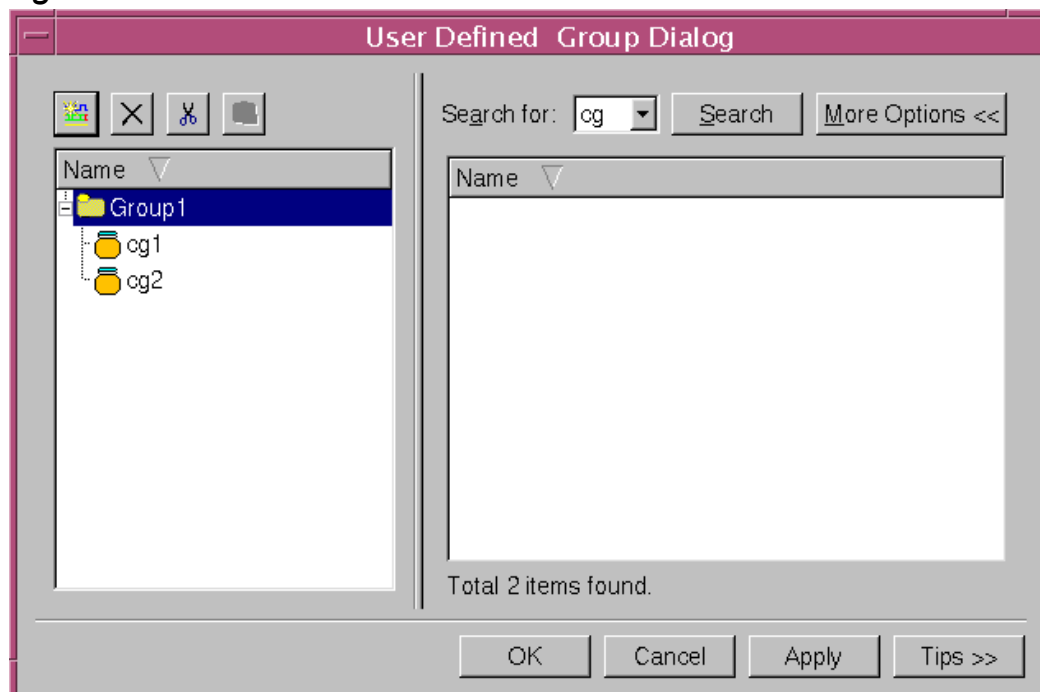
To create a user defined cover group


1. Click **Edit > Create/Edit User Defined Groups**.

The User Defined Group dialog box appears.

Cover groups and assertions in your project are listed in the left pane.

Figure 1-9



2. Click the Create new group icon , then accept the default name or name your group.
3. Select cover groups and assertions from the Name list, search using a search string, or click More to specify search options and criteria.
4. Drag and drop desired items into your created group.
5. Click **OK**.

Your group gets created and the dialog box closes. You can click **Apply** to save the group. The dialog box remains open and you can make more groups.

DVE Coverage Source / Database File Relocation

If you change the location of a coverage file, and then attempt to load that file with DVE, DVE displays a dialog asking you to provide the new location of the file.

After you provide DVE with the new location of the file, DVE subsequently uses both the built-in location and the new location information you have provided.

Moreover, DVE compares the two locations to make a more intelligent attempt at locating the file.

For example, assume that the location of a file `foo.v` (as specified in the coverage database) is as follows:

```
/A/B/C/foo.v
```

But assume that you moved the file to this location, which you provided to DVE:

```
/X/Y/C/foo.v
```

Thereafter, DVE compares these two locations and, working from right to left, determines that the following location is the new root directory for design files is as follows:

```
/X/Y
```

Assume now that DVE performs a search for the following file, based on database information:

```
/A/B/E/bar.v
```


Instead of looking in /A/B, DVE searches in /X/Y, the correct location of this file:

```
/X/Y/E/bar.v
```

If DVE cannot find the file using the default location, then DVE displays a dialog asking for the correct location of the file.

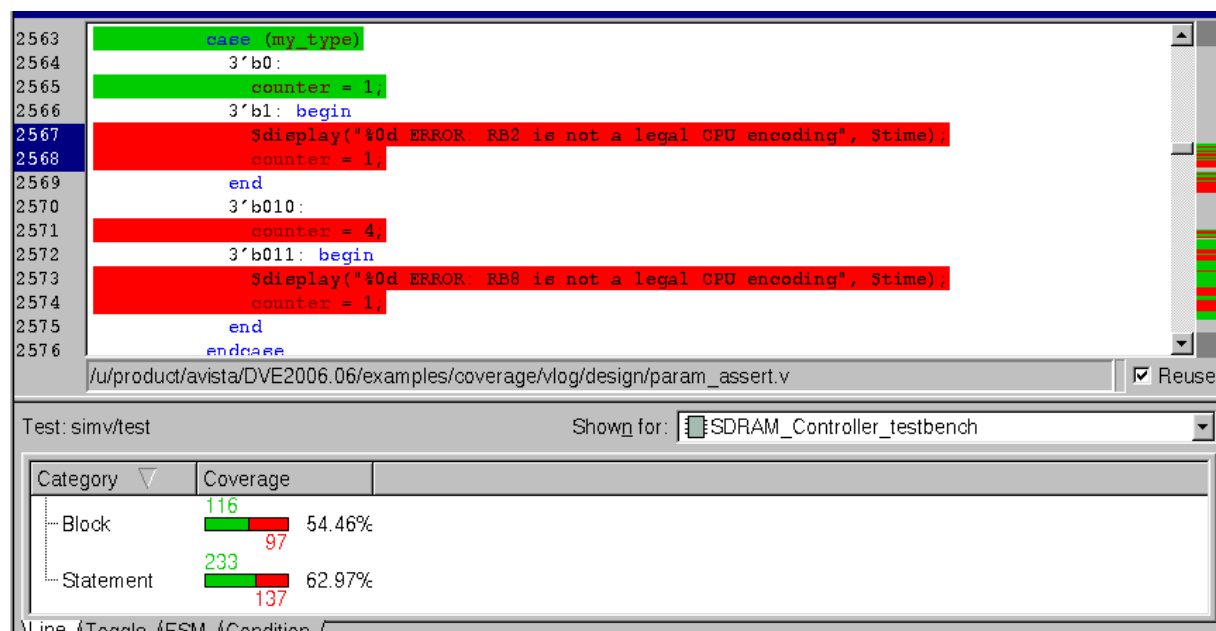
DVE can only remember one root location directory at a time. DVE replaces any earlier root location directory with whichever new location you specify.

Displaying Code Coverage

Displaying Line Coverage

The Line Coverage Detail window displays annotated source code and coverage percentage for total block and statement coverage.

Figure 1-10



- The Annotated source window allows you to view the source files annotations that shows you which code has and has not been covered. The highlighted colors can be changed using the Colors tab in the Preferences dialog box as described in the section [“Setting Display Preferences”](#) . Color banding adjacent to the scroll bar aid in navigating the code. [Figure 1-7](#) shows the Source Window with annotations, and the default annotation colors.
- The Detail Window for line coverage displays metrics for coverage of blocks and statements. A block of code is a group of statements that always will be executed together.

This display tells you how much of your code is actually executed by the tests. The number of covered and uncovered lines give you the information you need to write the tests to complete the coverage.

- Use the **Shown for** drop-down to look at the coverage of other instances or module variants corresponding to the current instance or module.

Displaying Toggle Coverage

This section describes toggle coverage and contains the following sections:

- [“Nets and Registers”](#)
- [“Multi-Dimensional Arrays”](#)

Nets and Registers

The **Details Window Toggle** tab, shown in [Figure 1-11](#), displays results for each net and register for any value transition from 0 to 1 and 1 to 0.

The Annotated Source window displays nets and registers highlighted in colors indicating their coverage range as shown in the figure.

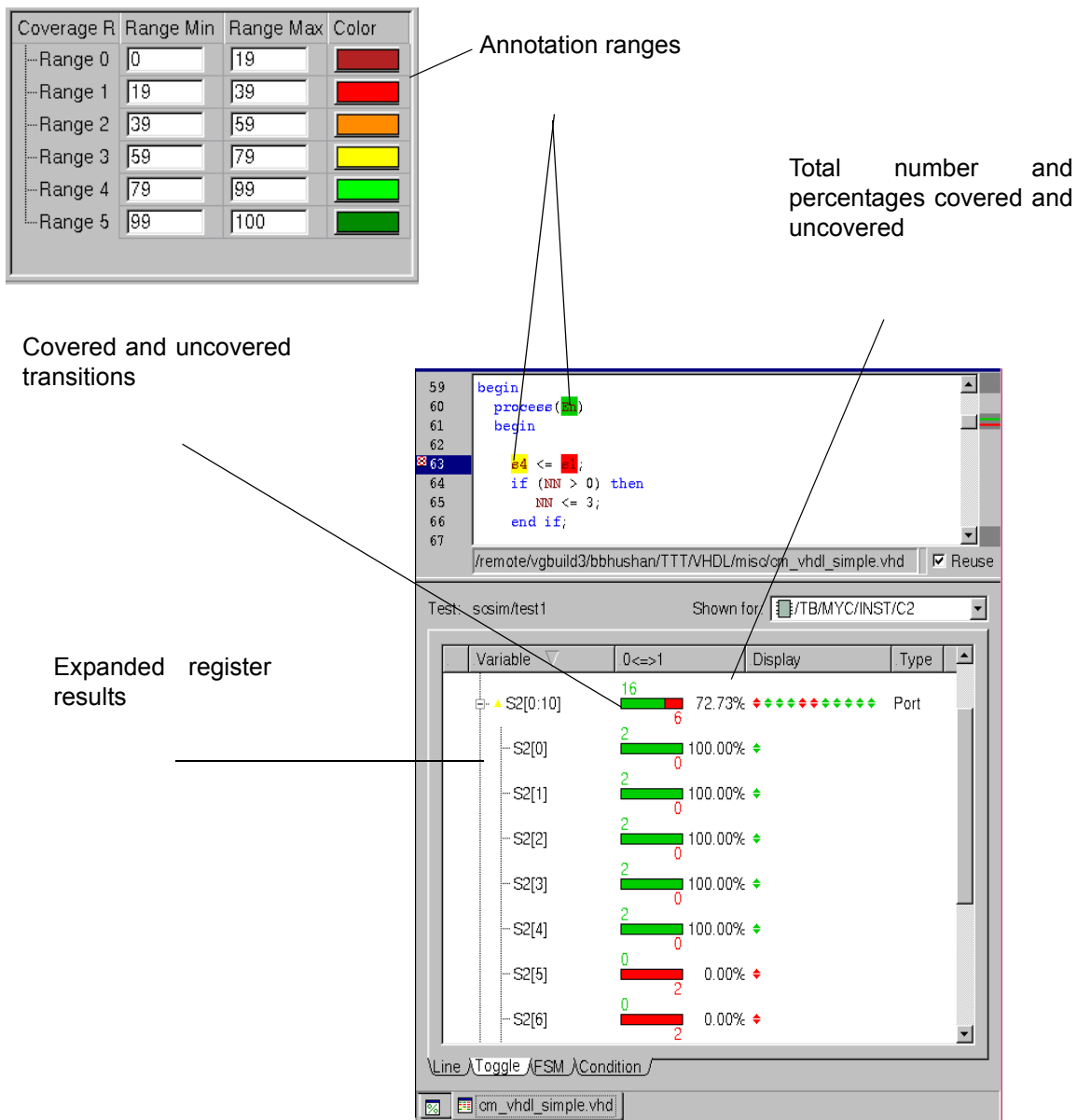
Use the **Shown for** drop-down to look at the coverage of other instances or module variants corresponding to the current instance or module.

Metrics for total coverage of nets and registers tell you how much activity is occurring on all the elements, thus giving you a clear indication of how much testing is actually being performed at the gate. The Variable column identifies the nets and registers and expands to display each bit in a register.

An up arrow indicates the coverage of a toggle from 0 to 1, and a down arrow indicates coverage for a toggle from 1 to 0. The arrow is colored green if the toggle is covered and red if it is uncovered. The column displays the total number covered and uncovered in the design and the coverage percentage.

In the following example, the register S2(0:10)) has been expanded to display results for each of the bits. The Display column indicates covered possible transitions for the register with pairs of up-down arrows and each bit's two transitions are shown.

Figure 1-11 Toggle Detail Window



Multi-Dimensional Arrays

Multi-dimensional arrays (MDAs) are displayed as toggle signals. The top level toggle signal is named with all its indices. Its value are a contiguous string of values of all its leaf level words.

The top level MDA is expandable to its next level of indices just like a vector. Each of the MDA's layers will be expandable until the bit level. Just like that for a vector, the coverage information of each layer of the MDA consists of coverage for 0->1 and 1->0 transitions.

The cumulative score of each layer in MDA is determined by calculating the covered/total bits in that MDA layer.

Tooltip on the coverage graphic indicates the bit name on which the mouse is pointed.

- A green arrow indicates covered.
- A red arrow indicates uncovered.
- A grey arrow indicates excluded bit.

The top level MDA, any mda layer in the middle, the word and bits - any of these may be excluded.

In the **Type** column, identifier MDA indicates that it may be expanded into multiple layers.

Displaying Condition Coverage

The Condition Detail Window helps you monitor whether both true and false states of the conditional expressions and sub-expressions in your code are covered.

- The Condition list expands to show coverage by possible vectors or subconditions, percent coverage, and the line number of the expression.. By default, condition coverage only lists sensitized vectors. For more information about sensitized vectors or how to monitor coverage of all vectors, see the *VCS/VCS MX Coverage Metrics User Guide*.

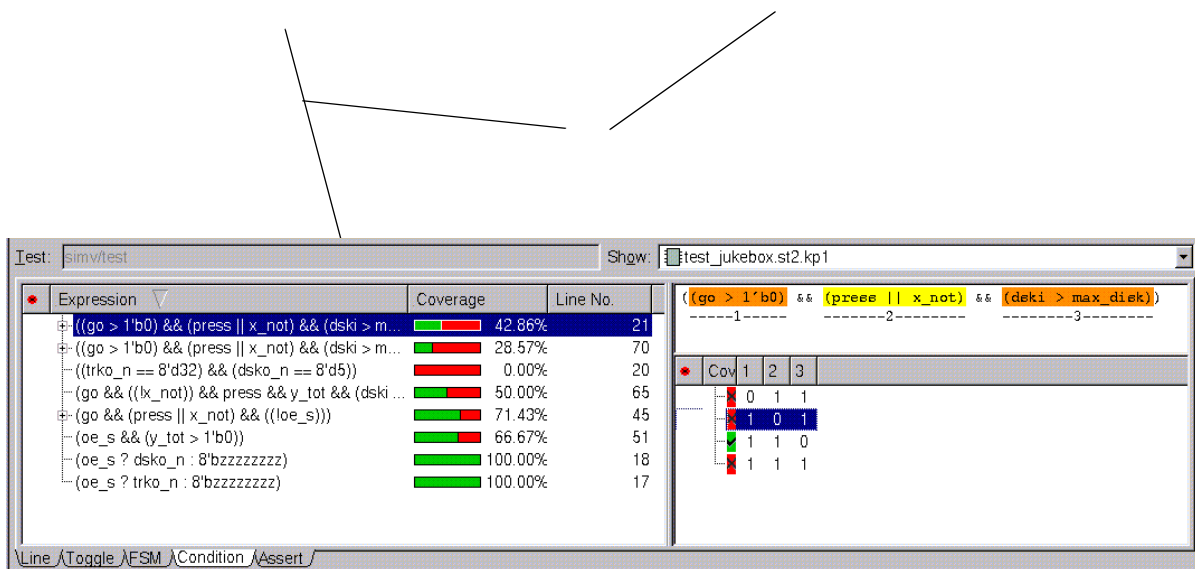
- The right pane displays the selected expression above with possible conditions and their coverage status below.
- Use the `Show` drop-down to look at the coverage of other instances or module variants corresponding to the current instance or module.

Figure 1-12 shows condition results for a simple expression.

Figure 1-12 Condition Detail Window

Expression coverage
totals and line numbers

Selected conditional expression



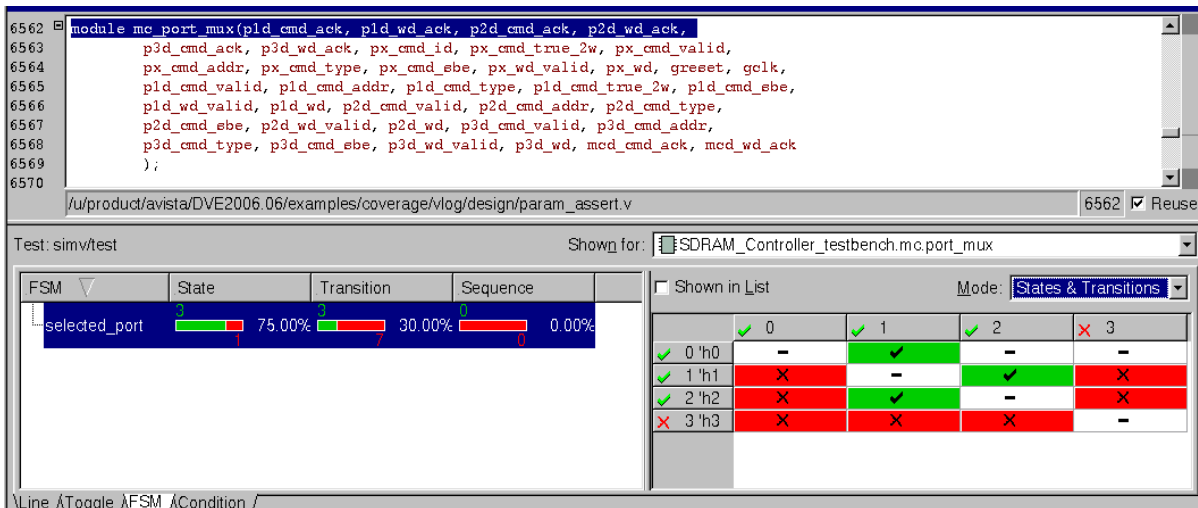
Displaying Finite State Machine (FSM) Coverage

The default FSM coverage tab has two views:

- The left pane displays state, transition, and sequence coverage for each FSM with a percentage bar annotated with the total number of covered and uncovered and the percentage of each.

- The right pane displays transition or sequence results for all possibilities in table or list format.

Figure 1-13 Default FSM Detail Display



Displaying Transition Details

The Transition Mode for the selected FSM displays all the possible transitions and whether the current state of the FSM made any of these transitions. You can display the results in table or list format. [Figure 1-14](#) shows the table view with states and transitions displayed with covered transitions in green and uncovered transitions in red. Transitions that are not possible are white with a dash.

Figure 1-14 FSM Transition Detail Table

<input type="checkbox"/> Shown in List	Mode: States & Transitions			
	✓ 0	✓ 1	✓ 2	✗ 3
✓ 0 'h0	-	✓	-	-
✓ 1 'h1	✗	-	✓	✗
✓ 2 'h2	✗	✓	-	✗
✗ 3 'h3	✗	✗	✗	-

To view the transition details in list format with coverage ranges, select **Shown in List**. Figure 1-15 shows transitions with covered transitions displayed in green and uncovered in red. Not-possible transitions are not listed in the table.

Figure 1-15 FSM Transition List

<input checked="" type="checkbox"/> Shown in List	Mode: Transition	
Coverage ▾	Transition	
✓	'h0->'h1	
✗	'h1->'h0	
✓	'h1->'h2	
✗	'h1->'h3	
✗	'h2->'h0	
✓	'h2->'h1	
✗	'h2->'h3	
✗	'h3->'h0	
✗	'h3->'h1	
✗	'h3->'h2	

Displaying Sequences

The Sequences view shows states and sequences and their possible coverage numbers that an FSM can follow to reach from one state to another state. Holding the cursor on a cell displays a tooltip showing the sequences and their coverage.

Figure 1-16 FSM Sequence Table

☐ Shown in List Mode: States & Sequences

	✓ 0	✗ 1	✓ 2	✓ 3	✓ 4	✓ 5
✓ 0 idle	2/23	0/1	1/2	1/3	1/5	2/12
✗ 1 five	0/14	0/14	0/7	0/4	0/9	0/21
✓ 2 ten	1/8	0/8	1/16	1/5	1/4	1/15
✓ 3 ten	1/4	0/4	1/8	1/12	1/5	1/12
✓ 4 twenty	1/2	0/2	1/4	1/6	1/10	1/8
✓ 5 paid	1/1	0/1	1/2	1/3	1/5	2/12

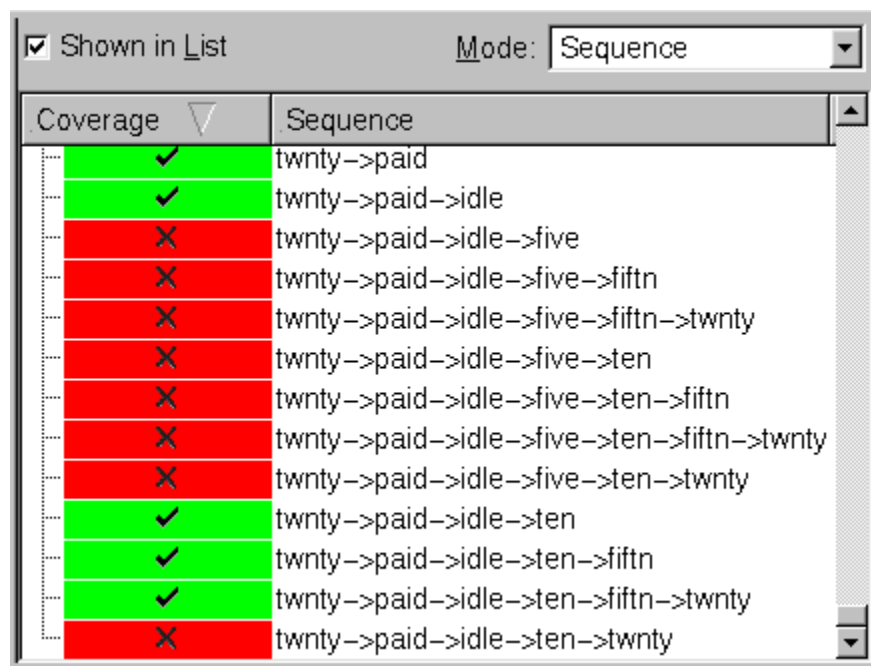
To view the state details in list format, select **Shown in List**.

Figure 1-17 shows sequences with covered sequences displayed in green and uncovered in red.

Note:

You must turn on the sequences at compile time to view them in DVE.

Figure 1-17 FSM Sequence List



The screenshot shows a window titled "FSM Sequence List". At the top, there is a checkbox labeled "Shown in List" which is checked, and a dropdown menu labeled "Mode:" with "Sequence" selected. Below this is a table with two columns: "Coverage" and "Sequence". The "Coverage" column contains green checkmarks for full coverage and red X's for partial or no coverage. The "Sequence" column lists various state transitions.

Coverage	Sequence
✓	twnty->paid
✓	twnty->paid->idle
✗	twnty->paid->idle->five
✗	twnty->paid->idle->five->fiftn
✗	twnty->paid->idle->five->fiftn->twnty
✗	twnty->paid->idle->five->ten
✗	twnty->paid->idle->five->ten->fiftn
✗	twnty->paid->idle->five->ten->fiftn->twnty
✗	twnty->paid->idle->five->ten->twnty
✓	twnty->paid->idle->ten
✓	twnty->paid->idle->ten->fiftn
✓	twnty->paid->idle->ten->fiftn->twnty
✗	twnty->paid->idle->ten->twnty

Displaying Branch Coverage

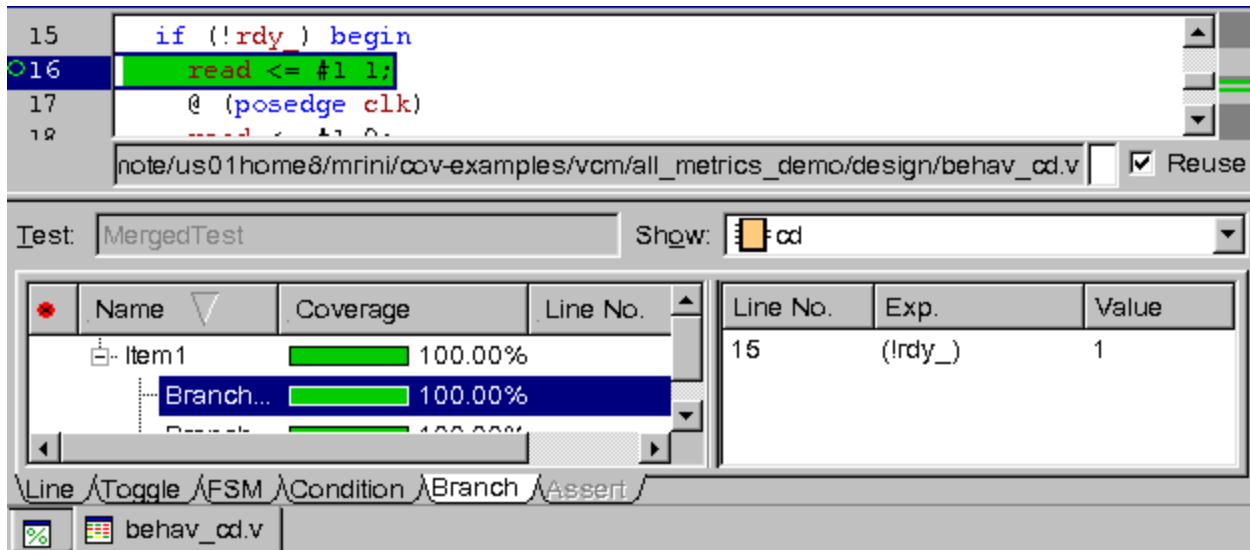
This section describes branch coverage. You can view the branch coverage information in the following windows:

- Summary Table - The branch coverage information is displayed in the **Branch** column.
- Treemap - The **Branch** tab is displayed in the Treemap view inline with other tabs.
- Application Preferences - A metric **Branch** is displayed in the Coverage Settings - Coverage Weights view.
- Detail View - The **Branch** tab is displayed inline with other tabs. Unlike other metrics, you can view the source code line numbers to see the branch coverage information.

The Branch Coverage Detail window

The Branch Coverage Detail window displays annotated source code and coverage percentage for branch coverage.

Figure 1-18



It consists of the following panes and options:

- Source pane - allows you to view the source files annotations that shows you which code has and has not been covered. The source code lines are color-annotated according to their coverage values. You can see only the line of branches as color-annotated and not the line of terms.

You can also see the missing Else/Default statements annotated in the Source pane.

- The Detail List View - allows you to view the items and branches on the left side and the branch path list on the right side. When you select the items, the Value column will be empty. When you select the branch, the value column gets populated with the branch's value.

- Show drop-down - allows you to look at the coverage of other instances or module variants corresponding to the current instance or module.

Note:

When there are more than 50 branches for an item, DVE changes the report from complete table format to just the related expression paths for a branch.

Displaying Assertion Coverage

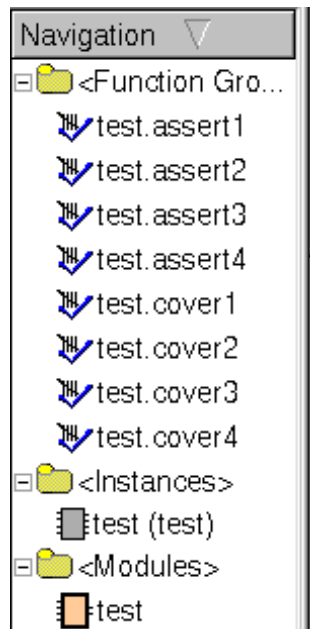
Currently, DVE can display limited information about SystemVerilog Assertion (SVA) and OpenVera Assertion (OVA) coverage.

SVA `assert` and `cover` statements and OVA `assert` and `cover` directives appear in the Navigation pane and in the **Details Window Covers** tab.

There is no coverage information in the database for `assume` statements.

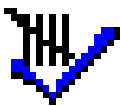
- To monitor SystemVerilog `assert` statements, you must include the `-cm assert` compile-time and runtime option and keyword argument.

Figure 1-19 SVA *assert* and *cover* statements in the Navigation Pane



The statements appear in the Function Groups folder in the Navigation pane. In this example, the `assert` and `cover` statements are in top-level module `test` with the block identifiers `assert1` to `assert4` and `cover1` to `cover4`.

DVE uses the following symbol to indicate an `assert` or `cover` statement or directive:



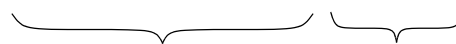
Note:

DVE uses the same symbol for SystemVerilog and OpenVera `assert` or `cover` statements and directives.

SVA Naming Convention


DVE lists these statements by their hierarchical name, for example:

top.dev1.specdev1.cover1


Hierarchical name of the module instance that contains the
assert or cover statement

If you bound an SVA module to a design module, the hierarchical name would include the instance name of the SVA module, for example::

top.dev1.specdev1.checker1.assert1


Hierarchical name of the
module instance to which
the SVA module is bound
unit is bound

Instance name
of the SVA
module


Block identifier

The **Detail Window Covers** tab puts the block identifier in a separate column from the rest of the hierarchical name.

OVA Naming Convention

DVE lists these directives by their hierarchical name, for example:

top.dev1.specdev1.oa_unit_instance_name.directive_name





Hierarchical name of the
module instance to which
the OpenVera assertion
unit is bound

Instance name for the
OpenVera assertion
unit

Name of the directive

If you do not specify an instance name in the `bind module` construct that binds the unit to the module, VCS and VCS MX generates an instance name for the unit and puts it in the hierarchical name, for example:

`top.dev1.specdev1.assert1_inst00000000.directive_name`

		
Hierarchical name of the module instance to which the OpenVera assertion unit is bound	Generated instance name for the OpenVera assertion unit	Name of the directive

The generated instance name begins with the name of the OpenVera assertion unit, followed by `_inst` and then an eight digit number that is unique to the instance.

The **Detail Window Covers** tab puts the name of the directive in a separate column from the rest of the hierarchical name.

The Covers Tab

This section describes the fields and columns in the **Covers** Tab.

Figure 1-20 *SystemVerilog `assert` and `cover` statements in the Covers Tab*

Instance	Cover Name	Attempts	Real Successes	Failures	Incompletes
<Function Groups>	User defined Group	745	32	422	2
operator_preced_and_delay1_testbench.d0	operator_preced_and_delay1	5	4	1	0
operator_preced_and_delay2_testbench.d0	operator_preced_and_delay2	5	4	1	0
operator_preced_and_delay_define1_testbench.d0	operator_preced_and_delay_define1	5	4	1	0
operator_preced_and_or1_testbench.d0	operator_preced_and_or1	5	2	3	0
operator_preced_and_or_delay1_testbench.d0	operator_preced_and_or_delay1	5	2	3	0
operator_preced_and_throughout_delay_testbench.d0	operator_preced_and_throughout_delay	100	0	100	0
operator_preced_or_delay1_testbench.d0	operator_preced_or_delay1	5	1	4	0
operator_preced_or_delay2_testbench.d0	operator_preced_or_delay2	5	1	4	0
operator_preced_or_throughout_delay_1_testbench.d0	operator_preced_or_throughout_delay	100	0	100	0
operator_preced_or_throughout_delay_testbench.d0	operator_preced_or_throughout_delay	100	0	99	1
packed_array_bit1_testbench.d0	packed_array_bit1	30	29	1	0
packed_array_bit2_testbench.d0	packed_array_bit2	30	29	1	0
packed_array_bit_define1_testbench.d0	packed_array_bit_define	30	29	1	0
packed_array_bit_parameter1_testbench.d0	packed_array_bit_parameter1	30	29	1	0
packed_array_define1_testbench.d0	packed_array_define1	30	29	1	0
packed_array_expression1_testbench.d0	packed_array_expression1	30	29	1	0
packed_array_parameter1_testbench.d0	packed_array_parameter1	20	20	0	0
packed_array_part1_testbench.d0	packed_array_part1	20	20	0	0
packed_array_part2_testbench.d0	packed_array_part2	30	29	1	0
packed_array_whole1_testbench.d0	packed_array_whole1	0	0	0	0
packed_array_whole2_testbench.d0	packed_array_whole2	0	0	0	0

Figure 1-21 *OpenVera `assert` and `cover` Directives in the Covers Tab*

Instance	Cover Name	Attempts	Real Successes	Failures	Incompletes
<Function Groups>	User defined Group	56	2	20	34
testbench1.mux.t1	funnyassert	21	2	3	16
testbench2.mux.t1	funnyassert	17	0	6	11
testbench3.mux.t1	funnyassert	13	0	6	7
testbench4.mux.t1	funnyassert	5	0	5	0

The **Instance** column lists the module instance that contains the `assert` and `cover` statement or directive. If you bound an SVA module, this name also include the instance name for the SVA module. Each entry in this column also begins with the symbol for one of these statements.

The **Cover Name** column lists the block identifier for `assert` or `cover`.

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

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

The Real Successes column is color coded with a 0 considered uncovered and a non-0 considered covered. By default, covered is displayed in green and uncovered is red. You can change the colors in the Preferences dialog box.

The **Failures** column shows the number of times `assert` does not succeed. `cover` does not have failures, so the entry for `cover` statements in this column will always be 0.

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

Displaying Branch Coverage and Implied Branches

The DVE Coverage GUI shows implied branches in the source window (when you use `dve -cov`) only when you view branch coverage or are analyzing the branch metric. The implied branches

of `case` and `if` statements are indicated by `MISSING_ELSE` and `MISSING_DEFAULT` tags. You can exclude any branches tagged in this fashion.

Displaying Testbench Coverage

DVE displays testbench coverage information as specified in SystemVerilog `covergroup` constructs in SystemVerilog `program` blocks and the OVA `coverage_group` construct in the OpenVera code.

The SVA and OVA constructs specify the clocking event that defines the coverage data reported during simulation and specifies the coverage points monitored.

When you open a database, DVE displays the SVA covergroups and OVA `coverage_groups` in the Functional Groups folder in the Navigation pane.

The **Covergroup** tab lists covergroup items with coverage score, goal, weight, and a coverage map view. The Covergroup Items pane is shown in [Figure 1-22](#). Double-clicking a covergroup displays the covergroup source code in the Source Window. Expanding the covergroup displays coverpoints, crosses, bins, and instances.

In the Covergroup detailed window, you can view the Covergroup Definition and Instances using the Show drop-down. You can also use the CSM to locate instance for covergroup definition, and definition for covergroup instance.

The screenshot displays the DVE Coverage GUI. The top section is a Verilog code editor showing the following code:

```

336 covergroup cross_2 @(posedge clk);
337     test1 : coverpoint wb0_we_i + wb1_we_i;
338     axb : cross a, test1;
339 endgroup
340
341 cross_2 cr_2 = new();
342
343 covergroup cross_3 @(posedge clk);
344     test1 : coverpoint {1'b1,wb1_we_i} { bins b[] = {[0:3]}; }
345
346 /remote/vgrnd3/bgao/BM/Wishbone_Svtb_Fcov_Svd.bak/bench/sv_tb_wrapper.v

```

Below the code editor, the 'Test' dropdown is set to 'MergedTest'. The 'Show' dropdown is set to 'Instances', and the 'Auto bins and holes' checkbox is checked. The 'Table' icon is also visible.

The main table displays the covergroup summary:

Cover Group Item	Definition	Score	Bin Name	Status	At Least	Hit Count
<Function Groups>		47.20%				
cr_4	dut_tb_test.tb::cross_4	69.70%				
cr_sample	dut_tb_test.tb::cross_sample	36.17%				

The bottom status bar shows 'Covergroup / Assert'.

Figure 1-22 Covergroup Item Pane

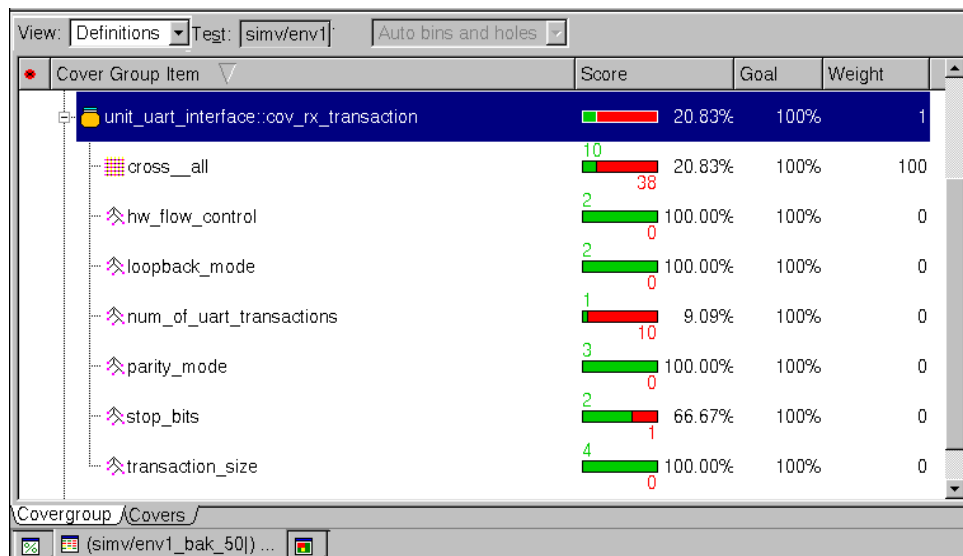
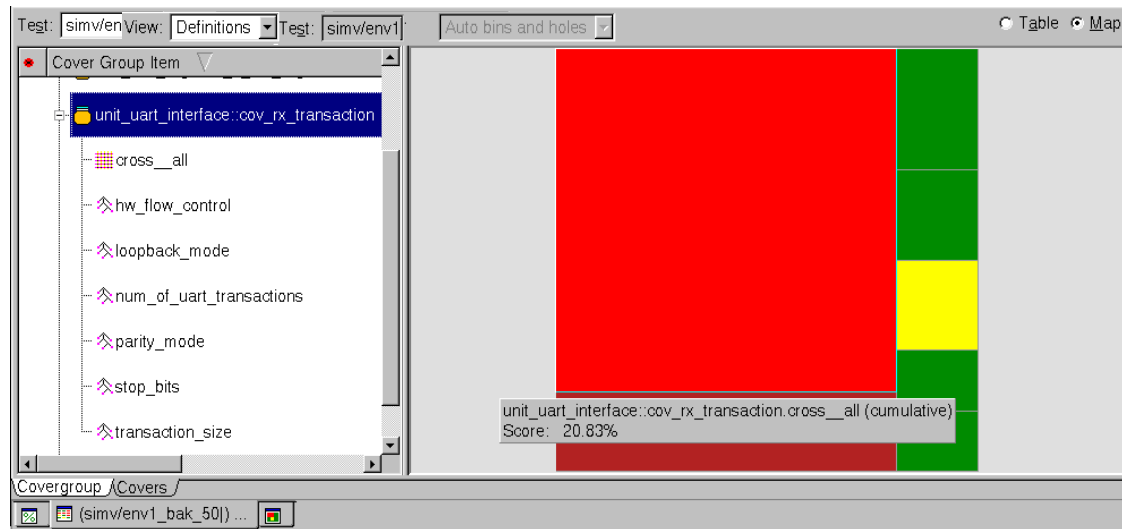


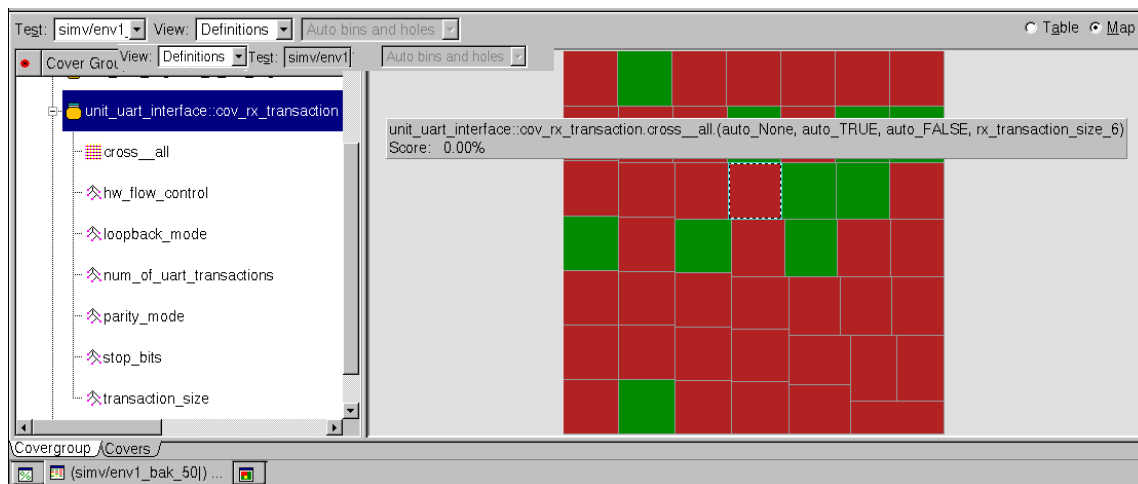
Figure 1-23 shows the map view results for covergroup unit_uart_interface::cov_rx_transaction. The cumulative score for the cross below the mouse pointer in the map view is 20.83, as displayed in the tooltip.

Figure 1-23 The Covergroup Tab



To drill down for details of the cross or cover point results, you can double-click on a rectangle in the map view or drag and drop, or select a cross in the left pane.

Figure 1-24 Results for Crosses



To view results in table format, select **Table**. [Figure 1-25](#) shows the tabular results for a cross listing bins and their coverage. Click the Size column header to sort and bring the holes to the top.

Figure 1-25 Coverpoint Table View

Test: MergedTest View: Definitions Auto bins and holes Table Map

Cover Group Item	active_cnt	priority_cnt	At Least	Size	Hit Count
wb_dma_cfg_cov::config_c	[CH_CNT_13, CH_CNT_14]	PR_CNT_1, PR_CNT_2, PR_CNT_3	1	104	0
activeXpriority	[CH_CNT_01, CH_CNT_02]	PR_CNT_1, PR_CNT_2, PR_CNT_3	1	96	0
	[CH_CNT_0e, CH_CNT_0f]	PR_CNT_1, PR_CNT_2, PR_CNT_3	1	24	0
	[CH_CNT_11]	PR_CNT_1, PR_CNT_2, PR_CNT_3	1	7	0
	[CH_CNT_12]	PR_CNT_1, PR_CNT_2, PR_CNT_3	1	6	0
	[CH_CNT_0d]	PR_CNT_1, PR_CNT_2, PR_CNT_3	1	5	0
	[CH_CNT_0d]	PR_CNT_7, PR_CNT_8]	1	2	0
	[CH_CNT_12]	PR_CNT_8]	1	1	0
	CH_CNT_12	PR_CNT_7	1	1	1
	CH_CNT_11	PR_CNT_8	1	1	2
	CH_CNT_0d	PR_CNT_6	1	1	1

Covergroup /Covers

Working with HVP Files

You can use the DVE Coverage GUI to create HVP files. An HVP (Hierarchical Verification Plan) is a comprehensive model that allows you to hierarchically describe a verification plan. You can then annotate that plan with live verification data using the Unified Report Generator (URG) tool.

For more information about HVP files and how to use URG to annotate the live verification data, see the *VMM Planner User Guide* and the *URG User Guide* respectively in the VCS Online Documentation.

Working with Coverage Results

Running Scripts

You can do any DVE operation with Tcl commands, including using Tcl scripts to modify the display of your coverage data. For example, you can omit display of statement, instance, or module coverage.

To select and source a TCL script

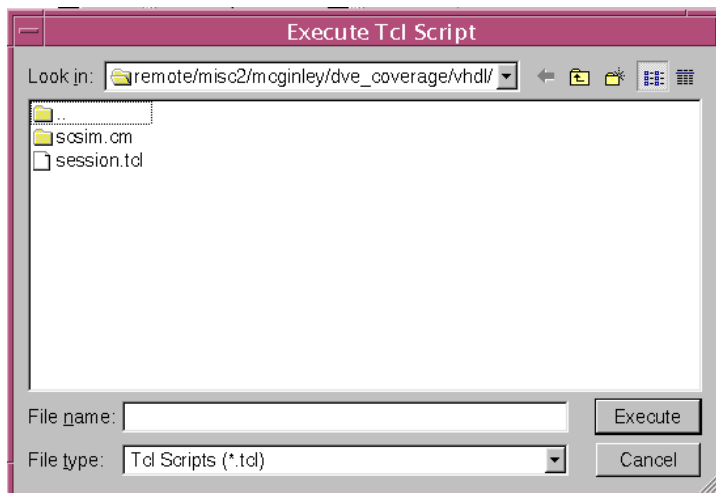
1. Select **File > Execute TCL Script**.

