

Coverage Technology

User Guide

G-2012.09
September 2012

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

SYNOPSYS®

Copyright Notice and Proprietary Information

Copyright © 2012 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

"This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____. "

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AEON, AMPS, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, CoMET, Confirma, CODE V, Design Compiler, DesignWare, EMBED-IT!, Formality, Galaxy Custom Designer, Global Synthesis, HAPS, HapsTrak, HDL Analyst, HSIM, HSPICE, Identify, Leda, LightTools, MAST, METeor, ModelTools, NanoSim, NOVeA, OpenVera, ORA, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Sonic Focus, STAR Memory System, Syndicated, Synplicity, the Synplicity logo, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, ARC, ASAP, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomExplorer, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclypse, Encore, EPIC, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIM^{plus}, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Intelli, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Macro-PLUS, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, ORAengineering, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, RippledMixer, Saturn, Scirocco, Scirocco-i, SiWare, Star-RCXT, Star-SimXT, StarRC, System Compiler, System Designer, Taurus, TotalRecall, TSUPREM-4, VCSI, VHDL Compiler, VMC, and Worksheet Buffer are trademarks of Synopsys, Inc.

Service Marks (sm)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.

ARM and AMBA are registered trademarks of ARM Limited.

Saber is a registered trademark of SabreMark Limited Partnership and is used under license.

All other product or company names may be trademarks of their respective owners.

Contents

1. Introducing Coverage Technology	1-1
Coverage Metrics Overview	1-2
Coverage Flow	1-3
Types of Coverage	1-4
Line Coverage	1-4
Line Coverage for Verilog	1-5
Line Coverage for VHDL	1-11
Toggle Coverage	1-12
Condition Coverage	1-14
Verilog Conditional Coverage Not Monitored	1-17
VHDL Conditional Coverage Not Monitored	1-18
Stable Names for Conditions in a Design	1-19
Naming a Condition in a Module	1-20
Advantages of Assigning Stable Names for Conditions	1-20
Specifying Tags in the Source	1-21
Multiple Conditions in a Statement	1-21
Multiple Statements on a Line	1-22
Names in the Generated Reports	1-23

EI File for Condition Coverage	1-24
Limitations of Stable Names for Conditions	1-25
FSM Coverage	1-25
FSM Coverage for Verilog	1-27
FSM Coverage for VHDL	1-29
Branch Coverage	1-31
Functional Coverage	1-33
Generating the Coverage Database	1-34
Generating Coverage Reports	1-34
Unified Report Generator	1-34
DVE Coverage GUI	1-35
Using Unified Coverage API	1-36
Post-processing Techniques	1-37
2. Generating Coverage Database	2-1
Unified Coverage Database	2-1
How to Obtain Coverage Data for your Design	2-2
Compiling Design Files	2-2
Running the Simulation and Monitoring Coverage	2-3
Viewing Coverage Reports	2-4
Coverage for Protected Modules	2-5
Generating Coverage Metrics	2-5
3. Viewing the Coverage Report Using Unified Report Generator ..	3-1
Supported Metrics	3-2

Invoking URG	3-3
Merging Coverage Results of Several Runs of the Same Executable	3-5
Merging Coverage Database Across Versions of VCS	3-5
Format to Display Coverage Results.....	3-6
What is Covered in Each Coverage Metrics	3-8
The Line Coverage Report	3-9
The Toggle Coverage Report.....	3-12
Support for Interface Signals on Port Boundary	3-13
Excluding Interface Signals from Design	3-17
Interface Expansion Limitations	3-18
The Condition Coverage Report	3-20
Condition Coverage Report Splitting for Large Number of Conditions.....	3-23
The FSM Coverage Report	3-25
The Branch Coverage Report	3-27
The Assertion Coverage Report	3-34
Summary Tables.....	3-34
Detail Report for Assertions	3-35
Detail Report for Cover Sequence	3-38
Detail Report for Cover Properties	3-39
The Covergroup Report.....	3-39
Viewing Results for Coverage Group Variants	3-42
Understanding Covergroup Page Splitting	3-43
Correlation Report: Which Tests Covered Which Coverage Bins	3-52
Covered Objects.....	3-54
Tests Page	3-56

Unsupported Arguments	3-57
Difference Reports for Functional Coverage	3-57
Diff Results Shown in the Dashboard and Test Pages	3-59
What is Shown as Covered	3-60
Report for Default Mode (-diff or -diff object)	3-60
Report for Count Mode (-diff count)	3-61
Unsupported Flags	3-62
Exclusions	3-62
Reporting Only Uncovered Objects (-show brief)	3-62
Brief Report for Line Coverage.	3-63
Brief Report for Condition Coverage:....	3-65
Brief Report for FSM Coverage	3-66
URG Support for Uncovered Only (–show brief) Feature on Selected Metrics	3-67
Uncovered Only Examples	3-69
Report Changes.....	3-73
Dashboard	3-74
Module List.....	3-74
Hierarchy	3-75
Groups	3-76
Tests.....	3-76
HVP	3-76
Assertions Report.....	3-77
Module and Instance Header Sections	3-77
Summary Only URG Reports	3-78
Use Model	3-78
Coverage Report Files	3-79

Common Report Elements	3-80
The Dashboard File	3-81
The Hierarchy File	3-83
The Modlist File	3-85
The Groups File	3-86
The modN File	3-87
Module with Parameters.	3-90
The grpN Files	3-90
The Asserts File	3-92
The Tests File.	3-92
Analyzing Trend Charts	3-93
Quick Overview	3-93
Generating Trend Charts	3-94
Customizing Trend Charts	3-97
Navigating Trend Charts.	3-102
Trend Chart Linkage	3-104
Organization of Trend Charts	3-105
Top-Level Chart	3-105
Metric-Wide Breakdown Linkage	3-108
Hierarchical Linkage.	3-113
Links to Previous Sessions	3-116
4. Viewing Coverage Reports Using the DVE Coverage GUI	4-1
Starting the DVE in Coverage Mode	4-2
Opening a Database	4-2
Loading Multiple Coverage Tests Incrementally	4-8
Loading and Saving Sessions	4-10

Saving a Session	4-10
Loading a Session	4-11
DVE Coverage Source / Database File Relocation	4-12
Using the Coverage GUI	4-13
The Navigation Pane	4-14
The Coverage Summary Table Window	4-15
The Coverage Summary Map Window	4-17
The Coverage Detail Window	4-18
Navigating in the Source Pane	4-20
Creating User-Defined Groups	4-21
Displaying Code Coverage	4-23
Displaying Line Coverage	4-23
Displaying Toggle Coverage	4-24
Displaying Nets and Registers	4-24
Displaying Multi-Dimensional Arrays	4-26
Displaying Condition Coverage	4-27
Viewing Condition IDs	4-28
DVE Coverage with Condition Coverage Observability	4-29
Displaying Finite State Machine (FSM) Coverage	4-30
Displaying Transition Details	4-31
Displaying Sequences	4-32
Displaying Branch Coverage	4-34
The Branch Coverage Detail window	4-35
Displaying Implied Branches	4-36
Displaying Assertion Coverage	4-36
SVA Naming Convention	4-38
OVA Naming Convention	4-38

The Covers Tab	4-39
Displaying Testbench Coverage	4-41
Filtering Instances with No Coverable Objects	4-45
Working with HVP Files	4-49
Working with Coverage Results	4-50
Running Scripts	4-50
Filtering the Hierarchy Display	4-51
Generating URG Report from DVE Coverage GUI	4-52
 5. Coverage Post-processing Techniques.....	5-1
Merging	5-1
Serial Merging	5-3
Parallel Merging.....	5-3
Specifying the Machines that Perform the Jobs	5-6
Using a GRID Computing Engine.....	5-7
Using LSF.....	5-7
Specifying the Number of Tests in a Merging.....	5-8
Flexible Merging	5-8
Merge Equivalence.....	5-9
Rules for Flexible Merging Databases	5-10
Example	5-11
Flexible Merge Database	5-14
Grading and Coverage Analysis	5-15
Examples Using the Grading Option	5-15
Scoring	5-15
Quick Grading	5-16

Greedy Grading	5-17
Grading and the <code>-scorefile</code> Option.....	5-18
Using Index-Based Test Grading for Code Coverage	5-20
Limitations of Index-Based Grading Feature	5-22
Difference Reports.....	5-23
Showing the diff Tests in Dashboard and Test Pages	5-24
Objects Covered in the diff Report.....	5-25
Differences in diff and Regular Report.....	5-27
Unsupported flags	5-27
Support for Exclusion.....	5-28
Determining Which Test Covered an Object Report	5-28
Covered Objects.....	5-29
Tests Page	5-31
Unsupported Arguments	5-31
Resetting or Deleting Covergroup in the Database	5-32
Using the Reset Command	5-33
Using the Delete Command.....	5-34
Resetting and Deleting Assertion Coverage	5-34
Using the Reset Command	5-34
Using the Delete Command.....	5-35
Mapping Coverage	5-35
Using <code>-map</code> for Assertion Coverage Mapping	5-39
Mapping Subhierarchy Coverage in VHDL	5-40
Understanding Instance-Based Mapping.....	5-41
Examples of Instance-Based Mapping.....	5-43
Using <code>-mapfile</code> for Assertion Coverage Mapping	5-44
Using Multiple <code>-map</code> Options	5-46

Mapping Relocated Source File.....	5-47
Exclusion.....	5-48
Exclusion Using DVE Coverage GUI.....	5-49
Coverage Exclusion with DVE	5-49
Toggle Coverage MDA Exclusion	5-72
Elf file Syntax for MDA Exclusion.....	5-73
Limitations	5-74
Excluding Covergroups	5-74
Cross-of-a-Cross Exclusion Support.....	5-79
Excluding Cross-of-a-Cross.....	5-79
Enhanced Syntax of Covergroup Exclusion.....	5-82
Generating URG Reports Using Exclude Files	5-83
Exclusion in Strict Mode	5-85
Covergroup Exclusion in URG	5-85
Editing Exclude Files	5-86
Exclusion Annotations in URG	5-97
Exclusion Mark in URG	5-98
Exclusion in UCAPI	5-102
Hierarchy Exclusion Files.....	5-102
Exclusion by Object Handle	5-102
Default vs. Strict Exclusion.....	5-104
Editing the Covergroup Coverage Database.....	5-105
db_edit_file Syntax	5-105
Resetting Covergroups or Coverpoints	5-106
Removing Covergroups from the Database.....	5-107
Editing the Assertion Coverage Database.....	5-107

Resetting the Coverage Scores for Assertions	5-108
Removing Assertions from the Coverage Database	5-108
6. Unified Coverage API	6-1
Overview	6-1
Database Contents	6-4
Persistence	6-4
UCAPI Setup and Compilation Requirements	6-5
Enabling Error Reporting	6-7
Dynamic UCAPI Library	6-8
Data Model	6-8
UCAPI Object	6-9
Common Properties	6-9
Coverable Objects	6-12
Block	6-13
Value Set	6-13
Integer and Scalar values	6-15
Interval values	6-15
BDD values	6-16
Vector values	6-16
Value Set	6-19
Sequence	6-20
Cross	6-23
Stable Names for Coverable Objects	6-23
Naming the Next Statement in a Module	6-24
Using Stable Names for Coverable Objects	6-25
Specifying Tags in the Source	6-25
Multiple Conditions in a Statement	6-26

Multiple Statements on a Line	6-27
Names in the Generated Reports	6-27
El File for Condition Coverage	6-29
Design Objects	6-29
Design and Test Name	6-29
Tests and Metrics	6-30
Source Definition and Source Instance	6-31
Test-qualified Source Regions	6-40
Other Objects	6-42
Container	6-42
Predefined Coverage Metrics	6-44
Line Coverage	6-44
Condition Coverage	6-47
Branch Coverage	6-53
Finite State Machine Coverage	6-59
Toggle Coverage	6-63
Assertion Coverage	6-67
Testbench Coverage	6-76
Setting Hit Count on Bins	6-77
Usage	6-78
Set covdbStatusCovered	6-78
Contents of the Bin Tables	6-79
Limitations	6-84
Loading a Design	6-84
Available Tests	6-85
Loading Tests	6-85

Coverage Correlation : Determining Which Test Covered an Object	6-87
--	------

1

Introducing Coverage Technology

This chapter contains the following sections:

- “Coverage Metrics Overview”
- “Coverage Flow”
- “Types of Coverage”
- “Generating the Coverage Database”
- “Generating Coverage Reports”
- “Post-processing Techniques”

Coverage Metrics Overview

VCS can monitor and evaluate the coverage metrics of Verilog, VHDL, and mixed HDL designs during simulation to determine which portions of the design have not been tested. The results of the analysis are reported in a number of ways that allow you to see the shortcomings in your testbench and improve tests as needed to obtain the complete coverage. Functional coverage is the determination of how much functionality of the design has been exercised by the verification environment.

VCS writes both code and functional coverage databases during the simulation. Using VCS, you can generate the following two types of coverage databases:

- **Code Coverage**

You can get two types of code coverage information.

- The first is control-flow coverage that tracks which lines and branches have taken the flow of execution through the Verilog or VHDL code. Control-flow metrics include line coverage and branch coverage.
- The second type is value coverage, which monitors what values signals and expressions take on during simulation. Value-coverage metrics include toggle coverage and FSM coverage, which track the values (or value-transitions) for signals and variables.