The Execute TCL Script dialog box appears.

2. Browse to the script, select it, then click **Execute**.

The Tcl script is executed.

Figure 1-26



Filtering the Hierarchy Display

To filter the display in the Navigation pane

1. Select **Edit > Filters > Add**.

The Filter dialog box appears.

2. Enter a string or expression to filter.
3. Select any of the filtering options as follows:
 - Match case
 - Match whole word only
 - Use regular expression
4. Click **OK** or Apply to filter according to the criteria you specified.

The Navigation pane displays the modified results.

Filters are applied cumulatively. A filter added to an already filtered result further filters the result display. If you want to apply the filter to the unfiltered results, first apply a * filter to display the unfiltered results.

Setting Display Preferences

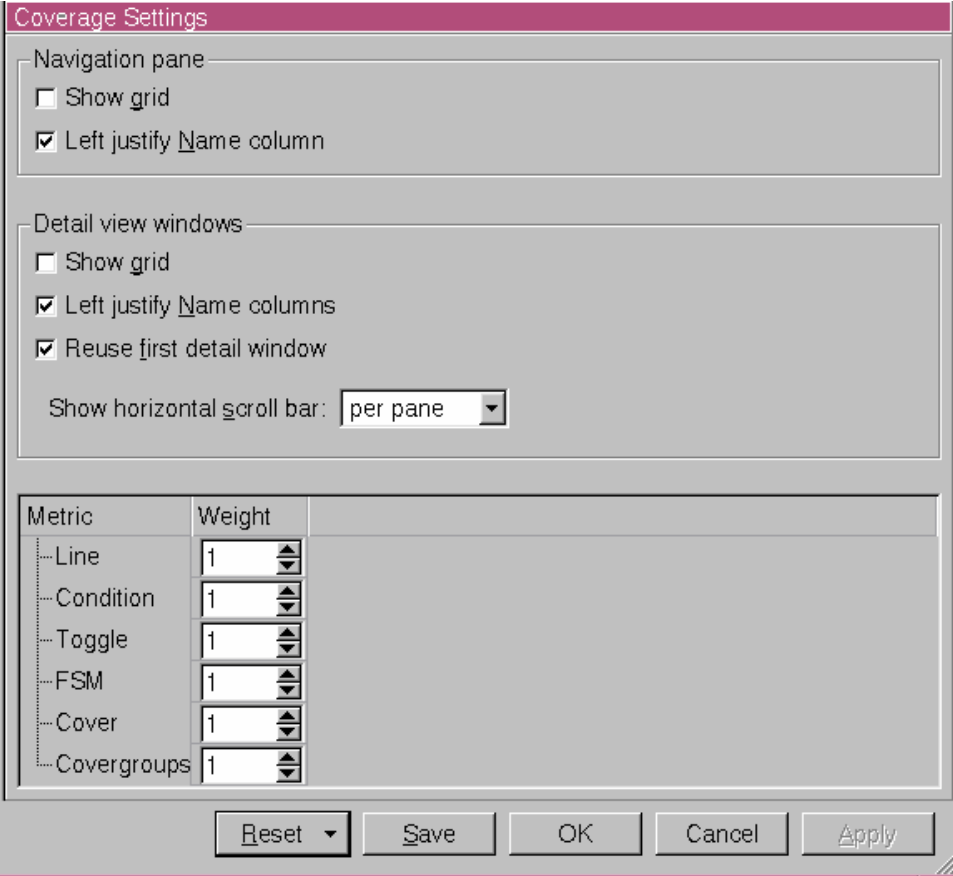
To set your display preferences

1. Select **Edit > Preferences**, then select **Coverage Settings**.

The Coverage Settings window appears.

2. Select appearance display options for the Navigation pane as shown in [Figure 1-22](#).

Figure 1-27



The Coverage Settings dialog box is divided into three main sections. The top section, 'Navigation pane', contains two checkboxes: 'Show grid' (unchecked) and 'Left justify Name column' (checked). The middle section, 'Detail view windows', contains three checkboxes: 'Show grid' (unchecked), 'Left justify Name columns' (checked), and 'Reuse first detail window' (checked). Below these is a label 'Show horizontal scroll bar:' followed by a dropdown menu set to 'per pane'. The bottom section is a table with two columns: 'Metric' and 'Weight'. The table lists six metrics: Line, Condition, Toggle, FSM, Cover, and Covergroups, each with a weight of 1. At the bottom of the dialog are five buttons: 'Reset' (with a dropdown arrow), 'Save', 'OK', 'Cancel', and 'Apply'.

Metric	Weight
Line	1
Condition	1
Toggle	1
FSM	1
Cover	1
Covergroups	1







3. 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.







Color Display

Customize the color display of Source Window cover states and the number of coverage ranges and their associated colors in the Color Settings Window.

Color Settings Preferences

Coverage Colors		
Source Window Cover state	Visibility	Color
Uncovered	<input checked="" type="checkbox"/>	
Partial Covered	<input checked="" type="checkbox"/>	
Covered	<input checked="" type="checkbox"/>	
Excluded and Uncovered	<input checked="" type="checkbox"/>	
Excluded and Covered	<input checked="" type="checkbox"/>	
Unreachable (For FSM...	<input checked="" type="checkbox"/>	

Number of coverage ranges:

Coverage R	Range Min	Range Max	Color
Range 0	<input type="text" value="0"/>	<input type="text" value="19"/>	
Range 1	<input type="text" value="19"/>	<input type="text" value="39"/>	
Range 2	<input type="text" value="39"/>	<input type="text" value="59"/>	
Range 3	<input type="text" value="59"/>	<input type="text" value="79"/>	
Range 4	<input type="text" value="79"/>	<input type="text" value="99"/>	
Range 5	<input type="text" value="99"/>	<input type="text" value="100"/>	

Coverage Menu and Toolbar Reference

This section describes the Menu and Toolbar commands and options in the DVE.

Menu Bar

The Menu bar contains the following menus and options:

File Menu

The following items comprise the **File** menu:

Open Database	Displays the Open 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.
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.
Execute TCL Script	Displays the Execute TCL Script dialog box, which enables you to select and source a TCL script.
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.
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.
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	Enabled	Turns on exclusion.
	Exclude/Unexclude	Selects item for recalculation.
	Recalculate	Displays result with item excluded or unexcluded.
	Uncovered Mode	Determines whether or not you can exclude covered items.
Test Grading		Defines design, test, criteria, and post operations for grading coverage results.
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

The following are the main options in the View menu:

Show Values		Annotates the display with coverage totals.
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.

Using the DVE Coverage GUI

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	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 previous scope.
Forward		Goes to next scope.
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.

	Grading Window	Opens a new Grading Viewer.
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.
Panes	Console	Displays new console pane.
	Navigation	Displays new Navigation pane.
New Top Level Frame	Coverage	Displays new frame.
Default Layout		Returns display to default layout.
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 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.
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







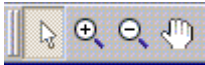
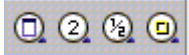
The following items comprise the Help menu:

DVE Help	Displays help for DVE Coverage.
A Quick Start Example	Loads a DVE example coverage database.
About	Displays DVE Coverage version and copyright information.

Toolbar Reference

This section describes all Toolbar text fields, menus, and icons

Icon	Description
 Open Database or File	Displays the Open Database or Open File dialog box, depending on the DVE window displayed, and enables you to select and open a VPD file.
 Search	Searches for selection.
 Search Forward/Back	Searches forward or back.
 Exclude	Excludes selected item.
 Unexclude	Unexcludes selected item.
 Recalculate	Recalculates results with excluded/Unexcluded items.

 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.
 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.
	<p>From left to right:</p> <p>Selection tool – Uses to select objects.</p> <p>Zoom In – Makes objects in current view twice as big so fewer objects will be viewable.</p> <p>Zoom Out – Makes objects in current view twice as small so more objects will be viewable.</p> <p>Pan tool – Moves the view so the selected object is centered and viewable in the current pane. It does not change the zoom.</p>
	<p>From left to right:</p> <p>Zoom Full – Fits all viewable objects into the current view.</p> <p>Zoom in 2x – Doubles current view scale.</p> <p>Zoom out 2x – Halves current view scale.</p> <p>Zoom to selection– Moves the view so the selected object is centered and viewable in the current pane. It does not change the zoom.</p>



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.

3

VMM Planner MS Doc Annotation

The VMM Planner Doc annotator enables creation of verification plans using Microsoft Office (.doc) documents, and back-annotation of coverage and other scores into the .doc plan. The .doc verification plan must be formatted properly with predefined styles and keywords built into the VMM Planner Doc annotator.

Either MS-Word 2003 or later can be used to create the .doc verification plan.

Note:

Unlike the VMM Planner Spreadsheet annotator flow, the Word Doc flow does not currently support OpenOffice input.

Back-annotation is an extension of the existing VMM Planner Spreadsheet annotator flow. The `hvp annotate` command and most of its switches can be used for back-annotation the same way that they are used with the spreadsheet annotator.

This section contains the following topics:

- [“Introduction” on page 74](#)
- [“Use Model” on page 75](#)
- [“hvp annotate Command Arguments” on page 77](#)
- [“Capturing a Verification Plan in Doc XML Format” on page 80](#)
- [“Debugging the Doc Plan” on page 89](#)
- [“How to Get Scores Back-Annotated in the Doc” on page 92](#)

Introduction

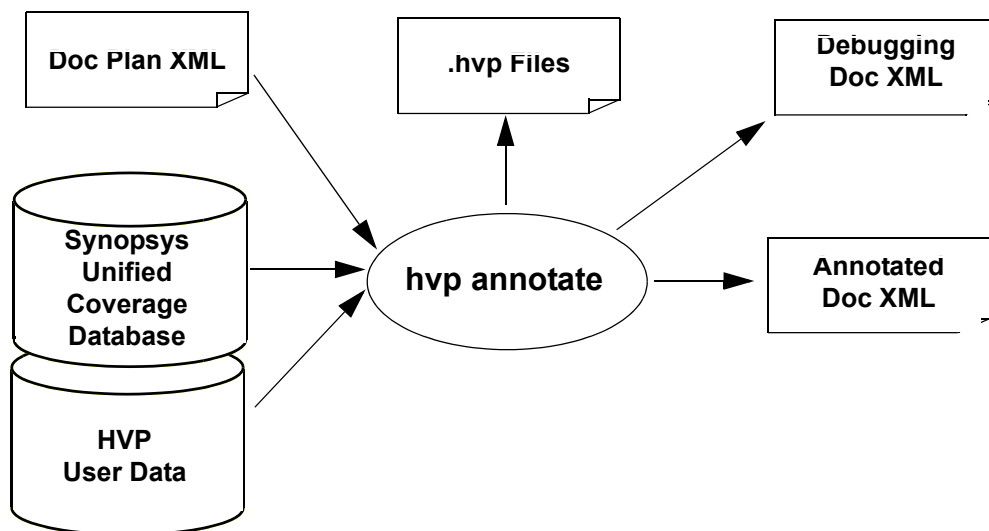
The VMM Planner Doc annotator is used:

- To provide simple hooks in MS Word-compliant documents to interface with VMM Planner-enabled applications.
- To provide flexibility in formatting verification plan doc XML for nicely formatted coverage reports.

The same doc XML file is used by VMM Planner as a verification plan and as an annotated report. Formatting features such as selectable fonts, text alignment, graphics, charts, tables and descriptions are applied in both the plan and the annotated output report.

Use Model

Figure 3-1 Word Doc annotator use model



This section contains the following topics:

- [“Process Flow” on page 76](#)
- [“License Model” on page 76](#)
- [“Supported Platforms” on page 77](#)
- [“Compatibility with the VMM Planner Spreadsheet Annotation Flow” on page 77](#)

Process Flow

1. Capture your verification plan in Doc XML format. This is typically done by starting with a Synopsys-provided template file and marking key paragraphs in the created doc with Synopsys-defined styles, and copying and pasting Synopsys-defined tables.
2. Prepare Synopsys coverage database and HVP user data files.
3. Run `hvp annotate` with the `XML plan`, `covdb`, `userdata` and other desired switches.
4. Fix any errors you see. You can find more error information in `filename.dbg.xml`. Even if you do not see any errors, open the `filename.dbg.xml` file and check to make sure that VMM Planner processed your verification plan hierarchy and contents as you intended. This debug check process only needs to be done on newly created or recently edited plans. Once the plan becomes stable, this debug check step can be skipped when annotating with new coverage databases.
5. If there are no errors, open `filename.ann.xml` to see the annotated scores.

License Model

VMM Planner in C-2009.06 Beta is a Beta feature, so a Beta license key is needed in order to use the Doc XML annotation flow.

Supported Platforms

linux
amd64
suse32
suse64
sparc64
sparcOS5

Compatibility with the VMM Planner Spreadsheet Annotation Flow

The VMM Planner Spreadsheet annotator flow and Doc annotator flow share the **hvp annotate** command and most of its switches. When an XML plan file is specified in **-plan *planfile***, VMM Planner automatically detects the XML file format and invokes the proper process depending on the format of the XML.

The **hvp annotate** command is completely backward compatible with the existing VMM Planner Spreadsheet annotator.

hvp annotate Command Arguments

Syntax

```
hvp annotate -plan planfile  
[-h]  
[-mod hvpfiles]  
[-plan_out annfile]  
[-feature "hierarchies" | -featurefile txtfile]  
[-dir covdbpath | -f txtfile]  
[-userdata vedata | -userdatafile txtfile]  
[-userdata_out outvedata]
```

```
[-metric_prefix prefix]  
[-group ratio|merge_across_scopes]  
[-show_ratio]  
[-show_incomplete]  
[-v]  
[-q]
```

Options

-plan *planfile*

Spreadsheet, doc XML or HVP file for your verification plan. This switch is mandatory.

Example: **-plan** myplan.xml

-h

Show this help message and exit.

-mod *hvpfiles*

Filter or override files in HVP language format multiple files can be specified. They are applied in the order in which they are entered.

Example: **-mod** override.hvp filter.hvp

-plan_out *annfile*

Specify the name for the output annotated spreadsheet or doc XML file. If **-plan_out** is not entered, a file with the original filename and `.ann.xml` extension is generated.

-feature "*hierarchy [hierarchy...]*"

Specify HVP scopes that you want to annotate with the given covdb coverage database or ve.data. Multiple scopes can be specified, and wildcards (*, **, ?) can be used. Enclose the string with double quotes. Subhierarchies of matched scopes are automatically annotated. if **-hier** is not used, covdb and ve.data are annotated to entire plan.

Example:

-feature "myplan.rec_feat.* myplan.play_feat.*"

-featurefile *txtfile*

A text file that contains list of hierarchical filters.

-dir *covdbpath*

Specify the path to a Synopsys coverage database (cm, vdb).
Multiple paths can be entered, separated by commas.

Example: **-dir** wishbone.cm wishbone.vdb

-f *txtfile*

Specify a text file that contains list of covdb coverage database paths.

-userdata *vedata*

Specify a ve.data file path. Multiple paths can be entered, separated by commas.

Example: **-userdata** result.txt bugcount.txt

-userdatafile *txtfile*

Specify a text file that contains a list of user database file paths.

-userdata_out *outvedata*

Dump an annotated score of all measures into the specified *outvedata* user data file.

-metric_prefix *prefix*

The given prefix is used to change a metric name in the output user database file used by **-userdata_out**.

-group ratio | merge_across_scopes

-group ratio: Aggregate the group metric score as a ratio type (covered/coverable) instead of as a percent.

-group merge_across_scopes: Merge the group score across scopes.

-show_ratio
Display ratio type scores in a ratio form instead of a percent.

-show_incomplete
Indicate incomplete scores with [inc].

-v
Verbose mode: Show progress status messages.

-q
Quiet mode: Turn off all warning messages.

Capturing a Verification Plan in Doc XML Format

This section contains the following topics:

- [“Built-In Styles” on page 80](#)
- [“Table Keyword” on page 84](#)
- [“Other Contents” on page 89](#)

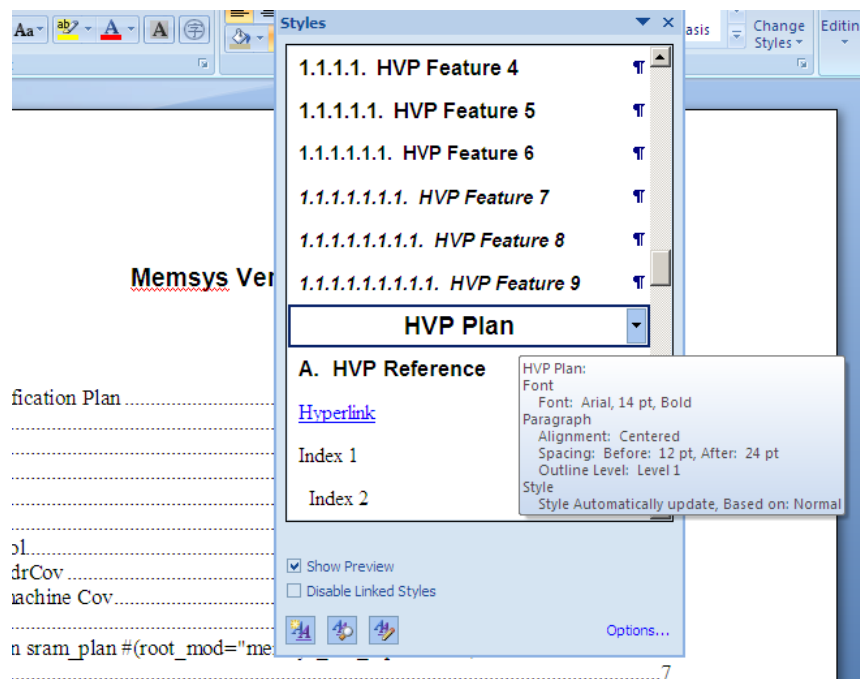
Built-In Styles

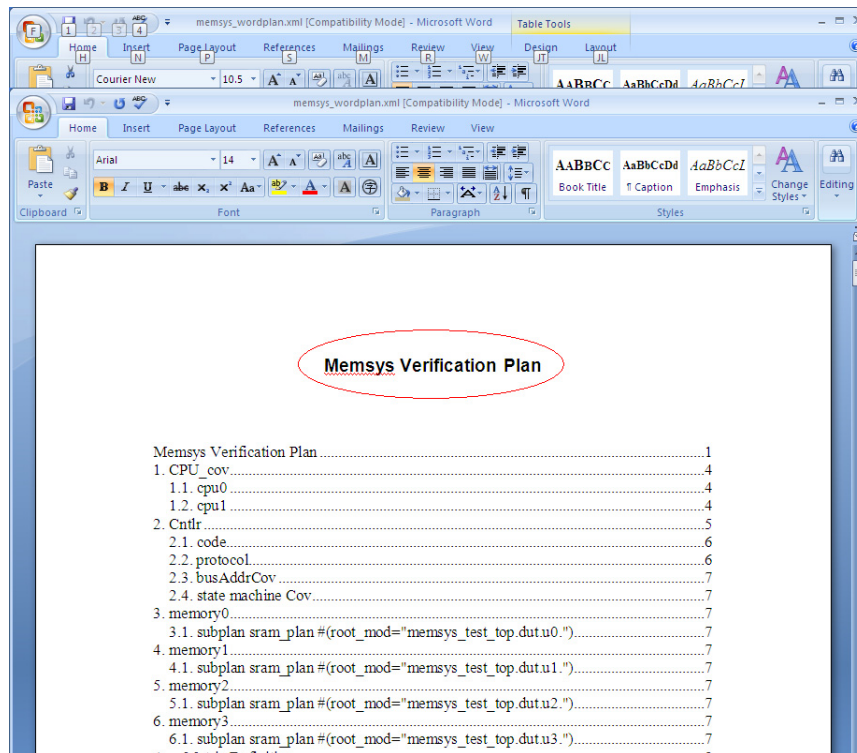
In the Doc XML plan template, there are predefined styles that are prefixed with “HVP”. VMM Planner extracts the contents and their styles to establish the HVP hierarchy. A template document

containing predefined styles is provided with the VMM Planner Doc annotator. Make a copy of that template and then use your copy as a basis to create a new plan.

HVP Plan Style

Content with the style “HVP Plan” is assigned as the name of the HVP Plan. Use this style only once in the plan, typically at the very top of the document.

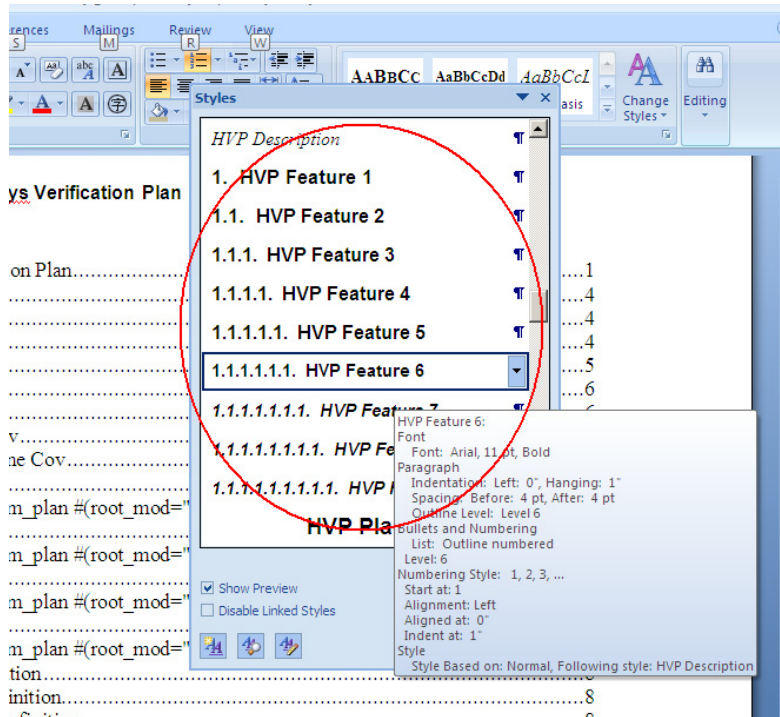




HVP Feature 1 to HVP Feature 9

There are nine levels of “HVP Feature” styles, which are used to represent HVP feature hierarchy. The index must be seamless in terms of hierarchy, which means that “HVP Feature 1” can be followed by another “HVP Feature 1” or “HVP Feature 2”, but not by “HVP Feature 3” or “HVP Feature 4”, for example. If “HVP Feature 2” is found, the feature will be a subfeature of the previous “HVP Feature 1”. If “HVP Feature 1” is found, it will be sibling of the previous “HVP Feature 1”, if one exists.

This limited number of styles also limits the number of levels of hierarchical depth in a verification plan. You cannot create a plan with more than nine levels of features.



You can also use “HVP Feature *number*” for subplan declaration. To instantiate a subplan, use **subplan** *name-of-plan*. You can also pass attributes and annotation value overrides the using # () syntax, which is the same syntax as in the HVP language.

Since VMM Planner does not allow the same instance name in the same node, you must add one more feature level to instantiate multiple subplans with the same plan, as shown in the following example.

```

3. memory0
3.1. subplan sram_plan #(root_mod="memsys_test_top.dut.u0.")
4. memory1
4.1. subplan sram_plan #(root_mod="memsys_test_top.dut.u1.")
5. memory2
5.1. subplan sram_plan #(root_mod="memsys_test_top.dut.u2.")
6. memory3
6.1. subplan sram_plan #(root_mod="memsys_test_top.dut.u3.")

```

HVP Description

The paragraph with the “HVP description” style is extracted as the built-in `description` attribute in the HVP feature. You can also use an “HVP Assignment” table to override description values.

Table Keyword

When a table is found, VMM Planner looks at the contents in the first cell of the table, and determines whether the table needs to be interpreted as part of the HVP hierarchy. The following keywords are reserved:

- [“HVP Assignment” on page 85](#)
- [“HVP Measure name” on page 85](#)
- [“HVP Metric name” on page 87](#)
- [“HVP Attribute name, HVP Annotation name” on page 88](#)
- [“HVP Include File” on page 89](#)

HVP Assignment

A table with an “HVP Assignment” style is used to override attributes, annotations and goals for metrics. The keyword `cell` is followed by two-column rows. The left cell must have the name of an attribute or annotation to override the value of the name or annotation, or the name of a metric to override a goal expression. All names of attributes, annotations or metrics in this table must be defined in the plan. Otherwise, VMM Planner generates warnings and ignores the undefined names.

HVP Assignment	
phase	2
owner	Snps

Note:

Although the HVP language is case sensitive, VMM Planner searches for the names of attributes, annotations, and metrics in its definition tables with case-insensitive matching, because Word sometimes capitalizes the first character of a word.

HVP Measure *name*

The table with the **HVP Measure** *name* keyword in the first cell is used to declare HVP measure statements. You can specify a unique measure name or leave it blank. If no measure name is entered, then VMM Planner automatically generates a measure name.

The keyword `cell` is followed by two-column rows. You specify at least one source string. To specify multiple sources, you can list multiple source strings in the same cell separated by commas, or add multiple rows with the `source` keyword in the left cell of each row.

After the source row, append one or more two-column rows, with a metric name in the left cell of each of those rows. Leave the right cell empty. The annotation process fills each empty cell with its score.

HVP Measure m1	
source	group instance: memsys_test_top.testbench::cpu::cpu_cov.my_cpu0
Group	

HVP Measure	
source	group: memsys_test_top
source	group: memsys_test_top2
Group	

HVP Measure	
source	group: memsys_test_top, group: memsys_test_top2
Group	
Assert	

HVP Metric *name*

A table with an **HVP Metric** *name* keyword in the first cell is used to define an HVP metric. In the keyword cell, a unique metric name must be specified after **HVP Metric** (“bugs” in the example table below).

The following are rules for creating an HVP metric table:

- One table per one metric definition.
- Following the first row, use two-column rows with predefined keywords in the left cell. Rows for **type** and **aggregator** are mandatory. A row for **goal** is optional.

Type can be one of: integer, ratio, real, percent, enum {entry1, entry2, ...}.

Aggregator can be one of: sum, average, max, min. Not all entries are available for all types. See “Using the HVP Language” for more information.

Goal is an expression for the coverage goal.

HVP metric bugs	
type	Integer
aggregator	Sum
goal	bugs < 3

HVP Attribute *name*, HVP Annotation *name*

A table with an **HVP Attribute *name*** or **HVP Annotation *name*** keyword in the first cell is used to define an HVP attribute or annotation, respectively. In the keyword cell, a unique name must be specified after the **HVP Attribute** or **HVP Annotation** keyword (“phase” in the first example table below, and “spec” in the second example table).

The following are rules for creating an HVP metric table:

- One table per one definition.
- Following the first row, use two, two-column rows with predefined keywords **type** and **default** in the left cells.

Type can be one of: integer, ratio, real, percent, enum {entry1, entry2, ...}.

- Default is the default value of the attribute or annotation.

HVP attribute phase	
type	Integer
default	1

HVP annotation spec	
type	String
default	

HVP Include File

A table with the **HVP Include File** keyword in the first cell is used to declare other XML plans to be included as subplans. The XML plans that are included in this table can be used as subplans. Add as many one-column rows containing XML filenames as you need.

HVP include file
sram_wordplan.xml
dram_wordplan.xml

Other Contents

All other content such as text, images, and charts, that is neither in a predefined HVP style nor in a table with a predefined keyword, is ignored during the annotation process, and appears in the annotated plan exactly as it appears in the original plan document. Formatting is limited to the formatting features that are allowed in Microsoft Word 2003 XML .doc format.

Debugging the Doc Plan

When an error is detected in a .doc verification plan, it is difficult to find where the error occurred. In addition, VMM Planner might interpret your verification plan in a different way than what you intended or expected due to reasons such as typographic errors, incorrect styles, and so on. This is because the paragraph styles are essentially invisible in the normal WYSIWYG view of the document. You might, for example, select the wrong style, not knowing that

VMM Planner will not recognize the paragraph properly. When you run `hvp annotate`, a `filename.dbg.xml` is created for each plan file, which provides detailed debugging information.

This section contains the following topics:

- “[[Style|Table](#)] Keyword Indicator” on page 90
- “Unique Error Code in Error Messages” on page 92

[[Style|Table](#)] Keyword Indicator

In the `filename.dbg.xml`, a blue term enclosed in square brackets, [[Style|Table](#)], indicates that the section was either a predefined HVP style or a predefined HVP table. Several examples are shown below. If you do not see [[Style|Table](#)] where you expect to see it, there is an error in your HVP file such as a wrongly used style or a typographic error.

[[HVP Plan](#)]Memsys Verification Plan

of verification. You may use an override file for setting the attributes so that no need to change the main plan during different stages. This attribute is defined in section “B Attribute Definition” ---

1. [[HVP Feature](#)]CPUcov

---Note that the style of the above “First top level feature” is “HVP Feature 1”. This can be chosen from “styles” in the word menus. There are 9 leves of HVP Features already defined in this word document template for you. All features are identified by styles of the form “HVP Feature *” where * is the hierarchical level of the feature.---

---below are some overrides of the top level attributes. This is optional---

[HVP Assignment]HVP Assignment	
phase	2
owner	sneg

---The HVP attribute phase is 2 since we are interested in this feature only during the top level verification and an override file will remove this feature for block level verification

---The table below defines a measurement for this feature. metrics to measure – group and bugs . ---

[HVP Measure]HVP Measure	
Source	group instance: <u>wwwvs_test_top_testbench</u> ::cpu::cpu_cov.my_cpu0
Group	

[HVP Measure]HVP Measure	
Source	searchench+
bugs	

3. [HVP Feature]memory0

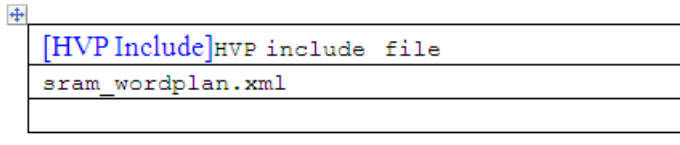
```
3.1. [HVP Subplan]subplan sram_plan
    #(root_mod="memsys_test_top.dut.u0.")
```

4. [HVP Feature]memory1

```
4.1. [HVP Subplan]subplan sram_plan
    # (root mod="memsys_test_top.dut.u1.")
```

[HVP Metric] HVP metric bugs	
Type	Integer
Aggregator	Sum
Goal	bugs < 3

[HVP Attribute]HVP attribute phase	
Type	Integer
Default	1



Unique Error Code in Error Messages

The following error while running `hvp annotate` indicates that an invalid metric, Group 1, was found at the location of ERR001:

```
Error: [DOC_060] memsys_wordplan.xml:ERR001: Invalid metric
name(Group1) was used in Measure(_m1) in Feature(1.2. cpu1)
```

To find where Group1 was used, you can open the file named `memsys_wordplan.dbg.xml` and search for the ERR001 string. ERR001 is a unique string in the `filename.dbg.xml` file, so you can easily find the location of the error. Then you can open the original plan and edit it to fix the error.

How to Get Scores Back-Annotated in the Doc

Running `hvp annotate`, creates one or more `filename.ann.xml` files. VMM Planner produces one `filename.ann.xml` file per plan or subplan instance in the HVP hierarchy. If your plan does not include any subplans, then only one `filename.ann.xml` is produced.

If a plan is used as subplan multiple times, then VMM Planner produces `filename.ann.1.xml`, `filename.ann.2.xml` and so on.

It is not necessary to open the `*ann.*.xml` file to view a subplan. You can open only the top-level plan `.ann.xml` file and navigate to any subplan `.ann.xml` from there.

This section contains the following topics:

- [“Plan/Feature Score Summary” on page 93](#)
- [“Measure Score Table” on page 94](#)
- [“Navigating Down to a Subplan” on page 95](#)

Plan/Feature Score Summary

Running `hvp annotate` produces a score summary table for each plan or feature. Each score cell in the score summary table is filled in with a color determined by the score and the goal. The meanings of the colors are:

- If a goal was specified,
 - Green indicates that the goal was met
 - Red indicates that the goal was not met
- If no goal was specified:

For Synopsys coverage,

 - Green indicates 100% coverage
 - Red indicates 0% coverage
 - A coverage score between 0 and 100% is indicated by one of six different colors (see the *Unified Coverage Reporting User Guide* for information about setting the colors)

For a test metric,

- Green indicates a pass/total ratio of 1 (100%)
- Red indicates a pass/total ratio of 0 (0%)
- A coverage score between 0 and 100% is indicated by one of six different colors (see the *Unified Coverage Reporting User Guide* for information about setting the colors)

Some examples are shown below.

Memsys Verification Plan

Line	Cond	Assert	Group	bugs
100.00%	93.62%	100.00%	100.00%	1

1. CPU_cov

Group	bugs
100.00%	1

Measure Score Table

In the measure score table, the score cell is colored using the same scheme as the “Plan/Feature Score Summary”. If no score is available, then the cell contains “N/A”.

HVP Measure	
Source	group instance: memsys_core_scorebench:cpu:cpu_cov.my_cpu1
Group	100.00%
bugs	N/A

If the `-show_incomplete` switch is used, “[incomplete]”, in orange, appears in front of any source string that does not match any of the regions in the coverage database or user data.

HVP Measure	
Source	group instance: memsys_test_top.srambench::cpu::cpu_cov.my_cpu0
Source	[incomplete]property: non_existing_region
Group	100.00%

Navigating Down to a Subplan

In a `filename.ann.xml` file, clicking on a subplan name that uses the **HVP Feature** style opens the `ann.xml` file for the subplan instance. Hold down Ctrl and click the subplan name (for example, the name starting with `subplan sram_plan` in the example below) to open the file.

3. memory0

Line	Cond	Assert	Group
100.00%	100.00%	100.00%	100.00%

3.1. subplan sram_plan #(root_mod="memsys_test_top.dut.u0.")

Line	Cond	Assert	Group
100.00%	100.00%	100.00%	100.00%

3

SV Cover Property Path Coverage

SystemVerilog allows a rich set of operators and expressions as part of cover property construct intended to capture control oriented coverage measures of the design. VCS collects and reports coverage statistics on such cover property by default. With the properties becoming more complex, the amount of information that a user would like from such cover property expressions is more than mere covered, uncovered. A complex cover property may contain several alternate paths and repetition ranges. VCS can now report extra, value added coverage metric on such properties with several paths. This feature is referred to as "Path Coverage on cover property".

The following are covered:

- Values from ranges in `##`, `[*]`, `[->]` and `[=]` operators
- Branches of a top-level sequence combined with `sequence_or` (OR) operator

Restrictions

- Implication operators are not supported in any form.
- Only top-level OR branches distinguished, no nested OR operators. If these are present the coverage will not be collected.
- Plain coverage on "cover prop" is still collected; only 'path cov" is not implemented.
- If cover property is on a sequence, all matches are registered, if on a property, only first match is registered.
- Design variable values and user-defined local variable values are not collected.
- If a property is inlined in "cover property", then the path coverage is not applied. Only if it is inside a property or a sequence then it is collected.

Using Cover Property Path Coverage

You compile using the following option:

```
vcs ... -cm property_path ...
```

Example 1: Delay range coverage

```
module m;
reg a,b,clk;
    property p;
        @clk
        a##[2:4]b;
    endproperty
cover_p: cover property(p);
endmodule
```

Example 2: here are 2 branches (2 top level constructs connected through or)

```
module m;
bit clk=0;
bit a=0,b=0,c=0;
    always@(posedge clk) begin
        a <= {$random %2};
        b <= {$random %2};
        c <= {$random %2};
    end
    initial begin
        #100 $finish;
    end
sequence s1;
@(posedge clk) (a&c)##[2:4]b;
endsequence
sequence s2;
@(posedge clk) a ##1 c[*1:$] ##1 a;
endsequence
    property p;
        @(posedge clk)
            s1 or s2;
    endproperty
cover_p: cover property(p);
endmodule
```

Assertion Path and Cover Property Path Coverage Reporting in URG

This section explains how assertion path coverage will be reported in the Unified Report Generator (URG). It also specifies how built-in checkers will be reported. It contains the following sections:

- [“Summary table of properties with path coverage ” on page 85](#)
- [“Details for a Property ” on page 85](#)
- [“Details for built-in checkers ” on page 88](#)
- [“Other Changes ” on page 90](#)
- [“Computing the score in URG ” on page 91](#)

Assertion path coverage collects data not just on which assertions/cover properties are covered or not covered, but along which paths they were covered. Although there are many different paths through an assertion, only these types of paths will be monitored:

- “or” alternatives created at the top level of the assertion
- alternatives created by time ranges in the assertion (values from ranges in `##`, `[*]`, `[->]` and `[=]` operators)

URG will not add a new metric for assertion path coverage. There will still be a single overall score called “Assert” reported. See section 3 for how this is computed.

Summary table of properties with path coverage

We will have a new table of cover properties for which paths were monitored. This will be separate from the table for regular properties, since the columns will be different. The assertion path table will include the number of paths and how many were uncovered/covered instead of listing the number of attempts/matches:

NAME	CATEGORY	SEVERITY	PATHS	UNCOVERED	COVERED	PERCENT
modu.i1.p	0	0	39	17	22	56.41
modu.q	0	0	24	20	4	16.67
modu.m3.j2.r	0	0	144	54	90	62.50

This new table will be added to the asserts.html report, in a new table titled “Assertion Path Coverage”. The tables for cover properties, sequences and assertions will follow it in the page.

Note that assertion path coverage can be recorded for assertions, properties or sequences (But only certain ones; it depends on which constructs were used in the expression. For example, implication cannot be used). All of these types - assertions, properties and sequences - for which path coverage is recorded will be listed in this new table shown above.

Details for a Property

For each property with assertion path information, we will show the annotated source of the property along with coverage information for each path.

Consider the following cover property:

```
property p;
```

```

int var_1;
@clk
a ##[2:4] b
or
(a, var_1=ex) ##1 c[*1:$] ##1 !a && (ex==var_1) [*1:8] ##1 a;
endproperty

```

URG first shows the text of the property:

```

property p;
  int var_1;
  @clk
  a ##[2:4] b
  or
  (a, var_1=ex) ##1 c[*1:$] ##1 !a && (ex==var_1) [*1:8] ##1 a;
endproperty

```

It then shows each top-level “or” branch, annotated with index numbers on any values leading to separate paths. Underneath the annotated text, we will create a table that indicates graphically which values were covered or uncovered, giving the counts for non-compressed bins. In the example below, the value 2 is covered with 8 hits, and the values 3 and 4 each had only 3 hits, not enough to count as covered.

BRANCH 1

a ##[2:4] b			
-1-			
-1-	COUNT	AT LEAST	STATUS
2 3 4			
<div><div></div><div></div><div></div></div>	8	5	Covered
<div><div></div><div></div><div></div></div>	3	5	Not covered
<div><div></div><div></div><div></div></div>	3	5	Not covered

In some cases uncovered values may be compressed into a single line in the table. This is shown by coloring in multiple blocks under the relevant index number. For example, in the table below, the

cases where -1-is either 1 or 2 and -2-is 2, 3, 4, 5, or some other value are all uncovered, and they are shown in a single line in the report (the first red line):

BRANCH 2

```
(a, var_1=ex) ##1 c[*1:$] ##1 !a && (ex==var_1) [*1:8] ##1 a;
--1-                                --2-
```

-1-						-2-						COUNT	AT LEAST	STATUS
1	2	3	4	5	+	1	2	3	4	5	+			
█						█						5	5	Covered
█							█					7	5	Covered
			█				█					5	5	Covered
				█			█					5	5	Covered
					█		█					5	5	Covered
█	█					█	█	█	█	█	█	—	5	Not covered
		█				█	█	█	█	█	█	—	5	Not covered
			█			█	█	█	█	█	█	—	5	Not covered

This same format is used when the index numbers are large, for example:

BRANCH 2

```
(a, var_1=ex) ##1 c[*50:$] ##1 !a && (ex==var_1) [*100:108] ##1
--1--                                ----2---
```

-1-						-2-						COUNT	AT LEAST	STATUS
50	51	52	53	54	+	100	101	102	103	104	+			
█						█						5	5	Covered
█							█					7	5	Covered
			█				█					5	5	Covered
				█			█					5	5	Covered
					█		█					5	5	Covered
█	█					█	█	█	█	█	█	—	5	Not covered
		█				█	█	█	█	█	█	—	5	Not covered
			█			█	█	█	█	█	█	—	5	Not covered

However, if one of the ranges contains too many different values, we will use a simple table showing the values instead, to avoid blowing up the table horizontally. For example, in the table below, values from 1-50 were collected, which would make the table form above too wide. So we just show the numbers directly:

BRANCH 2

```
(a, var_1=ex) ##1 c[*1:$] ##1 !a && (ex==var_1) [*1:8] ##1 a;
--1-                                --2-
```

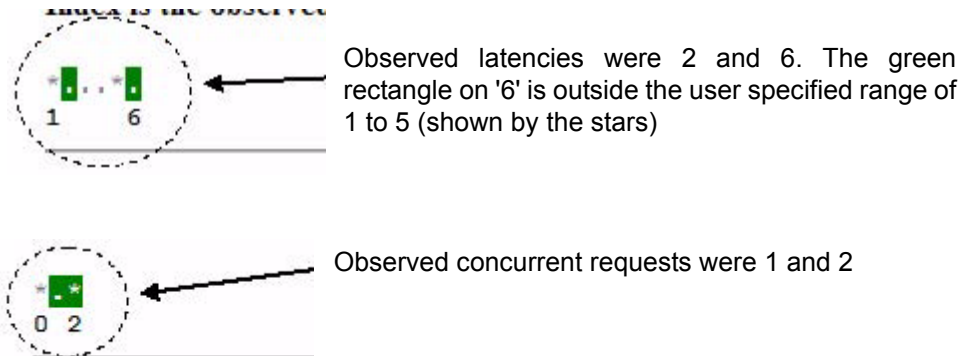
-1-	-2-	COUNT AT LEAST STATUS		
1	1	5	5	Covered
2	1	7	5	Covered
4	2	5	5	Covered
5	2	10	5	Covered
other	2	7	5	Covered
1,2	2-50, other	—	5	Not covered
3	1-50, other	—	5	Not covered
4-50, other	1,3-50, other	—	5	Not covered

Details for built-in checkers

There is a special class of assertion objects called built-in checkers. These checkers have both cover properties/assertions in them as well as collecting some “bin hit” type data. For example, the `assert_arbiter` checker reports this data:

- Coverage of property `cover_abandoned_req` (number of attempts, matches, incompletes)
- Which latency values were observed in the arbiter
- The observed numbers of concurrent requests

The `assertCovReport` tool reports these values like this:



As shown above, the number line displays under `observed_latency` and `concurrent_requests` show which numbers were covered. This project will add reporting for checkers that contains this same information. Currently URG reports an error if checkers are found in the database and does not report them at all.

't know how many times 2 concurrent requests were observed during simulation). URG will not report counts for these values, either.

The different built-in checkers have different coverage levels as part of their names. In the example above, the property is "`cov_level_2_0`". These levels indicate which properties will be enabled in a given simulation run; they are controlled by the user with special parameters. In URG we will just report these with their full names (as `assertCovReport` does).


If a checker has level 2, it means a special report will be generated, like the one above for `assert_arbiter`. Only level 2 checkers have these special reports. They will be included in the details section of the instance in which the checker lives inside the design.

The details for level 2 checkers will be shown in the instance in which the checker was instantiated. It will be the last section in the “Assertion Coverage for Instance” section (after “Assertion Details”, “Cover Directives for Properties Details”, etc.). There will be one of these details sections for each level 2 checker in the instance.


For example, the `assert_arbiter` whose `assertCovReport` page is shown above would look like this in the URG report:

Level 2 Checker `assert_arbiter`

`cov_level_2_1.observed_latency`
Index is the observed latency in clock cycles



`cov_level_2_2.concurrent_requests`
Index is the number of observed concurrent requests



note the counts for properties belonging to this Level 2 checker are repeated here

Name	Attempts	Matches	Vacuous Matches	Incomplete
<code>loop_A[0].cov_level_2_0.cover_abandoned_req</code>	10	1	0	0
<code>loop_A[1].cov_level_2_0.cover_abandoned_req</code>	10	1	0	0

Go to top

Other Changes

The `asserts.html` report will change such that each assertion name is a link to the assertion coverage section of its containing instance. This is being added for two reasons:

- Assertions are always associated with an instance, and this lets the user go directly to the instance containing an assertion to see other details

- Level 2 checkers have additional information to be display, as explained in section The link for level 2 checkers will take the browser directly to those details.