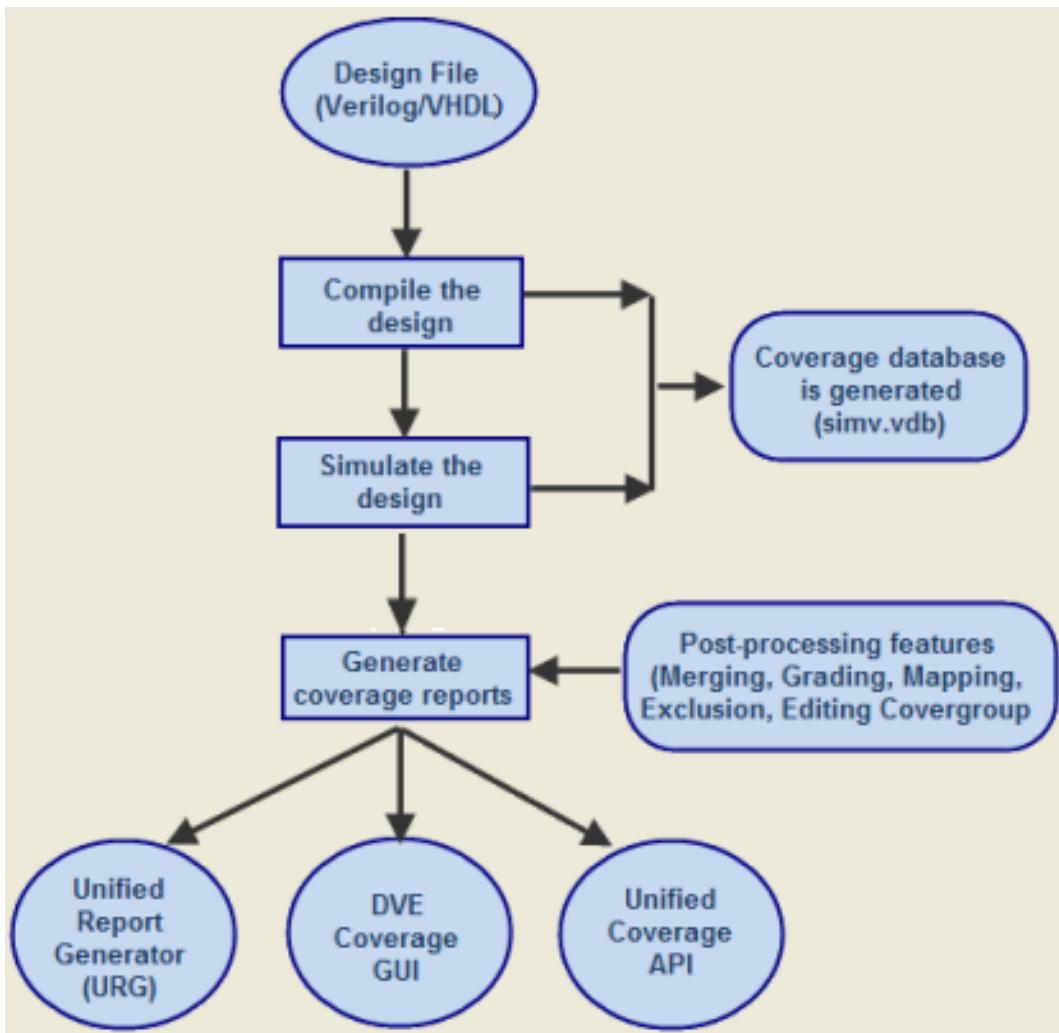
- **Functional Coverage**

Functional coverage provides you with information of your testbench variable and signal values and their state transitions. You can enable cross-coverage between variables and signals.

The VCS implementation of SystemVerilog supports the covergroup construct, which you specify as the user. These constructs allow the system to monitor values and transitions for variables and signals. They also enable cross-coverage between variables and signals.

Coverage Flow

Following is a high level picture of the VCS Coverage flow:



The Coverage Technology User Guide is organized based on this flow.

Types of Coverage

This section provides introductory descriptions of the following types of coverage metrics.

- “Line Coverage”
 - “Toggle Coverage”
 - “Condition Coverage”
 - “FSM Coverage”
 - “Branch Coverage”
 - “Functional Coverage”
-

Line Coverage

Line coverage (or statement coverage) shows you which lines of code are exercised — and which ones are not — by your testbench during a simulation run. VCS generates metrics for total coverage of lines, blocks, and branches that tell you how much of your code is actually executed by the tests. Line coverage is applied to signal and variable assignments in HDL code and gives an indication of the number of times each assignment statement was executed when the design was simulated. A zero execution count pin-points a line of code that has not been exercised that could be the source of a potential design error.

Line coverage also produces annotated listings, which identify the unexercised statements, therefore, giving you the information you need to write more tests to complete the coverage. However, coverage reports do not tell you how many times a line of code is executed.

In line coverage, VCS keeps track of the following in the source code:

- Individual procedural statements
- Procedural statement blocks
- Procedural statement block type
- Missing (implied) conditional statements

Note:

- Statement blocks consist of a set of individual procedural statements executed sequentially and always executed together in the same time step.
- By default, line coverage does not keep track of continuous assignment statements, but you can enable it; see “[Enabling Line Coverage for Continuous Assignments](#)” .

Line Coverage for Verilog

VCS tracks the following types of Verilog procedural statements:

- Statements that cause a simulation event, such as a procedural assignment statement, or a system task.
- Statements that control which other statements are executed, such as a `while` or an `if` statements.

The following code example illustrates what is covered in Verilog:

Example 1-1 Monitored Statements in Line Coverage

```
module top;
reg clk;
reg [8:0] data;
wire [8:0] results;

initial
begin
    $display("Assigning initial values");
    clk=0; data=256;
    #100 $finish;
end

always
#3 clk=~clk;

dev dev1 (results,clk,data);

endmodule

module dev (out,clock,in);
output [8:0] out;
input clock;
input [8:0] in;
reg [8:0] out;
reg en;

task splitter;
input [8:0] tin;
output [8:0] tout;
begin
    tout = tin/2;
end
endtask

always @ (posedge clock)
begin
if (in % 2 !== 0)
    $display("error cond, input not even");
```

```

else
    out = in;
if (in % 2 == 0)
en =
    1;
while (en)
forever
    case (out % 2)
        !0 : #0 $finish;
        0  : #5 splitter(out,out);
    endcase
end
endmodule

```

In [Example 1-1](#), the boldface lines contain statements that VCS tracks for line coverage.

VCS line coverage also includes information about always and initial blocks and other types of blocks of code. In line coverage, a block is a nesting level in the code. It is more of a level of control of the execution of the procedural code. Typically, you change the indentation of your source code for a nesting level. [Example 1-2](#) illustrates the code blocks in the source code in [Example 1-1](#). It is interrupted in a number of places to explain the blocks.

Example 1-2 Code Blocks in Line Coverage

```

1 module top;
2     reg clk;
3     reg [8:0] data;
4     wire [8:0] results;
5
6 initial
7 begin
8     $display("Assigning initial values");
9     clk=0; data=256;
10    #100 $finish;
11 end

```

The initial block is a distinct block of code in this top-level module. Inside the initial block is a procedural delay on the \$finish system task. This delay and system task constitutes a separate block of code, so that if simulation ends before time 100, the other statements in the initial block can be covered, just not the \$finish system task.

Notice that the assignment statements to regs clk and data are on the same line. This illustrates the difference between line and statement coverages. In this case, line 9 is covered only if both the statements clk=0; and data=256; are covered.