Note that finding the design instance containing a given property can be confusing. For example, clicking on `top.arbiter_inst.loop_A[0].cov_level_2_0.cov_abandoned_req` will link to the assertions coverage report for instance "top.arbiter_inst". The other parts of the property's pathname are artifacts of the checker library and not instances in the design.

Computing the score in URG

URG will compute the assertions coverage score of region as follows. Let P_T be the total number of paths for all assertions with paths monitored and P_C be the total of those that were covered, and let A_T be the total number of assertions for which paths were not monitored and A_C be the total number of those assertions that were covered. Then the assertions coverage score for the region is:

$$(A_C + P_C) / (A_T + P_T) * 100$$

In other words, assertion paths are treated like coverable objects.

5

Using the SVAPP Utility

This chapter describes the SystemVerilog Assertions post-processing (SVAPP) utility available with VCS . This chapter contains the following sections:

- [“Overview”](#)
- [“Use Model”](#)
- [“Example using SystemVerilog Testbench”](#)
- [“Limitations”](#)

Overview

The SVA post-processing (SVAPP) utility enables you to debug your SystemVerilog assertions (SVA) code without having to recompile or re-simulate your design (and testbench).

You first create a VPD file using VCS during the simulation of your design or testbench (but without SVA) and then you input the VPD file and your SVA code to SVAPP. SVAPP uses the VPD file as a design description when it compiles your SVA code. SVAPP does not need to parse or compile your design code.

When SVAPP compiles your SVA code, an executable named `simv.svapp` is created by default that runs after the SVAPP utility compiles your assertion code.

SVAPP displays SVA messages to help you debug your code. It displays information about failed assertions, the time when they failed and the signal value that caused the failure.

SVAPP can also write another VPD file, so that you can examine the assertions with DVE.

Use Model

To use the SVAPP utility, you need to perform the following tasks:

1. Simulate your design using the `$vcdpluson` system task to create the VPD file.
2. Run SVAPP by entering the `svapp` command line.

The syntax for the `svapp` command line is as follows:

```
svapp assertions_filenames [-vpdin filename]
[-vpdout filename] [-vcsflags "compile-time_options"]
[-simflags "runtime_options"] [-h] [-f filename]
[-o filename]
```

The arguments and options are as follows:

`assertions_filenames`

One or more SVA code files. As an alternative you can specify these files with the `-f` option.

`-vpdin filename`

Specifies an alternative name and location for the VPD file that SVAPP uses for the design description. Enter this option if the input VPD file is not `vcdplus.vpd` in the current directory.

`-vpdout filename`

Specifies another VPD files, one that SVAPP writes. You can use this file to debug your SVA code in DVE.

`-vcsflags "compile-time_options"`

Passes VCS compile-time options to SVAPP. You use this option for passing compile-time options for SVA code such as `-assert enable_diag`, `-cm assert`, and `-assert dve`.

`-simflags "runtime_options"`

Passes VCS runtime options (also called simulation options) to SVAPP. You use this option to pass runtime options such as `-assert success`.

`-h`

Displays the required and optional arguments and options.

`-f filename`

Specifies a file containing a list of your SVA code files.

`-o filename`

By default, SVAPP writes an executable file named `simv.svapp` in the current directory. You use this option if you want a different location or name for this executable file.

Example using SystemVerilog Testbench

The following is an example of using SVAPP.

This `my.sv` file contains a SystemVerilog testbench:

```
interface intfclk(sig);
    input clk;
    output logic [7:0] sig;
endinterface

program tst(intfclk ifc);
integer i = 0;
initial begin
    ifc.sig[4:0] = 4'h2;
    @(posedge ifc.clk);
    @(posedge ifc.clk);
    ifc.sig[4:0] = 4'h0;
    @(posedge ifc.clk);
    @(posedge ifc.clk);
    ifc.sig[7:0] = 8'h1;
    @(posedge ifc.clk);
    @(posedge ifc.clk);
    ifc.sig[7:4] = 4'h3;
    @(posedge ifc.clk);
    @(posedge ifc.clk);
    ifc.sig[7:0] = 4'h7;
    @(posedge ifc.clk);
    @(posedge ifc.clk);
    ifc.sig[3:0] = 4'h2;
    @(posedge ifc.clk);
    @(posedge ifc.clk);
end
endprogram
```

This my.v file contains a design module:

```
module test;
initial begin
    $vcdpluson;
end

parameter simulation_cycle = 100;
reg SystemClock;
wire [7:0] ifc_sig;
intfc ifc(SystemClock,ifc_sig);
tst drive(ifc);
initial begin
    SystemClock = 1;
    forever begin
        #(simulation_cycle/2) SystemClock = ~SystemClock;
    end
end
initial $monitor($time,,ifc_sig);
endmodule
```

Compile and simulate the design and testbench with the following command line:

```
vcs -sverilog my.v my.sv -debug_pp -R
```

This assert.sva file contains the following SVA code:

```
module my_checker(input clk, input reset);
property p;
    @(posedge clk) reset== 0;
endproperty

a : assert property(p);
endmodule

bind test my_checker inst (test.ifc.clk, test.ifc.sig[0]);

module my_checker2(input clk, input logic [7:0] expr);
property check_expr_true;
    $countones(expr);
endproperty
```

```

a_check_prop: assert property(@(posedge clk)
check_expr_true);

endmodule

bind test my_checker2 inst2 (test.ifc.clk, test.ifc.sig);

```

Run SVAPP with the following command line:

```
svapp assert.sva
```

SVAPP displays the following SVA messages:

```

"assert.sva", 6: test.inst.a: started at 0s failed at 0s
    Offending '(reset == 1'b0)'
"assert.sva", 16: test.inst2.a_check_prop: started at 0s
failed at 0s
    Offending '$countones(expr)'
"assert.sva", 6: test.inst.a: started at 100s failed at 100s
    Offending '(reset == 1'b0)'
"assert.sva", 16: test.inst2.a_check_prop: started at 100s
failed at 100s
    Offending '$countones(expr)'
"assert.sva", 6: test.inst.a: started at 200s failed at 200s
    Offending '(reset == 1'b0)'
"assert.sva", 16: test.inst2.a_check_prop: started at 200s
failed at 200s
    Offending '$countones(expr)'
"assert.sva", 16: test.inst2.a_check_prop: started at 300s
failed at 300s
    Offending '$countones(expr)'
"assert.sva", 16: test.inst2.a_check_prop: started at 400s
failed at 400s
    Offending '$countones(expr)'
"assert.sva", 6: test.inst.a: started at 500s failed at 500s
    Offending '(reset == 1'b0)'
"assert.sva", 6: test.inst.a: started at 600s failed at 600s
    Offending '(reset == 1'b0)'
"assert.sva", 6: test.inst.a: started at 700s failed at 700s

```

```

        Offending '(reset == 1'b0)'
"assert.sva", 6: test.inst.a: started at 800s failed at 800s
        Offending '(reset == 1'b0)'
"assert.sva", 6: test.inst.a: started at 900s failed at 900s
        Offending '(reset == 1'b0)'
"assert.sva", 6: test.inst.a: started at 1000s failed at
1000s
        Offending '(reset == 1'b0)'
"assert.sva", 6: test.inst.a: started at 1100s failed at
1100s
        Offending '(reset == 1'b0)'
"assert.sva", 16: test.inst2.a_check_prop: started at 1100s
failed at 1100s
        Offending '$countones(expr)'
"assert.sva", 6: test.inst.a: started at 1200s failed at
1200s
        Offending '(reset == 1'b0)'
"assert.sva", 16: test.inst2.a_check_prop: started at 1200s
failed at 1200s
        Offending '$countones(expr)'

```

Limitations

SVAPP has the following limitations:

- SVAPP is not supported on 64-bit platforms
- bind statements cannot include function calls. The following would be invalid in SVAPP:

```

module design_mod (input clk, reset);
.
.
.
function myfunc;
input clk,reset;
.
.
.

```

```

endfunction

endmodule

module assert_mod (input clk, input reset);
.
.
.
endmodule

bind design_mod assert_mod dm1 (clk, myfunc(clk,reset));

```

You can, however, use a function local variable in the bind statement, for example:

```

module design_mod (input clk, reset);
.
.
.
function myfunc;
input clk,reset;
logic funclog1;
.
.
.
endfunction

endmodule

module assert_mod (input clk, input reset);
.
.
.
endmodule

bind design_mod assert_mod dm1 (clk, myfunc.funclog1);

```

- **Cross module references (XMRs) are supported only in the assertion expression or in the bind statement, for example:**

```

module assert_mod (input clk, input reset);

```



```

.
.
.
property p2;
    @(posedge clk) test.dm1.log1 == 0;
endproperty

a2 : assert property (p2);

endmodule

bind design_mod assert_mod dm1 (.clk(clk),
    .reset(reset));

```

Here is a cross module reference (XMR) to signal log1 in design module instance test.dm1. The XMR is in the property definition. We do *not* recommend this technique. We suggests that you avoid XMRs and you read and write values using the assertion module ports instead.

Using XMRs from the assertion module to assign values to design signals is not supported at all, for example:

```

module assert_mod (input clk, input reset);
property p1;
    @(posedge clk) reset== 0;
endproperty
a1 : assert property(p1)
begin
.
.
.
end
else
begin
.
.
.
test.dm1.log1=1;
end

```

```
endmodule  
bind design_mod assert_mod dm1 (.clk(clk),  
.reset(reset));
```

Assigning a value to a design signal in an action block will not work.

- SVAPP does not support dynamic variables.

6

Using the let Construct

The `let` construct facilitates the creation of a modeling layer for SystemVerilog Assertions and assertion-based checkers.

This chapter contains the following topics:

- [“let Construct Overview”](#)
- [“let Construct Examples”](#)
- [“let Construct Syntax”](#)

let Construct Overview

The purpose of the `let` construct is to allow macro-like parameterized definitions of symbols that have the following characteristics:

- `let` statements can define parameterized expressions (like macros) that can be instantiated in both assertions and procedural code.

Note:

To define property or sequence expressions, use existing sequence and property declarations.

- As in Mantis items 928 and 1601 for sequences and properties, the formal arguments can optionally be typed. To declare a type for a formal argument of a `let` statement, it is required to prefix the argument with a type. A formal argument that is not prefixed by a type will be untyped. A type name can refer to a comma-separated list of arguments.
- The supported data types for `let` formal arguments are the types that are allowed for operands in assertion Boolean expressions (see 17.4.1). There are two ways to achieve implicit typing of arguments. The first is to write the implicitly typed arguments at the beginning of the formal argument list, prior to any typed argument. The second is to use the context type. Because a type applies to multiple comma-separated arguments, the context type is required if an implicitly typed argument is to be placed after a typed argument in the formal argument list. The context type specifies that the semantics for binding to the argument shall be as though the argument were written at the beginning of the formal argument list, prior to any typed argument.
- The formal arguments can have optional default values.

- Scoping rules are such that referenced identifiers or names in the `let` expression which are not formal arguments bind in the declarative scope of the `let` statement; this is similar to `sequence` and `property` declarations. Such names must be declared before they are used. In the scope of the declaration, `let` must be defined before it is used.
- The `let` expression is inlined in the place of instantiation without any partial evaluation of the expression. References to identifiers or names in the declarative scope of the `let` statement are replaced by hierarchical references if they are different from the instantiation scope. This is to avoid collisions with local declarations. Recursive `let` instantiations are not permitted.
- `let` expressions can contain sampled value function calls (`$rose`, `$fell`, `$past`, `$stable`). If the clock is not explicitly specified, it is inferred from the instantiation context in the same way as if the functions were used in `sequence` or `property` definitions.
- `let` expressions are checked for valid syntax in both the definition and the instantiation context.
- Individual instances of `let` expressions can be viewed as variables in the debug environment provided the instance is not dependent on the instantiation context, such as if the bit width is not self-determined or the `let` statement contains formal arguments.
- `let` statements can be referenced using a hierarchical reference in another scope, but as mentioned above, any names that are not formal arguments will be bound in the declarative scope of the `let` statement.

- Like `sequence` and `property` declarations, `let` statements can be placed in generated blocks (conditional, loops) and referenced like any other statement.
- `let` statements can be declared in an interfaces and accessed by a port reference to the interface or modport in a module.
- `let` statements can be imported from a package. Binding of identifiers or names that are not formal arguments must be satisfied by the declarations in the package.
- `let` statements are not synthesized by Design Compiler: `let` is not part of design code, but the result of its expansion in the instantiation context is synthesized (provided that the code is synthesizable).
- If assertions are compiled into RTL for emulation, the result of inlining `let` constructs is compiled as well.

The `let` statement is expanded in the instantiation context in a similar way as a sequence declaration is, variables that are not formal arguments are bound in the scope of the `let` definition (declarative context.)

A `let` instance can appear wherever expression is allowed.

Basic syntax checking is done on the definition itself, to check that it is a valid expression (for example, that there is no `always` statement in `let_expr`.)

If there are syntactic errors caused by wrong instantiation then an error message relative to the instantiation context is issued, containing a reference to the corresponding `let` definition. For example:

```
let concat(x, y) = {x , y};
```

```
...
always concat(b, c); // erroneous instance
```

A debug variable is not created for `let` statements that do not have self-determined bit width or contain formal arguments.

Names of the additional `let` variables for debugging may consist of the following concatenation of strings:

```
let_let-identifier_line-number-of-context
```

There is only one variable declaration required for each `let` statement. If there is more than one `let` statement declared on the same line, an occurrence number may be appended to the variable name. The occurrence number is the occurrence index of the `let` statement on the line. The name then has the following form:

```
let_let-identifier_line-number-of-context_occurrence-
number
```

let Construct Examples

Example 6-1 let with arguments and without arguments

```
module m;
  bit a, b;
  let eq(x, y=b) = x == y; // with formal arguments and
    // default value on y
  let tmp = a && b; // without parameters, binds to
    // a, b above
  sequence s(x);
    x ##1 b;
  endsequence
  property p;
    bit a;
    @(posedge clk) eq(a) | => (tmp==0) ##0 s(a); // default
    // value b used for y
```

```

        endproperty : p
    al: assert property (p);
endmodule

```

The effective code after inlining:

```

module m;
    bit a, b;
    // let eq(x, y=b) = x == y;
    // let tmp = a && b;
    var type(a && b) let_tmp_4; assign let_tmp_4 = a && b;
    // debug var for tmp
    sequence s(x);
        x ##1 b;
    endsequence
    property p;
        bit a;
        @(posedge clk) a == m.b | => (m.a && m.b == 0)
            ##0 (a ##1 m.b);
    endproperty : p al: assert property (p);
endmodule

```

Example 6-2 Declarative context binding of `let` arguments

```

let x = 1'b1;
let y = !x;
bit z = y;
...
begin
    let x = 1'b0; // redundant definition, y binds to
        // preceding definition
    a <= y; end

```

The resulting code after `let` expansion:

```

// let x = 1'b1;
var type(1'b1) let_x_1; assign let_x_1 = 1'b1;
// let y = !1'b1;
var type(1'b1) let_y_2; assign let_y_2 = !1'b1;
bit z = ! 1'b1;
...

```



```

begin
    a <= ! 1'b1;
end

```

Example 6-3 Sequences (and properties) in structural context

```

begin : top
    logic a, b;
    let x = a;
    sequence s;
        @clk x ##1 b;
    endsequence : s
    generate
        begin : mid
            logic a, b;
            ap: assert property(@clk s.ended |-> s);
            ...
        end : mid
    endgenerate
end : top

```

After let inlining:

```

begin : top
    logic a, b;
    // let x = a;
    var type(a) let_x_3; assign let_x_3 = a; // debug variable
    sequence s;
        @clk a ##1 b;
    endsequence : s
    generate
        begin : mid
            logic a, b;
            ap: assert property(@clk s.ended |-> s);
            ...
        end : mid
    endgenerate
end : top

```

After sequence inlining (full variable path names used to show exact binding):

```

begin : top
  logic a, b;
  // let x = a;
  var type(a) let_x_3; let_x_3 = a;
  sequence s; // form that runs in s.ended
    @clk top.a ##1 top.b;
  endsequence : s
  generate
    begin : mid
      logic a, b;
      // both s.ended and explicit s use top declarations
      ap: assert property(@clk s.ended |-> top.a ##1 top.b);
      ...
    end : mid
  endgenerate
end : top

```

Example 6-4 Sequences (and properties) used in procedural context

```

begin : top
  logic a, b;
  let x = a;
  sequence s;
    @clk x ##1 b;
  endsequence : s
  always @clk begin : mid
    logic a, b;
    ap: assert property(s.ended |-> x); ...
  end : mid
end : top

```

After let inlining and clock inference:

```

begin : top
  logic a, b;
  // let x = a;
  var type(top.a) let_x_3; assign let_x_3 = top.a;
  sequence s;
    @clk a ##1 b;
  endsequence : s
  always @clk begin : mid

```

```

        logic a, b;
        ap: assert property(@clk s.ended |-> top.a); ...
    end : mid
end : top

```

After sequence inlining:

```

begin : top
    logic a, b;
    // let x = a;
    var type(a) let_x_3; assign let_x_3 = a;
    sequence s; // form that runs in s.ended
        @clk top.a ##1 top.b;
    endsequence : s
    always @clk begin : mid
        logic a, b;
        ap: assert property(@clk s.ended |-> top.a );
        ...
    end : mid
end : top

```

Example 6-5 *let declared in a scope and referenced using a hierarchical reference:*

```

module m0(...);
    bit a, b;
    let my_let(x) = !x || a && b;
    ...
endmodule
module m(...);
    bit clk;
    reg a, b, c;
    my_assert: assert property (@(posedge clk) m0.my_let(c));
    ...
endmodule

```

After let expansion it becomes:

```

module m0(...);
    bit a, b;

```

```

        let my_let(x) = !x || m0.a && m0.b;
        ...
    endmodule
module m(...);
    bit clk;
    reg a, b, c;
    // the let expression refers to a, b from m0 scope
    my_assert: assert property (@(posedge clk) !c ||
        m0.a && m0.b);
    ...
endmodule

```

Example 6-6 *let declared in a generate statement*

```

module m(...);
    bit clk, a, b;
    bit [2:0] c;
    for (genvar i = 0; i < 3; i++) begin : L0
        if (i != 1) begin : L1
            let my_let(x) = !x || b && c[i];
            my_assert: assert property (@(posedge clk)
                my_let(a));
        end : L1
    end : L0
endmodule

```

This will resolve to the following equivalent code:

```

module m(...);
    bit clk, a, b;
    bit [2:0] c; begin : L0[0]
        begin : L1
            let my_let(x) = !x || m.b && m.c[0];
            my_assert: assert property (@(posedge clk) !m.a ||
                m.b && m.c[0]);
        end
    end begin : L0[2]
    begin : L1
        let my_let(x) = !x || m.b && m.c[2];
        my_assert: assert property (@(posedge clk) !m.a ||
            m.b && m.c[2]);
    end
end

```

```

    end
endmodule

```

The *let* statements can also be referred to hierarchically using the paths `m.L0[0].L1` and `m.L0[2].L1`. For example, `L0[0].L1.my_let(c)` used inside `m` will expand to `m.c || m.b && m.c`.

Example 6-7 Reference to a let statement in an interface

```

interface itf;
    logic a, b;
    let my_let = a && b; // a and b resolve within itf
    wire c = my_let;
    modport mp1(input a, output b, c);
    modport mp2(output a, input b, c);
endinterface
module m(itf bus); // cannot access via modport, must be the
    // interface type
    bit clk, a, b;
    // bus.my_let will expand into bus.a && bus.b
    my_assert: assert property @(posedge clk) bus.my_let;
    ...
endmodule

```

Example 6-8 Import from a package should follow the same rules as when importing functions from packages.

```

package pack;
    function bit my_fn(bit x); ...; endfunction
    let my_let(x, y) = x && my_fn(y);
endpackage
module m1 (...);
    import pack :: *;
    bit clk, a, b;
    // my_let(a, b) will expand into m1.a && my_fn(m1.b)
    my_assert: assert property @(posedge clk) my_let(a, b);
    ...
endmodule

```

Example 6-9 *Using sampled value functions*

```
begin : top
  logic a, b;
  let x = $past(a);
  sequence s;
    x ##1 $rose(b);
  endsequence : s
  generate
    begin : mid
      logic a, b;
      ap: assert property(@clk s.ended |-> s); ...
    end : mid
  endgenerate
end : top
```

After let inlining:

```
begin : top
  logic a, b;
  // let x = $past(a);
  sequence s;
    $past(a) ##1 $rose(b); // No clock yet, a and b bound
  endsequence : s
  generate
    begin : mid
      logic a, b;
      ap: assert property(@clk a |-> s); ...
    end : mid
  endgenerate
end : top
```

After sequence inlining and clock inference:

```
begin : top
  logic a, b;
  // let x = $past(a);
  // There is no debug variable since $past depends on
  // context for its clock
  sequence s;
    $past(a) ##1 $rose(b); // No clock inferred
```

```

endsequence : s
generate
  begin : mid
    logic a, b;
    ap: assert property(@clk top.mid.a |->
      $past(top.a) ##1 $rose(top.b));
      // clk used in $past and $rose
    ...
  end : mid
endgenerate
end : top

```

Example 6-10 *Typed formal arguments and named actual argument association type of x and y is bit, type of z is determined from the actual argument.*

```

module m;
  bit a, b; int v;
  let eq(bit x, y=b, context z=3) = (z == v) &&(x == y);
  // default on y and z
  let tmp = a && b; // without parameters, binds to
  // a, b above
  ...
  sequence s(x);
    x ##1 b;
  endsequence
  property p;
    bit a;
    @(posedge clk) eq(.x(a)) |=> (tmp==0) ##0 s(a);
    // default b used for y , 3 for z
  endproperty : p
  a1: assert property (p);
endmodule

```

The effective code after inlining:

```

module m;
  bit a, b;
  // let eq(bit x, y=b, context z=3) = (z == v) &&(x == y);
  // let tmp = a && b;

```

```

var type(a && b) let_tmp_4; assign    let_tmp_4 =
    a && b; // debug var for tmp
sequence s(x);
    x ##1 b;
endsequence
property p;
    bit a; @(posedge clk) (3 == m.v) && (a == m.b) | =>
        (m.a && m.b == 0) ##0 (a ##1 m.b);
endproperty : p a1: assert property (p);
endmodule

```

let Construct Syntax

```

let_declaration ::=
    let let_identifier[ ( [let_port_list] ) ] = let_expr;

let_identifier ::=
    identifier

let_port_list ::=
    let_port_item {, let_port_item}

let_port_item ::=
    { attribute_instance } let_formal_type identifier
    [= expression]

let_formal_type ::=
    data_type_or_implicit | context

let_expr ::=
    expression

let_instance ::=
    let_identifier[ ( [let_list_of_arguments] ) ]

let_list_of_arguments ::=
    [let_actual_arg] {, [let_actual_arg] }
    {, . identifier ([let_actual_arg]) }
    | . identifier ([let_actual_arg] )

```



```
      { , . identifier ( [let_actual_arg] ) }  
let_actual_arg ::=  
    let_instance  
    | expression
```


7

New std::randomize() Function

The `randomize()` function randomizes variables that are not class members.

Syntax

```
[std::] randomize(variable-identifier-list)  
    [with constraint-block]
```

Description

SystemVerilog defines extensive randomization methods and operators for class members. Most modeling methodologies recommend the use of classes for randomization. However, there are situations where the data to be randomized is not available in a class. SystemVerilog provides the `std::randomize()` function to randomize variables that are not class members.

The `std::randomize()` function can be used in the following scopes:

- module
- function
- task
- class method

Arguments to `std::randomize()` can be of integral types including:

- integer
- bit vector
- enumerated type

Object handles and strings cannot be used as arguments to `std::randomize()`.

The variables passed to `std::randomize()` must be visible in the scope where the function is called. Cross-module references are not allowed as arguments to the `std::randomize()` function.

All constraint expressions currently available with `obj.randomize()` in VCS can be used as constraints in the *constraint-block*.

Only constraints specified in the constraint block are honored. Any rand mode specified on the class members is ignored when `std::randomize()` is called with the given class member.

The `pre_randomize()` and `post-randomize()` tasks are not called when `std::randomize()` is used within a class member function.

The “`std::`” prefix must be explicitly specified for the `randomize()` call.

The `std::randomize()` function is supported in VCS. Files containing `std::randomize()` calls can be compiled with `vlogan`.

The function using `std::randomize()` can be declared in a task inside a package that can be imported into modules and programs.

Example

```
module M;
    bit[11:0] addr;
    integer data;

    function bit genAddrData();
        bit success;
        success = std::randomize(addr, data);
        return success;
    endfunction

    function bit genConstrainedAddrData();
        bit success;
        success = std::randomize(addr, data)
            with {addr > 1000; addr + data < 20000;};
        return success;
    endfunction
endmodule
```

The `genAddrData` function uses `std::randomize(addr, data)` to assign random values to `addr` and `data` variables. The `std::randomize()` function randomizes any variables that are visible in the scope.

The `getConstrainedAddrData()` function uses `std::randomize(addr, data)` to assign random values to `addr` and `data` variables. In this case there is an additional constraint given to the call, which is that `addr` is greater than 1,000 and `addr+data` is less than 20,000.

8

Support for IEEE Verilog Encryption

Three new options enable the VCS compiler to function as an encryptor application compliant with the “IEEE Std1364-2005” standard for encryption envelopes. These switches are:

- `-ipkey <key>`
- `-ipprotect <protect.v>`
- `-ipout <filename.vp>`

This section describes the above three switches in detail.

`-ipkey <key>`

This option instructs the VCS compiler to enter into encryption mode. In this mode, VCS does not compile the (Verilog) source file(s), but instead encrypts each source file specified in the command line into

a separate encrypted Verilog file. Each encrypted file is saved under the same base name, but will add a suffix 'p' to its extension (see the `-ipout` option to modify this behavior).

Using the `-ipkey` option allows IP providers to specify their private key to encrypt the data block. At the moment, `<key>` is limited to 128 bits in length with the AES method. If the `<key>` argument is longer than the maximum allowed for a particular encryption method (or system imposed limit), it will be truncated and a warning will be issued.

`-ipprotect <protect.v>`

This option instructs VCS to enter into encryption mode. In this mode, VCS does not compile the (Verilog) source file(s), but instead encrypts each source file specified in the command line into a separate encrypted Verilog file. Each encrypted file is saved under the same base name, but will add a suffix 'p' to its extension (see the `-ipout` option to modify this behavior).

Using the `-ipprotect` option allows IP providers to specify a protection header file that contains various protection pragmas. This mechanism allows IP providers to either specify a private encryption key (using the `data_decrypt_key` pragma), or instruct VCS to use its own private encryption key (by not specifying the `data_decrypt_key` pragma). If the user specified key is longer than the maximum allowed for a particular encryption method (or system imposed limit), it will be truncated and a warning will be issued.

-ipout <filename.vp>

This option instructs VCS to save the encrypted file under the specified name, <filename.vp>. This option applies only to the 1st file specified in the command line; any subsequent source files (or library files or ``include` files) are unaffected by this option.

The `-ipout` option is only active when either `-ipkey` or `-ipprotect` options are specified; otherwise it has no effect.

Important: The `-ipkey` and `-ipprotect` options are mutually exclusive. If both of these options are specified, VCS will exit with an error. If neither the `-ipkey` or `-ipprotect` options are specified, VCS will operate as a compiler, that is, it will compile (or decrypt and compile) the specified source files.

All ``include` directives in the encrypted sources will get modified with a suffix 'p' to its extension. The modified `include` directives are left as clear text. In addition, every file included via an ``include` directive will also be encrypted and saved under the modified filename (with a suffix 'p' to the extension).

In encryptor mode, VCS also accepts the following options:

`+incdir+<directory>`

Specifies the directories to search for files included via the ``include` compiler directive. Multiple directories can be specified by separating each path name with the `+` character.

Any included file will also be encrypted and saved in the directory in which it was found; the base name of the encrypted file will be the same but will add a suffix 'p' to its extension.

`-f <filename>`

Specifies a file that contains a list of path names to source files and compile-time options.

The specified filename will not be encrypted.

`-F <filename>`

Same as the `-f` option but allows specification of a path to the file and the source files listed in the file do not have to be absolute path names.

`+define+<MACRO>=<VALUE>`

Defines the given text macro to the indicated value.

Any macro specified via the `+define` option to the encryptor will disallow end-users from overriding the given macro at compile time. If user's attempt to change the value of the macro, the VCS compiler will silently ignore the new value and instead use the value specified at encryption time.

The following options may be needed in combination with the above ones.

`+v2k`

Enables the use of new Verilog constructs in the 1364-2001 standard.

`-sverilog`

Enables the use of the SystemVerilog language extensions in the P1800 standard.

Files referenced through ``include` options are not explicitly listed on the command line. Hence, when VCS encrypts these files, they will have a suffix 'p' to their extension.

Encryption Pragmas

The VCS encryption mechanism supports the following standard encryption pragmas:

- `author`
- `author_info`
- `data_method`
- `encoding`
- `key_keyowner`
- `key_method`
- `key_keyname`
- `comment`
- `data_decrypt_key`

When specified in a `<protect.vp>` file via the `-ipprotect` option, these pragmas have the following behavior:

`author`

This pragma is fully supported.

`author_info`

This pragma is fully supported.

`data_method`

If the method specified is not supported by a particular VCS version, an error will be generated. Currently, VCS only supports the DES (64-bit) and AES (128-bit) algorithms. Hence, the data method may only specify one of the following methods:

“des-cbc” supports 64 bit (8 bytes) key

“aes128-cbc” supports 128 bit (16 bytes) key

`encoding`

This pragma is parsed and ignored.

Currently, only the “uuencode” encoding is supported.

`key_owner`

This pragma must specify “VCS” to the encryptor. It is an error for this pragma to not specify “VCS” when used in conjunction with the – ipkey option.

`key_method`

This pragma is parsed and ignored (VCS uses its built-in method).

`key_keyname`

This pragma is parsed and ignored.

`comment`

This pragma is fully supported.

`data_decrypt_key`

This pragma must specify the user’s private key in accordance to the active encryption method and any system imposed restrictions.

Note:Currently, this option is ignored. Only the `-ipkey` option allows specification of private keys.

Important:These encryption pragmas are only supported in the `protect.v` file, which is specified via the `-ipprotect` option. If these are specified anywhere else (e.g., the Verilog source), they will trigger a syntax error.

Example `<filename.vp>` (for `-ipprotect`)

Here's the recommended protect header for VCS to supply the private data encryption key:

```
`pragma protect data_method = "des-cbc"
`pragma protect key_keyowner = "VCS"
`pragma protect author = "Xilinx", author_info = "temac.003"
`pragma protect begin
```

Note that the second line above could specify a `key_method` and/or `key_keyname`, but VCS will simply ignore those options and use its internal method.

10

Sequential Distance Coverage for Assertions

This feature provides coverage of sequential elements for assertions by measuring the sequential distance between an assertion and a sequential element (a latch, a register, or a primary input).

This feature is supported for designs written in

- Verilog
- SystemVerilog
- OVA using bind statements to bind OVA units to the design. An OVA unit shall be considered as an assertion, and the inputs of the OVA unit shall be considered the inputs of that assertion.

Note:

This capability is part of VCS Design Checker, provided as a rule to be checked during compile-time static checking. Other rules provided by VCS Design Checker can also be invoked during this analysis. For more information about how to use VCS Design Checker, see the *VCS Design Checker User Guide*.

Overview

When you write assertions to check the behavior of a design, it is equally important to know what portion of the logic is being tested by the assertions as it is to know what behavior is being captured by those assertions.

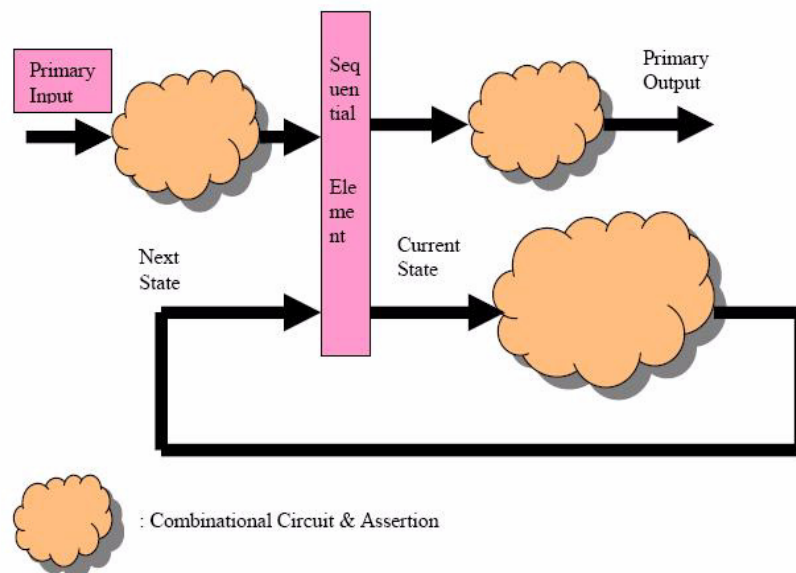
For any complex design, it is not practical to write assertions such that all signals are included in one or more assertions. The number of assertions required for such inclusion of all signals is neither productive nor provides effective use of resources to test a design. Given these consideration, an effective measure is needed that can be an indicator of how well logic is being addressed by the assertions.

One such measure is the notion of sequential distance between an assertion and a sequential element. You can call this sequential distance coverage of sequential elements by assertions. A sequential element is a latch, a register or a primary input.

Sequential Distance Analysis

Consider the case of typical logic as conceptualized by a series of sequential elements with combinational logic between the sequential elements. This is shown in the following figure.

Figure 10-1 Structural view of a digital circuit



An input to an assertion is a signal included in any expression that is evaluated to determine the truth of the assertion. This excludes clocking event expressions and expressions used in task/function calls of sequence match items.

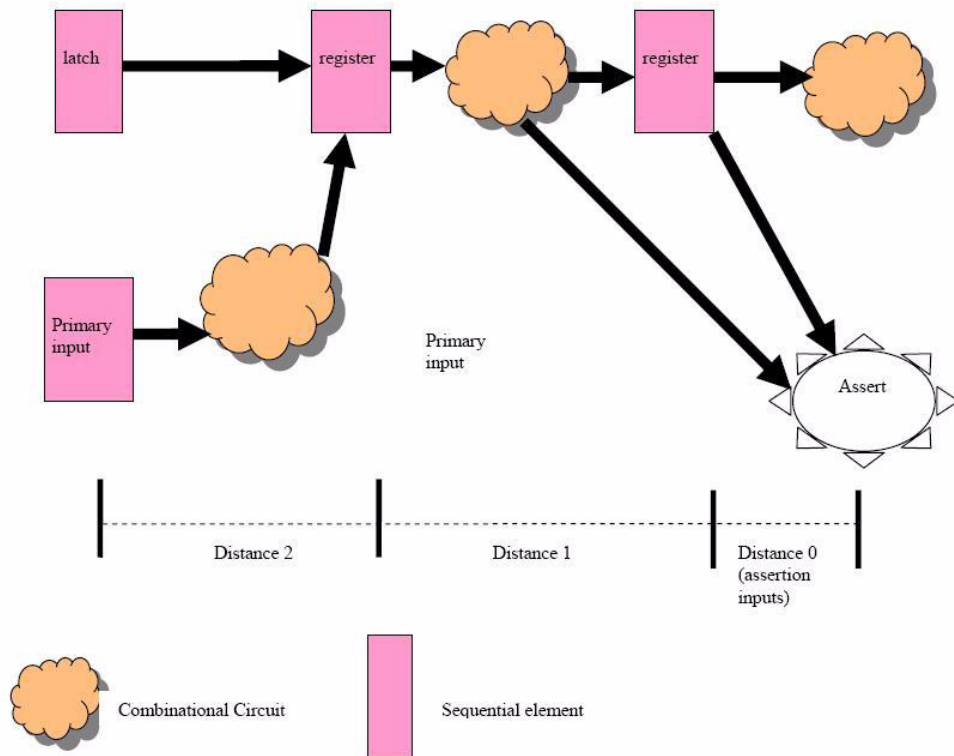
Sequential distance from a sequential element to an assertion is the minimum number of sequential elements in any path from an input of the assertion to that sequential element, excluding the assertion input. The sequential element itself is counted in the distance. For example, if there is no sequential element between the input and the

sequential element, then the distance is 1. If there is one sequential element between the input and the sequential element, then the distance is 2.

If the sequential element is an input to the assertion, then the sequential distance is said to be 0. The sequential distance of a sequential element from an assertion is calculated as follows.

If a sequential element is part of an expression of the assertion, then the distance is 0. Otherwise, all signals in the expressions of the assertion are examined to determine if the fanin logic cone of any signal includes the sequential element. If so, the distance is considered to be 1. If no such signal is found, then the sequential elements included in these fanin cones of all signals are examined. These sequential elements have a distance of 1. Once again, if a sequential element is found in any fanin logic cone of the sequential elements with distance 1, then the distance of that element is considered to be 2. In this manner, the distance of all sequential elements is measured with respect to each assertion.

Figure 10-2 **Sequential Distance**



From this analysis, following information is of prime interest:

1. List of sequential elements which are not within a **specified sequential distance** from any assertion.
2. Sequential distance of a sequential element from any assertion.
3. Number of assertions from which a sequential element is found with a **specified sequential distance**.
4. Sequential elements with a **specified sequential distance** from an assertion.
5. Assertions with identical inputs.

The above measures indicates how encompassing an assertion is to include logic that is observed by that assertion and whether any sequential logic is being left out of assertions. Based on these measures, you can write additional assertions, so that full logic of the design is effectively monitored by the assertions. As it is with coverage functionality, an important trade-off in these measures is the amount of information vs. the time needed to analyze the provided information. To assist in this trade-off, following capabilities are provided:

- An option to exclude portions of hierarchy from the analysis.
- An option to exclude assertions from the analysis.
- A parameter to select the maximum sequential distance to be considered. Above the maximum distance, an element is considered not monitored by the assertion.
- An option to count the number of assertions that cover a sequential element with a specific distance. This distance you can select.
- An option to select a cover, assume, or assert directive.

Rules for Specifying Sequential Distance

The following rules control the specification of sequential distance.

NTL_COV_ASSERT01

Reports the sequential elements that are not covered by any assertion within the specified sequential depth. The sequential elements that are within the hierarchy of the specified `design_top`

are analyzed. The list of sequential elements not covered is reported according to two sorting criteria: hierarchy and depth. The default order of applying the sorting criteria is first by hierarchy, and then by depth.

DEPTH

Specifies the sequential depth that needs to be reached from an assertion.

Value: Integer

Default: Any depth up to the primary inputs

Usage: Optional

INCLUDE_COVER_ASSERT

Controls whether the cover assertion should be included in the analysis.

Value: `true` or `false`

Default: `false`

Usage: Optional

INCLUDE_LATCH

Controls whether latches are considered as sequential elements in calculating sequential depth.

Value: `true` or `false`

Default: `true`

Usage: Optional

STATS_ONLY

Reports only the numbers of covered and uncovered elements at each depth when applying this rule.

Value: `true` or `false`

Default: `false`

Usage: Optional

SORT_DEPTH

Sorts the list of reported sequential elements, first by depth and then by hierarchy.

Value: `true` or `false`

Default: `false`

Usage: Optional

Output reporting occurs in one of two ways:

- Sorted by hierarchy and then by depth (default)
- Sorted by depth and then by hierarchy (use `SORT_DEPTH`)

Examples of the Tcl commands for the configuration file:

```
set_rule -rule NTL_COV_ASSERT01
set_rule_parameter -rule NTL_COV_ASSERT01 -parameter INCLUDE_LATCH -value "true"
set_rule_parameter -rule NTL_COV_ASSERT01 -parameter DEPTH -value "4"
```

NTL_COV_ASSERT02

Calculates the sequential depth from a given sequential element to all assertions. Specify the sequential element(s) with full hierarchical path(s).

ELEMENT

Defines one or more sequential elements, using the full hierarchical path(s), in a comma-separated list.

Default: `None`

Usage: Mandatory

You can specify this parameter with multiple commands. The parameters are appended to the existing list, which allows a number of elements to be specified in one run. You can specify multiple sequential elements with the wildcard notation (*).

Examples of the Tcl commands for the configuration file:

```
set_rule -rule NTL_COV_ASSERT02
set_rule_parameter -rule NTL_COV_ASSERT02 -parameter ELEMENT -value
"top.ml.reg1"
```

NTL_COV_ASSERT03

Calculates the number of assertions from which a sequential element can be reached within the specified sequential depth. The sequential elements that are within the hierarchy of the specified `design_top` are analyzed.

DEPTH

Specifies the sequential depth that needs to be reached from an assertion.

Value: Integer

Default: Any depth up to the primary inputs

Usage: Optional

MAXASSERTS

Reports only violations of sequential elements that can be reached from more than the specified number of assertions.

Value: Integer

Default: None

Usage: Optional

Examples of the Tcl commands for the configuration file:

```
set_rule -rule NTL_COV_ASSERT03
set_rule_parameter -rule NTL_COV_ASSERT03 -parameter MAXASSERTS -value "5"
set_rule_parameter -rule NTL_COV_ASSERT03 -parameter DEPTH -value "4"
```

NTL_COV_ASSERT04

Reports all sequential elements within the transitive fanin of an assertion.

DEPTH

Specifies the sequential depth that needs to be reached from an assertion.

Value: Integer

Default: Any depth up to the primary inputs

Usage: Optional

ASSERTION_NAME

Defines the path name of the assertion.

Value: Hierarchical path to the assertion

Default: None

Usage: Mandatory

Examples of the Tcl commands for the configuration file:

```
set_rule -rule NTL_COV_ASSERT04
set_rule_parameter -rule NTL_COV_ASSERT04 -parameter ASSERTION_NAME
"top.c2.rcheck"
set_rule_parameter -rule NTL_COV_ASSERT04 -parameter DEPTH -value "3"
```

NTL_COV_ASSERT05

Reports groups of assertions whose inputs are identical. For this rule to apply to assertions, all signals that each assertion monitors must be identical.

Example of the Tcl command for the configuration file:

```
set_rule -rule NTL_COV_ASSERT05
```

NTL_COV_ASSERT06

Reports a violation if the transitive fanin of the given assertion with the given depth consists of inputs greater than the given size.

ASSERTION_NAME

Defines the path name of the assertion.

Value: Hierarchical path to the assertion

Default: None

Usage: Mandatory

DEPTH

Specifies the sequential depth that needs to be reached from an assertion.

Value: Integer

Default: Any depth up to the primary inputs

Usage: Optional

SIZE

Defines the limit of the number of inputs in the transitive fanin of the assertion with the given depth.

Value: Integer

Default: None

Usage: Mandatory

Examples of the Tcl commands for the configuration file:

```
set_rule -rule NTL_COV_ASSERT06
set_rule_parameter -rule NTL_COV_ASSERT06 -parameter DEPTH -value "4"
set_rule_parameter -rule NTL_COV_ASSERT06 -parameter SIZE -value "10"
set_rule_parameter -rule NTL_COV_ASSERT06 -parameter ASSERTION_NAME -value
"top.m1.a1"
```

Rule Output

The reporting will be in the same format as the other rule output of VCS Design Checker. Here are some sample of textual report shown below. The VCS Design Checker outputs will be available in the file check.log.

In addition, you can view these error in DVE, in which each error can be linked back to the source window where the relevant element (assertion, sequential element) is present.