Also, in terms of the statement coverage, the procedural delay is considered a separate wait statement in addition to the \$finish system task on that line. Here #100 and \$finish are different statement present in one line. So the initial block above contains five statements in three lines.

```
always
#3 clk=~clk;

dev dev1 (results,clk,data);

endmodule
```

The always block that contains an assignment statement with a procedural delay is one block instead of two.

```
module dev (out,clock,in);
output [8:0] out;
input clock;
input [8:0] in;
reg [8:0] out;
reg en;
```

```

task splitter;
  input [8:0] tin;
  output [8:0] tout;
begin
  tout = tin/2;
end
endtask

```

The procedural statements in a task definition (in this case, one statement) are a separate block.

```

always @ (posedge clock)
begin
  if (in % 2 != 0)
    $display("error cond, input not even");
  else
    out = in;
  if (in % 2 == 0)
    en =
      1;
  while (en)
    forever
      case (out % 2)
        !0 : #0    $finish;
        0  : #5    splitter(out,out);
      endcase
    end
  endmodule

```

A block for this `always` block contains three statements: `if-else`, `if`, and `while` statements.

The `$display` system task is controlled by the `if` construct of the `if-else` statement and is in a separate block.

The procedural assignment to reg `out` is a separate block because it is controlled by the `else` construct.

The procedural assignment to reg `en` is a separate block because it is controlled by the second `if` construct. This is one statement on two lines.

The `forever` statement is in a separate block because it is controlled by the `while` statement.

The `case` statement is in a separate block because it is controlled by the `forever` statement.

The case item statements are separate blocks and the procedural delays on these case item statements are considered separate `wait` statements. There can also be blocks for missing constructs. If you use an `if` statement instead of an `if-else` statement, VCS considers the `else` construct to be missing. If you use a `case` statement and do not enter a default case, VCS considers the default case to be missing.

Enabling Line Coverage for Continuous Assignments

By default, line coverage does not include Verilog continuous assignments. You can enable line coverage for continuous assignments with the `-cm_line` `contassign` compile-time option and keyword argument. For example:

```
vcs -f design_files -cm line -cm_line contassign
```

When you include this compile-time option and keyword argument, URG shows the continuous assignment statement, including any delay specification, as a block of code.

If a continuous assignment assigns a constant value, VCS does not compile or monitor it for line coverage. For example:

```
module dev;
  wire w1,w2,w3;
  reg r1,r2;
  .
  .
  .
  assign #5 w1=1;
  assign #3 w2=r1;
  assign #10 w3=r1 && r2;
endmodule
```

VCS compiles and monitors the continuous assignments to w2 and w3 for line coverage, but not to w1.

Line Coverage for VHDL

In line coverage, VCS tracks the following in the VHDL source code:

- Individual statements
- Statement blocks
- Statement block type

Note:

- Statement blocks consist of a set of individual statements executed sequentially and always executed together in the same time step.
- Concurrent signal assignments and component instantiations are not reported in the coverage reports because they are always executed.

- Coverage information is not reported for design regions running in cycle mode.
 - Coverage information is not reported for execution lines in entity or configuration declaration statements.
 - VCS monitors line coverage inside for-generates, but individual instances of the generate loop are not monitored separately. Therefore, a line is reported as covered if it is covered in any iteration of the loop, and a line is reported not covered only if it is not covered in any iteration of the loop.
-

Toggle Coverage

Toggle coverage monitors value changes on signal bits in the design. When toggle coverage reaches 100%, it means that every bit of every monitored signal has changed its value from 0 to 1 and from 1 to 0. VCS generates metrics for total coverage of nets and registers, which tells 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 level.

Missing transitions of values provide definitive conclusions about inactive elements and unexercised portions of the design, and the statistics produced for each module can be examined to quickly determine areas of low coverage. This helps you to write tests to address the missing activity.

Note:

By default, toggle coverage does not display or report $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions where the signal returns to its original value during the same time step, in other words, a glitch during which zero simulation time passes.

Example 1-3 Example For Toggle Coverage

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

dut dut1 (w1,r1);

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

always
#10 r1=~r1;

always
#25 r2=r2+1;

endmodule

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

always @ in
dutr1=in;

assign out=dutr1;
endmodule
```

In [Example 1-3](#), the top-level module, `test`, instantiates module `dut`. The top-level module, `test`, contains two registers, `r1` and `r2`, and one net, `w1`. The module instance `dut1` of module `dut` contains one register `dutr1` and two nets, ports `in` and `out`.

In the top-level module, `test`, `reg r1` will have ten $0 \rightarrow 1$ or $1 \rightarrow 0$ transitions and vector `reg r2` will increment three times.

In module instance `dut1`, all the signals will have ten $0 \rightarrow 1$ or $1 \rightarrow 0$ transitions.

Condition Coverage

Condition coverage monitors whether certain expressions and subexpressions in your code evaluate to true or false. By default, the expressions and subexpressions are as follows:

- The conditional expression used with the conditional operator “`? :`” in a continuous or procedural assignment statement. For example, note the following continuous assignment statement for Verilog:

```
assign w1 = r1 ^~ r2 ? r2 : r3;
```

The conditional expression is `r1 ^~ r2` and the truth or falsity of this expression are conditions for condition coverage.

- The conditional expression used in conditional concurrent signal assignment or selected concurrent signal assignment, as shown in either of the following VHDL examples:

```
c <= '0' when (a='0' and b= '0') else
      '1' when (a='1' and b= '1'); -- conditional concurrent
                                -- signal assignment
```

or

```
with (a and b or d) select
      c <= '0' when '0',
      '1' when '1',
      'X' when others;
```

In this example, the conditional expressions are ($a = '0'$ and $b = '0'$) and ($a = '1'$ and $b = '1'$) conditional concurrent signal assignment, and (a and b or d) for selected concurrent signal assignment.

- Sub expressions are other conditional operands to the logical AND(&&), logical OR (||), or ternary operators. For example, see the following Verilog assignment statement:

```
r8 = (r1 == r2) && r3 ? r4 : r6;
```

Here, the conditional expression is $(r1 == r2) \&\& r3$ and the subexpressions are $(r1 == r2)$ and $r3$. The conditions for condition coverage are the truth or falsity of the conditional expression and these two subexpressions.

In the case of VHDL, expression $((a \text{ and } b) \text{ or } d)$ consists of two subexpressions: $(a \text{ and } b)$ expr1 $(a \text{ and } b)$. The conditions for condition coverage are the truth or falsity of the whole expression and these two subexpressions.

- Subexpressions that are the operands to the logical AND && or logical OR || operators in the conditional expression in an if statement. For example, in the following Verilog if statement:

```
if ((r1 ^ (!r2)) && (r3 == r4))
    begin
        .
        .
        .
    end
```

The subexpressions that are the operands of the logical and operator && are $(r1 ^ (!r2))$ and $(r3 == r4)$ and the conditions for condition coverage are the truth or falsity of these VHDL subexpressions.

```
if ( A = '0' ) and ( B = '0' ) then
...
end if;
(A = '0') and (B = '0') are the subexpressions
```

- Subexpressions that are the operands to the logical AND (`&&`) or logical OR (`||`) operators in an expression in a continuous or procedural assignment statement. For example, in the following Verilog continuous assignment statement:

```
assign w2 = r5 || r6;
```

Subexpressions `r5` and `r6` are operands of the logical OR (`||`) operator and their truth or falsity are conditions in condition coverage.

For another example, in the following Verilog procedural assignment statement:

```
r6 = (r6 != r7) || r8;
```

Subexpressions `(r6 != r7)` and `r8` are operands of the logical OR (`||`) operator and their truth or falsity are conditions in condition coverage.

- Subexpressions `(mba < mbb)` and `(mbb <= mbc)` are operands of the logical AND operator and their truth or falsity are conditions in condition coverage.
- Subexpressions `(b /= c)` and "d" are operands of the logical OR and their truth or falsity are conditions in condition coverage.

Verilog Conditional Coverage Not Monitored

Not all occurrences of conditional expressions with the right operators result in conditions for condition coverage. The following are cases where there are conditional expressions, but VCS do not monitor for condition coverage:

- In terminal connection lists in task-enabling statements:

```
#1 t1(r11 && r12,r12,r3);
```

- In instance connection lists:

```
dev d1(r11 && r12);
```

- In PLI routine calls:

```
$pli((r11 && r12);
```

- In `for` loops:

```
for (i=0;i<10;i=i+1)
  if(r1 && r2)
    r3=r1;
  else
    r3=r2;
```

In this example, VCS by default does not monitor the conditional expression (`r1 && r2`) for condition coverage, but it is possible to monitor conditional expressions in `for` loops.

For more information, see the topic *Enabling Condition Coverage in For Loops* in the Coverage Technology Reference Manual.

- In user-defined tasks and functions:

```
task mytask;
  input in;
  output out;
```

```

begin
if (r1 && r2)
    out=in;
else
    out=~in;
end
endtask

```

By default, VCS does not monitor for conditional expressions in user-defined tasks and functions.

For more information, see the topic *Enabling Condition Coverage in Tasks and Functions* in the Coverage Technology Reference Manual.

VHDL Conditional Coverage Not Monitored

In the following instances, VCS cannot monitor for condition coverage and URG cannot report condition coverage; no warning or error message appears.

- Conditions in sequential templates. For example:

```

if(clk'event and clk='1') then
.
.
.
.
```

- Conditions with a vector of variable size, though the size is known through further evaluation. For example:

```

variable N : integer;
DATA (N to N+3) <= DATA2 (7-N downto 4-N) AND DATA3 (7-N
downto 4-N) ;
```

- Conditions within a generate block.
- Conditions within a subprogram body.

- Conditions within an entity declaration region. For example, a subprogram defined in an entity.
- Conditions within a package body. For example, a subprogram defined in a package body.
- Conditions within a block construct.

Stable Names for Conditions in a Design

Code coverage automatically finds conditions in a design, and monitors whether they are covered in simulation runs. These objects are identified in an ad hoc way, depending on the type of object. For example, signals in toggle coverage and machines in finite-state machine coverage can be identified by signal names. Statements, branches, and conditions are identified by line numbers. Conditions are identified by a sequence number within a given module (for example, 1st, 2nd, and 3rd).

Following are the two categories of conditions:

Conditions with stable names — An object name is stable, if it is not sensitive to unrelated changes. For example, the name of a signal for toggle or FSM coverage is stable. The condition name can be changed by changing the signal name itself.

Conditions with unstable names — Object names that are sensitive to unrelated changes in the design are unstable. For example, any condition whose name depends on a source line number has an unstable name, since even adding a comment can change the name.

This feature allows you to specify stable names for conditions, whose names are unstable by default.

Naming a Condition in a Module

A new pragma is introduced to allow you to name a condition in a module. For example:

```
//coverage name a1  
if (a || (b && c))
```

In the example, the name `a1` is assigned to the statement `if (a || (b && c))`. Any conditions found in the statement can be addressed starting with `a1`, rather than with a sequence number in the module or source line number. Pragma should be just inserted immediately before the condition.

Advantages of Assigning Stable Names for Conditions

Assigning stable names for conditions is useful for excluding continuous conditions and vectors specified in el files for condition coverage.

At present, conditions and vectors can be excluded using the condids given in elfile, which is generated by DVE. However, the names of conditions and vectors in that file are sequence numbers generated by an incrementing counter. For example, condition 1, condition 2, and so on.

An issue in this approach is that if a new condition is added, then all the sequence numbers for conditions below the addition are invalidated. Where you once had conditions with IDs 5, 6, 7, the IDs are now 6, 7, and 8. Any file that contains a condition to 'exclude condition 6' is now excluding the wrong one.

If you assign a stable name to a condition, then that name should be shown in any coverage reports (and available through UCAPI).

Assigning stable names for conditions is accomplished by providing tags in the source and generating an el file using DVE for condition coverage metric. This metric will contain the stable names and excluded vectors in the generated file.

Specifying Tags in the Source

This section explains how tags are specified in the source, and how condition names are derived from those tags.

Condition and Subcondition

Consider the following example:

```
//coverage name a1
if (a || (b && c)) {
    ...
}
```

The top-level condition is `a || (b && c)`. It is broken down by code coverage into two terms. Its name is `a1`:

```
a || (b && c) Condition a1
1      --2--
```

There is also a subcondition with two terms, named `a1.1`:

```
b && c     Condition a1.1
1      2
```

Multiple Conditions in a Statement

Statements may contain more than one independent condition. For example:

```
//coverage name mycond
x <= (a || (b && c)) ? ( (d && (e || f)) ? 1'b1 : 1'b0 ) : 1'b1;
```

There are two conditions, (a || (b && c)) and (d && (e || f)) in the example. Both the conditions are part of the same statement, but there is only one name pragma.

VCS will assign names based on the name. The first condition is named as mycond:

```
a || (b && c)      Condition mycond  
1      --2---  
b && c      Condition mycond.1  
1      2
```

The second condition is named as mycond.2:

```
d && (e || f)      Condition mycond.2  
1      --2---  
e || f      Condition mycond.3  
1      2
```

If there were additional conditions, then they would be named as mycond.<serial number of condition on the line>. For example, mycond.2, mycond.3, and so on.

Note: This additional naming only holds for one statement.

Multiple Statements on a Line

If a single source line contains multiple statements, then only the first statement will be named by a preceding pragma. For example:

```
//coverage name a2  
x <= (a || b) ? 1'b0 : 1'b1; y = (c && d) ? 1'b0 : 1'b1;  
a || b      Condition a2  
1      2
```

There is no name for the second condition `c && d`, since it occurs in another statement that has no pragma preceding it. If you want to assign a name to `c && d`, then you should put the statement on a separate line:

```
//coverage name a2
x <= (a || b) ? 1'b0 : 1'b1;
//coverage name a3
y = (c && d) ? 1'b0 : 1'b1;
a || b Condition a2
1    2
c && d Condition a3
1    2
```

Names in the Generated Reports

Currently in the URG reports, condition ID numbers are shown if the `-cond_ids` option is specified. These appear inlined with the condition coverage reports. For example, in URG, you can view the following:

```
LINE    12
CONDITION_ID    1
STATEMENT      if ((a || (b && c)))
                1      --2---
EXPRESSION     -1-    -2-
                  0      0  | Not Covered
                  0      1  | Not Covered
                  1      0  | Not Covered
```

If the condition was named with a pragma in the source, then the name will appear instead:

```
LINE    12
CONDITION_ID    a1
STATEMENT      if ((a || (b && c)))
                1      --2---
EXPRESSION     -1-    -2-
```

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

Subconditions will be named as described above. For example, the subcondition `b && c` is labeled as `a1.1`:

```

LINE    12
CONDITION_ID  a1.1
STATEMENT    if ((a || (b && c)))
              1      2
EXPRESSION   -1-    -2-
              1      1  | Not Covered
              0      1  | Not Covered
              1      0  | Not Covered

```

You can use the `covdb_get_annotation(condObj, "id")` to get the stable name from the condition ID handle.

El File for Condition Coverage

If you use DVE to generate the el file, then the dumped file contains the stable name for tagged conditions instead of the Integer IDs. Expression IDs for subexpression in a new database may differ with respect to the old database.

While applying the el file, if the module or instance checksum has changed, then only the conditions with given stable names are considered for exclusion. The remaining non-stable IDs are ignored in URG and are under review in DVE.

The `auto_stablefile.el` file is generated when you specify the `-cond_ids` option in the `urg` command. In this el file, all the condition exclusions are commented. You can uncomment any condition exclusion that you need.

Limitations of Stable Names for Conditions

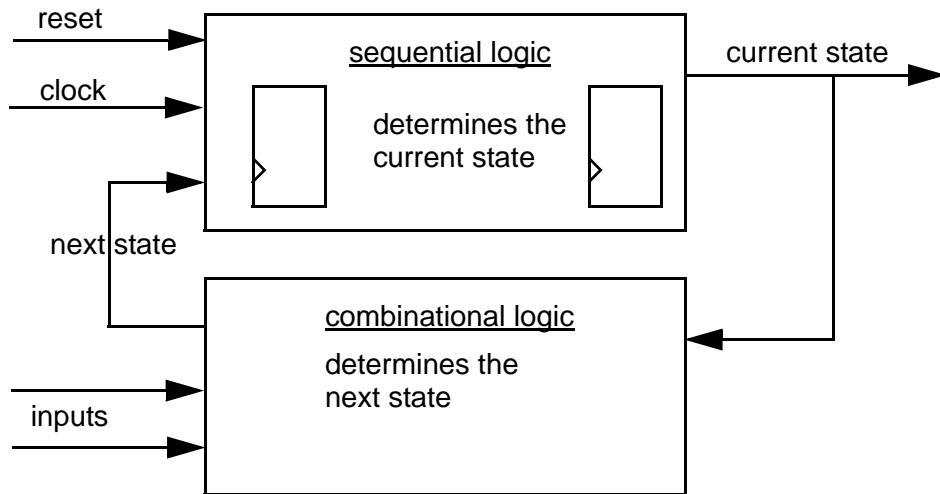
- The support for stable names is not currently available for VHDL conditions. It is available only for Verilog conditions.
- There should be no other lines (even blank lines) between the condition statement line and the pragma line.
- The stable name for condition cannot have a dot (.) character in it; it is reserved for condition coverage naming scheme for sub conditions.

FSM Coverage

In hardware, a Finite State Machine (FSM) is a sequential logic that outputs a current state and a combinational logic that outputs the next state. When VCS compiles your design for FSM coverage, it identifies a group of statements in the source code to be an FSM and tracks the states and transitions that occur in the FSM during simulation.

The sequential logic is driven by the next state signal and clock and reset signals. The combinational logic is driven by the current state and the inputs to the FSM.

Figure 1-1 Finite State Machine



In higher level Verilog or VHDL, a group of statements can be a higher level of abstraction of an FSM. VCS treats a group of statements as an FSM if the group contains a procedural assignment statement to assign the current state of the FSM. The values assigned in the group of statements must be clearly identifiable as constants and there must be a dependency between the current state and the next state. The following group of Verilog statements is an FSM:

```

input in;
reg [3:0] current,next;

initial
current=4'b0001;

always @ in
begin
case (1'b1)
current [0] : next=4'b0010;
current [1] : next=4'b0100;
current [2] : next=4'b1000;
current [3] : next=4'b0001;

```

```
endcase  
#2 current=next;  
end
```

FSM Coverage for Verilog

In Verilog, a group of statements can be used to describe a higher level of abstraction of an FSM. VCS treats a group of statements as an FSM if it meets the following criteria:

- The group of statements must contain a procedural assignment statement to a vector variable to assign the current state of the FSM.

The group can also contain one of the following (though this is not required):

- A procedural assignment to assign the next state to another variable.
- A concurrent assignment statement to assign the next state to a net.
- In the group, a dependency exists between the current value assigned and the next value assigned.

When you write such a group of statements, it is important to know if the statements made the assignments to all possible states of the FSM and all possible transitions between states.

VCS does not automatically extract an FSM when:

- The group of statements exists in a user-defined task (in VHDL, a function or procedure), including a SystemVerilog global task (it will extract if the group is in a user-defined function, including a SystemVerilog global function).

- There are less than three possible states.

Example 1-4 is a Verilog module definition that contains statements that function as an FSM:

Example 1-4 Verilog Module Definition Containing an FSM

```

module dev (clk,in,state);
  input clk,in;
  output [1:0] state;

  reg [1:0] state,next;
  parameter idle = 2'b00,
            first = 2'b01,
            second = 2'b10,
            third = 2'b11;

  initial
  begin
    state=idle;
    next=idle;
  end

  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;

endmodule

```

The FSM has four states: idle, first, second, and third. It is possible for a testbench to apply stimulus such that line coverage indicates that VCS executed all the executable lines, but FSM coverage indicates that there never was a transition from the third to the idle state.

The values that the statements in the group assign to a signal are the states of the FSM, and must be of an identifiable set of parameters, numeric constants, or text macros (where the macro text that VCS substitutes for the macro name is a numeric constant). The states can also be enumerated types in SystemVerilog.

FSM Coverage for VHDL

VCS automatically extracts FSMs from VHDL code when it sees the following:

- The state values are stored in enumerated, std_logic_vector, bit_vector, or integer data types.
- The state values of the FSM are either an enumerated type, VHDL constant, or literal constant. No expressions or function calls on the right side of assignment statements assigning the next state of the FSM.
- The code determines the next state of the FSM using a `case` statement. VCS does not extract the FSM if there is an `if` statement for determining the next state.
- The code for the FSM is entirely within a procedure or process or using one combinational process to determine the next state of the FSM and a sequential process to set the next state on the clock edge.

VCS cannot automatically extract the following types of FSMs:

- One-hot or hot-bit FSMs.
- FSMs using conditional assignment statements.

[Example 1-5](#) illustrates a VHDL architecture that models an FSM:

Example 1-5 VHDL Architecture Modeling an FSM