NTL_COV_ASSERT01

```
set_rule -rule NTL_COV_ASSERT01
set_rule_parameter -parameter INCLUDE_LATCH -value "true"
set_rule_parameter -parameter DEPTH -value "4"
Lint- [NTL_COV_ASSERT01] Unreachable sequential elements
within sequential
depth of 4
top.m1.a1 10 <misc_func.v , 35>
top.m2.ad2 5 <toArray.v , 54>
Sequential Number of Sequentials
Distance
-----
0 300
1 1500
2 5000
3 120
4 35
-- 2
```

NTL_COV_ASSERT02

```
set_rule -rule NTL_COV_ASSERT02 -element top.m1.reg1
Lint- [NTL_COV_ASSERT02] Sequential depth of top.m1.reg1 from
```

```
assertions:
5 top.m1.a1 <misc_func.v , 35>
3 top.m2.ad2 <toArray.v , 54>
```

NTL_COV_ASSERT03

```
set_rule -rule NTL_COV_ASSERT03 -depth 4
Lint- [NTL_COV_ASSERT03] Sequential elements within
sequential depth 4 to
assertions
3 top.m1.reg1 <misc_func.v , 12>
2 top.m1.reg2 <misc_func.v , 13>
8 top.m1.reg3 <misc_func.v , 14>
9 top.m1.reg4 <misc_func.v , 15>
set_rule -rule NTL_COV_ASSERT03 -maxasserts 5 -depth 4
Lint- [NTL_COV_ASSERT03] Sequential elements within
sequential depth 4
to assertions violating the maxassert limit
8 top.m1.reg3 <misc_func.v , 14>
9 top.m1.reg4 <misc_func.v , 15>
```

NTL_COV_ASSERT04

```
set_rule -rule NTL_COV_ASSERT04 -assert top.c2.rcheck -depth
3
Lint- [NTL_COV_ASSERT04] Sequential elements within
sequential depth 3 from
assertion top.c2.rcheck
1 /top/dut/arb/state[0] <design.v, 24>
2 /top/dut/arb/state[1] <design.v, 24>
2 /top/dut/fifo/iptr[3] <fifo.v, 3>
3 /top/dut/fifo/iptr[4] <fifo.v, 3>
```

NTL_COV_ASSERT05

```
set_rule -rule NTL_COV_ASSERT05
Lint- [NTL_COV_ASSERT05] Groups of assertions with identical
```

```
inputs
Group with identical inputs:
top.m1.a1
top.m2.a_1
Group with identical inputs:
top.rem.r23
top.reall.r2
```

NTL_COV_ASSERT06

```
set_rule -rule NTL_COV_ASSERT01 -depth 4 -maxinputs 10 -
assert top.m1.a1
Lint-[NTL_COV_ASSERT04] 10 inputs found in the fanin cone
with a depth of 4
or less for top.m1.a1
```

Type of Assertions

All concurrent assertions (assert, assume and cover) are supported.

11

Testbench Separate Compilation

This chapter explains how to use VCS to compile the SystemVerilog/NTB OpenVera testbench separately. It contains the following sections:

- [“Overview” on page 164](#)
- [“Use Model” on page 166](#)
- [“Usage Notes” on page 176](#)
- [“Testbench Separate Compile Flow Notes” on page 184](#)
- [“NTB OpenVera/SystemVerilog Interoperability” on page 187](#)

Overview

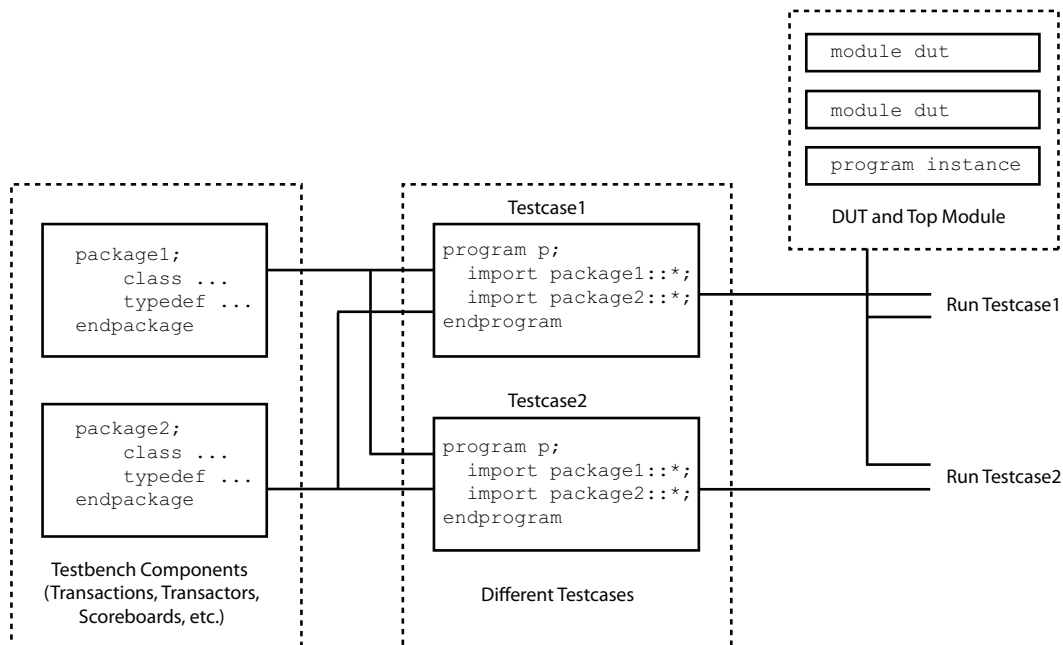
The testbench separate compilation feature accelerates the iterative process of testbench development, and reduces the disk space consumed by a large number of tests.

In a regular VCS compilation flow, a single `simv` executable contains both the Design Under Test (DUT) and testbench. Any change to source code requires VCS to analyze and regenerate the entire `simv`. Moreover, depending on how the testbench is structured, each test might require a separate executable.

Using the testbench separate compilation feature, you can now generate named shared object libraries, and at run time, dynamically link multiple shared object libraries to the `simv` containing the DUT. In other words, you do not have to regenerate the entire `simv` for every small change; instead, you can recompile only the library that has been modified. This significantly reduces the compilation time. It also reduces the disk space consumed as the testbench components can be reused, and a single copy of the DUT is sufficient for all the tests.

For example, consider a typical SystemVerilog verification environment as shown in [Figure 11-1](#). It includes a testbench, DUT, top module to link the DUT with the testbench, and various testcases.

Figure 11-1 Typical SystemVerilog Verification Environment



The testbench includes the necessary components of the verification environment such as transaction definitions, transactors, scoreboard, DesignWare Models, etc. In the testbench separate compile flow, you can implement this testbench as multiple SystemVerilog packages that are linked at runtime. These packages can contain one or more discrete testbench components such as BFMs and/or stimulus components. You must compile each package into a separate partition with the same name, which is used at runtime to dynamically load the code.

The testcases, which use the testbench components, can be constrained, random, or direct tests. As recommended by the VMM flow, you can implement testcases using program blocks -- a different program for each testcase. You must compile each testcase into a separate partition with a unique name even though the program block name remains the same for all testcases. This helps to distinguish each testcase at runtime while dynamically loading the code.

While running the simulation, the different program partitions can be linked to `simv` dynamically to run different testcases without recompiling the entire verification environment. You can compile only the packages that have been modified or added.

Use Model

The SystemVerilog testbench separate compilation flow mainly consists of three steps -- code analysis, library generation, and runtime linking.

The following is an example of how you can use this flow:

1. Decide how to organize your testbench into packages/partitions.
2. Compile the testbench components for a given package.
 - Analyze the code with `vlogan`. See the [“Specifying Logical Libraries”](#) and [“Analyzing the Code”](#) sections for more information.
 - Generate shared object libraries for each package with VCS. See the [“Generating Shared Object Library”](#) section for more information.
 - Repeat these steps for each package.
3. Compile the testcases.
 - Analyze the code with `vlogan` and specify a unique partition name for each testcase. See the [“Specifying Logical Libraries”](#) and [“Analyzing the Code”](#) sections for more information.
 - Generate shared object libraries for different program partitions with VCS. See the [“Generating Shared Object Library”](#) section for more information.

4. Generate a program shell file from the program block. This shell file is used for compiling the main `simv` in order to provide a hook for dynamic linking of the testbench at runtime. The testcases must have the same program block name so that only one shell file is required to load the testcases. See the [“Generating Shell File”](#) section for more information.
5. Generate `simv`, which includes the DUT, program shell file, and top module to link the DUT with the program shell file. See the [“Generating simv”](#) section for more information.
6. Run simulations using the generated `simv`, specifying the partitions that need to be loaded. See the [“Linking Partitions Dynamically at Runtime”](#) section for more information.
7. Repeat steps 2 or 3 followed by step 6 to further develop the testbench or testcases, and later in the testing cycle, to run your tests and regressions.

Note:

The testbench separate compilation feature also supports OpenVera/SystemVerilog interoperability with a similar use model.

Specifying Logical Libraries

To use the testbench separate compilation feature effectively, you must specify the logical libraries in the `synopsys_sim.setup` file. This enables VCS to partition the code and avoid unnecessary recompilation.

The `synopsys_sim.setup` file maps library identifiers (logical names) to the physical location of the library in the UNIX file system (physical name). Refer to the *VCS MX/VCS MXi User Guide* for more information on the `synopsys_sim.setup` file and the VCS search order.

The following is a sample `synopsys_sim.setup` file:

```
WORK > DEFAULT
COMPLEX_BFM : ../lib/opp_bfm_lib
USEFUL_UTILS : ../lib/utilities.lib
TEST37 : ../lib/test37.lib
DEFAULT : ../lib/default.lib
```

If you do not use the `synopsys_sim.setup` file, VCS places the entire testbench in the default `WORK` partition as a single logical library.

Analyzing the Code

During analysis, VCS checks the syntax and analyzes all the dependencies.

The command to analyze the code is:

```
% vlogan [vlogan_options] -sep_cmp -sverilog
    -partition partition_name
    [-work logic_library]
    [-liblist logic_lib1+logic_lib2+...]
    [-timescale scale]
    Verilog_filelist
```

Here:

`-partition partition_name`

Specify a unique partition name for each testcase program block. This helps in compiling multiple testcases separately and dynamically linking them during simulation. You can compile each partition in a different library.

You do not need to use this option while generating a package partition. `vlogan` automatically generates a separate partition for each package. The partition has the same name as the corresponding package.

The minimum granularity for a NTB-SystemVerilog partition is a SystemVerilog package or program. All the classes must be defined in either a package or a program.

Multiple SystemVerilog partitions can be placed in the same logical library by invoking `vlogan` multiple times. Ensure that you do not have too many partitions as maintaining and tracking them can be tough. Moreover, if the partitions are very small (for example, each package containing only one class), loading them will take a lot of time even though the compile time will be less. Ideally, place the code that does not change in a single package, and the code that is edited or substituted frequently (for example, program block level code) in separately named partitions.

`-work logic_library`

Specify a logical library that is mentioned in the `synopsys_sim.setup` file. `vlogan` determines the corresponding physical library from the `synopsys_sim.setup` file, and dumps the intermediate data to that directory.

`-liblist logic_lib1+logic_lib2+...`

Specifies the logical libraries to be searched, and alters the logical library search order. This option overrides the libraries and the search order specified in the `synopsys_sim.setup` file.

If you do not use this option, `vlogan` searches for any unresolved packages in the libraries specified in the `synopsys_sim.setup` file.

Examples

- If `package1`, `package2`, and `package3` are defined in the `package1.sv`, `package2.sv`, and `package3.sv` files respectively, the following command analyzes these packages and generates partitions for `package1`, `package2`, and `package3` in the WORK logical library.

```
% vlogan -sep_cmp -sverilog package1.sv package2.sv  
package3.sv
```

- If the `program_test1.sv` file contains the `tb_main` program block, the following command analyzes the code and generates a program partition, `TEST_A`.

```
% vlogan -sep_cmp -sverilog -partition TEST_A  
program_test1.sv
```

- Consider the following files:

```
// FILE test1.sv  
package P1;  
    ...  
endpackage  
  
// FILE test2.sv  
package P1;  
    ...  
endpackage  
  
// FILE test3.sv  
package P2;  
    ...  
endpackage  
  
// FILE test4.sv  
program M1;  
    import P1::*;  
    import P2::*;  
    ...  
endprogram
```

The `synopsys_sim.setup` is:

```
// FILE synopsys_sim.setup
WORK> DEFAULT
DEFAULT : work
L1 : ./lib1
L2 : ./lib2
L3 : ./lib3
L4 : ./lib4
```

The packages can be analyzed using the following commands:

```
% vlogan -sep_cmp test1.sv -work L1
% vlogan -sep_cmp test2.sv -work L2
% vlogan -sep_cmp test3.sv -work L3
```

```
% vlogan -sep_cmp test4.sv
Package P1 is picked from library L1 while package P2 is picked
from library L3.
```

```
% vlogan -sverilog test4.sv -work L4 -liblist L2
Package P1 is picked from library L2 while search for package P2
fails.
```

```
% vlogan -sverilog test4.sv -work L4 -liblist L2 -liblist
L1 -liblist L3
Package P1 is picked from library L2 and package P2 is picked
from library L3.
```

```
% vlogan -sverilog test4.sv -work L4 -liblist L2+L1+L3
Package P1 is picked from library L2 and package P2 is picked
from library L3.
```

Note:

Synopsys recommends that you specify the `-ntb_opts rvm` option only to the lowermost partition.

For example, consider the following code:

```
//package1.sv
package package1;
    `include "vmm.sv"
    ...
endpackage

//package2.sv
package package2;
    import package1::*
    ...
endpackage
```

The recommended analysis commands for this example are:

```
//code analysis for partition package1
% vlogan -sep_cmp -sverilog -ntb_opts rvm package1.sv ...

//code analysis for partition package2
% vlogan -sep_cmp -sverilog package2.sv ...
```

Generating Shared Object Library

VCS creates a shared object library for partitions during library generation. The command to generate the shared object library is:

```
% vcs -sep_cmp [logic_library.]partition_name
               [standard_library_generation_options]
```

If you specify the `logic_library`, VCS generates a shared object library for the partition specified in that logical library. If you do not specify the `logic_library`, VCS uses the specified partition located in the `WORK` library.

If you modify only one package/testcase, you need to analyze the code and generate the shared library only for that package/testcase.

For example, the following commands generate shared object libraries for package partitions and program partition, TEST_A.

```
% vcs -sep_cmp package1
% vcs -sep_cmp package2
% vcs -sep_cmp package3
% vcs -sep_cmp TEST_A
```

If you modify the code in `tb_main`, issue the following commands:

```
% vlogan -sep_cmp -sverilog -partition TEST_A program_test1.sv
% vcs -sep_cmp TEST_A
```

If you modify one of the packages in the `package2.sv` file, issue the following commands:

```
% vlogan -sep_cmp -sverilog package2.sv
% vcs -sep_cmp package2
```

Note that VCS generates the shared library files in the physical UNIX directory specified in the `synopsys_sim.setup` file. You can later move these UNIX directories to another location, provided the internal directory structure is unaltered and the new location is specified in the `synopsys_sim.setup` file.

Generating Shell File

For the program block, use VCS to generate a SystemVerilog shell file with the `-ntb_option genShellOnly` option:

```
% vcs -sep_cmp -ntb_opts genShellOnly partition_name
```

The generated shell file is named *shell_file_name.svshell*, where *shell_file_name* is the same as program block name. For example, the following command generates the shell file for the `tb_main` program block, and is named `tb_main.svshell`:

```
% vcs -sep_cmp -ntb_opts genShellOnly TEST_A
```

Use this shell file as a testbench placeholder while compiling the `simv` containing the DUT. It must be treated as if it were the program block inside the `simv`.

In SystemVerilog, the program block name must be the same across all tests. VCS uses the `-partition` option to identify different tests in `vlogan`. The named partition is then used in library generation and at runtime to identify the specific test to use.

You need to generate the shell file only once with any one of the program partitions. However, if you change the arguments of the program, you must regenerate the shell file.

Generating simv

Use the following command to compile the DUT, top module, and the generated program shell file, and generate `simv`:

```
% vcs -sep_cmp -ntb_vl -sverilog \  
    [standard_compile_options]\  
    program_shell_file\  
    Verilog_filelist
```

For example, to compile `dut.sv`, `top.sv`, and `tb_main.svshell`, and generate `simv`, issue the following command:

```
% vcs -sep_cmp -ntb_vl -sverilog \  
    tb_main.svshell \  
    dut.sv top.sv
```

Linking Partitions Dynamically at Runtime

Use the following command to dynamically link and run the testcase associated with the partition specified during analysis.

```
% simv -sep_cmp=partition_name[+partition_name2+...]
```

Here, `partition_name` is one of the program partitions. Before runtime linking, ensure that the `simv` is compiled with the shell file.

During runtime linking, VCS performs a dependency check to ensure that partitions are correctly compiled and generated. It also enforces timescale consistency.

For example, to run the simulation with the testcase implemented in the `program_test1.sv` file, issue the following command:

```
% simv -sep_cmp=TEST_A
```

Usage Notes

This section describes a few commonly used testbench separate compile flow scenarios.

DesignWare VIP

DesignWare VIPs are supported with the testbench separate compile flow. You can compile VIPs in one partition and dynamically link them during simulation. VIPs do not need to be recompiled for each testcase.

For this release, you must compile all the VIPs used in the testbench environment in one partition. The partition name for VIPs is `SYNOPSYS_VIP_PACKAGE`.

The VIPs are provided through `.pkg` files for SystemVerilog testbench. Therefore, all the `.pkg` files must be analyzed together and `.pkg` files must not be given during program analysis.

The testbench separate compilation feature supports two VIP use models based on how Virtual ports are dealt in VIPs:

- [“Compiling VIPs with Interface Files”](#)
- [“Compiling VIPs without Interface Files”](#)

Virtual ports (the signals' sizes are not defined in Virtual ports) are used in some VIP modules to enable the reuse of VIPs in different environments. If VIPs modules are compiled separately, VCS will not have information on the width of each virtual port signal. By default, VCS assumes the maximum signal width to be 32. However, the actual signal connected to the port signal can have a larger width (for example, 1024 for a 1024 width AHB system). VCS provides the `-sc_bind_width` option to set the default signal width for Virtual ports.

If the VIP modules are compiled with the interfaces signals, VCS can detect the port widths from the interfaces, and the `-sc_bind_width` option is not required.

Compiling VIPs with Interface Files

In this use model, VIP models are compiled with interface files.

For example:

```
//DW AHB VIP analysis with interfaces
vlogan -sep_cmp -sverilog \
  +define+SYNOPSISYS_SV -ntb_define NTB \
  -ntb_opts rvm \
  -ntb_opts use_sigprop \
  -ntb_opts dtm \
  -ntb_opts dw_vip \
  -ntb_vipext .ov \
  +define+NTB \
  ../../include/svtb/AhbMasterInterface.svi \
  ../../include/svtb/AhbSlaveInterface.svi \
  ../../include/svtb/AhbMonitorInterface.svi \
  ../../include/svtb/AhbBusInterface.svi \
  ../../examples/ahb_rvm_sys/svtb/extra_itf.sv \
  +pkgdir+../../include/svtb \
  AhbSlave_rvm.pkg \
  AhbMonitor_rvm.pkg \
  AhbMaster_rvm.pkg \
  AhbBus_rvm.pkg \
  -ntb_incdir ../../include/vera \
  .....

//DW AHB VIP library generation
vcs \
  -sep_cmp -sverilog \
  -ntb_opts rvm \
  -ntb_opts use_sigprop \
  -ntb_opts dtm \
  -ntb_opts dw_vip \
  -ntb_vipext .ov \
  SYNOPSISYS_VIP_PACKAGE

//SV Program block
program automatic AhbSystemTest(...);
  // import DW VIP modules
  import SYNOPSISYS_VIP_PACKAGE::*;
  ...
```

Compiling VIPs without Interface Files

In this use model, VIP modules are compiled without interface files. In this case, if the width of the actual signal connected to the port is larger than 32, VCS errors out. To overcome this problem, add the following option during library generation:

```
-sc_bind_width=actual_signal_size
```

Synopsys recommends you to compile VIP models with interface files.

If you want to create several analyzed copies of VIP packages (for example, with different `-sc_bind_width` options), then you must use the `-liblist` option during program analysis and elaboration to specify the library to pick up the VIP package.

XMR in Testbench-to-DUT or Top-Module Tasks

The SystemVerilog LRM does not permit cross module references (XMR) to DUT or top-module tasks. However, in some cases, XMR to DUT or top module tasks is required in testbench components. For example, it is required for back door access to the memory of the DUT through tasks, etc.

In such cases, you can create DPIs as the bridge between the testbench component and DUT. In the DPIs, the DUT tasks can be called through export DPI.

The following example shows how a testbench component calls a DUT task, `dut_task`.

```
//dut.sv
module dut;
    //declare export DPI for the DUT task
    export "DPI-C" task dut_task;
    task dut_task(input int i);
        $display("DUT:dut_task i is %d",i);
    endtask
endmodule

//xmr_dpi.c
#include "svdpi.h"
#include "vcsuser.h"
//dut task to be called
extern void dut_task(int i);
void xmr_dut_task(char *task_scope,int i) {
    svSetScope(svGetScopeFromName(task_scope));
    dut_task(i);
}

//package.sv
package pkg;
    `define XMR_PATH "top.dut_inst"
    //import DPI
    import "DPI-C" task xmr_dut_task(string task_scope, int i);
    class cl_c;
        task cl_task();
            $display("Called in package");
            xmr_dut_task(`XMR_PATH,10);
        endtask
    endclass
endpackage
```

Parameterized Programs

Parameterized programs are supported if the parameter values are not overridden after the code generation of the program block.

For example, consider the following code and script:

```
//prog.sv
program prog #(int a = 10,
string test_name="TEST");
    initial begin
        $display(test_name,," a is %0d",a);
    end
endprogram
```

```
//top1.sv
module top;
    prog #(20,"TEST1") prog_inst();
endmodule
```

```
//top2.sv
module top;
    prog #(30,"TEST2") prog_inst();
endmodule
```

```
//script
vlogan -sep_cmp -sverilog test1.sv \
-partition test1
vcs -sep_cmp test1
vcs -sep_cmp test1 -ntb_opts genShellOnly
vcs -sep_cmp -ntb_vl -sverilog \
prog.svshell top1.sv
./simv -sep_cmp=test1

vcs -sep_cmp -ntb_vl -sverilog \
prog.svshell top2.sv
./simv -sep_cmp=test1
```

These two simulations incorrectly produce the same result:

```
TEST  a is 10
VCS Simulation Report
```

The workaround for this limitation is to generate multiple partitions with different sets of parameter values depending on how the program is instantiated. You can use the `-pvalue` or `-parameters` option during code generation to set parameter values. Refer to the VCS/VCSi User Guide for more information.

For the above example, compiling the following code will produce correct results.

```
//compile the program block to different partitions
vlogan -sep_cmp -sverilog test1.sv \
-partition test1_20_TEST1
vlogan -sep_cmp -sverilog test1.sv \
-partition test1_30_TEST2

//set parameter values with -pvalue during code generation
vcs -sep_cmp -pvalue+prog.a=20 \
-pvalue+prog.test_name='\"TEST1\"' test1_20_TEST1
vcs -sep_cmp -pvalue+prog.a=30 \
-pvalue+prog.test_name='\"TEST2\"' test1_30_TEST2

vcs -sep_cmp test1_20_TEST1 -ntb_opts genShellOnly

//generate and run simulation with different partitions
vcs -sep_cmp -ntb_vl -sverilog \
prog.svshell top1.sv
./simv -sep_cmp=test1_20_TEST1
vcs -sep_cmp -ntb_vl -sverilog \
prog.svshell top2.sv
./simv -sep_cmp=test1_30_TEST2
```

In this script, the program shell file is generated from any one of the program partition and shared by all other partitions. This works well if the parameter is not used to specify the size of program port arguments.

In case the parameter is also used to specify the program port size, you must also generate different shell files from different partitions and use the matched shell file for generating `simv`.

Parallel Compilation

Typically, a number of testcases need to be run to verify the correctness of SoC/ASIC designs. Using the separate compilation feature, you can compile all these testcase partitions in parallel after compiling the necessary package partitions.

You can generate `simv` even before compiling all the testcases. After generating the necessary program shell files, you can generate `simv` from the DUT/top modules and program shell files.

Random Stability

The testbench random stability and repeatability are accomplished through thread and instance based random number generators and a hierarchical seeding mechanism. Every new thread and object containing random data has its own separate random number generator, where the starting seed is from its parent's random number generator. In situations where the thread execution order is free to change (for example, a fork join of two or more threads with nonblocking code), different optimization strategies can impact random number repeatability. The single compile and testbench separate compile flow optimizations can cause nonblocking thread execution order to be different, potentially impacting random number repeatability. Therefore, random data repeatability is not guaranteed when a testcase is moved between the two flows.

Testbench Separate Compile Flow Notes

Consider the following guidelines to enable the testbench separate compile flow:

- XMRs (cross-module references)
 - XMR-to-DUT or top-module signals in the program block are not permitted. Alternatively, you can use the program port signals and connect those signals to the desired DUT or top-module signals during program instantiation.
 - XMR-to-DUT or top-module tasks in program blocks are not permitted. See [“XMR in Testbench-to-DUT or Top-Module Tasks”](#) for information on the use model.
 - XMRs to signals of interface instances defined in the DUT or top module are not permitted.
 - XMRs to interface instances are permitted in the testbench separate compile flow.
 - The following code shows what is and is not permitted:

```
interface intf();
    logic x;
endinterface

program p1;
    virtual intf v_intf;
    initial begin
        top.intf_inst.x=1'b1;//Not permitted
        intf.x=1'b1;//Not permitted
        v_intf = top.intf_inst;
        v_intf.x=1'b1;//Correct
    end
endprogram

module top;
    intf intf_inst();
```

```

        p1 p1_inst();
    endmodule

```

- All testbench classes must be inside a package or program.
- `$root` and `$unit` calls are not permitted from the testbench.
- Modport is supported only when the program instance and the formal argument for the port definition are identical. Other uses of modport are not permitted. The argument list calling the stub must be identical to what the program block is expecting. For example:

```

program prog(intf.mp intf1);
...
endprogram

module top;
    intf intf_inst(..);
    prog prog_inst(intf_inst); //Not permitted
    prog prog_inst(intf_inst.mp); //Correct
endmodule

```

- Testcases must be implemented with program block(s) instead of module(s). Testbench separate compile for module is not yet supported.
- Shared packages between the testbench and DUT are not permitted.
- Parameterized programs are supported but the parameters' values cannot be overridden. See [“Parameterized Programs”](#) for information on how to use different parameter values with parameterized programs.
- When you use `-partition` during program compilation, you cannot have multiple program blocks in one file.
- Interface methods are not yet supported.
- Direction coercion for testbench ports is not permitted. Testbench ports must have directions such that no port direction is coerced.
- Aspect-oriented extensions are only supported in program partitions.

- Compilation options must be the same across all partitions and the DUT. For example, if you use `-debug_all` for one partition, you must use it for all partitions.
- Parameterized interfaces are supported, but the parameters' values are not permitted to be overridden in the testbench. In the DUT, parameterized interface values can be overridden. For example:

```
//interface
interface ifc #(int SIZE=10) (input clk);
    bit[SIZE-1:0] c;
...
//program
program p1;
    virtual ifc#(20) v_ifc;//Not permitted
    virtual ifc v_ifc;//Correct
    initial begin
        v_ifc = top.sv_ifc;
    ...
```

- An interface cannot be used only in the DUT. As an alternative, declare a virtual interface variable of that interface in the program block.
- All interfaces used must be included in both the DUT and the testbench even if the package or partition uses only some of the interfaces.
- Only mixed-language designs with a Verilog top are currently supported.
- Function calls in constraints are not permitted.
- You must not use the `-Mdir` option for testbench code generation because all the testbench codes are compiled and saved to logical library directories. You can use the `-Mdir` option for generating `simv`.
- You must use the same set of PLI tab files and C or object files for both the testbench and the DUT.

- DVE interactive debug does not support classes that have later been manipulated with aspect-oriented extensions.
- Waveform dump calls cannot be called from the testbench separate compile flow. This includes `$dumpports` and `$dumpvars`.

NTB OpenVera/SystemVerilog Interoperability

The testbench separate compilation feature supports OpenVera and SystemVerilog interoperability. Refer to the VCS/VCSi User Guide for more information on OpenVera/SystemVerilog interoperability.

Analyzing the Code

For analyzing OpenVera code, issue the following command:

```
vlogan -sep_cmp -ntb -partition partition_name
      [-work logic_library]
      [-liblist logic_lib1+logic_lib2+...]
      [ -timescale scale ]
      [standard_compile_options]
      Verilog_filelist
```

For analyzing SystemVerilog code, issue the following command:

```
vlogan -sep_cmp -sverilog
      [ -partition partition_name]
      [-work logic_library]
      [-liblist logic_lib1+logic_lib2+...]
      [ -timescale scale ]
      [standard_compile_options]
      Verilog_filelist
```

Note:

The files for each `vlogan` command must be either in OpenVera or SystemVerilog, but not both on the same command line.

Examples

- To analyze the OpenVera code defined in `class_ov.svr` and generate an OpenVera partition, issue the following command:

```
% vlogan -ntb -sep_cmp -ntb_opts interop class_ov.svr  
-partition OpenVera
```

- To analyze the SystemVerilog code of a SystemVerilog package defined in `package.sv`, which may use OpenVera class defined in other OpenVera codes, issue the following command:

```
% vlogan -sverilog -sep_cmp -ntb_opts interop package.sv
```

- To analyze the SystemVerilog code of a SystemVerilog program defined in `prog.sv` and generate a partition `p1` for this code, issue the following command. The program block may use other OpenVera classes.

```
% vlogan -sverilog -sep_cmp -ntb_opts interop prog.sv  
-partition p1
```

Generating Shared Object Library

Use the following command to generate shared object library for the specified partition:

```
vcs -sep_cmp [logic_library.]partition_name  
[standard_library_generation_options]
```

For example, to generate shared library for the OpenVera, pack, and p1 partitions respectively, issue the following command:

```
% vcs -sep_cmp OpenVera  
% vcs -sep_cmp pack  
% vcs -sep_cmp p1
```

Generating Shell File

The command syntax is explained in the section, [“Generating Shell File” on page 174](#).

For example, to generate the shell file for the program block of partition p1, issue the following command:

```
% vcs -sep_cmp -ntb_opts genShellOnly p1
```

Generating simv

The command syntax is explained in the section, [“Generating simv” on page 175](#).

For example, to compile the top module with the program shell, and generate `simv`, issue the following command:

```
% vcs -sep_cmp -sverilog -ntb_vl p1.svshell top.sv
```

Linking Partitions Dynamically at Runtime

The command syntax and examples are explained in the section, [“Linking Partitions Dynamically at Runtime” on page 175](#).

Testbench Separate Compile Flow Notes

Consider the following guideline to enable the testbench separate compile flow:

- All OpenVera code must be in one partition.

12

Unified Exclusion from Coverage Analysis

This chapter describes the exclusion of items from the coverage database. When an item is excluded, it is no longer counted when computing coverage scores. Excluding items helps you focus on only the uncovered items that matter for your verification task.

You can exclude items interactively using DVE, and your exclusions can be saved to files that can be reloaded into a later DVE session, into the Unified Report Generator (URG), or into the Unified Coverage API (UCAPI). This chapter discusses how to exclude items and how to use the resulting "exclude files" with these tools.

This chapter contains the following sections:

- [“Using DVE Coverage with Unified Exclusion”](#)
- [“Using URG with Unified Exclusion”](#)
- [“Editing Exclude Files”](#)

- [“Using UCAPI with Unified Exclusion”](#)

Using DVE Coverage with Unified Exclusion

This section contains the following topics:

- [“Coverage Exclusion with DVE”](#)
- [“Excluding Covergroups”](#)

Coverage Exclusion with DVE

In DVE, there are two steps required to exclude coverage items:

1. Marking the items to be excluded.
2. Recalculating the coverage scores.

Exclusion Modes

There are two modes available for exclusion:

Default mode - In Default mode, you can exclude anything, whether it is covered or not. Anything specified in the exclude file, including covergroups, covergroup instances, coverpoints, crosses, bins, modules, module instances, or any other coverable object will be excluded completely.

Strict mode - In Strict mode, you can exclude only uncovered objects. To enable Strict mode, you can use the DVE command line option `-excl_strict`, for example,

```
dve -covdir simv.cm -excl_strict
```

or choose the option "Do not allow Covered Objects to be excluded" in the DVE Application Preferences window. Using the preferences menu option will require that you reload the database to enable Strict mode.

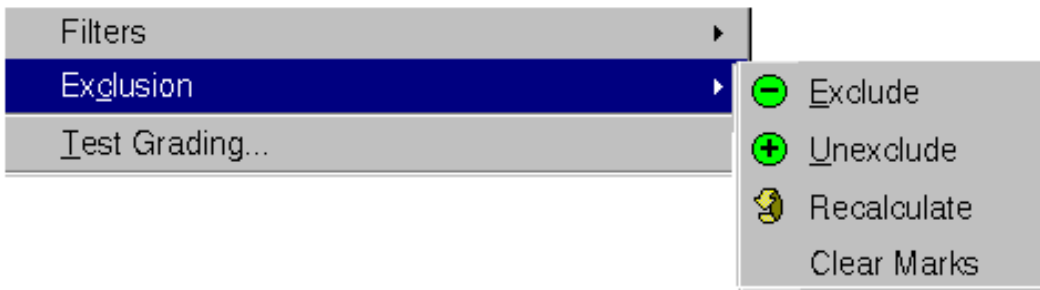
In Strict mode, containers, such as vector signals in toggle coverage, conditions, control statement in branch coverage, and FSMs, may be excluded, but only the uncovered objects within those containers will actually be excluded. The covered objects will be marked as "Attempted", meaning that an attempt was made to exclude them, but they were already covered.

After you have excluded objects in DVE, you can save those exclusions to a file. The mode you are in (Default or Strict) when you create an exclude file is saved with the rest of that file's contents. If you load an exclude file in a different mode than the mode in which it was created, a warning message will be printed, but the mode in which you are currently in will be in effect.

For example, if DVE is in Strict mode and you load an exclude file created in Default mode, you will get a warning, and only the uncovered objects specified in the exclude file will actually be excluded, since DVE is in Strict mode. In addition, a file "attempts.log" will be created listing all the covered objects that were in the exclude file but which could not be excluded.

Exclusion Commands

The **Edit > Exclusion** menu commands are shown below:.



- **Exclude** - Marks the selected item for exclusion.
- **Unexclude** - Marks the selected excluded item for reinclusion.
- **Recalculate** - Reruns coverage calculations and displays results to include results of pending exclusion/inclusion items.
- **Clear Marks** - Reverts all pending exclusion/inclusion items to their original state.




Context Sensitive Menu

Right-click in the Summary window to display the context-sensitive menu (CSM) and Exclude or include items:



Exclusion Toolbar

The exclusion toolbar has three buttons:

-  Exclude Selected
-  Include Selected
-  Recalculate


Excluding Items

You can mark the items by first selecting them and then exclude or include them either through the CSM or the main menus. You can select and mark multiple items simultaneously.

When all of the selected items are excluded, only the Include menus are enabled. When all of the selected items are included, then only the Exclude menus are enabled. If there is a mix of included and excluded items, both menus are enabled.

Recalculating Coverage Scores

Once items are marked, the system needs to recalculate the

coverage scores. Select  from the toolbar or **Edit > Exclusion > Recalculate** to view results with the excluded/included results.

The Recalculate command is disabled until an item is marked. After recalculation, it is once again disabled.

To save time, it is recommended that you mark multiple items before recalculating rather than recalculating after marking each item.

Exclusion States

Every coverable item has an exclusion state associated with it. The state can be:

- Included (default) - All items are included unless explicitly excluded.
- Excluded - Excluded items are those items which have been ignored for the purpose of calculating coverage metrics.

- Partially Excluded - Regions or items that contain multiple coverable objects will be marked "Partially Excluded" if some but not all of the contained objects are excluded. The objects that may be marked Partially Excluded include covergroups, covergroup instances, coverpoints, crosses, and code coverage objects such as signal vectors in toggle coverage, conditions, vectors, branch statements, or FSMs.
- Attempted - The Attempted state is only used in Strict mode. If an object or region is marked Attempted, it means that you, or an exclude file you loaded, tried to exclude items that were marked covered. You can only see the Attempted mark in the detailed view, where items marked as Attempted are shown in the "Attempted" column.

Exclusion State Markers

Marker icons in DVE are circles with symbols inside. You can view the exclusion markers in the Detail window and Source window.

- Pending items marked to be excluded are indicated by a minus sign in a green circle. Pending items marked to be included are indicated as a plus sign in a green circle. The following are the inclusion, exclusion, and partial exclusion icons:



- Excluded items are indicated with an x in a red circle.



- When a line is partially marked or excluded with the Partial Exclusion symbol:



- The following is the exclusion marker for “Attempted” state:



Tooltips on these symbols indicate the statement or metric that is marked.

List view panes (summary and detail) have an additional column showing the exclusion markers.

For additional clarity, excluded items have their coverage bars grayed out and values removed. The bars will continue to display as they still impart useful information.

Test: MergedTest		Show: Design Hierarchy		Hierarchical coverage	
Name	Score	Line	Toggle	FSM	Condition
test_jukebox	45.97%	66.78%	49.79%	34.83%	32.47%
cd1	84.21%	100.00%	68.42%		
fifo1	80.78%	88.24%	73.33%		
jb1	71.43%	96.00%	46.88%	100.00%	42.86%
st0					
st1	64.68%	89.09%	55.10%	70.59%	43.96%
st2	27.77%	40.74%	41.89%	11.76%	16.67%
coin1	24.55%	40.74%	29.03%	11.76%	16.67%
kp1					
st3	53.19%	76.36%	52.38%	41.18%	42.86%
coin1	51.97%	71.60%	58.06%	41.18%	37.04%
kp1	62.53%	89.66%	46.58%		51.35%
st4	20.54%	37.27%	30.61%	0.00%	14.29%

Container Exclusion State Markers

The following DVE exclusion markers indicate the exclusion state of items within a container, without the need to open the container to inspect the exclusion state of the individual items in the container.

A *container* is a structural element that contains other data, for example, a module or entity.

Note:

In this discussion of “Attempted” state markers, be aware that DVE displays the “Attempted” state marker only in *strict* mode, that is, when you use the `-excl_strict` switch. When DVE marks a container as “Attempted” in strict mode, this means that the item was already covered and cannot be excluded.

- Partially Excluded—When some, but not all, of the items in a container are excluded, the container is marked with the “Partially Excluded” state marker:



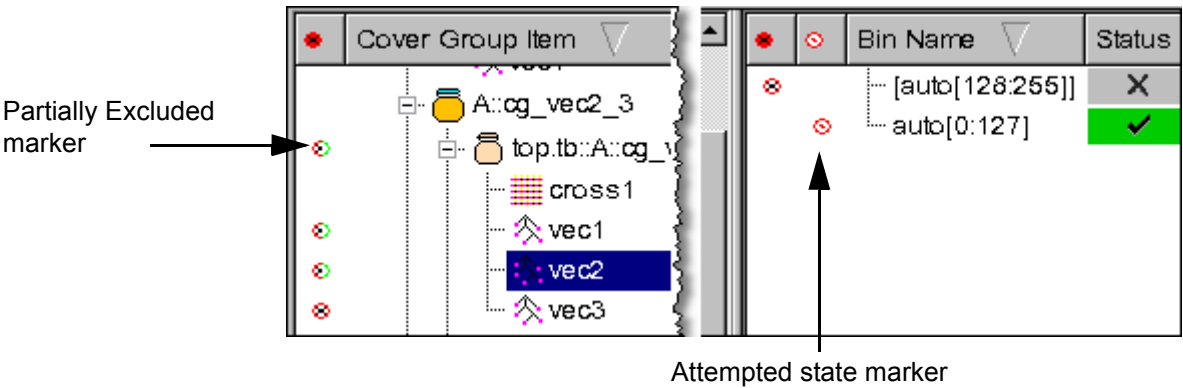
The following illustration shows how the Partially Excluded marker appears in the DVE GUI in relation to the container:

Partially Excluded state marker

- Partially Excluded and Attempted—When some of the items in a container are excluded, and some have been marked as attempted, the container is marked with the “Partially Excluded” and “Attempted” state markers:



The following illustration shows how the Partially Excluded and Attempted state markers appear in the DVE GUI in relation to the container and its contents:



- Attempted—When DVE has attempted to exclude some or all covered items in a container, the container is marked with the “Attempted” state marker:

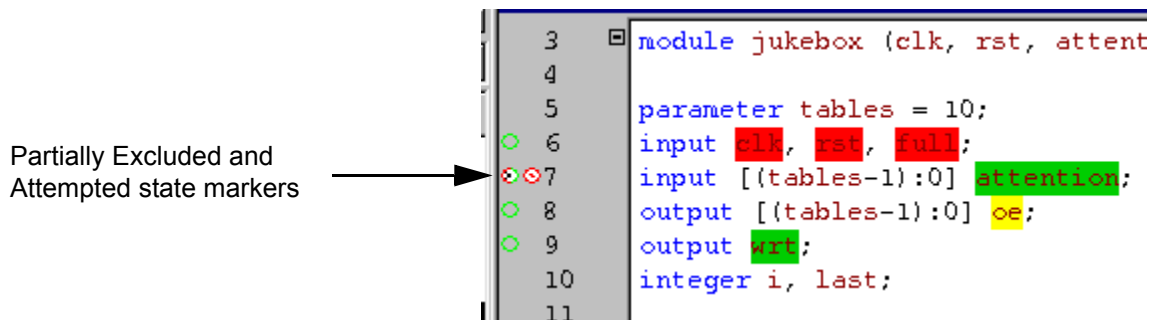


The following illustration shows how the Attempted state marker appears in the DVE GUI in relation to the container:

Variable	Type	Coverage	Display	
clk	SVBit	100.00%		
dem_rx_active	Net			
mod_demod_off	Logic	100.00%		

Attempted state marker

The Source window also displays container exclusion information:



Setting Preferences

You can set your exclusion preferences in the Application Preferences window.

To set exclusion preferences

1. Click **Edit > Preferences**.

The Application Preferences dialog appears.

2. 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.

3. Click **Apply** to save the settings or **OK** to close the Applications Preferences window.

Interactive Marking within Detail Views

Within the *source* windows, items can be marked in various ways:

- Menus - Select the item and use the CSM, Edit menu, or toolbar in the standard fashion.
- Margin - Click in the left hand column of the margin to toggle the mark. Depending upon the contents of the line being marked the system will mark all related items. All excludable items are initially marked with an empty green circle. Lines that do not contain this icon do not represent excludable items.
- Lines containing multiple excludable items are marked with a special icon when the items on the line are at different exclusion states. This line is referred to as being partially excluded.

Excludable items are marked with the  .

Figure 12-1 Detail view showing markers in Source and List windows

The screenshot shows a software interface with two main windows. The top window is the Source window, displaying Verilog code for a module named `kp_fsm`. The code includes input/output declarations, register declarations, and parameter definitions. The bottom window is the List window, which displays a table of variables and their coverage data.

Source Window Code:

```

3 module kp_fsm (trki, dski, press, clk, rst, oe, go, kp_hold, trko, dsko);
4
5 input[7:0] trki, dski;
6 inout[7:0] trko, dsko;
7 input press, clk, rst, oe, go;
8 output kp_hold;
9
10 reg kp_hold, oe_s;
11 reg x_not, y_bot;
12 reg[7:0] trko_n, dsko_n, max_disk;
13 reg[2:0] state, n_state;
14 parameter [2:0] idle = 3'b001,
15                 hold = 3'b010,
16                 service = 3'b100;
17 tri[7:0] trko = oe_s ? trko_n : 8'bz;

```

List Window Table:

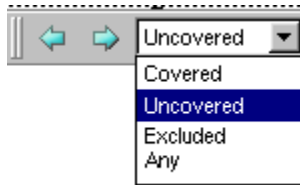
Variable	Type	Coverage	Display
trki[7:0]	Net	20.00%	12
dsko[7:0]	Net	68.75%	5
dsko_n[7:0]	Reg	0.00%	16
favrit_trk_led	Net	0.00%	2
go	Net	0.00%	2
illegal_dsk_led	Net	0.00%	2
kp_hold	Reg	0.00%	0
max_disk[7:0]	Reg	0.00%	0

Within the list views, you can toggle the exclusion markers by clicking them. The markers toggle as follows:

- Excluded -> Include Pending
- Included -> Exclude Pending
- Include Pending -> Excluded
- Exclude Pending -> Included

Exclusion Browser

Use the exclusion browser in the toolbar to navigate coverage items based on their exclusion status.



Saving and Loading Saved Exclusion States

There are two ways to save exclusion states:

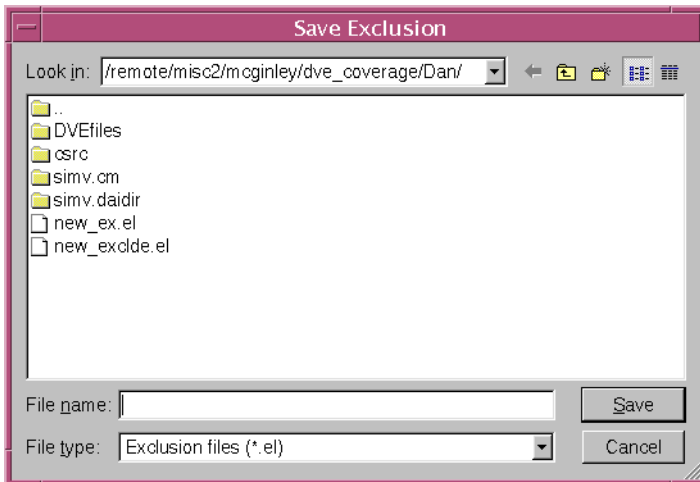
- Using the **File > Save Exclusions** command.
- Using the **Edit > Preferences > Exclusion**, then select **Save exclusion data when saving sessions**.

To save an exclusion state

1. Select **File > Save Exclusions**.

The Save Exclusion File dialog box appears.

2. Enter a name for the exclude file and enter identifying notes in the Comments box.



3. Click **OK**.

The exclusion state is saved.

To load a previously saved exclusion state

1. Select **File > Load Exclusions**.

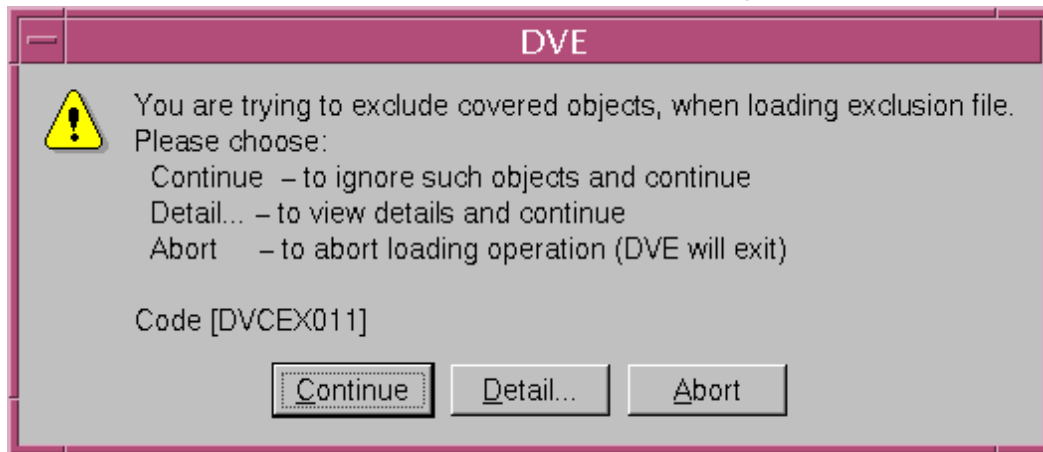
The Load Exclusion dialog box appears.

2. Select a previously saved exclude file.
3. Select **Overwrite existing exclusions** if you want only the saved exclusions to appear.
4. Click **OK**.

The exclusion state is loaded.

Attempt to Load Excluded Covered Items

In Strict mode, covered items cannot be excluded. When you load an exclude file containing covered items in Strict mode, a warning is given, and you can either select continue, which will continue without excluding the covered items, you can view details of the covered items, or abort the exclude file loading operation and exit DVE.



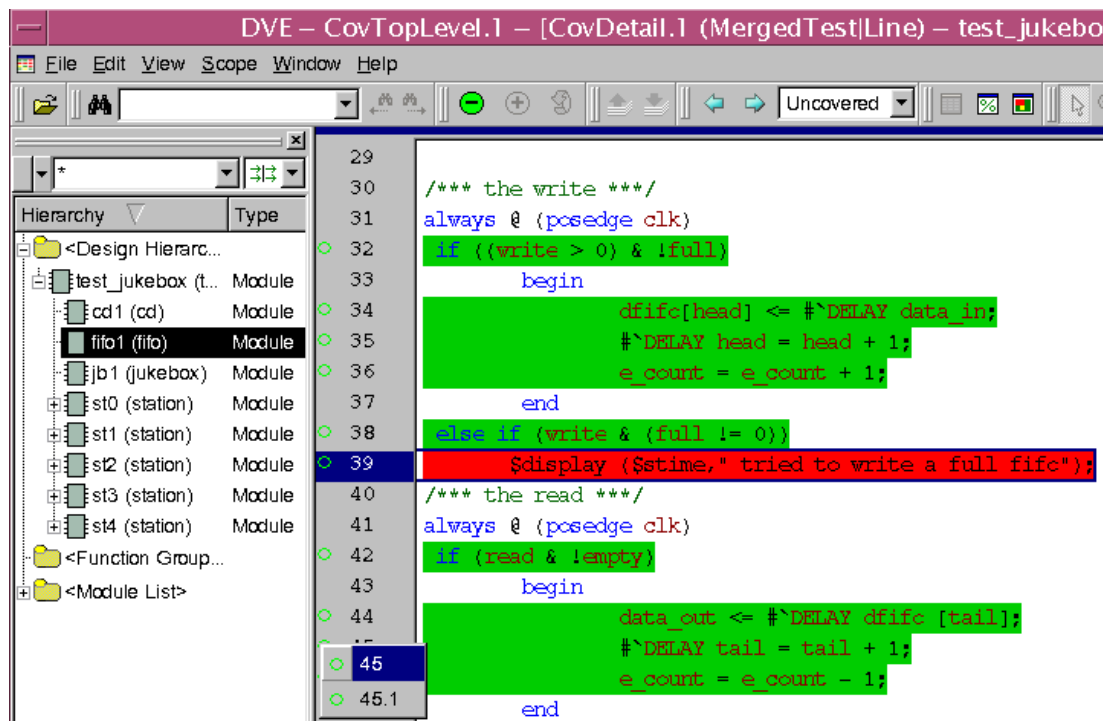
Excluding Multiple Objects in a Single Line

You can exclude multiple objects in a single line of source code. This feature is supported in all metrics.

To exclude/include an object in a line

1. Select the line of code in the Source window.
2. Right-click the green exclusion marker beside a particular line number.

A pop-up menu appears.



3. From the pop-up menu, select the desired object in the line that you want to exclude/include.

The object is excluded/included as desired.

Hierarchical Exclusion

Some views display object hierarchy as trees. When marking tree branches, all child objects within that branch will also be marked. This is true whether or not the child items are visible in the navigation pane at the time the parent object was marked.

Marking an object in one view will automatically mark the same object in all other views in which it appears.

For hierarchical excludable items in toggle, condition, branch, FSM, or assertion coverage, you can mark a branch of the hierarchy simply by selecting the root of the branch. In such a case, all items in the branch are marked, not just the root. This makes the exclusion explicit. In certain cases, child objects of an excluded hierarchy can be reincluded.

Note that excluding an instance will only exclude that specific instance itself, not other instances within it. You can use the "Exclude Tree/Unexclude Tree" options in the CSM to mark the hierarchy for exclusion or re-inclusion.

Different object types have different exclusion rules associated with them. For example, if a module is excluded none of its instances can be included.

Note:

If a module itself is excluded, coverable objects at the instance level can not be later included through DVE. However, if the module definition is not totally excluded, coverable objects at the instance level, which were excluded in the module can later be included in DVE.

Understanding and Excluding Half-Toggle Transitions

Toggle coverage monitors each net and register for any value transition from 0 to 1 and 1 to 0. Such monitoring provides an indication of the amount of activity on each element.

Without half-toggle exclusion, when you exclude an element from toggle coverage, VCS excludes both transitions, even if one of the transitions is covered. This can have an impact on the overall toggle coverage score. Consider the following example:

```

reg [1:0] p;

p = 2'b10;
#1 p = 2'b01;

```

In this simulation, `p[0]` has a 0-to-1 transition and `p[1]` has a 1-to-0 transition. Full-toggle exclusion for a signal `p[1]` gives a coverage toggle report of 50%, which can be broken down as follows:

```

p[0] 0->1 Covered
p[0] 1->0 Not covered
p[1] 0->1 Excluded (not counted)
p[1] 1->0 Excluded (not counted)

```

Half-toggle exclusion gives you better control of the granularity of the transitions to be ignored. In this example, you can exclude the `p[1]` 0->1 transition and still cover the `p[1]` 1->0 transition. The overall coverage score becomes 66.67% (2/3), as shown below:

```

p[0] 0->1 Covered
p[0] 1->0 Not covered
p[1] 0->1 Excluded (not counted)
p[1] 1->0 Covered

```

This is the BNF syntax for the half-toggle exclusion functionality:

```

toggle_spec : mod_or_inst : mod_or_inst_name {toggle_data}
mod_or_inst : MODULE | INSTANCE
toggle_data : Toggle {direction} {signal_spec}
direction: 0to1 | 1to0 | NULL
signal_spec : name [index]
index: ID r_id
r_id: COLON ID | NULL

```

This is an example of exclusion for toggle coverage:

```

INSTANCE: top
Toggle y[0]

```

```

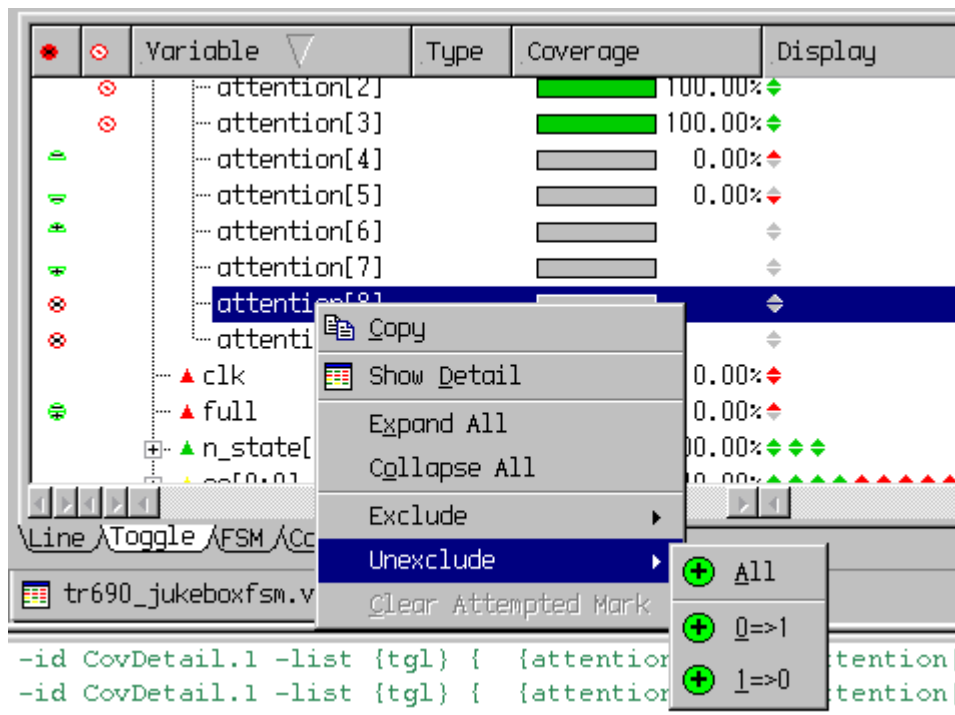
Toggle 1to0 x [0]
Toggle res[3]
Toggle res[4]
Toggle res[5]

```

In the above example, full-toggle exclusion is active for $y[0]$, $res[3]$, $res[4]$, and $res[5]$. Half-toggle exclusion is active for $x[0]$.

To exclude the toggle transition

1. Select the transition and right-click in the Summary window to display the CSM.
2. Select **Exclude** or **Unexclude** to display the transition sub-menu.
3. Select any of the following sub-menu as desired:



- **All** - Marks "Exclude/Unexclude" for the whole selected item (signal or vector).

- **0=>1** - Marks "Exclude/Unexclude" for all 0=>1 transitions in the selected item.
- **1=>0** - Marks "Exclude/Unexclude" for all 1=>0 transitions in the selected item.

Excluding Covergroups

You can exclude covergroups, cover items, or the constituent bins in DVE Coverage GUI.

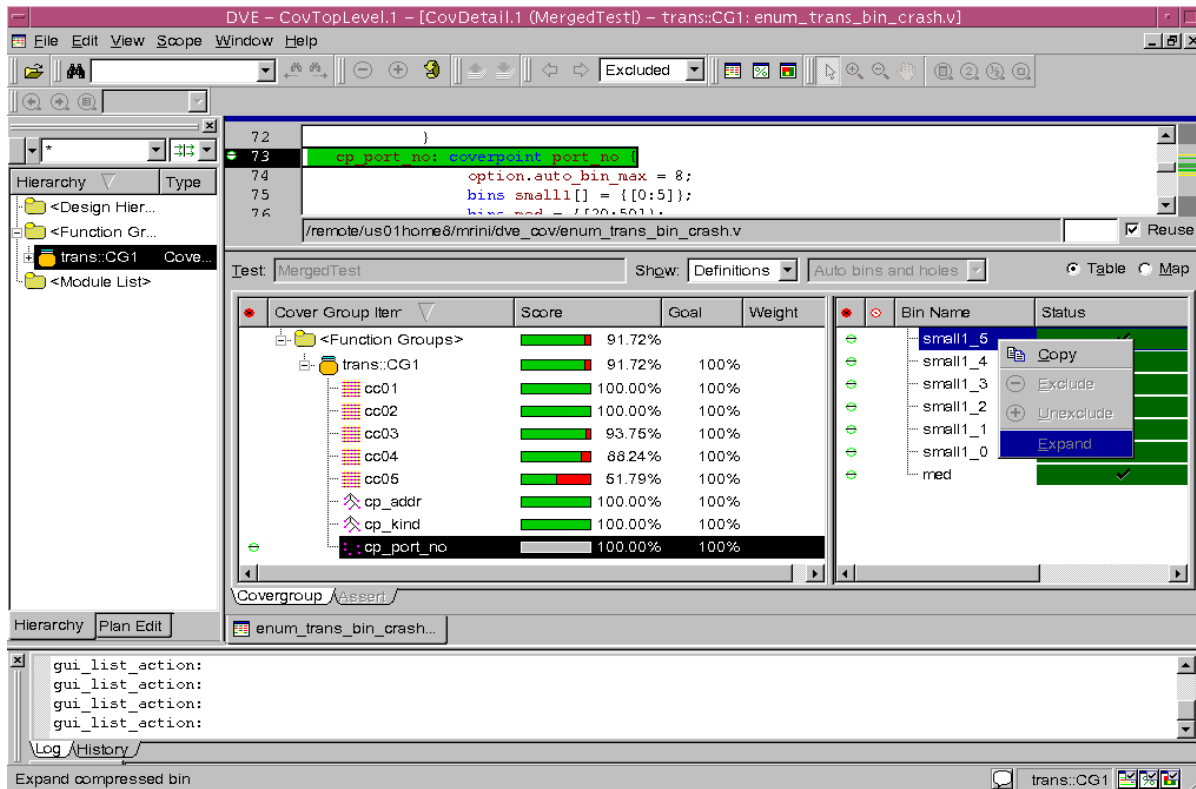
To exclude covergroups in DVE

1. Start DVE in coverage mode.
2. Load the Coverage Database.

The DVE coverage windows are populated with the coverage data.

3. Select the covergroup in the Navigation pane.

The covergroup items (coverpoints and bins) are displayed in the Summary window.



4. Modify the covergroup data with the following steps:

- Select the coverpoints/crosses under the covergroup in the Summary window and exclude/include using the CSM.
- View the source file with annotations indicating covered and uncovered lines or objects in the Annotated Source window.
- View the excluded coverpoints/crosses in Table or Map view in the Detail window.
- Save the exclusion data as elfile.
- Load/Reload saved exclude files.

For more information about how to save/load the exclusion data, see [“Saving and Loading Saved Exclusion States” on page 204](#).

Note:

- In Strict mode, you cannot exclude a cover point/cross once the score is 100%. When the score is greater than 0 and less than 100, its uncovered children are excluded and covered children are marked as attempted.

Exclusion Propagation

Coverpoint and coverpoint bin exclusion are propagated to the crosses automatically. If you exclude coverpoint or coverpoint bin, cross or cross bins containing the coverpoints are excluded as well.

Variants

Variants are different shapes of the covergroup definition due to module instance or parameterized covergroup instance.

The screenshot shows the DVE interface with the following components:

- File Hierarchy:**
 - <Design Hierarchy>
 - test_jukebox (test_jukebox) - Module
 - cd1 (cd) - Module
 - fifo1 (fifo) - Module
 - jb1 (jukebox) - Module
 - st0 (station) - Module
 - coin1 (coin_fsm) - Module
 - kp1 (kp_fsm) - Module
 - st1 (station) - Module
 - st2 (station) - Module
 - st3 (station) - Module
 - st4 (station) - Module
 - <Function Groups>
 - coin_fsm:Cvr - Covergroup definition
 - test_jukebox.st0.coin1::Cvr - Covergroup variant
 - state_cvr - Covergroup instance
 - test_jukebox.st1.coin1::Cvr - Covergroup variant
 - test_jukebox.st2.coin1::Cvr - Covergroup variant
 - test_jukebox.st3.coin1::Cvr - Covergroup variant
 - test_jukebox.st4.coin1::Cvr - Covergroup variant
 - coin_fsm:atm:packet - Covergroup definition
 - test_jukebox.fifo1.bsg_empty - Cover Property

- Code Editor:**

```

20  atnl.randomize;
21  atnl.packet.sample();
22  end
23  end
24  covergroup Cvr(ref [6:0] state,n_state);
25  option.per_instance = 1;
26  RD: coverpoint state {
27      bins LOW = { [1:6]};
28      bins MED = { [7:10]};
29      bins HIGH = { [10:41]};
30      bins WAYHIGH[] = { [10:41]};
31      bins SUPERHIGH[] = { [42:$]};
32  }
33  WD: coverpoint n_state {
34      bins LOW = { [1:6]};

```
- Test Results:**
- Test: MergedTest Show: Definitions Auto bins and holes Table Map
- Cover Group Item: coin_fsm:Cvr, test_jukebox.st0.coin1::Cvr, RD
- Bin Name Status:
 - HIGH ✓
 - LOW ✓
 - MED ✓
 - SUPERHIGH_2a ✗
 - SUPERHIGH_2b ✗
 - SUPERHIGH_2c ✗

In the above illustration, the covergroup definition coin_fsm::Cvr defined in module coin_fsm has 5 different variants created for each instance of the module.

Excluding Auto Cross Bins

You can exclude the auto cross bins either directly in the Detail window or using the Edit Exclusion sub-menu.

To exclude auto cross bins

1. Select auto cross bin in the Summary window.
2. Right-click and select Expand.

The compressed bins are visible.

3. Select Exclude/Unexclude icons to exclude/include the sub-bins.
- 4.

Using URG with Unified Exclusion

This section describes how to use exclude file with URG to remove excluded items from URG-generated reports. The file created using DVE, or manually, is passed to URG using the command line option `-elfile`. Excluded items are marked “excluded” in the generated reports, and not taken into account for computing coverage scores.

Generating URG Reports Using Exclude Files

To pass the exclude file to URG, the switch `elfile` is used:

```
urg -dir ... -elfile filename.el
```

The figure [Figure 12-2](#) shows a normal URG report with the total coverage summary. This report displays the total coverage numbers for the line, condition, toggle, FSM and branch coverage. For example:

Figure 12-2 Example of a Report without Exclusion

Total Coverage Summary					
SCORE	LINE	COND	TOGGLE	FSM	BRANCH
49.97	72.66	33.98	50.61	34.83	57.77

Hierarchical coverage data for top-level instances						
SCORE	LINE	COND	TOGGLE	FSM	BRANCH	NAME
49.97	72.66	33.98	50.61	34.83	57.77	test_jukebox

[dashboard](#) | [hierarchy](#) | [modlist](#) | [groups](#) | **[tests](#)** | [asserts](#)

When the exclude file is specified using the switch `elfile`, URG generates the following report [Figure 12-3](#). The total coverage numbers have increased since some of the uncovered coverage metric items specified in the exclude file are omitted from the coverage calculation, which results in higher coverage numbers:

```
urg -dir simv.cm -elfile filename.el
```

Figure 12-3 Example of a Report generated with an exclude file

Total Coverage Summary					
SCORE	LINE	COND	TOGGLE	FSM	BRANCH
51.06	72.77	33.99	50.72	34.83	62.98

Hierarchical coverage data for top-level instances						
SCORE	LINE	COND	TOGGLE	FSM	BRANCH	NAME
51.06	72.77	33.99	50.72	34.83	62.98	test_jukebox

[dashboard](#) | [hierarchy](#) | [modlist](#) | [groups](#) | **[tests](#)** | [asserts](#)

Exclusion in Strict Mode

By default, URG allows any objects to be excluded with an exclude file. To disallow the exclusion of covered objects, use the `-excl_strict` switch:

```
urg -dir simv.cm -elfile e2.el -excl_strict
```

If the exclude file you load contains any objects that are covered, those objects will not be excluded, and will be reported as “attempted” in the file “attempts.log.”

Covergroup Exclusion in URG

You can do the following things for covergroup exclusion in URG:

- You can specify a covergroup within the design hierarchy or a fully-qualified covergroup name as seen in URG report in the elfile. URG report includes the complete design hierarchy name also as a part of the covergroup name.

- You can specify the bin level exclusion in terms of the bin names.
- For a compact and easy specification especially for automatic bins, URG report prints an identifier (which is a number in an increasing order) for each of the bins in a cover point and a cross. This numbering scheme is generated for all bins in each of the coverpoint.

You can specify bin level exclusion in terms of list of these identifiers.

- You can specify automatic cross bins in terms of their identifiers or in terms of the bin identifiers of the constituent cover items of that particular cross.

Editing Exclude Files

You can use a text editor to modify exclude files created with DVE. This section describes the format to use when working with an exclude file. DVE saves the exclude files with a ".el" extension.

The Exclude File Format

When editing an exclude file, you must follow the format as described in this section.

The Timestamp

The timestamp is created by DVE. The timestamp identifies the source file and the creation date.

It is better to create exclude files using DVE and, if desired, edit them. However, if you do create your own file from scratch without timestamps, then synchronization checks are not done.

A timestamp is shown as follows:

```
// TimeStamp: /coverage_test/elfile_test.v Thu Apr 10
07:43:03 2008
```

For covergroup metric, the timestamp is obtained from the checksum information and names of the excluded covergroups. An example is shown below:

```
// Checksum: 300200840 3953929699 cg4_i | 3562878031
3865826166 cg3_i | 1282958077 2393771040 cg2_i
```

Any changes in the design pertaining to a covergroup is reflected in its checksum.

Line Exclusion Format

The line exclusion format example is shown below. It identifies the file, the instance, module, and the lines excluded. Note that lines 76 and 77 are if statements with two coverage possibilities.

Verilog File

```
INSTANCE: top3.D2
MODULE: top.test
Line 76.0:76.1
Line 77.0:77.1
Line 84
```

Example

```
71 always @ in
    72 begin
```

```

73 next = state; // by default hold
74 case (state)
75 idle : if (in) next = first;
76 first : if (in) next = second;
77 second : if (in) next = third;
78 third : if (in) next = idle;

```

URG Report

```

76          excluded      first : if (in) next = second;
                          ==> MISSING_ELSE
77          excluded      second : if (in) next = third;
                          ==> MISSING_ELSE
78          2/2           third : if (in) next = idle;
                          ==> MISSING_ELSE
                          ==> MISSING_DEFAULT
79                          endcase

```

Toggle Exclusion Format

The toggle exclusion format shown below identifies the instance, module, and toggle ID.

Verilog File

```

1 module top;
2   reg [26:10] top_var1;
3   reg [26:10] top_var2;
4   reg top_var3;
5   reg top_var4;
6   reg top_var5;
7
8   wire [26:10] top_net1, top_net2;
9   wire  top_net3, top_net4, top_net5;
... ..
Endmodule

```

Exclusion Format

```

INSTANCE: top
Toggle  top_net1
Toggle  0to1 top_net2[26:10]

```

```
Toggle 1to0 top_net3
```

URG Report

Net Details

	Toggle	Toggle 1->0	Toggle 0->1	Direction
top_net1[26:10]	Excluded	Excluded		Excluded
top_net2[14:10]	No	No		Excluded
top_net2[20:15]	Yes	Yes		Excluded
top_net2[23:21]	No	Yes		Excluded
top_net2[25:24]	No	No		Excluded
top_net2[26]	No	Yes		Excluded
top_net3	No	Excluded		Yes

FSM Exclusion Format

The FSM exclusion format below identifies the instance, module, and the FSM, and also an optional state and transition exclusion.

Verilog File

```
reg [1:0] state,next;
parameter idle = 2'b00,
           first = 2'b01,
           second = 2'b10,
           third = 2'b11;

... ..
always @ in
begin
next = state; // by default hold
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;
```

Exclusion Format

```
INSTANCE: top1
Line 14
MODULE: dev
FSM state
State first
State second
State third
Transition idle->first
Transition first->second
Transition second->third
Transition third->idle
```

URG Report

```
State, Transition and Sequence Details for FSM :: state
states      Covered
idle        Covered
first       Attempted
second      Attempted
third       Excluded

transitions  Covered
idle->first   Attempted
first->second Attempted
second->third Excluded
third->idle   Excluded
...
```

Condition Exclusion Format

The excluded condition format shows the instance, module, condition, and condition vectors.

Verilog file

```
56 input clk,in;
57 output [1:0] state;
58 wire e;
59 assign e = clk ? 1'b0:1'b1;
60 reg [1:0] state,next;
```


Exclusion Format

```
INSTANCE: top3.D3
Condition 1 (1)
Condition 1 (2)
```

URG Report

```
LINE          59
  EXPRESSION (clk ? 1'b0 : 1'b1)
              -1-
-1-  Status
0    Excluded
1    Excluded
```

Branch Exclusion Format

The excluded branch format shows the instance, the module, and the vector ID representing the excluded branches.

Verilog file

```
24  always@(posedge clk)
25    begin
26      // nested if's
27      if ( e && f)
28          begin
29              $display("s3");
30              if ( g && h)
31                  $display("s4");
32          end
33  end
```

Exclusion Format

```
MODULE: test
Branch 2 (0)
Branch 2 (1)
```

URG Report

```
27          if ( e && f)
```

```

-1-
28         begin
29             $display("s3");
30             if ( g && h)
-2-
31                 $display("s4");
                    ==>
                    MISSING_ELSE
                    ==>
32         end
        MISSING_ELSE
        ==>

```

Branches

```

-1- -2- Status
1    1    Excluded
1    0    Excluded
0    -    Covered

```

Assertion Exclusion Format

The excluded assertion format shows the instance, the module, and the assertion name that is excluded.

```

INSTANCE: tb.dut
MODULE: top.test
Assert a_one_yellow

```

Covergroup Exclusion Format

Given below is a snapshot of covergroup code and various elfile examples:

```

module M;
    ...
    bit [3:0] x;
    integer y;
    integer z;

```

```

covergroup cov1;
  cp1: coverpoint x;
  cp2: coverpoint y {
    bins b1 = {100};
    bins b2 = {200};
    bins b3 = {300};
    bins b4 = {400};
    bins b5 = {500};
  }
  cp3: coverpoint z {
    bins zb1[] = {[10:18]};
    bins zb2 = {4543};
  }

  cc1: cross cp1, cp2, cp3 {
    bins cb1 = binsof(cp2.b2);
  }
  cc2: cross cp1, cp2, cp3;
endgroup

covergroup cov2;
  coverpoint cp1 {
    ...
  }
  coverpoint cp2 {
    ...
  }
  coverpoint cp3 {
    ...
  }
  cc1: cross cp3, cp2;
  cc2: cross cp2, cp1;
endgroup

cov1 cov1_inst1 = new;
cov2 cov2_inst1 = new;
...
endmodule

```

```

module top;
    M i1();
    M i2();
    ...
endmodule

```

Example elfile1

```

covergroup top.i1::cov1
covergroup top.i2.cov2_inst1

```

The above elfile specifies that covergroup cov1 in design hierarchy top.i1 and instance cov2_inst1 of covergroup cov2 in design hierarchy top.i2 should be excluded. The hierarchy of an instance is same as that of its definition; In covergroup definitions, '::' serves as the resolution operator, while it is '.' in instances. For instance, covergroup cov2 in the above example is represented as top.i1::cov2, while its instance cov2_inst1 is specified as top.i1.cov2_inst1.

Example elfile2

```

covergroup top.i1::cov1
    coveritem cp2
        bins b1, b2 //Multiple bins grouped in a comma
                    separated list
    coveritem cc2

covergroup top.i1::cov2
    coveritem cp1, cp2 //Multiple coverpoints/crosses
                        grouped in a comma-separated list

covergroup top.i1::cov2, top.i1::cov3 //Multiple cover group
                                      definitions grouped in a
                                      comma-separated list
    coveritem cp2

```

Multiple covergroup instances having common exclusion items can similarly be grouped in a single comma-separated list.

Example elfile3

```
covergroup top.i1::cov1
    coveritem cp1
        bin auto[8], auto[10], auto[11], auto[12], auto[13]
    coveritem cp2
        bin b3, b5
    coveritem cp3
        bin zb2
    coveritem cc1
        bin cb1
    coveritem cc2
        bin {{auto[1], auto[4]}, {b2}, {zb1[13], zb1[14],
            zb2}}
```

```
covergroup top.i2.cov2_inst1
    coveritem cc1, cc2
```

In the above example, the five auto bins auto[8], auto[10], auto[11], auto[12], auto[13] can be specified in a compressed form: auto[8], auto[10-13] or auto[8], auto[10]-auto[13]. This representation is particularly useful to specify coverage holes, which are reported in compressed format by URG.

The auto cross bin specification {{auto[1], auto[4]}, {b2}, {zb1[13], zb1[14], zb2}} subsumes six auto bins of the cross cc2, corresponding to the following six combinations of coverpoint bins:

cp1	cp2	cp3
auto[1]	b2	zb1[13]
auto[1]	b2	zb1[14]
auto[1]	b2	zb2
auto[4]	b2	zb1[13]
auto[4]	b2	zb1[14]

```
auto[4]  b2          zb2
```

Example elfile4

```
covergroup top.i1::cov1
  coveritem cp2
  coveritem cc1
    bin {{auto[1], auto[ 8]}, {}, {zb1[18]}} // {} -
    Consider all bins of cp2 in cross exclusion
  coveritem cc2

    bin {{},{},{}} //Equivalent to excluding all bins
    of the cross cc2
```

Example elfile5

Consider the uncovered bins printed for the cross cc1 in URG report as shown below:

```
cp1 cp2 cp3 COUNT AT LEAST NUMBER
* [b5 , b4 , b3] * -- -- 1056
```

The above compressed representation of uncovered bins can be excluded as shown below:

```
covergroup top.i1::cov1
  coveritem cc1
  bin {{},{b3,b4,b5},{}}
```

Thus, URG will automatically exclude these uncovered bins while generating URG report.

Using UCAPI with Unified 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
<code>covdb_load_exclude_file</code>	loads an exclude file
<code>covdb_save_exclude_file</code>	saves all the excluded coverage data into a .el file
<code>covdb_unload_exclusion</code>	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.”

covdb_set

The covdb_set function is the call to set a cover_element (covergroup/coverpoint/cross/bin) as excluded or attempted.

```
int covdb_set(covdbHandle obj, covdbHandle region,
              covdbHandle test, covdbPropertiesT property,
              int value);
```

Example

```
covdb_set(covObj, region, test, covdbCovStatus,
          covdbStatusExcludedAtReportTime);
covdb_set(covObj, region, test, covdbCovStatus,
          covdbStatusAttempted);
```

covdb_get

The covdb_get function is used to find whether a cover_element is excluded or attempted.

```
int covdb_get(covdbHandle obj, covdbHandle region,
              covdbHandle test, covdbPropertiesT property);
```

Example

```
int stat = covdb_get(covObj, region, test, covdbCovStatus)
int excludedAtReportTime = stat &
    covdbStatusExcludedAtReportTime;
int attempted = stat & covdbStatusAttempted;
```

Auto Coverpoint Bin Exclusion

While iterating over individual coverpoint bins, UCAPL returns compressed auto-bin handles whenever a set of auto-bins can be combined together.

In such cases, if you want to exclude only a few bins from the compressed bin, you can obtain iterator over the compressed bin handle using the 'covdb_iterate' call with a property of 'covdbObjects' and perform exclusion as follows.

```
covdbHandle iter = covdb_iterate(compressed_bin_handle,  
                                covdbObjects);
```

Auto Cross Bin Exclusion

While iterating over individual cross bins, UCAPL returns compressed auto-bin handles whenever a set of auto-bins can be combined together.

In such cases, if you want to exclude only a few bins from the compressed bin, you can obtain iterator over the compressed bin handle using 'covdb_iterate' call with a property of 'covdbComponents' and perform exclusion as follows:

```
covdbHandle iter = covdb_iterate(compressed_bin_handle,  
                                covdbComponents);
```

13

Back Tracing X Values for Gate-level Designs

Back Tracing helps you debug a particular signal that has a value of X by traversing the design backwards both structurally and temporally. You can back trace an X value to its source signals, for example, across gates to identify the signal that caused the X value.

Note:

- Trace X is another feature in DVE, which is an older version of this feature and will be replaced with Back Tracing in an upcoming release.
- It is recommended that you use the Back Trace feature on a gate-level design. If you use it on a non-gate level design, then it will not Back Trace sufficiently to be useful.
- Back Trace may stop tracing for various reasons (for example, multiple X's at the input of a cell). If this happens, you need to manually select the input pin to continue tracing

- Even for gate-level designs, Back Trace may not be able to automatically trace through flip-flops. You may need to manually select the input of the flop, that is X, to continue tracing.

Back Tracing in DVE is performed in the Back Trace Schematic view. You can invoke the Back Trace Schematic view from any of the following windows:

- Waveform
- Schematic
- Path Schematic
- Data Pane

The Back Trace Schematic consists of two views - Wave View and Path Schematic. The Path Schematic view is the main structural view. The Wave view provides a temporal view and provides information to decide which signals need further tracing. You can close the Wave view if not needed.

To back trace a signal

1. Select a signal that is known to have an X value at a particular time (which is the current time in DVE).

For example, in the Wave view, you should select the signal that has an X value and click on the time where the signal is X.

2. Right-click the signal and select **Show Back Trace Schematic**.

The Back Trace Schematic view opens. A wave group and Waveform view of the selected signal is automatically created at the current simulation time. The current simulation time and value pair are annotated on the output pins of the driving cell. The input pins of the driving cell are annotated with the values and times of the next signal transition.

Back Trace will trace as far back as it can based upon your preferences.

3. Select the left most cell/pin in the back trace schematic to find out why tracing stopped.

A reason will be displayed in the top of the schematic.

For example, a common reason is "Trace endpoint pin 'Top.dut.a' status: Multiple X input pins on cells".

4. To continue back tracing, double-click on the input pin of the leftmost cell.

Note:

The current simulation time is moved to the earliest time over all driver inputs.

Setting the Back Trace Properties

The Back Trace Properties window is used to add multiple levels of trace, so that tracing X signals can automatically trace back multiple levels following the X value over time. Instead of expanding one level, you can draw multiple levels and add multiple signals on the traced path to the Wave view.

To set the Back Trace properties

1. Select **Show Back Trace Options** button on the toolbar.

The Back Trace Properties dialog box opens that contains the following options:

- Add traced signals to Back Trace Wave view - Adds the signals on the traced path to the wave group and displayed in the Wave view.
- Add end signals of trace result to Back Trace Wave view
- Stop trace at bus ripper - Controls the time to stop the trace. Once a signal is selected, the default is the time annotated by back tracing. There are several ways to search the waveform for the specified signal. You can specify to stop at "Any Edge", "Rising", "Falling", "Match", "Miss Match", "X Value", or "Value".
- Stop if multiple X inputs - Stops the trace when multiple inputs are provided.
- Number of levels to trace - Controls the maximum number of levels to search backwards when automatically searching for X values.

Using the Back Trace Schematic Toolbar

The Back Trace Schematic toolbar contains the following options:

Table 13-1 Back Trace Schematic toolbar

Options	Description
Show/Hide Wave View	Shows or hides the Wave view in the Back Trace Schematic.
Show back trace options	Opens the Back Trace Properties dialog box.
Start back trace on selected signals	Starts back tracing the selected signal.

14

Constraints Features

- [“Using String Indexed Associative Arrays in Constraints”](#)
- [“Using the array.exists\(\) Function in Constraints”](#)

Using String Indexed Associative Arrays in Constraints

To allow randomizing the values of elements in string indexed associative arrays, arrays can be used in constraints. This enables iterating over the string indexes to constrain the values of the associative array elements.

Syntax

```
class class-name;  
    rand integer associative-array-name[string];  
endclass
```

Description

Only string constants or state strings are allowed in the associative array index expressions. This is because the use of random variables would require the introduction of ordering constraints that force those random variables to be solved before the system method, which is not supported.

Random string index associative arrays are not allowed.

The behavior of string indexed arrays is the same as for integer indexed arrays in the case when size is not constrained; that is, in subsequent calls to randomize, the values of the array elements are randomized and the indexes are preserved.

A `foreach` statement over the array indexes is legal, as in the following example:

```
foreach (aa, i) {  
    aa[i] ...      // aa is an associative array with index i  
}
```

When an associative array is printed, its indexes are printed as strings. Debug messages and error messages show the string constants as the array indexes.

The index usage must be limited to referring the index of the array. Using the index as a guard, as in the following, results in an error message:

```
i == "mystring" => a[i].data == 10; // Error
```

Variable string indexes are not allowed.

If none of the array elements are assigned values, the size of the array is considered to be 0.

Constraining the size of the associative array is not allowed, because random string indexes cannot be generated. Each of the following `constraint` constructs causes an error:

```
constraint c {  
    aa.size() == 3;  
}
```

```
constraint c {  
    x == aa.size();  
}
```

If no elements have been assigned values, but constraints are written on any of the element's values, an error like the following is issued:

```
Error-[CNST-VOAE] Constraint variable outside array error
```

Example

```

program test ;

class C ;
    rand integer aa[string];
endclass

initial
begin
    integer res;
    C c = new;

    c.aa["s5"] = 10;
    c.aa["ss8"] = 15;
    c.aa["sss10"] = 20;

    foreach(c.aa[i])
        $display("%s %d\n", i, c.aa[i]);

    res = c.randomize();

    foreach(c.aa[i])
        $display("%s %d\n", i, c.aa[i]);
end

endprogram

```

Using the `array.exists()` Function in Constraints

The `assoc-array.exists()` system function can be used in a constraint to check whether a particular element exists in an associative array.

Syntax

```
assoc-array.exists(index-expr) -> constraint-expr;
```

Description

For every possible element of an associative array, the `assoc-array.exists()` function can be used to set a bit variable that indicates whether the element exists.

The `index-expr` can be of the type of the associative array key, such as integer or string. Only integral types (non-complex classes) are allowed as keys for associative arrays.

Only a constant or state variable can be used as the index expression. Rand variables are not allowed as arguments to `assoc-array.exists()`.

Example

```
class C;
  string s1;
  bit inUse[string]; // State string indexed assoc array.
  rand integer x;
  constraint c_1 {
    inUse.exists("tmp") && inUse["tmp"] -> x == 10;
    inUse.exists(s1) && inUse[s1] -> x == 20;
    foreach (inUse, key) {
      inUse.exists(key) -> guarded-constraints
    }
  }
```

```
}  
endclass
```

15

Coverage Features

This chapter contains the following topics:

- [“Turning Coverage Collection Off for Covered Objects”](#)
- [“URG Difference Reports for Functional Coverage”](#)
- [“Correlation Report: Which Tests Covered Which Bins”](#)
- [“Reporting Only Uncovered Objects”](#)
- [“Hierarchical Covergroups”](#)
- [“Cross of Crosses”](#)

Turning Coverage Collection Off for Covered Objects

A covergroup or a coverpoint might be hit multiple times in a test, increasing its score. It might be the case, however, that only the first hit (or the number of hits specified with the `at_least` option) are of interest and any hits after that are not interesting. In such cases, you can turn off coverage collection dynamically at runtime for an object (such as a coverpoint or covergroup) if the desired number of hits for the object has been reached. This is done by using the runtime option: `-cg_coverage_control=2`.

This section contains the following topics:

- [“General Use Model Flow of `cg_coverage_control=2`”](#)
- [“Limitations”](#)

General Use Model Flow of `cg_coverage_control=2`

The following is the general procedure turn coverage objects on or off at runtime.

1. Use `$load_coverage_db ("simv.vdb")` to load a user-specified coverage database. This is a standard system task as documented in the IEEE 1800 SystemVerilog LRM. Now this loaded database contains coverage data for some of the coverage objects in the current simulation run.
2. Optionally, use the following three additional VCS system tasks to help to achieve finer control on the coverage database loading:

```
$coverage_load_cumulative_data  
$coverage_load_cumulative_cg_data  
$covgLoadInstFromDbTest
```


See “Unified Coverage Directory and Database Control” in the *VCS/VCSi User Guide* for information about using those system tasks.

3. Use the `-cg_coverage_control=2` option at runtime to disable coverage data collection for objects that are already hit in the loaded database.

Examples

Example 15-1

```
initial
begin
$coverage_load_cumulative_data ("test", "simv.vdb");
end
```

The `$coverage_load_cumulative_data` system task call loads the database at the start of simulation, and checks if the coverage goal has been achieved. If the goal is achieved during the run, coverage collection is turned off and no coverage updating occurs from the next sampling event onward.

Example 15-2

```
initial
begin
#5
$coverage_load_cumulative_data ("test", "simv.vdb");
end
```

Normal sampling occurs until `$coverage_load_db_monitor_covg_data` is called. After the coverage database is loaded at #5, the functionality is the same as in Example 2-21

Limitations

- The `goal` attribute of a covergroup does not have any impact on this feature: The `-cg_coverage_control=2` flag ignores the `goal` attribute value and assumes the goal to be 100%.
- This feature can be used only in conjunction with the one of the load coverage database system tasks listed in [“General Use Model Flow of cg_coverage_control=2”](#). If used in stand-alone mode it may produce unexpected results.

URG Difference Reports for Functional Coverage

Coverage driven verification is an iterative process of running some tests, analyzing coverage results, adding new tests and repeating the cycle until the required level of coverage is achieved. During this iterative process, it sometimes helps to understand the differential value that a new test adds to an existing coverage result. That is, it is helpful to compare the coverage results of a new test run with an existing base or a reference test run. This is referred to as "Difference Report" in URG.

Given two tests `test1` and `test2`, a diff report shows the difference in the objects covered by the two tests. The difference is the set of objects covered by one test minus the set of objects covered by the other test.

To create a diff report, use the `-diff` flag of the `urg` command. The following is an example of the a `urg` command with the `-diff` flag:

```
urg -dir mydir.vdb mydir.cm -tests myfile -diff
```

The filename specified with the `-tests` flag (`myfile` in the example above) must contain the names of the two tests. If more or fewer than two tests are specified, or if the `-tests` flag is not given at all, URG reports an error exits.

The order in which the two tests are specified matters. The first test specified is called the diff test. The second test is called the base test. The diff operation is *diff test – base test*.

The default for `-diff` is to report a number of objects. However, a `count` option can be specified with the `-diff` flag to change the operation from object-based to count-based. The two forms are:

```
urg -dir mydir.vdb -tests myfile -diff [object]  
urg -dir mydir.vdb -tests myfile -diff count
```

A sample myfile looks like this:

```
simv/test  
simv/test_gen_1
```

If the `-diff count` flag is used, then the differences between the hit counts are shown for any metrics that have counting data in tests: The hit count in the base test is subtracted from the hit count in the diff test for each object. If the result is greater than 0, the result is shown as the hit count in the diff report. If the result is greater than or equal to the hit count goal (1 by default), then the object is shown as covered.

Only the assert and covergroup metrics are supported with the `-diff` flag. If data from other metrics is present, URG prints a warning message, and the other metrics data are ignored. No scores, objects, or other values from other metrics appear in a report generated with `-diff`.

Diff Results Shown in the Dashboard and Test Pages

Details about the diff being processed are shown at the top of the dashboard and test pages. For example, assume that the `-tests` file contains the following:

```
simv/test2  
simv/test1
```

Then the dashboard page would show this:

```
Date: date-and-time  
User: username  
Version: VCS-version  
Command line: urg -diff -dir simv.vdb -tests mytests
```

This report was generated with the `-diff` flag. The tests used were:

```
base test: simv/test1  
diff test: simv/test2
```

The only objects not shown as covered in this report are those that are covered in `simv/test2` that are not covered in `simv/test1`.

In a diff report, there are no list of tests in the `tests.html/txt` file, since the only two tests are the base test and the diff test, and they are already specified out explicitly.

What is Shown as Covered

Report for Default Mode (`-diff` or `-diff object`)

In a report generated with the `-diff` or `-diff object` flag, the only things shown as covered are those objects that were covered in the diff test but not covered in the base test. The overall scores, the summary tables, and the detailed reports all show only what was covered in the diff test but not covered in the base test.

For example, suppose the following information appears in the dashboard:

Hierarchical coverage data for top-level instances

SCORE	LINE	COND	TOGGLE	FSM	ASSERT	NAME
9.84	15.11	0.00	1.51	29.06	3.54	vitv

This means that 15.11% of all line coverage objects in the design that were not covered in the base test are covered in the diff test.

The following table shows how a given object is shown in the diff report for different combinations of coverage status in the base and diff tests:

Diff Test	Base Test	In Diff Report
Covered	Covered	Not covered
Not Covered	covered	Not covered
covered	Not Covered	Covered
Not covered	Not covered	Not covered

Note that in the default mode, the hit counts do not matter. For metrics that support hit counts in the report, the hit count of the diff test is shown for any objects that are covered in the diff test but not covered in the base test. For all other objects a hit count of 0 is shown.

Report for Count Mode (-diff count)

In `-diff count` mode, hit counts are computed as follows for each coverable object:

```

diff = diff test count – base test count
if (diff >= 0)
    displayed count = diff
else displayed count = 0

if (diff >= hit count goal for this object)
    displayed result = covered
else displayed result = not covered

```

The following table shows some examples for a given object:

Hit Count Goal	Diff Test	Base Test	In Diff Report
1	7	10	Not covered, count 0
1	10	7	Covered, count 3
4	10	7	Not covered, count 3
1	10	10	Not covered, count 0
1	10	10	Not covered, count 0
1	0	10	Not covered, count 0
1	1	0	Covered, count 1

For metrics for which counting is not enabled, the same reporting rules are used as for default mode (that is, `-diff object`), even when `-diff count` was given.

Unsupported Flags

The following flags are not compatible with the `-diff` flag. If any of these flags are given along with the `-diff` flag, an error is reported and URG exits.

- `-grade`

- `-hvp`
- `-annotate`
- `-trend`
- `-show tests`
- `-show maxtests N`
- `-show availabletests`
- `-dbname` (You cannot save a merged diff db)
- `-map`

Exclusions

All types of exclusion are supported with the `-diff` flag, including the `-hier` file and the loading of `exclude` files.

Correlation Report: Which Tests Covered Which Bins

For a group of tests that are being merged, the `-show tests` option of the `urg` command creates a report that shows the correlation between tests and the coverage achieved, by showing which tests covered which coverage bins.

By default, up to three tests that produced hits in a particular bin are listed for that bin. To remove this limit, you can use the command line argument `-show maxtests N` instead of `-show tests`. The *N* value is a positive integer that specifies the maximum number of tests you want to list as contributing to each bin.

A `-tests file` argument is required along with `-show tests` or `-show maxtests N`. The *file* is a regular text file that contains a list of the names of tests that are to be merged.

Following are examples of the usage of `urg` options `-show tests` and `-show maxtests`. The order of command line arguments does not matter.

```
urg -dir mydir.vdb mydir.cm -tests file -show tests
urg -dir mydir.vdb mydir.cm -tests file -show maxtests N
```

In test correlation mode, only the functional coverage and assertion coverage matrixes are applicable. In this mode, URG generates a report that differs from the normal report only as described in the following sections:

- [“Covered Objects”](#)
- [“Tests Page”](#)
- [“Unsupported Arguments”](#)

Covered Objects

For functional coverage, normal tables of covered objects are expanded by pairs of TEST and COUNT columns. The TEST column shows the contributing test numbers in the format T1, T2, ..., and the COUNT column shows the corresponding hit counts. For example, a covered bin correlation table may look like [Figure 15-1](#):

Figure 15-1 Functional coverage correlation table

Name	Count	At Least	Test	Count	Test	Count	Test	Count
auto[0:3]	16	1	T1	4	T2	8	T3	4
auto[4:7]	3	1	T2	1	T3	2	-	-
auto[8:11]	12	1	T3	3	T2	6	T1	3
auto[12:15]	16	1	T1	4	T3	8	T2	4
auto[16:19]	4	1	T1	4	-	-	-	-
auto[20:23]	20	1	T1	5	T2	10	T3	5
auto[24:27]	20	1	T1	5	T2	10	T3	5

For assertions, additional tables are added after the original assertion tables.

A separate table is created for every assertion with non-zero attempts, with the assertion name as the header. The first row of a detailed table is the total hit count, and it is followed by rows of contributing tests.

For example, an assertion correlation table may look like [Figure 15-2](#):

Figure 15-2 Assertion coverage correlation table

Cover Directives for Properties: Details				
Name	Attempts	Matches	Vacuous Matches	Incomplete
T1C1	40	36	0	4
T1C1				
Name	Attempts	Matches	Vacuous Matches	Incomplete
Total	40	36	0	0
T1	20	18	0	0
T2	20	18	0	0

At most, the `maxtests` *N* number of tests is shown for every covered object. Empty table cells are indicated by "-".

When the cursor hovers over any test number, a hint of its test name pops up. The popup in [Figure 15-3](#) shows that the test name for test T2 is `simv2.vdb/test`.

Figure 15-3 Popup showing the test name

Name	Count	At Least	Test	Count	Test	Count	Test	Count
auto[0:3]	16	1	T1	4	T2	8	T3	4
auto[4:7]	3	1	T2	simv2.vdb/test...	T3	2	-	-
auto[8:11]	12	1	T3		3	T2	6	T1
auto[12:15]	16	1	T1	4	T3	8	T2	4
auto[16:19]	4	1	T1	4	-	-	-	-
auto[20:23]	20	1	T1	5	T2	10	T3	5

Tests Page

In test correlation mode, list of tests in the tests page (tests.html/txt) have two columns: TEST NO and TEST NAME, showing all the test numbers (T1, T2, ...) and their corresponding test names. For example, a tests page may contain data shown in [Figure 15-4](#):

Figure 15-4 Tests page for test correlation

Total Coverage Summary	
Score	Group
30.73	30.73
Total tests in report: 3	

Data from the following tests was used to generate this report	
TEST NO	TEST NAME
T1	simv2/test
T2	simv2/all
T3	simv/test

Unsupported Arguments

The `-diff` command line argument of the `urg` command is not compatible with test correlation mode. If `-diff` is entered along with `-show tests` or `-show maxtests`, an error is reported and URG exits.

Reporting Only Uncovered Objects

This section describes how URG allows you to create reports only for uncovered objects (instead of creating reports for all coverable objects).

Command-Line Access

URG supports a command-line option that causes only uncovered objects to be shown in the report:

```
% urg -show brief
```

You can use this option in combination with text mode to generate a brief text report:

```
% urg -show text -format brief
```

Report Changes

This section describes how each report section changes when you use the `-show brief` option.

Dashboard

The dashboard report does not change. The total coverage summary for the design is still shown, as well as the summary for each top-level instance.

Module List

Only modules with an overall coverage score less than 100% are shown. Modules with no coverable objects in them (for any reported metric) are not listed.

For example, assume the module list report looks like this:

SCORE	LINE	COND	TOGGLE	NAME
98.72	100.00		97.44	HnSidop
98.84	100.00	100.00	96.52	AMosp16
99.64	100.00		99.29	MospOusn
100.00	100.00		100.00	WNamv8by8
100.00		100.00	100.00	NEff16_8
100.00		100.00	100.00	NEffMux
100.00	100.00		100.00	Namv
				Mosp16by8
				MospNEff16

In brief mode the module list report would only show the following modules:

SCORE	LINE	COND	TOGGLE	NAME
98.72	100.00		97.44	HnSidop
98.84	100.00	100.00	96.52	AMosp16
99.64	100.00		99.29	MospOusn

Hierarchy

The hierarchy report omits any instances that have no coverable objects in their subtree or whose entire subtree has a score of 100% (for all reported metrics). URG does not produce a comment or other indication that such an instance has been omitted.

In other words, the only parts of the hierarchy that will be deleted from the report are those *entire subtrees* that have coverage scores of 100%, or that have no coverable objects in them at all.

Assume the full report looks like this:

SCORE	LINE	COND	TOGGLE	
87.04	83.38	88.11	89.61	HNWOOVISDIPV_0
SCORE	LINE	COND	TOGGLE	
83.33	100.00	100.00	50.00	HNFIDUAPMI_0
95.77	100.00	100.00	99.81	HNGOGU_0
81.21	73.10	76.58	93.96	HNOOVISTIDV_0
SCORE	LINE	COND	TOGGLE	
75.59	70.24	59.26	97.27	HNOOVISTIDV_GTN
				HNUCKIDVOXUSFT_0
SCORE	LINE	COND	TOGGLE	
100.00	100.00	100.00	100.00	HNPSIDEDJI_0
SCORE	LINE	COND	TOGGLE	
100.00	100.00	100.00	100.00	HNUCKIDVOX_1

In brief mode, the report would show only the following modules:

SCORE	LINE	COND	TOGGLE	
87.04	83.38	88.11	89.61	HNWOOVSDIPV_0
SCORE	LINE	COND	TOGGLE	
83.33	100.00	100.00	50.00	HNFIDUAPMI_0
95.77	100.00	100.00	99.81	HNGOGU_0
81.21	73.10	76.58	93.96	HNOOVISTIDV_0
SCORE	LINE	COND	TOGGLE	
75.59	70.24	59.26	97.27	HNOOVISTIDV_GTN

Note that HNUCKIDVOXUSFT_0 is omitted because it has no coverable objects, and the subtree rooted at HNPSIDEDJI_0 is omitted because it has 100% coverage.

Groups

The groups report omits any groups that have a 100% score.

Tests

The tests report does not change.

HVP

The HVP format omits any features whose subtrees are either 100% covered or which have no coverable measures in them. Like the hierarchy, only entire subtrees are omitted (if any).

Assertions Report

URG lists in any of the tables only assertions that are not covered or that failed. Only covers and assertions that are not covered are listed.

The total number of assertions is still reported the same way in `asserts.html`.

Module and Instance Header Sections

In `brief` mode, URG generates module, instance, and group reports only for those regions that have coverage scores less than 100%.

Module Header

URG lists the same header for modules, including all self-instances—even those self-instances that have 100% coverage. However, the self-instances with 100% coverage do not link to any report because their reports are not generated. URG shows all self-instances because you can see how URG computed the scores for the module.

If a module is hidden from the module list page, but its instances are reported, the module header still appears at the top of the module page because the module header contains useful information for the whole page.

Instance Header

URG shows the same instance header, including a list of all subtrees.

Detailed Coverage Reports

URG shows detailed coverage reports only for regions that do not have 100% coverage. Within each region, URG shows the detailed report for a given metric only if that metric does not have 100% coverage in that region.

For example, if a module has 100% line coverage and 50% condition coverage, URG generates a detailed report for condition coverage but no detailed report for line coverage. Note also that the condition coverage report only shows uncovered objects.

URG shows the entire summary table for a metric it reports. For example, the following summary table for toggle coverage would not omit the “1 -> 0” rows even though they are all at 100%:

Toggle Coverage for Module : HnOpvistidv_gtn

	Total	Covered	Percent
Totals	27	23	85.19
Total Bits	330	325	98.48
Total Bits 0->1	165	160	96.97
Total Bits 1->0	165	165	100.00
Nets	14	12	85.71
Net Bits	166	163	98.19
Net Bits 0->1	83	80	96.39
Net Bits 1->0	83	83	100.00
Regs	13	11	84.62
Reg Bits	164	162	98.78
Reg Bits 0->1	82	80	97.56
Reg Bits 1->0	82	82	100.00

However, if the module had 100% toggle coverage in all rows, URG would not provide either this table or any toggle coverage details.

Line Coverage

For line coverage reports (which provide annotated source code that shows coverable objects and which objects are not covered), URG shows only the lines that contain uncovered statements (“uncovered lines”), along with two lines of context before and after each uncovered line.

Showing some context is important because the coverage database does not know the extent of a statement—a statement could cross over several lines, in which case the statement would be truncated. URG provides two lines of context above and below a statement:

```
if (some condition)
    if (some other condition)
        a <= (b ? c :
            ( d ? e :
                ( f ? g :
                    ( h ? i :
                        ( j ? k : l ) ) ) ) ) );
```

If the statement is uncovered, URG shows only this information:

```
if (some condition)
    if (some other condition)
        a <= (b ? c :
            ( d ? e :
                ( f ? g :
```

If several uncovered lines are grouped together, they are all grouped together in the report. For example:

```
if (some condition)
begin
    if (some other condition)
    begin
        a <= b;
```

```

        if (yet another condition)
        begin
            x <= y;
        end
    end
end

```

The report for the above example would show the following text, rather than reporting the information in two separate sections:

```

if (some other condition)
begin
    a <= b;
    if (yet another condition)
    begin
        x <= y;
    end
end
end

```

Condition Coverage

If a condition and all of its subconditions are completely covered (all vectors), then URG omits the report for it and its subconditions.

If a condition is not fully covered, then URG shows the condition's complete report (all vectors). If a condition is fully covered but one of its subconditions is not, URG still shows the complete report for that condition (and the not-fully-covered subcondition).

For example, consider the following report:

```

LINE          505
EXPRESSION    (x && y)
               1   2

```

-1-	-2-	Status
0	0	Covered
0	1	Covered
1	0	Covered

```

LINE          510
EXPRESSION    (a || (b && c))
               1   ----2---

```

-1-	-2-	Status
0	0	Covered
0	1	Covered
1	0	Covered

```

LINE          510
SUB-EXPRESSION (b && c)
                1   2

```

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

The brief report would appear as follows:

```

LINE          510
EXPRESSION    (a || (b && c))
               1      ----2---
```

-1-	-2-	Status
0	0	Covered
0	1	Covered
1	0	Covered

```

LINE          510
SUB-EXPRESSION (b && c)
                1      2
```

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

Note that URG shows the first condition because its subcondition (the second) is not fully covered. The condition at line 505 is omitted in the brief report because the condition was fully covered, with no (uncovered) subconditions.

The brief report shows both the condition on line 510 and its subcondition—even though the condition itself is fully covered—because one vector of the subcondition is not covered.

Toggle Coverage

In the detailed table, URG lists only signals that are not fully covered. For example, consider the following full report:

	Net Details		
	Toggle	Toggle 1->0	Toggle 0->1
Clk	Yes	Yes	Yes
Reset	No	Yes	No
IntersectCurrentY	Yes	Yes	Yes
DValidIn	Yes	Yes	Yes
DIn[0:63]	Yes	Yes	Yes
GothamID[0:1]	No	No	No
IntersectCurrent	Yes	Yes	Yes
IntersectNext	Yes	Yes	Yes
ActiveBitMapRow	Yes	Yes	Yes
BitMapRow[0:3]	Yes	Yes	Yes
ShortObject	Yes	Yes	Yes
LastWord	Yes	Yes	Yes
Stall	Yes	Yes	Yes
OpCode[0:2]	Yes	Yes	Yes

The brief report shows only these items:

	Net Details		
	Toggle	Toggle 1->0	Toggle 0->1
Reset	No	Yes	No
GothamID[0:1]	No	No	No

FSM Coverage

URG omits from the report any FSMs that are fully covered (all states and transitions). However, these FSMs are shown in the FSM summary table—with only the FSM name and its score.

If an FSM is fully covered except for its sequences, URG omits the FSM it will be omitted (even if the sequence score is less than 100).

If an FSM is not fully covered, URG lists all of the FSM's states, and only its uncovered transitions and sequences.

Assertion Coverage

URG list only assertions that have failed or are not covered. Only uncovered “covers” are listed.

Group Report

Inside a group report, URG lists only bins that are not covered.

Hierarchical Covergroups

SystemVerilog supports the covergroup construct to capture the functional coverage model. One of the requirements for a cross is that all the crossed elements must be declared within the same covergroup. This allows seamless and unambiguous sampling, among other benefits.

However, advanced coverage modeling requires the ability to specify crosses of elements in different covergroups. This is called the hierarchical covergroup features. The VCS SystemVerilog implementation has extended the IEEE 1800 standard to enable this feature.

Crosses can be defined between coverpoints that reside in separate covergroup instances. This enables object composition-like behavior for covergroups, in which an instance of a covergroup can make use of the coverage collected by its constituent covergroup instances.

[Example 15-3](#) shows a sample of a hierarchical covergroup, `hier_cg`.

Example 15-3

```
class cl_1;

covergroup cg_0 @(e0);
    cp0 : coverpoint a;
endgroup : cg_0

covergroup cg_1 @(e1);
    cp1 : coverpoint b;
endgroup : cg_1

covergroup hier_cg;
    cr_0 : cross cg_0.cp0, cg_1.cp1;
```

```

        // Using implicit class member
        // covergroup instances: cg_0, cg_1
    endgroup : hier_cg
endclass: cl_1

```

Here, covergroup `hier_cg` crosses coverpoints `cg_0.a` and `cg_1.b`. The `hier_cg` instance is thus a hierarchical covergroup instance, with `cg_0` and `cg_1` as its constituent instances.

This section contains the following subsections:

- [“Constituent Covergroup Instances”](#)
- [“Sampling Hierarchical Covergroups”](#)

Constituent Covergroup Instances

A hierarchical covergroup definition can use any covergroup instance visible in its scope to define crosses. Alternatives to using implicit class member instances are shown in the following sections:

- [“Referring to Covergroups using Cross Module Referencing”](#)
- [“Passing Constituent Instances as Arguments”](#)

Referring to Covergroups using Cross Module Referencing

In [Example 15-4](#), the `cg1_inst` and `cg_2_inst` are visible only inside the initial block, named `blk1`. The `hier_cg` hierarchical covergroup refers to them via XMR (Cross Module Referencing):

Example 15-4

```

module hier_cg_ex;
    event cov_e1, cov_e2;

```

```

bit [5:0] a, b;

covergroup cg_1 @(cov_e1);
    cp0: coverpoint a;
endgroup: cg_1
covergroup cg_2 @(cov_e2);
    cp1: coverpoint b;
endgroup: cg_2

covergroup hier_cg;
    cr_1 : cross blk1.cg_1_inst.cp0, blk1.cg_2_inst.cp1;
endgroup: hier_cg

initial begin: blk1
cg_1 cg_1_inst = new();
cg_2 cg_2_inst = new();
hier_cg h_cg_inst = new();
end: blk1

endmodule: hier_cg_ex

```

Passing Constituent Instances as Arguments

It is possible to define a covergroup with constituent instances passed as arguments at the time of its instantiation, as shown in [Example 15-5](#).

Example 15-5

```

covergroup cg_0;
    cp0 : coverpoint p;
endgroup: cg_0

covergroup cg_1;
    cp1 : coverpoint q;
endgroup: cg_1

covergroup hier_cg (cg_0 a1, cg_1 b1);
    cr_0 : cross a1.cp0, b1.cp1;
endgroup: hier_cg

```

```

initial begin
integer p, q, r;
cg_0 a1 = new;
cg_1 b1 = new;
hier_cg x1 = new;

```

Sampling Hierarchical Covergroups

A hierarchical covergroup infers the sampling event from its constituents and cannot have its own sampling event.

A hierarchical covergroup is considered sampled at a given time stamp if and only if all of its constituent covergroups were sampled at that time stamp. If one or more of the constituent covergroups did not get sampled, the results are not collected for the hierarchical covergroup. The cross hit of a hierarchical cross is composed of the values recorded for the respective coverpoints. An interesting situation occurs when, at a given time stamp, one or more constituent covergroups gets sampled more than once. This is illustrated for [Example 15-6](#) in [Table 15-1](#).

Example 15-6

```

class cl_0;
covergroup cg_0 @(e0);
    cp0 : coverpoint p;
endgroup : cg_0

covergroup cg_1 @(e1);
    cp1 : coverpoint q;
endgroup : cg_1

covergroup hier_cg;
    cr_0 : cross cg_0.cp0, cg_1.cp1;
endgroup : hier_cg
endclass : cl_0

```

```

function new();
cg_0  = new();
cg_1  = new();
hier_cg = new();
endfunction
endclass
...

```

Table 15-1 Hierarchical Covergroup Sampling Results

Simulation Time Step	Events	Values Sampled		
		a1.p	b1.q	x1.cr0
1	e0	6	-	-
2	e1	-	a	-
3	e0, e1	7	b	(7,b)

Multiple Values Sampled in the Same Time Step

If any of the constituent covergroups is sampled multiple times during the same time step, more than one value might be recorded for the constituent coverpoints in a given time step. For such cases, crosses are composed as described below.

Table 15-2 Hierarchical Covergroup Sampling Results

Simulation Time Step	Events	Values Sampled		
		a1.p	b1.q	x1.cr0
1	e1#3	6, 7, 42	-	-
2	e0#1, e1#1	6	a	(6,a)
3	e0#3, e1#3	6, 7, 42	a, b, c	(6,a) (7,b) (42,c)
4	e0#3, e1#1	6, 7, 42	a	(42,a)
5	e0#3, e1#2	6, 7, 42	a, b	(42, b)

If all constituent coverpoints record the same number, say n , of values in the time step, covergroup `hier_cg` records n crosses with the i th cross composed from the i th recorded values of the each constituent coverpoint (see the row for Simulation Time Step 2 in [Table 15-2](#)). If the number of values sampled is not the same for all constituent coverpoints, the cross is composed using only the last recorded value of each coverpoint. This is done to ensure that crosses collected are meaningful for the majority of the interesting use cases (see the rows for Simulation Time Steps 4 and 5 in [Table 15-2](#)).

Cross of Crosses

IEEE 1800 SystemVerilog allows cross specification among two or more coverpoints. However, it does not allow the cross constituents to be a cross itself. Since advanced coverage models require this ability, the cross of crosses extension enables the definition of crosses that include other crosses as elements.

[Example 15-7](#) illustrates a cross of crosses.

Example 15-7

```
//Assuming cp0, cp1, cp2, cp3 are of type bit[3:0]
covergroup cg_1;
  coverpoint cp0;
  coverpoint cp1;
  coverpoint cp2;
  coverpoint cp3;
  cr_0: cross cp0, cp1 {
    bins low_b = binsof(cp0) intersect {[0:7]};
  }
  cr_1: cross cp2, cp3 {
    bins high_b = binsof(cp2) intersect {[8:15]};
  }
  cr_0_X_cr_1: cross cr_0, cr_1 {
    bins one_b = binsof(cr_0.low_b) && binsof(cr_1.high_b);
  }
  cp0_X_cr_1: cross cp0, cr_1 {
    bins two_b = binsof(cr_1.high_b) && binsof(cp0)
    intersect {[0:7]};
  }
endgroup: cg_1;
```

Here, apart from the fact that `cr_0_X_cr_1` is the cross of two crosses (`cr_0` and `cr_1`), it is like any other cross. It can define attributes and bins like regular crosses. However, the `binsof` syntax is restricted to disallow the use of `intersect` with cross

bins. It is also possible to have a mix of cross and coverpoint elements, for example, cross `cp0_X_cr_1` crosses `cr_1` with coverpoint `cp0`. Note that the `binsof` construct can use `intersect` when referring to a coverpoint bin.

Limitation: A cross of crosses records and reports its bins in terms of the bins of its elements, as shown in [Table 15-3](#).

Table 15-3 Cross coverage bins for crossed coverpoint values

Coverpoint values				Crosses		Bin hit in <code>cr_0_X_cr_1</code>
<code>cp0</code>	<code>cp1</code>	<code>cp2</code>	<code>cp3</code>	<code>cr_0</code>	<code>cr_1</code>	
1	10	14	15	<code>low_b</code>	<code>high_b</code>	<code>one_b</code>
14	10	14	15	14, 10	<code>high_b</code>	([14, 10], <code>high_b</code>)
8	10	7	15	8, 10	7, 15	([8, 10], [7, 15])

Note:

[14, 10], [8, 10] and [7, 15] are autocross bins in the respective crosses.

Limitation

Currently, this feature cannot be used together with the hierarchical cross feature.

16

Coverage Convergence Technology

This document describes the coverage convergence technology (CCT). This chapter describes all coverage-convergence-specific options.

This chapter also includes coding guidelines that allow you to get the most out of the coverage convergence technology. Some common user flows are also provided.

CCT is a technology which allows for auto-generation of a stimulus coverage model, stimulus coverage convergence, and the use of bias files to guide the convergence process. Convergence in VCS is done through the analysis of coverage holes during simulation and the specific targeting of those holes through automatically applied constraint biasing.

Compile-Time Options

This section describes the options you use to enable the coverage convergence technology when using the VCS compiler. Note that using these options does not guarantee that coverage convergence will occur. For coverage convergence to occur, the constraints and coverage model need to meet certain requirements.

You also need to specify the options described in this chapter (as appropriate) when you execute simulation (that is, when you run `simv`).

Coverage Convergence Option

`-ntb_opts cct`

Ensures that the required coverage-convergence-related processing is done for all relevant coverage points.

Example for OpenVera testbench:

```
vcs -ntb -ntb_opts cct foo.vr
```

Example for SystemVerilog testbench:

```
vcs -sverilog -ntb_opts cct foo.sv
```

Coverage Model Autogeneration Options

`-ntb_opts cct_cov_gen`

Generates a coverage model for a stimulus object—that is, for a class that contains random variables and/or constraints.

See [“Automatic Generation of a Coverage Model from Constraints” on page 288](#) for details on how a coverage model is inferred from constraints.

When you specify this option, coverage convergence writes automatically inferred cover groups (if any) to the directory `${SIMV_DIR}/cct_cov_gen`. You can specify a different directory with the `-ntb_opts cct_cov_gen_dir` option.

Note that if the `cct_cov_gen` directory already exists, any files in that directory will be overwritten when you rerun coverage convergence with the `-ntb_opts cct_cov_gen` option.

The names of the files created (and the names of the coverage groups themselves) depend on the names of the classes and constraints blocks in the source file.

Example for OpenVera testbench:

```
vcs -ntb -ntb_opts cct_cov_gen foo.vr
```

Example for SystemVerilog testbench:

```
vcs -sverilog -ntb_opts cct_cov_gen foo.sv
```

```
-ntb_opts cct_cov_gen_dir <directory_name>
```

Overrides the default directory (`./cct_cov_gen`) into which coverage convergence writes automatically inferred covergroups.

Example:

```
vcs -ntb -ntb_opts cct_cov_gen -ntb_opts cct_cov_gen_dir  
./my_dir foo.vr
```

Coverage Convergence Runtime Options

To enable the coverage convergence functionality, specify the following command-line options when executing the `simv` generated by VCS.

`-cct_enable`

Enables coverage convergence at runtime.

The `-cct_enable` option is required for running coverage convergence. The following arguments are optional.

`-cct_enable=on_start`

Tells coverage convergence to target coverage holes from the very first call to randomize. By default, the targeting of holes does not start until coverage convergence determines that merely running pure random stimulus (based on constraints) is no longer resulting in increasing coverage.

You can use this option when few randomize calls occurring in the test so each randomize call can be coverage-aware. This option is also enabled automatically when you use bias file options.

`-cct_bias_file=<file_name>`

Specifies the bias file to be used for the test run. The bias file specifies the coverage holes that the test should target in the current run. Different runs of the test can use different bias files as input. Although it is possible to create a bias file manually, we recommend that you generate the bias file using the URG utility. See [“URG Options for Bias File Generation” on page 285](#) for more details.

For the bias file to have any effect, you must enable other options (such as `-cct_enable`). When you specify the `-cct_bias_file` option, the `-cct_enable=on_start` option is automatically enabled.

URG Options for Bias File Generation

This section describes the URG options related to coverage convergence. These options generate bias files. You use the bias files to bias coverage convergence to target specific coverage holes. The URG options described in this section are in addition to other required URG options, such as `-dir` and `-format`.

```
-group cct_gen_bias <non-zero integer>
```

Instructs URG to partition all the coverage holes in the input database into the specified number of bias files.

Coverage convergence writes the bias files to the `./biasConfigs` directory. The bias files are text files that represent coverage holes. Avoid editing the bias files by hand. However, if you choose to edit the bias files, do not deviate from the bias file format. You can use each bias file generated with the `cct_gen_bias` option as an input to a test run.

See [“Understanding Bias Files and Coverage Convergence” on page 298](#) for more details.

```
-group cct_gen_bias_dir ./<my_dir>
```

Overrides the default directory setting that determines where the bias files are written. By default, the bias files are written into the `./biasConfigs` directory. Coverage convergence creates the directory if the directory does not exist.

Coding Guidelines

The following sections provide coding guidelines for using coverage convergence.

Constraints and Coverage Model on the Same Variables

Coverage convergence works when the coverage model is on the stimulus variables. In other words, the cover points (and cover point expressions) should be the `rand` variables that represent the stimulus. Constraints and the coverage model should be in the same class. Further, the randomize and coverage sampling should happen on the same instance of the generator class.

No Procedural Overwriting of Values Generated by the Solver

Coverage convergence works by generating constraints for all the coverage holes and by enabling these “coverage constraints” during consecutive calls to `randomize`. If there is procedural code (in `post_randomize` for example) which overwrites the values generated by the solver, then it is possible that coverage bins might not be hit, even though the solver is generating the right values for the random variables.

Coverage Should Be Sampled Between Consecutive Calls to `randomize`

The sampling event for the coverage model should be triggered between consecutive calls to `randomize`, to ensure that all generated values are sampled. An alternative is to use the `@randomize` sample event. Using this sample event ensures that the cover points are sampled as soon as randomization is completed.

Use Open Constraints

In an ideal scenario, you should specify only the legal (or environment) constraints when running coverage convergence. Do not use test constraints. Test constraints further constrain the legal environment to focus the solver on a specific part of the legal space. With coverage convergence enabled, the solver is automatically focussed on the portion of the legal space that is covered by the coverage model.

If you want to focus the solver in coverage convergence mode, then load a preexisting coverage database before starting the stimulus generation. Coverage convergence will not target anything that is covered in the loaded database.

Avoid Explicit or Implicit Partitioning in Constraints

Coverage convergence might not work efficiently when there are explicit or implicit partitions in the constraint problems. Explicit partitioning is enabled by using “solve before” constructs. Implicit partitioning is enabled when there are function calls in constraints or when object allocation is enabled.

If partitioning does happen and coverage convergence is enabled, then partitioning might lead to some loss of efficiency for coverage convergence: some coverage bins might be targeted but not covered.

Avoid In-line Constraints and the Use of “constraint_mode” and “rand_mode”

Coverage convergence is designed to work on the static structure of the constraints and coverage. If the structure of the constraints is changed using in-line constraints or turning constraints or randomness off and on using “constraint_mode” and “rand_mode”, then the performance of coverage convergence might be compromised.

Automatic Generation of a Coverage Model from Constraints

One of the goals of coverage convergence is to extract a functional coverage model from the constraint expressions, and automatically cover it. The intuition is that a better sampling of the stimulus space will exercise the design behaviors more exhaustively, increasing the likelihood of hitting coverage goals.

The plan is to provide a first-class “contract” to the user which specifies how the coverage model is inferred and how goals are named.

Coverage Groups

A cover group will be generated for each class by default.

A cover group will be generated for each class to contain the coverage model derived from variable declarations in the class. The cover group for the class that will contain the coverage model for variables will be called `covg_<class_name>`.

The members of the cover group (cover points, crosses) will be annotated with comment attributes in a manner that will allow tracking back to the original constraint expressions. The following sections detail the types of cover points and crosses that will be inferred.

The cover groups will contain items as described below.

Cover Points

Variable Coverage

Each `rand` variable will have a cover point associated with it, except for variables that have an unguarded set membership constraint at the top-level, in any of the constraint blocks of the class. For example:

```
enum PktType = {Type1, Type2};
rand bit[7:0] x;
rand PktType p;
PktType prevp;
```

```
constraint c1 {
x in {0:1, 5};
}
```

Here, a cover point will be inferred for `p`, but not for `x` or `prevp`. Note that a set membership cover point will be inferred for `x`.

The cover point expression will be just the variable. The following rules apply to the creation of bins:

- Autobinning for `enum` type variables.
- Equal volume autobinning, with a maximum of 64 bins, for variables with precision 8 bits or less.

For example:

```
rand bit[7:0] x;
rand PktType p;
rand bit[31:0] y;
```

In the above example, autobinned cover points will be created for `x` and `p`, with the bins for `x` having the ranges 0–3, 4–7, ..., 252–255, and the bins for `p` having the values `Type1` and `Type2`.

The built-in function `rand_mode` can be used to turn off the randomness of a `rand` variable. But the collection of coverage for such variables will continue irrespective of the mode change.

The cover points for variables will be called `covp_<cover_point_id>`. `<cover_point_id>` is a unique integer identifier for every cover point. The autobins for the cover points will be named by the usual naming convention for autobins. The comment for the cover point will indicate details about the variable.

Branch Coverage

Each conditional block will have an associated Boolean cover point. The cover point expression will be the conjunction of all the enclosing conditional guards. For example:

```
if (p) {  
  if (q) {  
    x == y;  
  }  
}
```

In the above example, there will be a coverpoint with the expression $p \&\& q$.

The comment for the cover point will have the expression string.

Condition Coverage (Sensitized)

Each sensitizing guard condition will have a Boolean cover bin associated with it. For example:

```
if (p || (q && r)) {  
  ...  
}
```

In the above example, there will be a cover point with expressions p $||$ $(q \&\& r)$ with one bin corresponding to the case when expression p is ON (or true) and one bin corresponding to the case when expression $q\&\&r$ is true. The idea is to make sure that the conditional expression becomes true at least once for every implicant in the expression being true.

The cover points will be guarded by the conjunction of all the enclosing conditional guards.

Note:

The subexpressions of the guard expression involving operators other than `&&`, `|`, `!` are considered as atomic, and the sensitizing conditions are defined over these atomic subexpressions.

When the condition expression is a set expression, the condition coverage model will be generated like the set membership coverage model. For example:

```
if (p in {0:4, 7}) {  
    ...  
}
```

Here the cover point expression will be `p`, and the bins `0`, `1-3`, `4`, `7` will be created.

The comment for the cover point will have the expression string.

Set Membership Coverage

CCT will infer a cover point for constraints coded using the `in` and `dist` operations. Equivalent ways of writing the same constraints using other operators might not result in an inferred coverage model. For example:

```
x in {0:5};
```

results in an inferred cover point, whereas

```
x >= 0 && x <= 5;
```

does not result in an inferred cover point. Also, use of set membership in complex constraints does not result in an inferred cover point.

For membership in sets with constant members:

- A single cover point will be created. The sample expression will be the same as the LHS
- value in the set membership constraint.
- A bin will be created for the low value of each range.
- A bin will be created for the high value of each range, for non-singleton ranges.
- A bin will be created for the range excluding the low and high values of each range, if the resulting range is non-empty.

For example:

```
x in {0:5, 7};
```

In the above example, a cover point with expression x and four bins, 0, 1–4, 5, 7 will be created.

The functional coverage language does not allow dynamic variables (those that are not parameters to the coverage groups) to describe bin ranges. Hence, for membership in sets with nonconstant members:

- A cover point with three Boolean cover bins will be created for each of the ranges as follows:
 - A bin with the expression (range_low_expression).
 - A bin with the expression (range_low_expression + 1 : range_high_expression - 1).
 - A bin with the expression (range_high_expression).

For example:

```
x in {a:5, b};
```

In the above example, a cover point will be created for variable x with the following bins:

- One bin with value a ;
- One bin with value range $(a+1:4)$;
- One bin with value 5 ;
- One bin with value b ;

Note:

If the set membership range expressions use random variables, then no coverage bin will be inferred for that range. For example, if a in the above example were specified as `rand`, then the first two bins would not be inferred.

Note:

All the cover points above will be guarded by the conjunction of all the enclosing conditional guards.

Crosses

Combinations of control variables in the class, will be used to generate crosses.

We will define control variables as variables that are of enumerated types or bit-vectors that are 8 bits wide or less, for which a variable coverage cover point is generated.

Cross on Variables Within a Class

If a class contains control variables x , y , then a cross x , y will be generated. The cross will be autobinned. The cover points and bins for x , y will be as described in the section on variable coverage.

The set of variables that participate in one cross is determined as follows:

Control variables will be added to a cross until the total number of expected bins (product of underlying cover point bins) of the cross does not exceed 64000.

If more control variables still remain after previous step, then a new cross will be created.

The crosses will be called `covc_<cross_id>`. `<cross_id>` is a unique integer identifier for the cross. The comment for the cross will indicate details about the crossed variables (class name, file, line, etc.).

Cross of Condition and Constraint Expressions

The sections on condition coverage and set membership described two kinds of cover points. In the case of set membership constraints inside conditionals, we will generate a cross of the cover points for coverage of the condition (sensitizing). This cross will be autobinned. For example:

```
if (p || (q && r)) {  
  x in {0:5, 7};  
}
```

In the above example, cover points for `x` will be generated from the set membership constraint. Cover points for the condition expression will be generated. A cross for the two cover points will be generated.

Sampling Event

A special built-in sample event called `@(randomize)` will be used to determine the sampling for the generated coverage groups.

This event can be used in user defined cover groups that are embedded in classes. The event is triggered when an instance of this class is randomized, either directly or through containment in an instance of another class being randomized. When the event is triggered, coverage is tracked for the instance and/or the cover group definition (that is, cumulative).

The `@(randomize)` event is triggered on all `rand` objects in the context being randomized, after the solver is done, and values have been populated in the objects. If objects are being allocated by `randomize`, then the allocation of the objects is done before the values are populated back, hence before `@(randomize)` is triggered.

Contribution to Coverage Scoring

The automatically generated coverage model will contribute to the coverage score by default with a weight of 1.0.

Coverage Model Inference for In-line Constraints

Since in-line constraints (`randomize with {...}`) is recommended for use in the specification of test constraints, we will not infer a coverage model from such constraints. In fact, we expect the use of test constraints to be minimized after coverage driven stimulus generation is implemented.

Use Model

Autogeneration of coverage model is done as part of the VCS compile step. The switch to enable coverage model generation is `-ntb_opts cct_cov_gen`. All the coverage models will be written out as text files in the `cct_cov_gen` directory. The `cct_cov_gen` directory will be present wherever the `simv.daidir` and `simv` files are generated by VCS compiler. Both System Verilog and OpenVera testbench formats are supported. The language in which the coverage model will be generated is the same as the syntax of the testbench.

Example command line for an OpenVera testbench:

```
vcs -ntb -ntb_opts cct_cov_gen foo.vr
```

Example command line for a SystemVerilog testbench.

```
vcs -sverilog -ntb_opts cct_cov_gen foo.sv
```

It is expected that the user will first create a coverage model using the `auto_gen` option. Then the user will include the coverage model into their testbench (using ``include` or `#include` directives) and then make sure the coverage model is instantiated in the new task for the enclosing class (that is, the class where the constraints and random variables are specified). The user is encouraged to view the autogenerated coverage model and make changes if required.

The VCS compiler does not compile the design when `cct_cov_gen` is specified (that is, no `simv` is created).

Understanding Bias Files and Coverage Convergence

Motivation

Coverage convergence is a new technology that enables coverage convergence for testbench functional coverage, that is, cover groups. Coverage convergence seeks to automatically target coverage holes by identifying the coverage holes and directing the constraints solver to generate stimulus such that the coverage holes will be hit. Coverage convergence works at a test level, that is, coverage convergence tries to hit coverage holes within a test. In most customer environments, verification tests are run on a regression farm, where multiple copies of the tests are being executed in parallel on different machines in the farm, with each simulation being assigned a different random seed. It is desirable that with coverage convergence enabled, each test be able to target different portions of the coverage space in order to meet the coverage goals in the shortest possible clock time.

What Is a “Test”?

A test for the purpose of this document is an executable that can be simulated (for example, `simv` generated by VCS). The source files (testbench files, design files etc.) are compiled by VCS compiler and linked with the simulation engine to create the executable. The executable can then be invoked multiple times with different runtime options, such as different random seeds or configuration files and the simulation results can be observed. Note that the executable is created once, that is, the source files (constraint, coverage models,

transactors, modules etc.) do not change with every execution. The different options passed with every execution lead to different behavior of the simulation.

From the coverage convergence point of view, a “test” should consist of a testbench with an “open” set of constraints: the constraints should represent the entire legal stimulus space. Note that in many existing methodologies, the constraints in a “test” consist of the legal constraints as well as a set of test constraints that limit the solution space of the legal stimulus. This is typically done to focus the test towards some verification targets such as coverage or specific features.

The bias file approach described below works best with tests using an open set of constraints and a full coverage model. Thus any invocation of the simulation executable can target any of the existing valid coverage holes.

Using Coverage Convergence Bias File to Target Coverage Holes

The user can influence the coverage holes being targeted by coverage convergence in a test run is by using a coverage convergence bias file. This is a file that contains a list of coverage holes which the coverage convergence engine will target when the test starts running. The coverage convergence bias file format is a text file representing coverage holes. This file can be modified by the user if needed.

Runtime Option to Specify a Bias File

The coverage convergence bias file will be passed in as a runtime argument for the test run, using

`-cct_bias_file=<path_to_cct_bias_file>`. When the test starts executing, the coverage convergence engine will read the bias file and populate its in-memory coverage hole database with the data from the coverage convergence bias file. Then it will systematically start targeting the holes. After all the holes are targeted, if more randomize calls occur, then they will not be reactive, that is, no coverage hole will be targeted.

- The holes in the coverage convergence bias file are all at a coverage group definition level. They will apply to all instances of the coverage group in the test, for which reactivity can be applied and can be enabled.
- Only the coverage holes present in the bias file will be targeted.
- The bias file will be read and processed by the coverage convergence runtime engine. The test does not need to be recompiled if the bias file is changed or a new bias file is to be used for the test.
- The bias file has no impact on the coverage engine and on coverage reporting. It will only influence coverage convergence. If the data in the bias file is for a coverage group which cannot be targeted by coverage convergence, then all the data for that coverage group is ignored by the coverage convergence engine.
- If the coverage convergence bias file as pointed to by the `-cct_bias_file` option cannot be read, then an error will be issued the file will be ignored. The test will run as though no bias file has been specified.

- There are no restrictions on the name of the coverage convergence bias file.
- The bias file will provide a guideline as to what coverage bins are targeted by coverage convergence, but coverage convergence can choose to override these guidelines in certain situations. For example, coverage convergence will not target coverage bins that have been assigned a zero weight by the user or coverage bins belonging to disabled by the user using the collect attribute or by using `coverage_control` system task.

Automatic Generation of Coverage Convergence Bias Files

URG automatically generates coverage convergence bias files for a particular test. URG can be invoked on a coverage database file and will generate N bias files, where N is supplied by the user. The utility can be invoked as follows:

```
$VCS_HOME/bin/urg -dir <coverage_db_dir> -group
cct_gen_bias <number_of_bias_files_required>
```

The utility will enumerate all the coverage bins in the input database files and will create N random partitions of the holes where N is supplied by the users using the `num_bias_files` argument. It is an error if N is greater than the number of coverage bins in the coverage database. The bias files will be written out in the `${PWD}/cctBiasConfigs` directory as `config0`, `config1`, `config(N-1)`. Other URG options (such as `-format`) can also be used simultaneously to process the coverage data. Since the coverage convergence bias file represents coverage bins at a coverage group definition level, it will be assumed that there is only one shape for

every coverage definition in the input coverage database. If multiple coverage definitions are detected, then an error will be issued and not bias files will be generated.

The bias files generated by URG can then be used as input to tests being run in parallel.

Repeatability of Test Results for Parallel Regression Runs

An essential requirement for the approach outlined above is that it should be possible to reproduce the results of a test run in a parallel regression environment in a stand alone manner. More specifically, if a test fails in the parallel regression run, it should fail in exactly the same manner when run on its own. Coverage convergence ensures that given the same inputs and the same command line arguments, the sequence of values generated by the solver will be exactly the same. Here the inputs are the bias file and the test source code, and the arguments are the random seed and the `-cct_bias_file` argument. *Note that this means that the regression system needs to preserve the bias file used by a test in addition to other test source code and scripts.*

Usage Scenarios

This section discusses coverage convergence usage in various common verification scenarios. In all the scenarios in this section, it is assumed that the guidelines mentioned in the previous section are being followed (for example, coverage and constraints are on the same variables, and so on).

You can load a preexisting coverage database in every scenario to screen out already-covered coverage bins. By doing so, the current run can focus only on uncovered bins. The coverage database must be loaded after all the testbench components have been instantiated, so that the coverage data from the database can be loaded in the proper (in-memory) runtime database.

Running a Single Test with Randomized Configurations

This scenario generally applies to multimedia devices, in which the device has many interfaces and many modes and the device can be configured to select some interfaces and some modes.

The object of verification is to exercise all the interesting configurations of the device. In the testbench, configurations are selected, and then data (possibly random) is passed through the device.

Configurations for a device are typically chosen (through `randomize`) a few times per test run. In the extreme case, only one configuration might be selected (for example, `config.randomize` is performed only once).

In this scenario it is important to preload the coverage database to ensure that already-selected configurations are not chosen.

Running a Single Test with Randomized Transactions

This scenario applies to transaction-based verification environments. This scenario is required for packet-processing devices or instruction-based processor verification.

The device is set in some mode, and then multiple transactions are generated and passed through the device. You use constraints to generate the transaction objects. In this scenario, you can use the default arguments for coverage convergence.

If you assume that many transactions will be generated (randomized), then it is appropriate for the convergence heuristics to take over and apply reactive calls as needed.

```
status = transObj.randomize();
```

or

```
repeat (1000) {  
    transObj = new;  
    status = transObj.randomize();  
}
```

Using a Bias File for a Parallel Regression

In this scenario, you use the bias file generation utility to bias inputs for different parallel test runs.

1. Run the first batch of tests using the following commands:

```
$SIM -cct_enable +ntb_random_seed = ${SEED}
```

Assume that each batch contains 100 tests.

2. Wait for first batch run to complete.
3. Merge the coverage results for all 100 tests in the first batch and store the results in `merged.vdb`.
4. Perform bias file generation for the next batch of tests using the following command:

```
$VCS_HOME/bin/urg -dir merged.vdb -group cct_gen_bias  
100
```

This command distributes the remaining coverage holes into 100 bias files.

5. Run the second batch of 100 tests with each test using a different bias file as an input:

```
$SIM -cct_enable -cct_bias_file=<bias_filename>
```

6. Merge results from the second batch of runs.
7. If coverage has not reached your coverage goal, then repeat steps 3 through 5.

Autogenerating a Coverage Model

Coverage convergence can generate a coverage model from the constraints specification. However, you must instantiate the coverage model that you intend to use in the simulation.

1. Autogenerate the coverage model:

```
vcs -ntb -ntb_opts cct_cov_gen foo.vr
```

Coverage convergence generates the model in the file
./cct_cov_gen/MyClass.vr.

2. Examine the generated model to determine if the model meets your expectations.
3. Add the declaration for the generated cover group as part of the class member declarations.

For example, if the name of the covergroup is `MyCov` and it is of the class `MyClass`, then you must add to the other member declarations for the class `MyClass`:

```
coverage_group MyCov;
```

4. The autogenerated coverage group must be instantiated in the new task of the enclosing class. Perform the instantiation with the following statement:

```
MyCov = new;
```

5. Make the cover group instantiation the last statement of the new task. If the new task does not exist, then you must add the task to the cover group instantiation.
6. Because the name of the autogenerated cover group is deterministic, the declaration and instantiation can be done once during testbench development phase.

Methodology and Flow Issues

This section addresses methodology and regression execution issues.

Scenario: All Tests Have the Same Constraints and Coverage Space (Recommended)

When all the tests being executed in parallel have exactly the same coverage space and the same legal stimulus space (that is, the constraints are exactly the same for all the tests), then the coverage convergence bias file approach can be used to achieve high efficiency test runs. Each test should have its own bias file. The input

database for the URG utility can be obtained by running any test and using the output coverage database. The number of test runs should be used as the value for the `cct_gen_bias` argument for URG. Once the bias files are generated, they can be used again and again for repeated regression runs. It is desirable that the number of times the stimulus objects are randomized in a test be at least equal to the average number of coverage bins in the bias files. After all the tests are done running, the coverage database from all the tests can be merged to get the final coverage number. It is possible but not efficient to use both the random seed and the bias file for every test.

Scenario: Tests Are Grouped Into Categories With Each Category Having Specific Test Constraints

In this scenario, tests have test specific test constraints which further constrain the legal stimulus space and thus also reduce the hittable coverage bins. (Coverage bins whose value ranges lie outside the space of the test constraints cannot be hit). A bias file can be created for every category of tests. The bias file should be such that all the coverage bins in the file are hittable given the test constraints for that category. Each test within this category can use the same bias file but with a different random seed. The URG bias generation utility can be used to generate a bias file template. (Note that the URG does not look at test constraints when generating the bias files). The test writer can then modify this template to include the hittable coverage bins. As in the previous scenario, assuming test constraints remain constant across regression runs, then the bias files have to be generated once and can be used for multiple regression runs. Even if all the bins in a bias file are not hittable, all it causes is less efficient utilization of resources. Coverage convergence may use some cycles in the test run to try to target unhittable bins, but remaining cycles will still target hittable coverage bins. In this scenario too, the

coverage data from all the tests can be collected and merged. Then a sequential run can be used to target any remaining coverage holes. For this final sequential run, the merged coverage database can be used as an input along with the fully open constraints.

Scenario: Coverage Database Being Loaded in the Beginning of a Test Run

If a coverage database is being loaded as part of the configuration for a test, then there is no need to use a coverage convergence bias file. Coverage convergence will only target coverage bins that are not covered in the loaded database, so it is already biased in some sense. If the same database is loaded in multiple tests, then a random seed can be used so that each test targets different holes. This is useful in a batch mode type of regression environment. Here all tests have the same constraints and coverage space, but one batch of tests is run at one time. After the batch is complete, the coverage databases are merged and the merged database is used as input for the next batch of tests. *Note that for repeatability purposes, the input database associated with a test needs to be preserved in case the test needs to be run in a stand alone mode.* If a coverage convergence bias file is used, then it will override the bias from the loaded coverage database, that is, holes in the bias file will be targeted even if they are marked as covered in the loaded coverage database. Hence it does not make sense to use both coverage database load and coverage convergence bias file approaches for a test.

17

VCS Multicore Technology Design Level Parallelism

VCS Multicore Technology takes advantage of the computing power of multiple processors in one machine to improve simulation turnaround time. For the current release, Design Level Parallelism is an LCA feature. For the sake of clarity, this document describes the whole VCS Multicore Technology feature, including both design level parallelism (DLP) and the GA Application Level Parallelism (ALP).

Candidates for DLP should be a long running simulation. Short running simulations of an hour or less may not show much value to user even if DLP can show some gain. It's the long running testcase typically several hours/days where you will see most value if DLP can demonstrate any gain.

To evaluate your design's suitability for DLP, run the serial profiler and generate vcs.prof data. Serial profile is enabled by adding `+prof` during the compile. Analyze the instance based profiling to determine if it is a good candidate for parallel simulation and identify partitions for use.

- If the Instance view shows that there are sub hierarchies that use most CPU time (instead of having few percents scattered across the design), then these sub hierarchies are good candidates for DLP partitions.
- If there are no uniform distribution or one instance takes most of the %Totaltime it is not suitable for DLP
- If there are too many instances and all of them take about 5% or less %Totaltime it is not suitable for DLP either. The distribution should be at least about 10% or higher per instance to indicate a good DLP opportunity.
- Cumulative simulation activity in the partitions chosen should be at least 50% or higher (e.g. you can have 2 partitions taking 25% each or 5 partitions taking 10% each)
- There should be at least 2 partitions to try out DLP apart from master partition.
- For multicore designs, the cores are typically the partitions.
- Ensure from serial profile output that there is minimal Testbench overhead (either in the form PLI or SVTB or any other test-bench language). If most of the time spent shows up in program block (test bench), it means not too much opportunity for DLP.

VCS Multicore Technology Options

You use the VCS `-parallel` option to invoke parallel compilation. The syntax is:

```
vcs filename(s).v -parallel [ +multicore_option(s)]  
[ -parallel+show_features ] [-o multicore_executable_name]  
[vcs-options]
```

These options and properties are as follows:

`-parallel`

When used without VCS Multicore options, `-parallel` enables all VCS Multicore Technology options, except for design level parallelism. When used with VCS Multicore options, `-parallel` enables only those option specified.

This option is available at compile-time only.

`+design=FILENAME`

Enables design level parallelism and specifies the name of the partition configuration file. Note: this option is available at compile-time only.

`+fc [=NCONS]`

This compile-time option, enables multicore Functional Coverage, and with `NCONS` specifies the number of PFC consumers. `NCONS` can be changed at run time.

```
vcs -parallel+fc ...  
vcs -parallel+fc=3 ...
```

`+profile`

Enables design and application level profiling.

`+profile_value`

Enables value-based design level profiling.

`+sva [=NCONS]`

This compile-time option enables multicore SVA, and with *NCONS* specifies the number of multicore SVA consumers. *NCONS* can be changed at run time.

`+tgl [=NCONS]`

Enables multicore Toggle Coverage, and specifies the number of multicore toggle coverage consumers. To enable the use of the same executable for both serial and parallel runs, use this option at runtime.

NCONS specifies the number of multicore SVA consumers. For ALP, *NCONS* can be changed at run time.

`+vpd [=NCONS]`

Enables multicore VCD+ Dumping, and specifies the number of multicore VCD+ consumers. To enable the use of the same executable for both serial and parallel runs, use this option at runtime.

NCONS specifies the number of multicore SVA consumers. For ALP, *NCONS* can be changed at run time.

`+vpd_sidebuf=MULT`

Sets the size of PVPD side buffer to *MULT* times the size of the main buffer. To enable the use of the same executable for both serial and parallel runs, use this option at runtime.

`[-o multicore_executable_name]`

Using the VCS `-o` option to specify the simulation executable binary filename allows work on multiple simultaneous VCS Multicore compiles and runs. VCS Multicore-specific data is stored in a directory *executable_name.pdaidir*. The default path name is *simv.pdaidir*.

Note:

If [NCONS] is not specified, the default is 1 client.

`-parallel+show_features`

Displays enabled VCS Multicore features. Note that you must enter the `-parallel` option with `+show_features`

Examples:

`-parallel+vpd` is equal to `-parallel+vpd=1`

`-parallel+tgl` is equal to `-parallel+tgl=1`

VCS Multicore option examples:

```
vcs -parallel+design=part.cfg ....
```

```
vcs -parallel+fc .... -o psimv
```

```
vcs -parallel+vpd+fc -parallel+tgl -o par_simv ....
```

```
vcs -parallel+design=part.cfg+sva ....
```

Use Model for Design Level Profiling and Simulation

Design level parallelism (DLP) performance gains depend on a number of factors, including

- The degree of parallelism in the design
- Design partition activity
- Location of clock logic in the master partition or replicated in each partitions. Avoid clock dependency between partitions in which one partition generate the clock and feeds into the other partition.

Design Suitability

Characteristics of design types suitable for DLP include:

- Parallel scan DFT designs
- Large BIST designs with parallel cores
- Multicore designs
- Large blocks instantiated multiple times
- Processor type of designs
- Design pipelines making the blocks run in parallel (e.g., network processors)

DLP Use Model

Follow these steps to determine design qualification criteria for VCS Multicore DLP.

1. Ensure the design (and tests) runs well with the latest VCS version (C-2009.06).

The selected testcase for DLP should be a long running simulation. Short running simulations of an hour or less may not show much value to user even if DLP can show some gain. It's the long running testcase typically several hours/days where you will see most value if DLP can demonstrate any gain.

2. Run the serial profiler and generate vcs.prof data. The VCS profiler tells you the subhierarchies in your design that use the most CPU time. See [“Profiling a Serial Simulation” on page 17-323](#).
3. Analyze the instance based profiling to determine if it is a good candidate for parallel simulation and identify partitions for use.

If the Instance view shows that there are sub hierarchies that use most CPU time (instead of having few percents scattered across the design), then these sub hierarchies are good candidates for DLP partitions. E.g.:

Instance	%Totaltime
top.A1	35.82
top.A2	33.31
top.A3	30.87

Here top.A1, top.A2 and top.A3 are good candidates for DLP partitions.

If there are no uniform distribution or one instance takes most of the %Totaltime, the design is not suitable for DLP.

If there are too many instances and all of them take about 5% or less %Totaltime it is not suitable for DLP either. The distribution should be at least about 10% or higher per instance to indicate a good DLP opportunity.

4. Cumulative simulation activity in the partitions chosen should be at least 50% or higher (e.g. you can have 2 partitions taking 25% each or 5 partitions taking 10% each)

There should be at least 2 partitions to try out DLP apart from master partition.

For multicore designs, the cores are typically the partitions.

Ensure from serial profile output that there is minimal Testbench overhead (either in the form PLI or SVTB or any other test-bench language). If most of the time spent shows up in program block (test bench), it means not too much opportunity for DLP.

5. Specify the partitions in the VCS Multicore configuration file. See [“Specifying Partitions” on page 17-324](#).

6. Run VCS Multicore compilation, specifying the VCS Multicore configuration file.. For more information, see [“Design Level Simulation” on page 17-317](#).

```
vcs -parallel+design=partition.cfg  
simv
```

If you wish to increase the performance gains from VCS Multicore, additional steps follow:

7. Run VCS Multicore simulation again and this time collect data for the VCS Multicore profiler. See [“Profiling a VCS Multicore Simulation” on page 17-327](#).

Use Model for Assertion Simulation

1. Run VCS Multicore compilation specifying the `sva` option.
2. Run VCS Multicore simulation.

Use Model for Toggle and Functional Coverage

1. Run VCS Multicore compilation specifying the VCS Multicore `tgl` option and coverage metric options for toggle coverage, and/or the VCS Multicore `fc` option for functional coverage. You can optionally specify the number of consumers for each.
2. Run the simulation to generate coverage results.
3. Generate coverage result reports.

Use Model for VPD Dumping

1. Run VCS Multicore compilation specifying the `vpd` option.
2. Run the simulation to generate the VPD file.

Running VCS Multicore Simulation

VCS Multicore Technology takes advantage of the computing power of multiple processors to improve simulation turnaround time

You can generate results for one of all the following VCS Multicore Technology options in a simulation:

- Design simulation
- Assertion simulation
- Toggle coverage
- Functional coverage
- VPD file generation

Design Level Simulation

Once you have profiled your design and created a partition configuration file, you can simulate your design with design level parallelism only or in combination with some of the ALP options.

1. Compile using the VCS Multicore `-parallel` option and other VCS Multicore and VCS options.

```
vcs filename(s).v -parallel+design=partition_filename.cfg
```

```
[multicore_options] vcs_options
```

2. Run the simulation with VCS and VCS Multicore run-time options.

```
simv
```

Assertion Simulation

You can process only assertion level results or assertion level results along with other VCS Multicore options with this compile-time option.

1. Compile using the VCS Multicore `-parallel` option, the assertion compilation option or options, and other VCS Multicore and VCS options.

```
vcs filename(s).v -parallel+[sva[=NCONS]]  
[ multicore_options vcs_options
```

2. Run the simulation with VCS and VCS Multicore run-time options.

```
simv
```

Toggle Coverage

Generate results for only toggle coverage or toggle coverage along with other results by compiling the design with VCS Multicore options that include the `+tgl` option and VCS coverage metrics options. You can use the `+count` option to report total executed transactions. After generating coverage results, you can examine them using the Unified Report Generator.

Note:

To enable the use of the same executable for both serial and parallel runs, use this option at runtime.

```
tgl [+count]
```


Report total executed transactions.

1. Compile using the VCS Multicore `-parallel` option, coverage option or options, and other VCS Multicore and VCS options.

```
vcs filename(s).v -parallel+tgl[=NCONS] -cm tgl  
[multicore_options] [vcs_options]
```

2. Run the simulation to generate coverage results.

```
simv -cm tgl [vcs_options]
```

3. Generate coverage result reports:

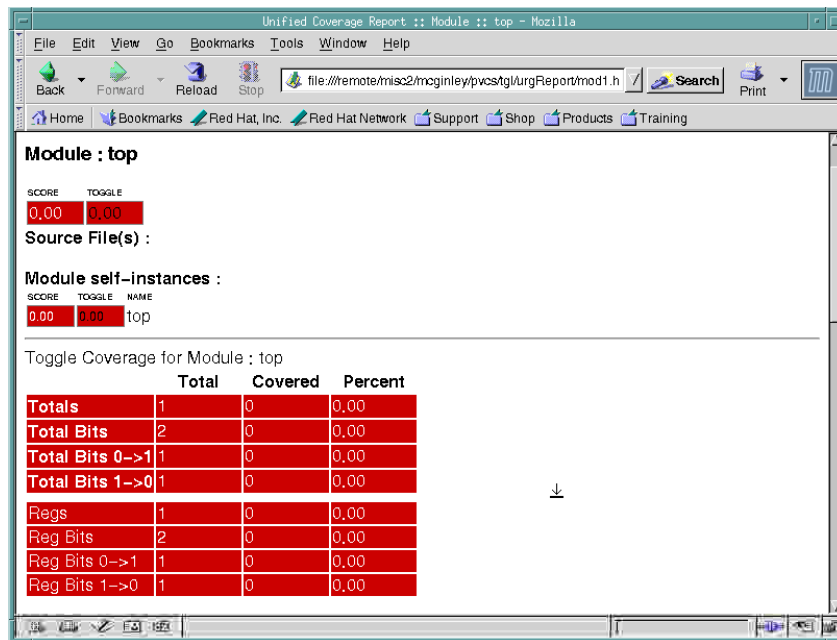
```
urg -dir coverage_directory.cm urg_options
```

Example

In this example, toggle coverage results only are generated and the URG report is produced in the default HTML format.

```
% vcs -cm tgl mda -q -cm_dir pragmaTest1.cm -cm tgl -sverilog  
-parallel+tgl=2 pragmaTest1.v  
% simv -cm tgl  
% vcs -cm_pp -cm_dir pragmaTest1.cm  
% urg -dir pragmaTest1.cm
```

Results can then be examined in your default browser.



Functional Coverage

Generate results for only functional coverage or functional coverage along with other results by compiling the design with VCS Multicore options that include the `+fc` option and VCS coverage metrics options. Note that this is a compile-time option. After generating coverage results, you can examine them using the Unified Report Generator.

1. Compile using the VCS Multicore `-parallel` option, coverage option or options, and other VCS Multicore and VCS options.

```
vcs filename(s).v -sverilog -parallel+fc[=NCONS]  
[parallel_vcs_options] [vcs_options]
```

2. Run the simulation to generate coverage results.

```
simv
```

3. Generate coverage result reports:

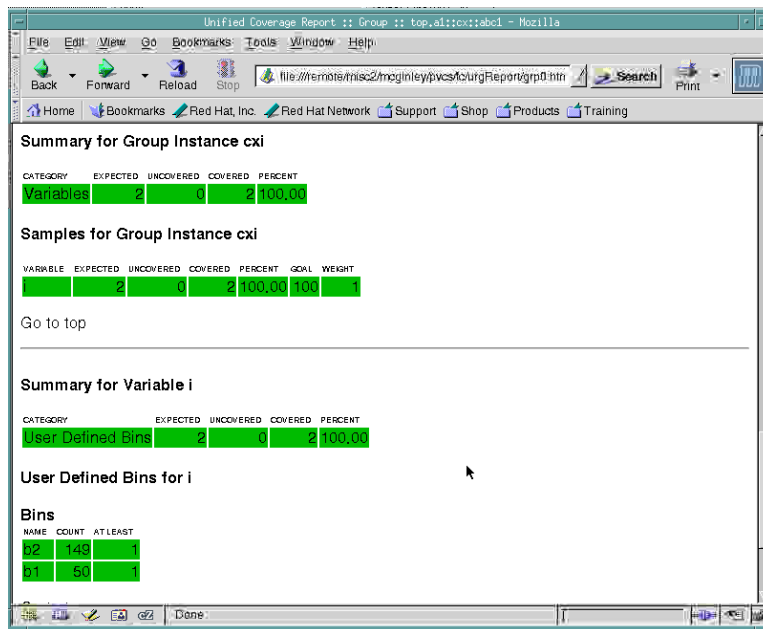
```
urg -dir coverage_directory.cm urg_options
```

Example

In this example, functional coverage results only are generated and the URG report is produced in the default HTML format.

```
% vcs iemIntf.v -ntb_opts dtm -sverilog -parallel+fc=2
% simv -covg_cont_on_error
% $urg -dir simv.vdb
% cat urgReport/gr*
%
```

Results can then be examined in your default browser.



VPD File

You can enable VCS Multicore VPD+ Dumping and specify the number of VCS Multicore VPD+ consumers using the VCS Multicore `vpd` option. To enable the use of the same executable for both serial and parallel runs, use this option at runtime.

Note:

When used with multiple consumers, VPD file size blow up might be an issue. Use `-parallel+vpd_buffer=<N>`, where `N=256, 512` etc.

1. Compile using the VCS Multicore `-parallel` option with the `vpd[=NCONS]` option, and other VCS Multicore and VCS options.

```
vcs filename(s).v -debug_pp -parallel+vpd[=NCONS]  
[multicore_options] [vcs_options]
```

2. Run the simulation.

```
simv
```

You can post-process the results with the generated +VPD database.

Example

In this example, a VPD+ file with three specified consumers is generated.

```
% vcs -debug_pp -parallel+vpd=3 design.v  
% simv
```

Profiling a Simulation

You can profile your VCS Multicore simulation to ensure efficient use of resources. This section details the profiling process. It contains the following sections:

- [Profiling a Serial Simulation](#)
- [Specifying Partitions](#)
- [Profiling a VCS Multicore Simulation](#)
- [Running VCS Multicore Examples](#)

Profiling a Serial Simulation

You can profile your simulation to determine if it is a good candidate for VCS Multicore simulation and identify partitions to use. You do this by telling VCS to run its profiler during simulation with the `+prof` compile-time option.

```
vcs +prof filename(s)
```

The VCS profiler tells you, among other things, what module instances use the most CPU time. The CPU time percentages are for the instance and all instances hierarchically under the instance. The CPU time percentage thus represents the subhierarchy in which the instance is the top-level instance.

The VCS profiler writes its profile information in the `vcs.prof` file in the current directory. Look at the INSTANCE VIEW section in this file. This section, or view, tells you the percentage of CPU time used by each subhierarchy. [Figure 17-1](#) shows this view.

Figure 17-1 Instance View in a vcs.prof File

```
=====
                                INSTANCE VIEW
=====
Instance                                %Totaltime

top.A1_inst                            (1)          31
top.A2_inst                            (2)          29
top.A3_inst                            (3)          30
-----
```

The subhierarchies that use the most CPU time are good candidates for VCS Multicore partitions. In [Figure 17-1](#) you would use separate partitions for the subhierarchies.

Specifying Partitions

The `-design` option requires a partition configuration file. You specify the partitions in the VCS Multicore configuration file using the following syntax:

```
partition {hierarchical_name(module_identifier),...} ;
partition {hierarchical_name(module_identifier),...} ;
partition {hierarchical_name(module_identifier),...} ;
.
.
.
```

The syntax is as follows:

`partition`

The keyword that specifies that what follows are the contents of one partition.

hierarchical_name

The hierarchical name of a Verilog module instance that is the top-level instance of the subhierarchy that you want to simulate as a partition.

module_identifier

The name of the module definition that corresponds to that top-level instance.

All parts of the design not covered in any of the specified partitions together form an implicitly defined master partition. The user-specified partitions are called slave partitions.

Example 17-1 VCS Multicore Configuration File

In this example, there is a separate line for each partition. Each hierarchical name for a top-level module instance must be followed by its module name in parentheses. You can specify more than one subhierarchy in a partition. That is, you can group multiple subhierarchies together to form a single partition.

```
partition {top.A1_inst(A1)};  
partition {top.A2_inst(A2)};  
partition {top.A3_inst(A3)};
```

Note: You can use the Verilog comment syntax to comment your partition file. For more information on creating configuration files, see the *VCS User Guide*.

Example 17-2 A Free Format Configuration File

You can also enter a free format of the specifications in a configuration file entering new lines, blanks, and comments.

```
partition { top.bus1 (MyBus),  
            top.cpu (CPU) }; // two instances per partition  
partition {  
    top.bus2 (MyBus)
```

```
} /* one instance per partition */
```

Profiling a VCS Multicore Simulation

Run the simulation by entering a command line with the name of the master executable (by default named `simv`) and runtime options. The syntax for this command line is as follows:

```
simv -parallel+profile [VCS_runtime_options]
```

VCS Multicore Profiling Options

If you entered the `profile` option on the `simv` command line, VCS Multicore Technology will write the data files that the VCS Multicore profiler needs to report on the VCS Multicore simulation.

You start the profiler with the `pvcSProfiler` command. Its syntax is as follows:

```
pvcSProfiler [profileDumpDir=simvName.pdaidir]  
[doTimeProfiling=1|0] [doToggleProfiling=0|1]  
[graphImHeight=integer] [graphImWidth=integer]  
[graphTnHeight=integer] [graphTnWidth=integer]  
[lowerSimTimeBound=float] [runVersion=string]  
[toggleCutOff=float] [upperSimTimeBound=float]  
[-help]
```

These arguments and properties are as follows:

```
profileDumpDir=simvName.pdaidir
```

Use `profileDumpDir` to specify the directory containing the dump files that the profiler reads. The directory is the name you specified for the simulation executable binary file with the extension `.pdaidir`. The default path name is `simv.pdaidir`.

The profiler writes HTML files that display profile information about the master and slave simulations. By default the profiler creates the `ppResults_0` directory and writes these files in this directory.

`doTimeProfiling=1|0`

Calculate the time accumulation and produce the graphs. The default argument is 1. If the argument is 0, the profiler does not make this calculation or produce the graphs.

`doToggleProfiling=1|0`

Calculate the partition port toggle counts and produce the high count listing. The default argument is 1. If the argument is 0, the profiler does not make this calculation.

`graphImHeight=integer`

Height of the full size graphs in pixels. The default height is 1000.

`graphImWidth=integer`

Width of the full size graphs in pixels. The default width is 1000.

`graphTnHeight=integer`

Height of the thumbnails size graphs in pixels. The default height is 250.

`graphTnWidth=integer`

Width of the thumbnails size graphs in pixels. The default width is 250.

`lowerSimTimeBound=float`

A floating point number for the simulation time when profiling starts. The default argument is 0.0.

`runVersion=string`

By default the profiler creates the `ppResults_0` directory in the current directory and writes its output HTML files in this directory. If you want a different name for this directory, enter this property. With this property the profiler creates the `ppResultsstring` directory.

`toggleCutoff=float`

Specifies that the port toggle count cutoff for a partition is equal to this percentage of highest toggle count. The default argument is 1.0.

`upperSimTimeBound=float`

A floating point number for the simulation time when profiling stops. The default argument is 1e+100.

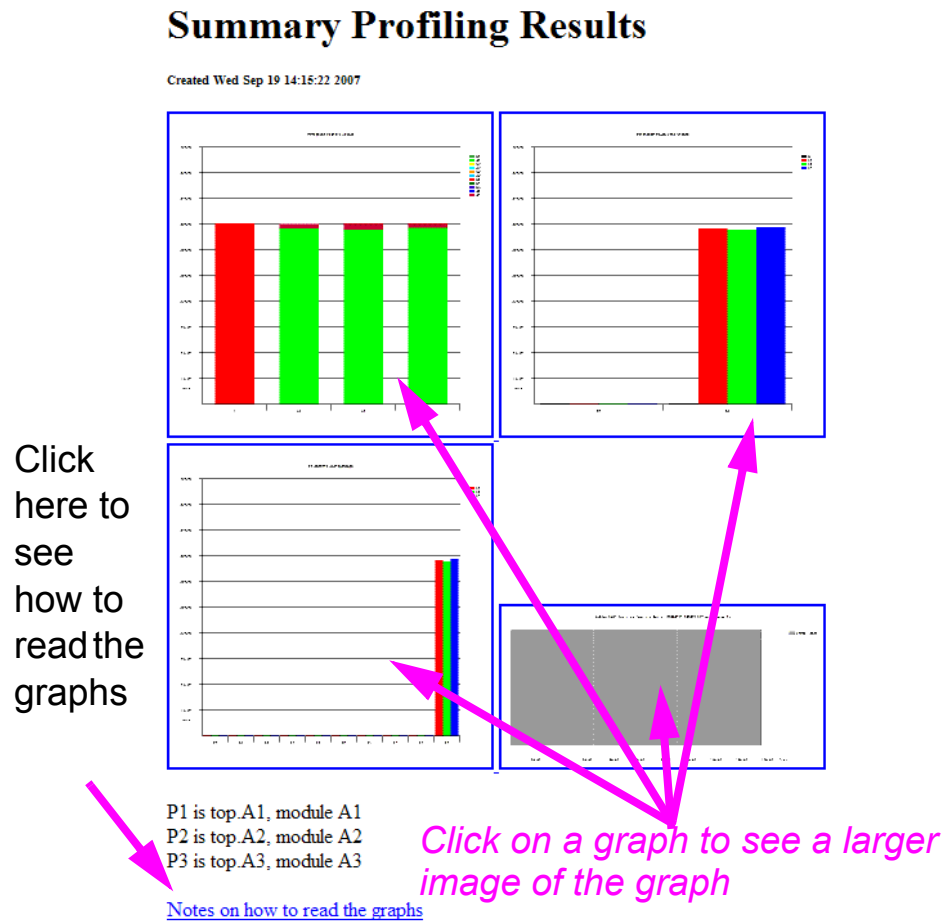
`-help`

Displays the valid properties and their definitions.

Examining the Profiler Results

The main results file that the profiler writes is named `results.html`. By default it writes it in a directory named `ppResults_0` in the current directory. The following is an example of a `results.html` file (as viewed in a regular web browser). Note that if `ppResults_0` already exists, it creates a directory `ppResults_#` where `#` is the next available number:

Figure 17-1 *results.html* File



The *results.html* file contains four graphs:

- The Processor Segment Totals (*graph1.html*)
- The Processor Delta Time Totals (*graph2.html*)
- The S1 Balance Distribution (*graph3.html*)
- The Active VCS Multicore Features (*graph4.html*)

Each image in *results.html* is a hypertext link to an HTML file with a larger image of the graph.

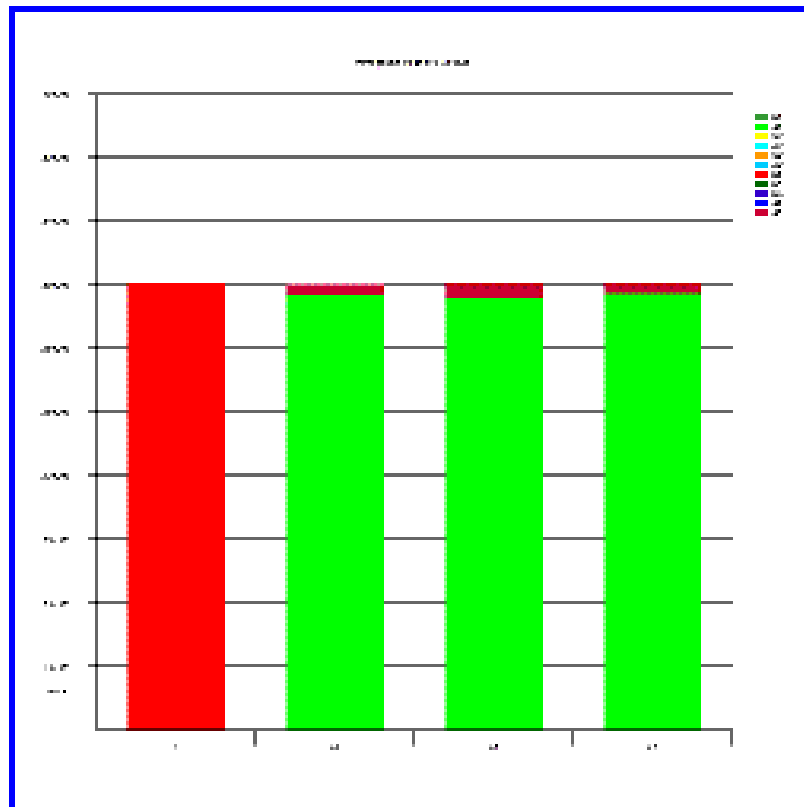
There is a link to the [howtoInterpretGraphs.html](#) file that explains how to interpret the graphs.

The following are a series of examples of these graphs that show different results from the profiler. The first three graphs are from a design where the profiler results are good, showing a balanced design with a good deal of parallelism.

Examples of Good Parallelism

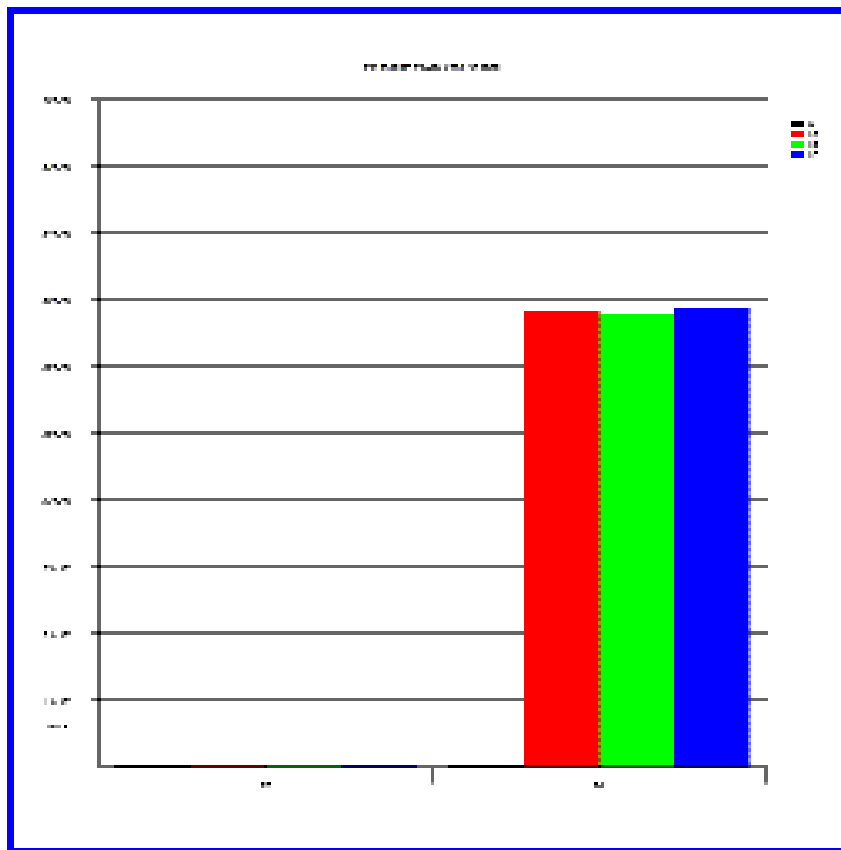
This Processor Segment Totals graph shows that each partition is doing the same amount of work.

Figure 17-2 A Good Processor Segment Totals Graph



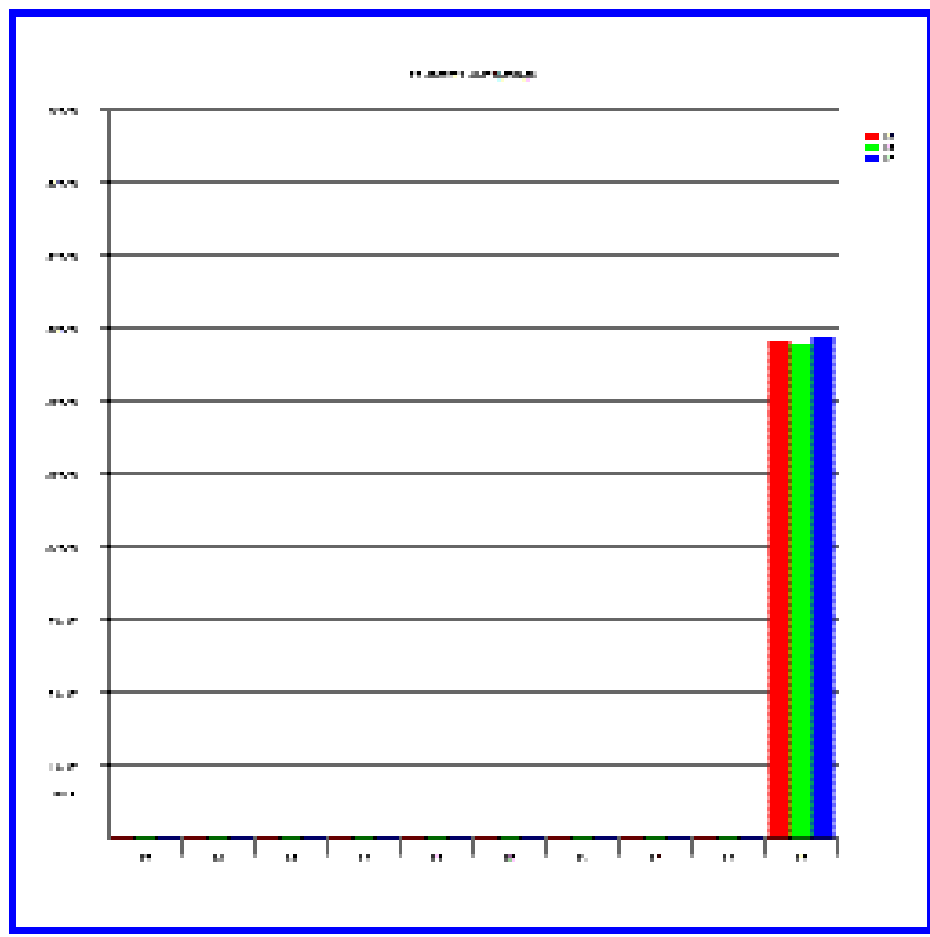
A delta is a set of events in one or more partitions that cause subsequent simulation events in other partitions. Figure 17-3 shows only two deltas, which is good. Delta d0, in which the design generated the clock signal, and d1, where each partition did its work in the same delta.

Figure 17-3 A Good Processor Delta Time Totals Graph



In [Figure 17-4](#) the bars are all to the right, indicating good parallelism.

Figure 17-4 A Good S1 Balance Distribution Graph



Examples of Uneven or Bad Parallelism

The next three graphs are from a design in which the amount of work done by the partitions is uneven or unbalanced.

Figure 17-5 shows that the partitions are not doing the same amount of work. Some partitions, as indicated by the dark color at the top of their bars, are doing a significant amount of waiting for events in other partitions.

Figure 17-5 An Uneven Processor Segment Totals Graph

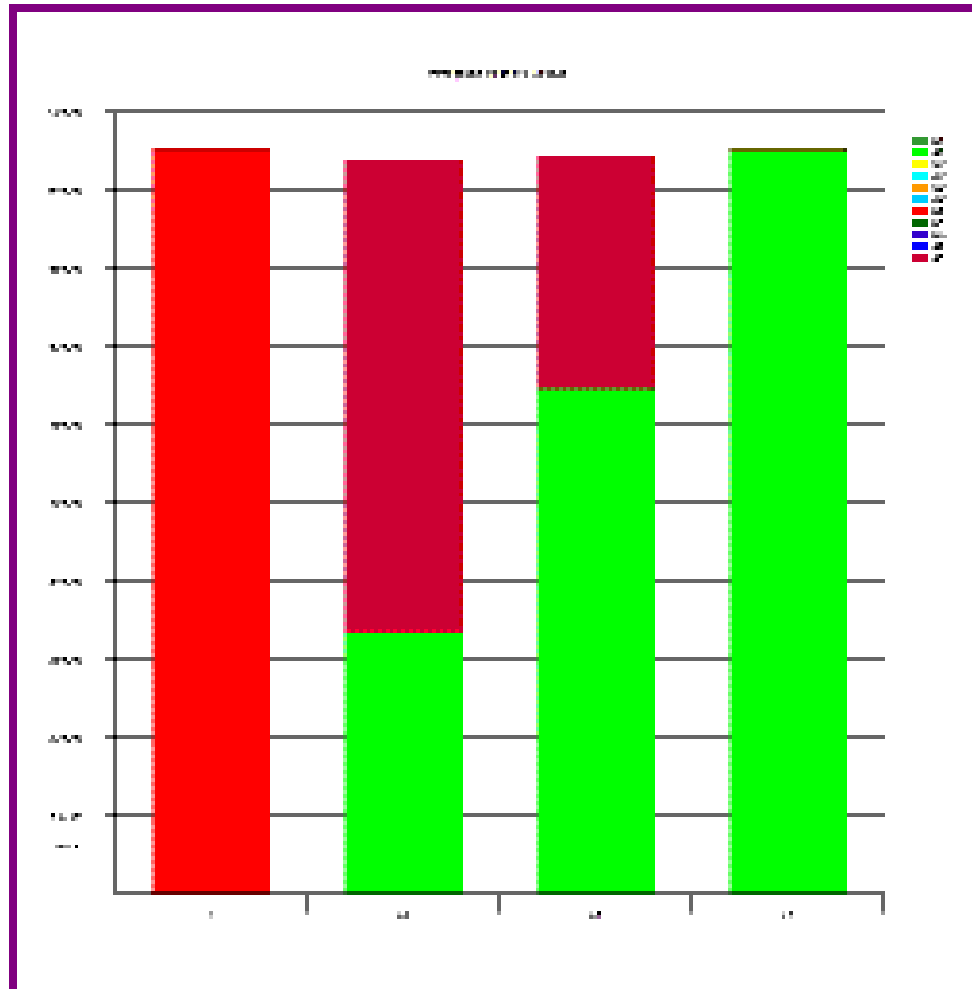


Figure 17-6 shows a small number of deltas, which is good, but also indicates that the partitions are doing different amounts of work.

Figure 17-6 An Uneven Processor Delta Time Totals Graph

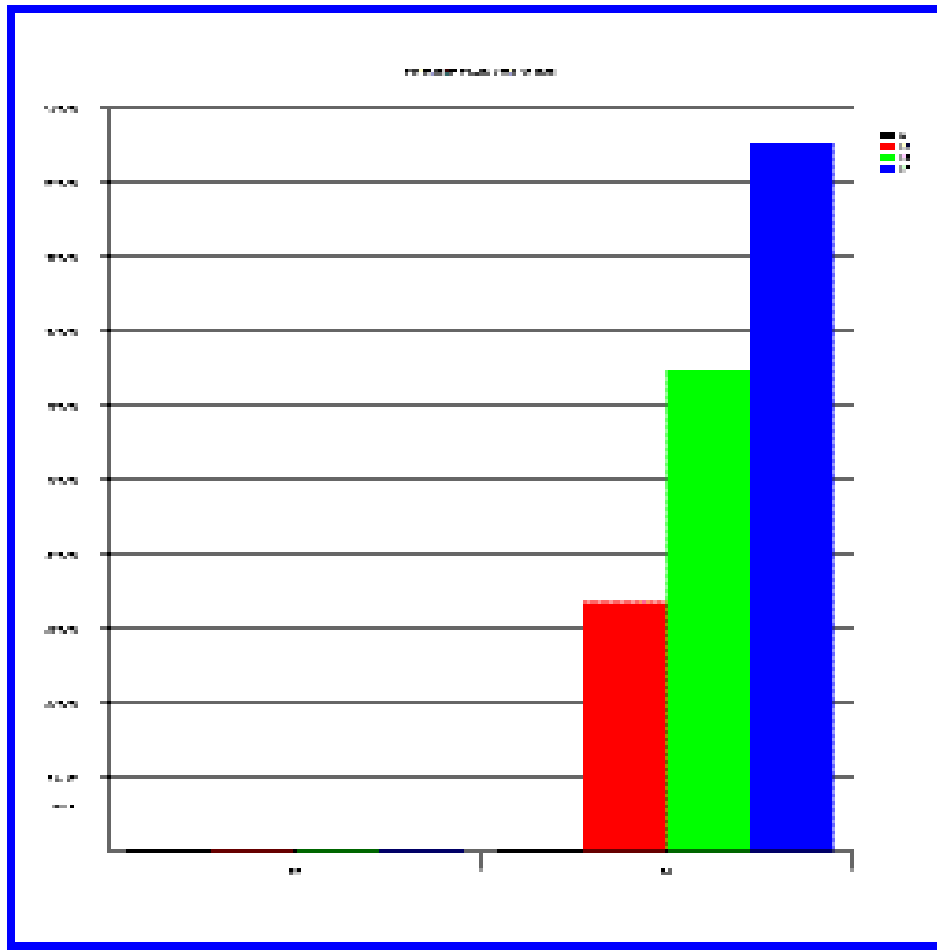
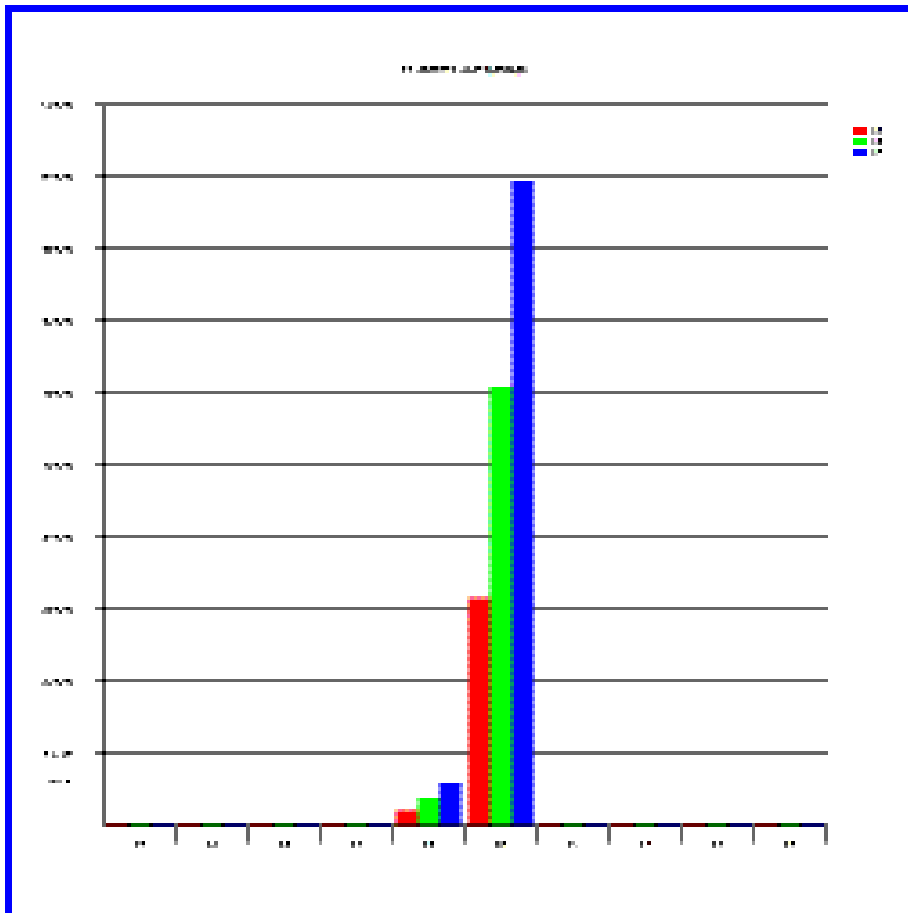


Figure 17-7 shows that the bars are more to the left, indicating less parallelism.

Figure 17-7 An Uneven S1 Balance Distribution



Example of a Design in Need of Clock Signal Analysis

The next four graphs are from a design in need of the clock signal optimization. Clock signal values propagate from the output of one partition to the input of another, causing each partition to wait for events in the other partitions.

Figure 17-8 Shows that the partitions are doing the same amount of work, but they are spending much of their time waiting for events in other partitions.

Figure 17-8 Clock Signals Processor Segment Totals Graph

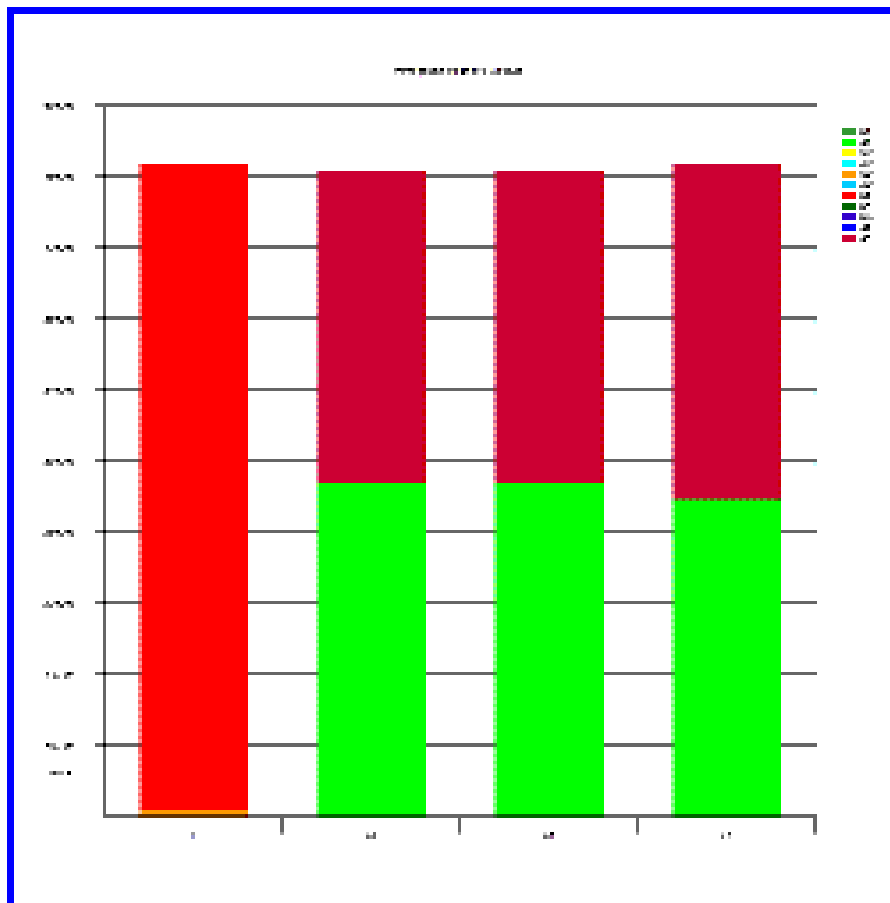


Figure 17-9 shows Partition A3 has a #1 delay before its activity

Figure 17-9 Clock Signal Processor Delta Time Totals Graph

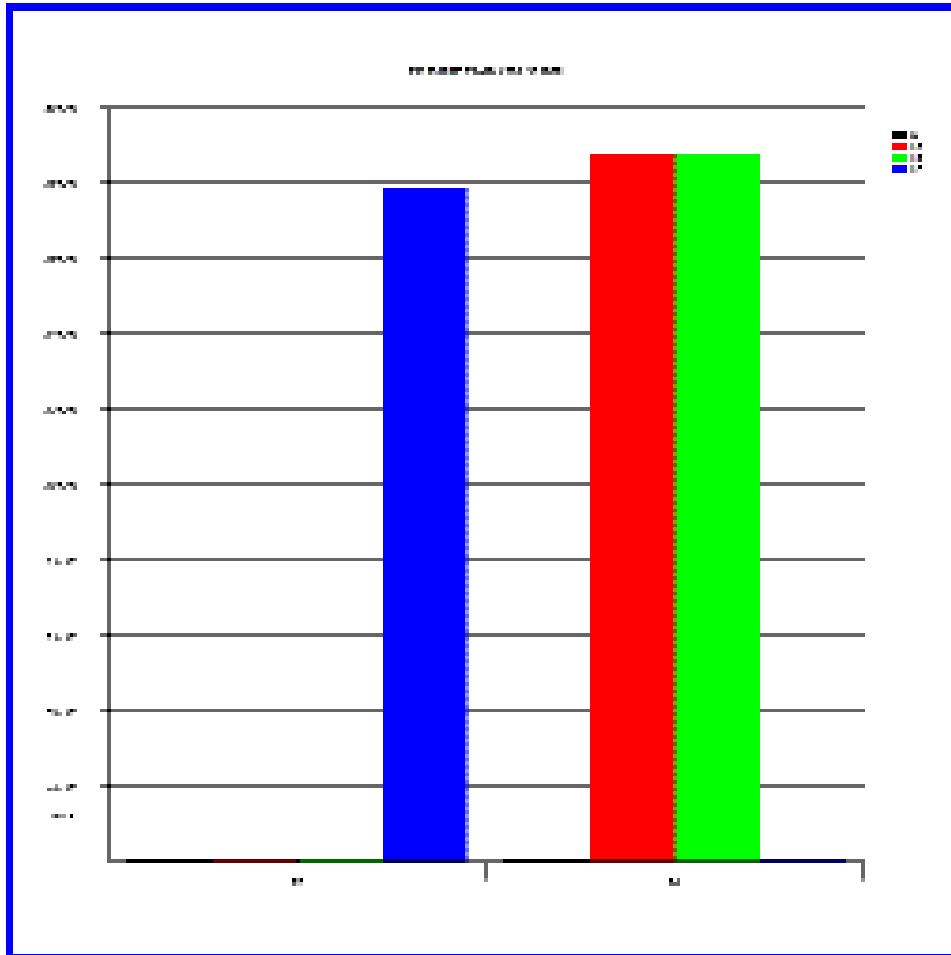
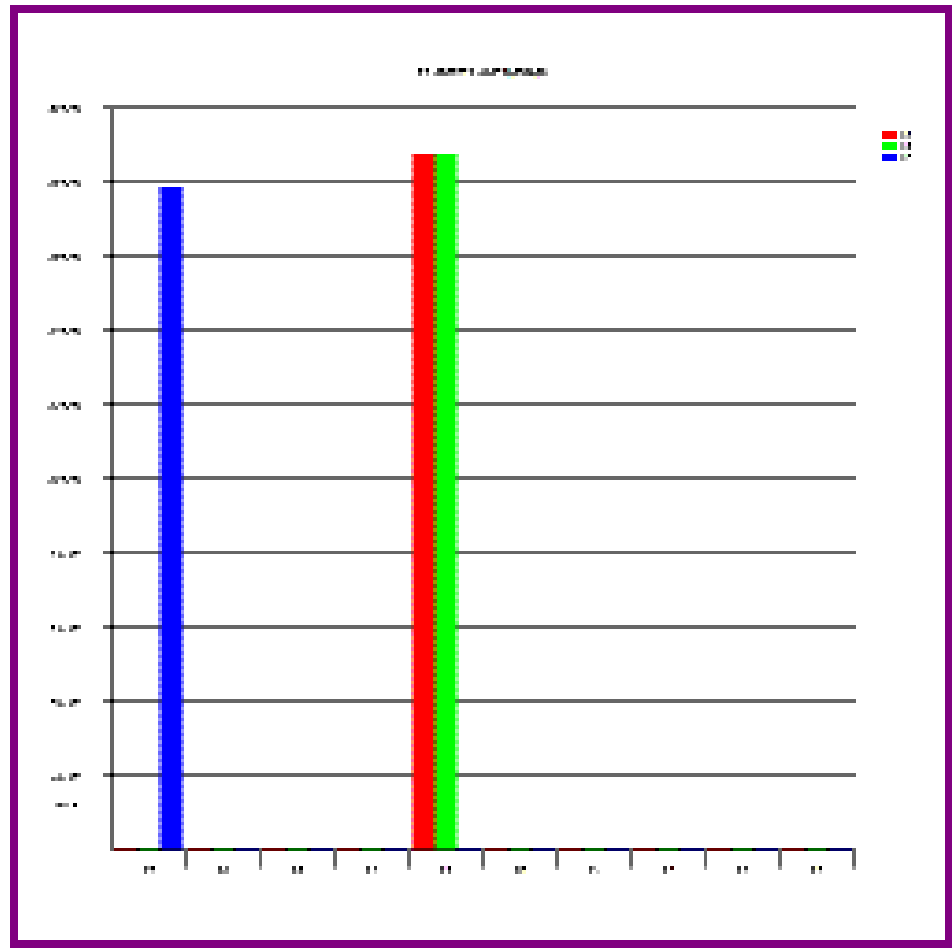


Figure 17-10 show workloads are balanced, but partition A3 does not execute in parallel with A1 and A2.

Figure 17-10 Clock Signal S1 Balance Distribution



Examples of Worst Case Scenario

The next four graphs are from a design in which the partitions get their clocks together. Workloads are balanced, but the partitions do not execute in parallel. This example is slower than a serial run.

Figure 17-11 Shows that the partitions are doing the same amount of work, but they are spending most of their time waiting for events in other partitions.

Figure 17-11 Clock Signals Processor Segment Totals Graph

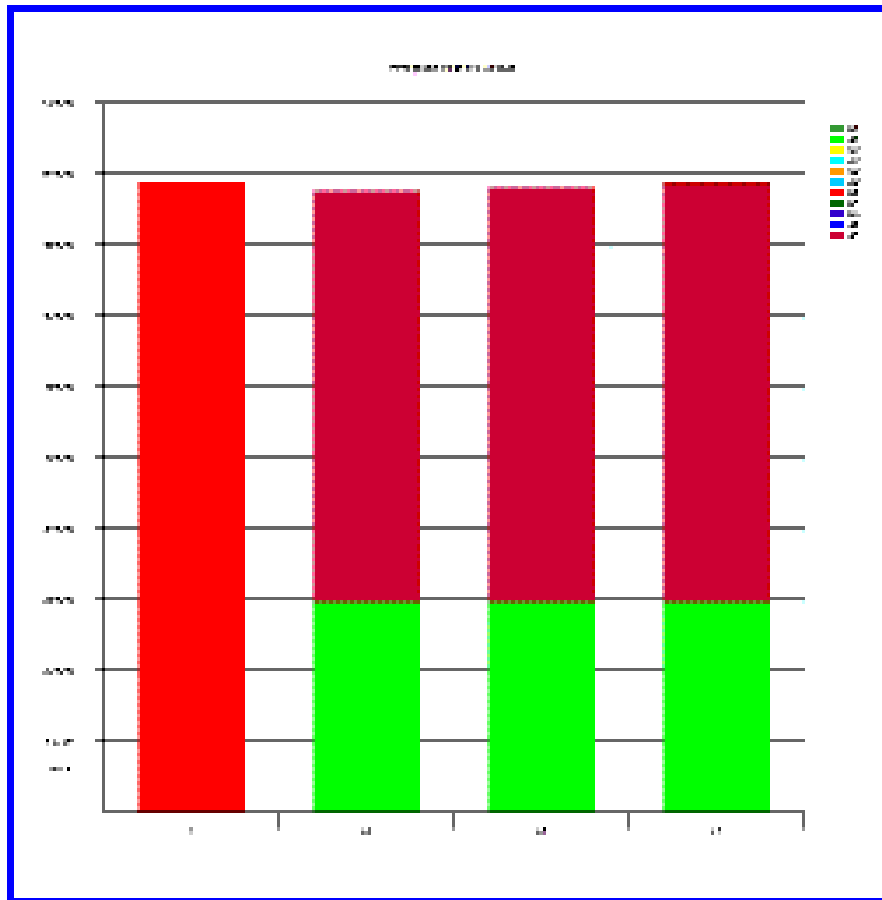


Figure 17-12 shows that all the work for each partition is done in a different delta.

Figure 17-12 Clock Signal Processor Delta Time Totals Graph

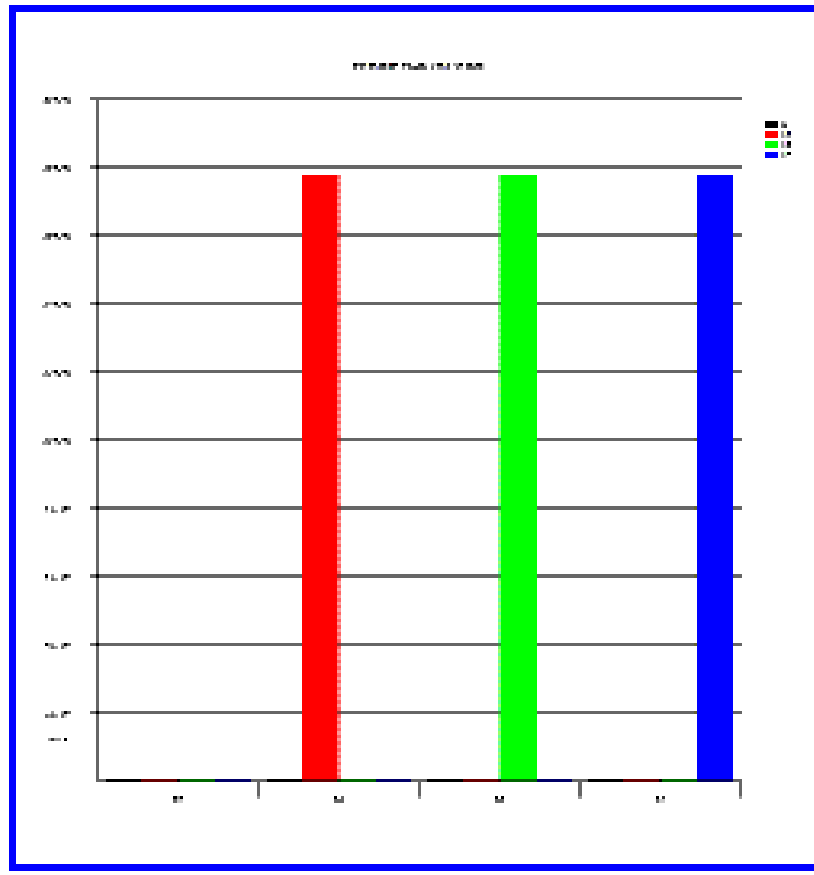
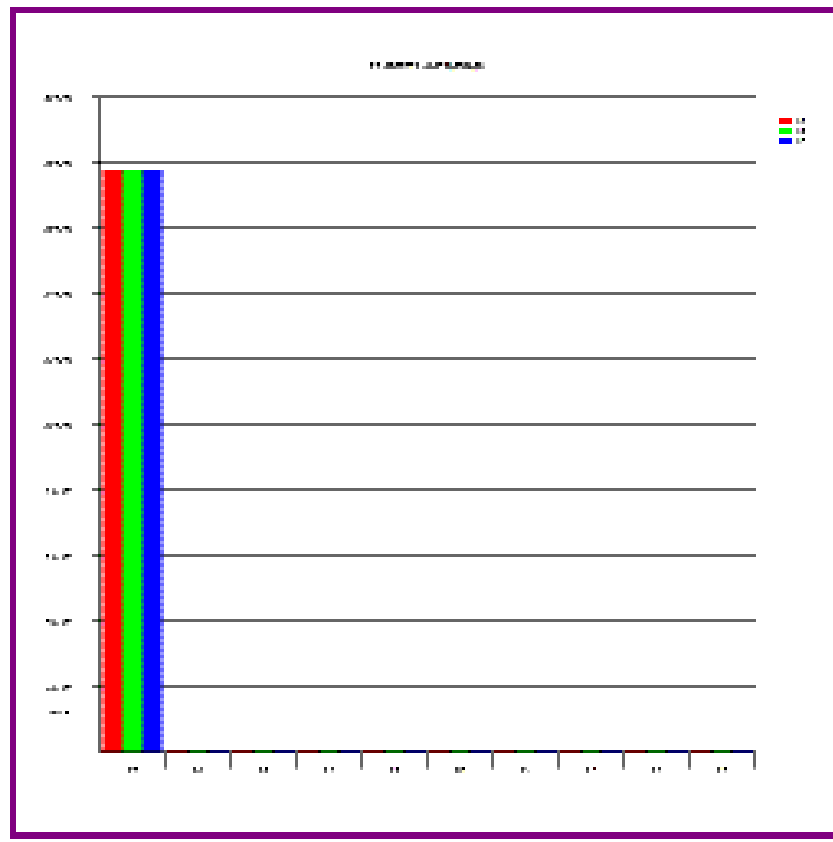


Figure 17-13 shows the bars completely to the left, indicating that there is no parallelism in the design.

Figure 17-13 Clock Signal S1 Balance Distribution



Running VCS Multicore Examples

Included in your installation is a design example and suite of scripts that allow you to perform serial / parallel performance comparisons on your system and profile the results. The tests include:

- A serial run
- A VCS Multicore run
- Profiler runs
 - A balanced workload VCS Multicore run
 - An unbalanced workload VCS Multicore run
 - A run with a delay in a partition causing one partition not to run in parallel with the other two partitions
 - A run in which a clock is passed from partition to partition causing non-parallel execution

Please contact VCS Support if you are looking for these examples.

Serial Run Time

The serial run example allows you to view wallclock time for comparison with the VCS Multicore run.

1. Run the script `run_serial.csh` .
2. Examine `DATE.serial` for the wallclock time taken for serial simulation as shown below.

```
Tue Oct 10 09:59:22 PDT 2006
Tue Oct 10 10:00:45 PDT 2006
```

VCS Multicore Run Time

The balanced VCS Multicore run example allows you to view wallclock time for comparison with the serial run.

1. Run the script `run_balanced.csh` .

2. Examine `DATE.parallel` for the wallclock time taken for VCS Multicore simulation as shown below.

```
Tue Oct 10 10:13:11 PDT 2006
Tue Oct 10 10:13:42 PDT 2006
```

Profiler Runs

Serial Run to Identify Partitions

This is example of a serial run used to identify partitions.

1. The script for the serial profiler run is `run_serial_prof.csh` .
2. Examine the INSTANCE VIEW section in `vcs.prof` to see that instances `top.A1_inst`, `top.A2_inst` and `top.A3_inst` are candidates for partitions, as shown in Figure 17-14.

Figure 17-14 Instance View from a Serial Run

```
=====
                        INSTANCE VIEW
=====
Instance                                     %Totaltime

top                                     (1)          100
top.A1_inst                           (2)           31
top.A2_inst                           (3)           29
top.A3_inst                           (4)           30
-----
```

VCS Multicore Runs

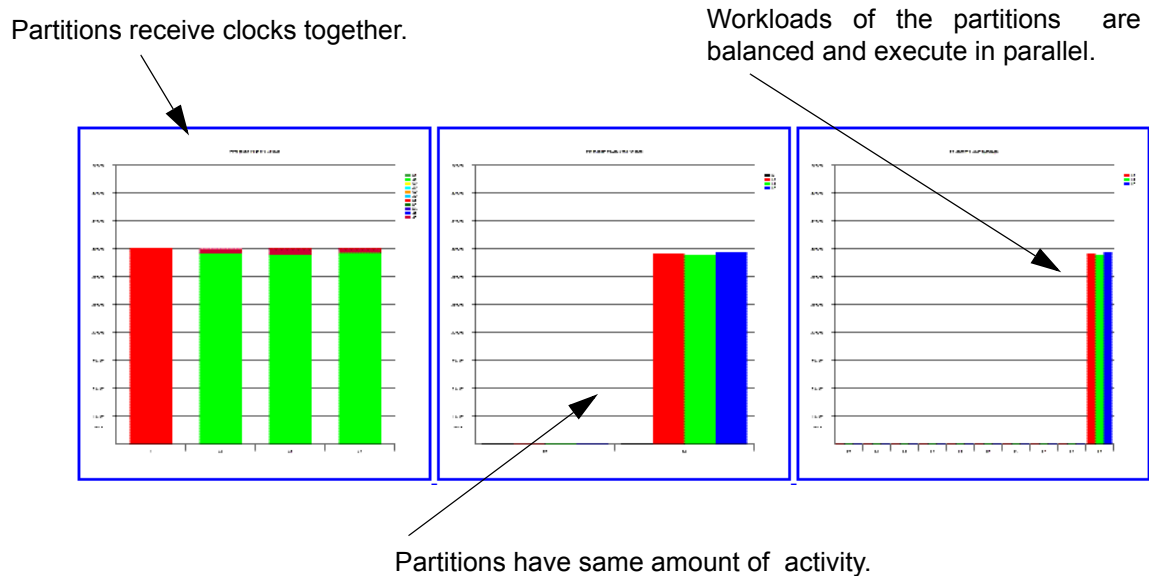
+define+BALANCED

This example is the best-case scenario.

1. Run the script `run_balanced_prof.csh` .

2. Examine the output in the `ppResults_balanced` directory by opening the file `results.html` as shown in Figure 17-15.

Figure 17-15 *Balanced VCS Multicore Run*



To view details, double-click on a graph in your browser to view the results in a full-screen.

In this example:

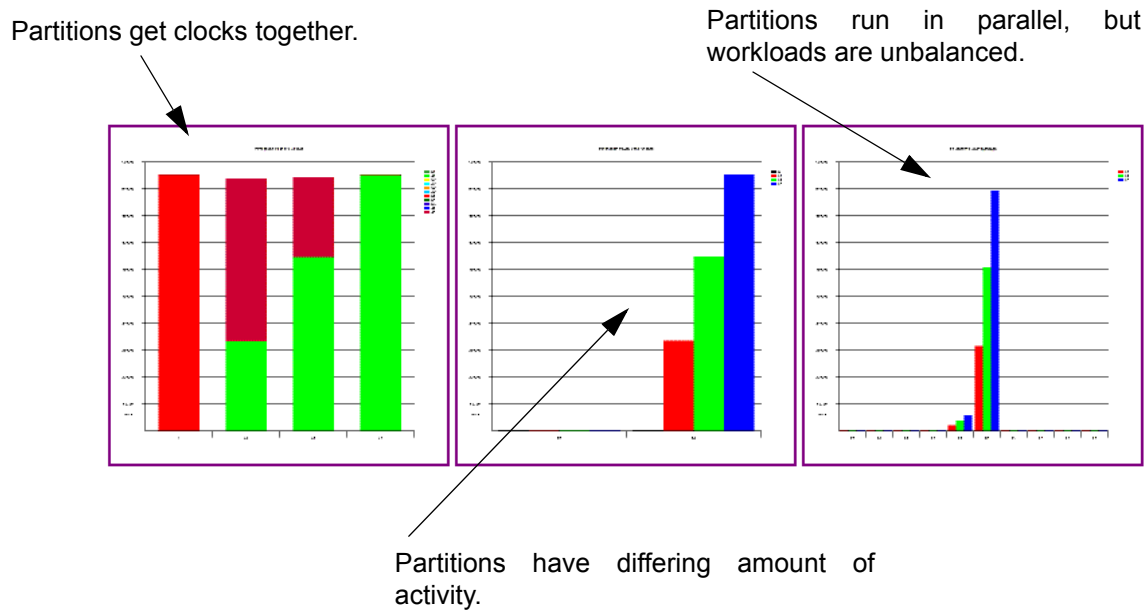
- On the left, the Processor Segment Totals graph shows that each partition is doing the same amount of work and the partitions receive their clocks together.
- In the center, the Processor Delta Time Totals Graph shows two deltas, `d0`, in which the design generated the clock signal, and `d1`, where each partition did its work in the same delta and has the same amount of activity.
- On the right, the Balance distribution graph shows that workloads of the partitions are balanced and execute in parallel.

+define+UNBALANCED

In this VCS Multicore run unbalanced workloads hinder performance.

1. Run the script `run_unbalanced_prof.csh`.
2. Examine the output is in the `ppResults_unbalanced` directory by opening the file `results.html` as shown in Figure 17-16.

Figure 17-16 Unbalanced VCS Multicore Run



In this example:

- The left chart, the Processor Segment Totals graph shows that the partitions are not doing the same amount of work. Some partitions are doing a significant amount of waiting for events in other partitions.
- The center graph, the Processor Delta Time Totals Graph, shows uneven processor delta times, indicating uneven work distribution among the partitions.

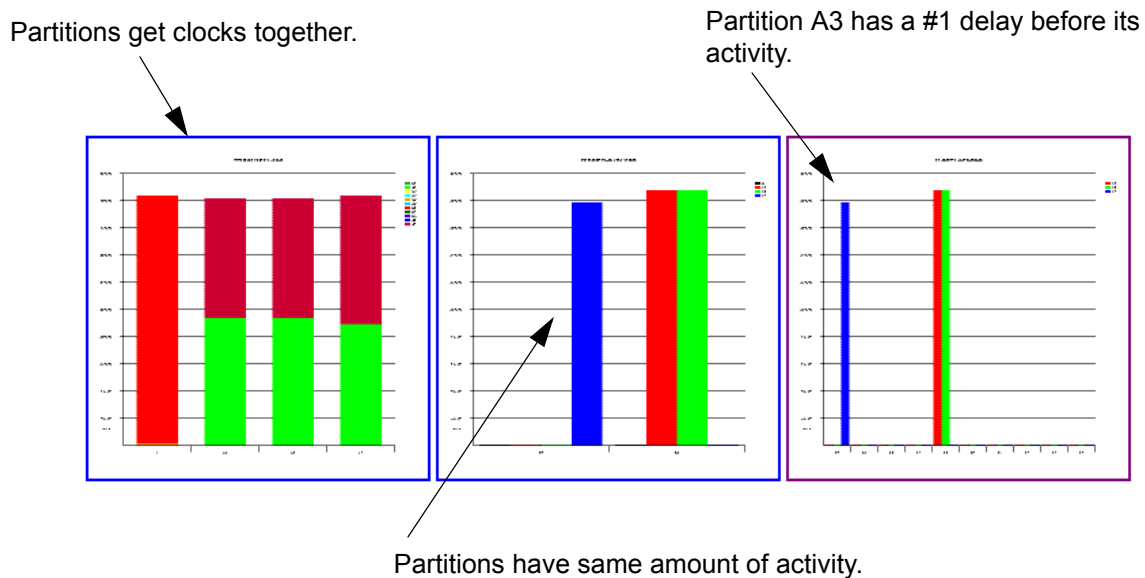
- On the right, the Balance distribution graph shows that partitions execute in parallel, but the workloads are unbalanced.

+define+BALANCED+DELAY

This is an example of a delay in a partition causing one partition not to run in parallel with the other two partitions.

1. Run the script `run_delay_prof.csh`.
2. Examine the output in the `ppResults_delay` directory by opening the file `results.html`. Figure 17-17 shows Partition A3 with a #1 delay before its activity. Hence workloads are balanced, but partition A3 does not execute in parallel with A1 and A2.

Figure 17-17 A Delay in a Partition



In this example:

- The left chart, the Processor Segment Totals graph shows that each partition is doing the same amount of work but are spends significant amounts of time waiting for events in other partitions.

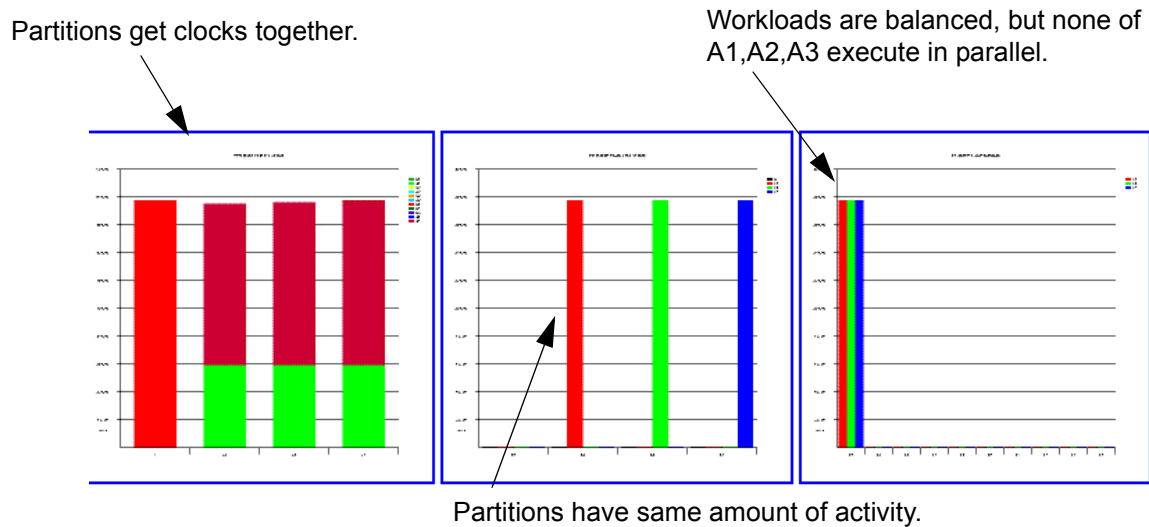
- The center graph, the Processor Delta Time Totals Graph, shows the partitions have the same amount of activity, but they execute across two deltas
- On the right, the Balance distribution graph shows that workloads of the partitions are balanced, but partition A3 does not execute in parallel with A1 and A2.

+define+BALANCED+CLOCK

This is an example of a clock being passed from partition to partition causing non-parallel execution. This is the worst-case scenario, and will be slower than the serial run.

1. Run the script `run_clock_prof.csh`.
2. Examine the output in the `ppResults_clock` directory by opening the file `results.html`. Figure 17-18 shows the example in which Partition A2 gets its clock from A1, and A3 gets its clock from A2.

Figure 17-18 A Clock Passed from Partition to Partition



In this example:

- The left chart, the Processor Segment Totals graph shows that each partition is doing the same amount of work but are spending most of the time waiting for events in other partitions.
- The center graph, the Processor Delta Time Totals Graph, shows the partitions have the same amount of activity, but the work for each partition is done in a different delta.
- On the right, the Balance distribution graph shows that workloads of the partitions are balanced, but none of the three partitions execute in parallel.

Supported Platforms

- Linux 32/64bit RH4.0 : Multi-core multi-processor machine.
- Solaris 32 bit: Multi-core Multi-processor machine.

Current Limitations

+race

+race is not supported

Partial Elab

This flow is not supported.

SystemVerilog

- Use of SV data types logic and bit types are allowed. Certain dynamic types are not allowed inside slave partitions.

VCS-MX

- Only Verilog instances can be added as partitions.
- All VHDL will run in the master partition.
For example, VHDL testbench and Verilog gate-level netlist will work very well.

VMC, SystemC-C, and AMS

These flows are not supported.

18

SystemVerilog Assertions

This chapter describes the implementation of LCA features for SystemVerilog Assertions. Many of these features are part of *IEEE P1800/D8-2009 Draft Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language* and detailed descriptions for them are available in the LRM.

Use Model

You must use the `-assert svaext` compile time option for all the new SVA features .

Note:

The `-assert svaext` option must be used at analysis phase (`vlogan`) for the UUM flow.

IEEE Std. 1800-2005 Compliant Feature

The following feature is implemented in compliance with the IEEE Std. 1800-2005 *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*. See the *SystemVerilog Language Reference Manual for VCS/VCS MX* for details.

- Use of local variables with property operators
- Recursive Properties

IEEE P1800-2009 Draft 8 Compliant Features

The following features are implemented in compliance with the *IEEE P1800/D8-2009 Draft Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language*. See that document for details.

- Overlapping operators in multiclock environment
- Deferred immediate assertions
- `weak` and `strong` sequence operators
- Implication and equivalence operators (`->` and `<->`)
- `until` operator in four variants
 - `until`
 - `until_with`
- `nexttime` operator in four variants

- nexttime property_expr
- nexttime [N] property_expr
- s_nexttime property_expr
- s_nexttime [N] property_expr
- **always operator in three variants**
 - always property_expr
 - always [cycle_delay_const_range_expression] property_expr
 - s_always [constant_range] property_expr strong
- **eventually operator in three variants**
 - eventually [constant_range] property_expr
 - s_eventually property_expr
 - s_eventually
- **followed-by operator (#-#, #-#)**
- **accept_on and reject_on abort conditions**
- **Inferred Value Functions**
 - \$inferred_clock
 - \$inferred_disable
 - \$inferred_enable

Note:

\$inferred_enable is a VCS extension to the Inferred functions and not a standard LRM feature.

- Local variable initialization and input
- Global clocking
- Global clocking past value functions
 - `$past_gclk(expression)`
 - `$rose_gclk(expression)`
 - `$fell_gclk(expression)`
 - `$stable_gclk(expression)`
 - `$changed_gclk(expression)`
- Global clocking future value functions
 - `$future_gclk(expression)`
 - `$rising_gclk(expression)`
 - `$falling_gclk(expression)`
 - `$steady_gclk(expression)`
 - `$changing_gclk(expression)`
- `let` operator

Limitations

This section describes known limitations.

Debug Support for New Constructs

Use `-assert dve` at compile/elab to enable debug for assertions. While basic debug support is available with this release, assertion tracing in DVE not supported completely. DVE provides information such as: `start_time`, `end_time` for every attempt and statistics for every assertion/cover. DVE also groups all signals involved in an assertion on tracing an attempt. However the extra "hints" that are provided for SVA 2005 constructs are not available for new constructs as of now.

UCLI support for new assertions is not fully qualified.

Note on Cross Features

Some of the features in new assertions have known limitations with cross feature support, such as Debug, Coverage. Please check with Synopsys support if there are unexpected results with cross feature behavior for these new constructs.

Some known issues:

- `-cm property_path` is not available for the new constructs
- New sequence operators when used as sampling event for covergroups may not function well.

19

Using HDL and SystemC Sync Loops

VcsSystemC enables you to simulate both HDL (Verilog, SystemVerilog, VHDL) and SystemC together. A sync loop drives the kernels of both HDL and SystemC parts and ensures that simulation events stay aligned. There are two different sync-loops to select from. They differ in simulation speed, accuracy of the alignment and other aspects.

The two sync loops are:

- The coarse-grained sync loop. This is the default.
- The fine-grained sync loop.

The Coarse-Grained Sync Loop

The default sync loop aligns HDL and SystemC at a coarse but efficient level. If there are multiple delta cycles on the SystemC side, then some or all of those SystemC delta cycles are executed consecutively before control is handed back to the HDL side. Similarly, multiple Verilog/VHDL delta cycles may happen before the next set of SystemC delta cycles will be started. This schema is efficient in terms of simulation time but the interaction between HDL and SystemC is difficult to predict.

The Fine-Grained Sync Loop

If a fine-grained and easy-to-predict alignment between HDL and SystemC is preferred, then use the fine-grained SC/HDL sync loop. This is done by specifying argument `-sysc=newsync` during elaboration, for example:

```
vcs -sysc ... -sysc=newsync ...
```

Run Time

The simulation speed may be affected by using the fine-grained sync loop. The difference depends on the individual design, so there is no simple rule-of-thumb. However, there is a general tendency that simulations will run slower when using the fine-grained sync loop.

Alignment of Delta Cycles

In the fine-grained SC/HDL sync loop, delta cycles of SystemC and Verilog are aligned. If at a given simulation time there are both SystemC and Verilog events present that span over multiple delta cycles each, then execution of events is aligned as follows:

1. Handle SystemC and Verilog events:
 - If SystemC events are present at current simulation time:
Execute one SystemC delta cycle;
 - If Verilog events present at current simulation time:
Execute all Verilog events at the current simulation time until there are only NBAs left;
2. Update all SystemC signals, execute all Verilog NBAs, and exchange all value updates between SystemC and Verilog;

The steps repeat until there are no more events at the current time, then proceed to the next simulation time. In short, SystemC delta cycles and Verilog NBAs are strictly aligned.

The order in which the step 1 operations are executed is not specified. However, step 2 happens only after both step 1 operations are done. The order should not matter because value updates are only done after both sides have finished their delta cycle. If there are no SystemC events in a specific delta cycle, then the SystemC event operation in step 1 is skipped. If there are no Verilog events exist then the Verilog event operation in step 1 is skipped.

If the simulation also contains VHDL processes, then VHDL and Verilog events are aligned as usual by VCS. This indirectly aligns VHDL and SystemC as well.

Example Syntax

```
vlogan verilog_dut.v verilog_top.v +define+VHDL_DUT

syscan -sysc=22 ./stimulus.cpp:stimulus ./
gen_clk.cpp:gen_clk -cflags "-g"

vhdlan vhdl_dut.vhd

vcs testbench -sysc=22 -sysc=newsync -debug_all -cflags "-g"
simv -ucli -i dump.tcl
```

Restrictions

The fine-grained SC/HDL sync loop is an LCA feature and has several restrictions:

- (IMPORTANT) No return from first `sc_start()` call: The execution flow will not return from the first call of `sc_start()`. All statements located after this call will never be executed. Furthermore, the program will never return from function `sc_main()`. This means that multiple calls to `sc_start()` are not supported. Also, class destructors may not be executed because VCS will call `exit()` before `sc_main()` returns.

Note that no warning is printed if there are multiple `sc_start()` calls or other important statements after the first `sc_start()` call.

- SystemC 2.2 must be used: an error is printed if another SystemC version is used.
- Pure SystemC mode (=no HDL modules) is not supported. An error is printed when the fine-grained sync loop is used in this situation.
- The time resolution between SystemC and HDL must match. An error is printed and the simulation is aborted during startup of simv when this restriction is violated.
- SystemC inout ports are not supported in combination with the fine-grained sync loop: no error message is printed and the simulation may hang.

Restrictions That No Longer Apply

The VCS slave-mode ("vcs -e ...") was never available with the default coarse sync loop. VCS slave-models are now available when the fine-grained sync loop is used

Index

A

- About the HVP Editor 2-60
- About VMM Planner 2-60
- Active Scope 1-53
- Adding User-Defined Attributes in HVP Editor 2-67
- annotated source code 1-21, 1-32

B

- Branch Coverage 1-32

C

- Cascade (Window menu selection) 1-54
- Close Database (File menu selection)
 - reference 1-51
- Close File (File menu selection)
 - reference 1-51
- Close Window (File menu selection)
 - reference 1-51
- color display, source window 1-49
- Condition Coverage Detail Window 1-26
- Console Pane 1-53, 1-54
- coverage 1-1

- coverage database, open 1-4
- Coverage Detail Window 1-16
- Coverage Map Window 1-15
- Coverage Settings Window 1-48
- Coverage Table Window 1-12
- coverage, viewing results 1-11
- Creating a Subplan in VMM Planner Editor 2-70

D

- Dock
 - Window menu selection 1-54
- DVE
 - starting 1-4

E

- Edit menu, coverage 1-51
- Edit menu, reference 1-51
- Edit Parent 1-53
- Edit Source 1-53
- exclusion, commands 12-194
- Execute Tcl Script (File menu selection)
 - reference 1-51
- Exit (File menu selection)
 - reference 1-51

F

- File menu
 - , coverage 1-51
- File menu, reference 1-51
- File Toolbar 1-52, 1-55
- filter coveredisplay 1-47
- Find (Edit menu selection)
 - reference 1-52
- FSM transition mode 1-28

G

- groups, creating coverage 1-18

I

- instance, displaying results by 1-13
- invokingDVE 1-4

L

- line coverage, displaying 1-21
- load coverage session 1-9
- load exclusion state 12-205
- Load Session (File menu selection)
 - reference 1-51
- Loading and Saving Sessions 1-9

M

- map window, coverage 1-15
- module, , displaying coverage results by 1-14

N

- net and register coverage results 1-23

O

- Open Database

- Toolbar icon 1-55

- Open Database (File menu selection)
 - reference 1-51

- Open File (File menu selection)
 - reference 1-51

- Overview 1-2

Q

- Quick Start Example
 - Toolbar icon 1-56, 1-57

R

- recalculate the coverage 12-196
- Running a Quick Start Example 1-4

S

- save exclusion state 12-204
- Save Session (File menu selection)
 - reference 1-51
- Scope menu, coverage 1-53
- sequences view, FSM 1-29
- Show 1-53
- Show Source 1-53
- Simulate Toolbar 1-52
- source code, displaying 1-21, 1-32
- Source Pane
 - Toolbar icon 1-56
- start coverage mode 1-4
- starting DVE 1-4
- SVA coverage 1-33
- SVA PP, SystemVerilog Assertions Post-processing, SVAPP, Post-processing SVA 5-109
- SVAPP Limitations, Limitations SVAPP 5-115
- SVAPP syntax, syntax for SVAPP 5-110

T

- TCL scripts 1-46
- Tile (Window menu selection) 1-54
- Toggle tab 1-23
- Toolbars 1-52, 1-55
- toolbarToolbar 1-55

U

- Undock 1-54
- user-defined coverage groups 1-18
- Using the VMM Planner Editor 2-59, 6-119

V

- View menu, coverage 1-52, 1-55
- View menu, reference 1-52
- Viewing Coverage Results 1-11

W

- Window Menu
 - reference 1-53
- Window menu, coverage 1-53
- Window Toolbar 1-52, 1-53
- Working with HVP Files 2-61