```

architecture exfsmarch of exfsm is
    type my_fsm_states is (idle, first, second, third);

    signal curr_state : my_fsm_states := idle;
    signal next_state : my_fsm_states := idle;

begin

    my_fsm : process(insig)
    begin
        next_state <= curr_state;
        case curr_state is
            when idle => if (insig = '1') then next_state <= first;
        end if;
            when first => if (insig = '1') then next_state <= second;
        end if;
            when second => if (insig = '0') then next_state <= third;
        end if;
            when third => if (insig = '1') then next_state <= idle;
        end if;
        end case;
    end process;

    advance_fsm : process
    begin
        wait until clk'event and clk = '1';
        curr_state <= next_state;
    end process;

end exfsmarch;
```

Branch Coverage

Branch coverage analyzes how `if` and `case` statements and the use of the ternary operator (`? :`) establish branches of execution in your HDL design. It shows you vectors of signal or expression values that enable or prevent simulation events.

Note:

VHDL branch coverage is an LCA feature. For more information about the LCA feature, see the chapter "Using VHDL Branch Coverage" under the LCA category in the VCS Online Documentation.

Consider the code in [Example 1-6](#):

Example 1-6 Branch Coverage Code Example

```
case (r1)
  1'b1    : if (r2 && r3)
              r4 = (r5 && r6) ? 1'b0 : 1'b1;
  1'b0    : if (r7 && r8)
              r9 = (r10 && r11) ? 1'b0 : 1'b1;
  default : $display("no op");
endcase
```

In this block of code, there are procedural assignment statements where the value assigned is controlled by the ternary operator (`? :`). These in turn are controlled by `if` statements and the `if` statements are controlled by a `case` statement.

In this block of code, the possible vectors of signal or expression values that result in simulation events or prevent simulation events are as follows:

```
r1    (r2 && r3) (r5 && r6) (r7 && r8) (r10 && r11)
```

1	1	1	-	-
1	1	0	-	-
1	0	-	-	-
0	-	-	1	1
0	-	-	1	0
0	-	-	0	-
default	-	-	-	-

For example, in the first vector, `r1` is `1'b1` and the expression `(r2 && r3)` is true, therefore, the value of `r4` depends on the value of `(r5 && r6)`. The values of `(r7 && r8)` and `(r10 && r11)` do not matter.

Another example, `r1` is `1'b1`, but the expression `(r2 && r3)` is false, and the values of the other expressions do not matter, therefore nothing happens.

Branch coverage shows you these vectors and then tells you whether these vectors ever occurred during simulation, in other words, whether they were covered.

By default, VCS does not monitor branch coverage for if and case statements and ternary operator`(?:)` if they are in user-defined tasks or functions or in code that executes as a result of a `for` loop. You can, however, enable branch coverage in this code.

For more information, see *For Loops and User-defined Tasks and Functions* in the Coverage Technology Reference Manual.

Branch Coverage With Unknown and High Impedance Values

If the conditional expression in an `if` statement evaluates to X or Z, branch coverage treats this as a false value and reports that 0 value for the expression is covered.

If the case expression in a case statement evaluates to X or Z, VCS executes the default case item. When this occurs, branch coverage reports that the vector for the default case item is covered. If there is no default case item, branch coverage reports a vector for the missing default case item and reports it as covered.

When the conditional expression for a ternary operator evaluates to X or Z, the vector for the expression is not covered.

Functional Coverage

You can gather the functional coverage metrics during your simulation runs. If your design has assertions or covergroup defined, VCS collects the coverage data and generates the coverage database in simv.vdb. Once you have simv.vdb, you can use the Unified Report Generator (URG) to generate text or HTML reports, DVE Coverage GUI (Cov GUI) to view your assertion or covergroup reports, or your own coverage API application. For more information about functional coverage metrics, see the *VCS User Guide*.

Generating the Coverage Database

To generate the coverage database, perform the following steps in VCS:

- Compile your design
- Run the simulation

For information on how to perform these steps, see the chapter [Generating Coverage Database](#).

Generating Coverage Reports

After you have generated the coverage database, you need to generate the coverage reports.

You can generate the coverage reports using any of the following tools:

Unified Report Generator

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

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

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

DVE Coverage GUI

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. DVE coverage also introduces more advanced feature enabling interactive operations, such as excluding parts of design.

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 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.

Using Unified Coverage API

Unified Coverage Application programming Interface (UCAPI) is a set of APIs that you can use to extract specific data from your coverage database. This is needed when you want to generate custom reports to analyze your coverage data. For more information about UCAPI, see the chapter [Unified Coverage API](#).

Post-processing Techniques

While generating the coverage database, you might want to analyze your coverage data, do some post-simulation procedures, or merge the data after multiple simulation runs. You can analyze the coverage data in the following ways:

- Edit the covergroup coverage database by resetting the covergroups or coverpoints, or removing the covergroups from the coverage database.
- Merge the coverage data to get combined report.
- Autograde to eliminate redundant tests.
- Map the subhierarchy coverage from one design to another.
- Exclude part of data.
- Access covergroup coverage data to modify it.

For more information about all these post-processing techniques and processes, see the chapter “[Coverage Post-processing Techniques](#)” .

2

Generating Coverage Database

This chapter contains the following section:

- “Unified Coverage Database”
- “How to Obtain Coverage Data for your Design”
- “Generating Coverage Metrics”

Unified Coverage Database

The Unified Coverage Database is to simplify data management with a single database while at the same time provide merge time and grade time performance. The Unified coverage database, as the name indicates, contains coverage data for Code coverage, Functional coverage, and Assertion coverage (various aspects of coverage).

With the unified database,

- coverage data of all the metrics (Line, Toggle, Branch, Condition, FSM, Assertion, and Functional coverage) is collected in a single directory `simv.vdb`.
- all the metrics are supported with the same post-processing tools. URG for batch merging, grading, and reporting, DVE Coverage GUI for interactive analysis, and UCAPI for custom developed applications.

Note:

The old and unified databases are not supported together, that is flow should be all old or all unified. URG, DVE, and UCAPI do not support mixing of old database and unified database.

How to Obtain Coverage Data for your Design

To obtain coverage metrics for your design, you must compile your design, run the simulation, and view coverage report in the coverage tools.

Compiling Design Files

You tell VCS the types of coverage for which you want it to compile your design. After a design is compiled for a type of coverage, VCS automatically collects coverage information for that type of coverage.

To compile the design for coverage, enter the following command:

```
%vcs [cover_options] [compile_options] source.v
```

Where,

[cover_options] are coverage related compilation options. For example, -cm line or -cm line+tgl.

[compile_options] are compilation options.

source.v is your design file.

In the process of compilation, VCS creates a directory simv.vdb, that contains code, assertion, and functional coverage data. You can also use -cm_dir and -cm_name compilation options.

-cm_dir directory_path_name, specifies an alternative name or location for the default simv.vdb directory. If not specified, VCS automatically generates the coverage database with the name simv.vdb or <exe_name>.vdb where, exe_name is the argument to -o option if present during compilation.

Note that if you compile the design with the -cm_dir option, and then move simv.vdb to a new location, you must use -cm_dir at runtime to point to the new location of simv.vdb.

-cm_name filename as a compile-time or runtime option, specifies an alternative test name instead of the default name. The default test name is "test".

Running the Simulation and Monitoring Coverage

To monitor for coverage during simulation, enter the following command:

```
% simv [cover_options] [run_options]
```

Where,

[cover_options] are coverage related compilation options. See “[Generating Coverage Metrics](#)” .

[run_options] are runtime options.

Viewing Coverage Reports

In URG

To invoke URG, you need to specify one or more coverage database directories. Under default coverage options, the coverage database directory can be the *.vdb directory containing code coverage, covergroup coverage, and assertion coverage data.

The following command line invokes URG:

```
% urg -dir dir1 [dir2 ....] [urg_options]  
% urg -dir simv.vdb -format both
```

Where,

dir1 and dir2 are the coverage database directories.

If you specify more than one coverage database directory in the URG command-line, URG merges the coverage database directories, and writes a merged coverage database in the current or specified directory.

[urg_options] are URG command-line options. For more information about URG options, see the chapter *URG Options* in the Coverage Technology Reference Manual.

By default, the above URG command writes html report files in the urgReport directory that is created in the current directory. dashboard.html is the main page, which gives you the overall

coverage information of the entire design or the testbench. For more information about URG, see the chapter [Viewing the Coverage Report Using Unified Report Generator](#).

In DVE Coverage GUI

To invoke the DVE Coverage GUI, you need to enter the `dve` command with a coverage command-line option (`-cov`) and then load the coverage database. You can also specify the directory name in the command-line while invoking the DVE Coverage GUI.

```
% dve -cov  
OR  
% dve -covdir *.vdb
```

For more information about the DVE Coverage GUI, see the chapter [Viewing Coverage Reports Using the DVE Coverage GUI](#).

Coverage for Protected Modules

VCS does not report coverage score for protected module for any coverage metric. If a module has some part of the code protected in it, then VCS does not monitor the module or its self-instance and the full hierarchy below it, for coverage.

Generating Coverage Metrics

To generate coverage metrics, use the following command during compile-time and runtime:

```
-cm <cov_metrics_name>
```

The arguments to the `-cm` option are as follows:

line

Specifies line coverage.

cond

Specifies condition coverage.

tgl

Specifies toggle coverage.

fsm

Specifies FSM coverage.

branch

Specifies branch coverage.

assert

Specifies assertion coverage.

You can include more than one argument using a plus (+) as a delimiter between arguments. For example,

```
vcs source.v -cm line+tgl+branch
```

Coverage for covergroup and cover property is automatically turned on in VCS, hence there's no need to specify any commands to collect the same.

The following examples show compilation and runtime options for one or more metrics and how to invoke URG and Coverage GUI. You can view the coverage report in batch mode using URG or in GUI mode using DVE Coverage GUI:

Example 2-1 Line Coverage Only

```
vcs -cm line source.v
simv -cm line
urg -dir simv.vdb //to generate URG report
dve -covdir simv.vdb //to view coverage report in DVE
                           Coverage GUI
```

In this example:

1. VCS compiles for line coverage.
2. During simulation, VCS looks for line coverage.

Example 2-2 Line, Condition and FSM Coverage

```
vcs -cm line+cond+fsm source.v
simv -cm line+cond+fsm
urg -dir simv.vdb
dve -covdir simv.vdb
```

In this example,

1. VCS compiles for Line, Condition, and FSM coverage.
2. During simulation, VCS looks for Line, Condition, and FSM coverage.

Example 2-3 Line, Condition, FSM, Toggle, and Branch Coverage

```
vcs -cm line+cond+fsm+tgl+branch -lca source.v
simv -cm line+cond+fsm+tgl+branch
urg -dir simv.vdb
dve -covdir simv.vdb
```

In this example,

1. VCS compiles for Line, Condition, FSM, Toggle, and Branch coverage.
2. During simulation, VCS keeps track of Line, Condition, FSM, Toggle, and Branch coverage.

Example 2-4 Line, Condition, and Assertion Coverage

```
vcs -cm line+cond+assert source.v  
simv -cm line+cond+assert  
urg -dir simv.vdb / dve -covdir simv.vdb
```

In this example,

1. VCS compiles for Line, Condition, and Assertion coverage.
2. During simulation, VCS keeps track of Line, Condition, and Assertion coverage.

Example 2-5 Compiling for FSM, Toggle, and Branch Coverage while Simulating only Branch Coverage

```
vcs -cm fsm+tgl+branch source.v  
simv -cm branch  
urg -dir simv.vdb / dve -covdir simv.vdb
```

In this example,

1. VCS compiles for FSM, Toggle, and Branch coverage.
2. You can choose to simulate VCS to track only few metrics than what you have specified during the compilation stage, such as Branch coverage.

However, choosing fewer metrics for coverage compilation and choosing more during simulation doesn't work.

Example 2-6 Specifying More Metrics at Compile-time and Runtime but Generating Reports for Fewer Metrics

```
vcs -cm line+fsm+branch+cond source.v  
simv -cm line+fsm+branch+cond  
urg -dir simv.vdb -metric line+fsm  
dve -covdir simv.vdb
```

In this example,

1. VCS compiles for and simulates Line, FSM, Branch, and Condition coverage.
2. While generating reports, you can choose to generate reports for fewer metrics than what you have specified during compilation and simulation and use URG to generate only Line and FSM coverage.

3

Viewing the Coverage Report Using Unified Report Generator

This chapter contains the following sections:

- “Supported Metrics”
- “Invoking URG”
- “Format to Display Coverage Results”
- “What is Covered in Each Coverage Metrics”
- “Difference Reports for Functional Coverage”
- “Reporting Only Uncovered Objects (-show brief)”
- “Coverage Report Files”
- “Analyzing Trend Charts”

Supported Metrics

URG generates reports that include the following metrics:

- Code coverage
 - “The Line Coverage Report”
 - “The Toggle Coverage Report”
 - “The Condition Coverage Report”
 - “The FSM Coverage Report”
 - “The Branch Coverage Report”
- “The Assertion Coverage Report”
- “The Covergroup Report”

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

- “The Dashboard File”
- “The Hierarchy File”
- “The Modlist File”
- “The Groups File”
- “The modN File”
- “The grpN Files”
- “The Asserts File”
- “The Tests File”

Invoking URG

The usage model to invoke URG is as follows:

1. Compile the test file.

```
% vcs [compile_options]
```

2. Simulate the test file.

```
% simv [runtime_options]
```

3. Run URG command

```
urg -dir coverage_directory.vdb urg_options
```

You must specify the directories containing coverage data files. Coverage directory contains code coverage, covergroup or assertion/property coverage data.

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

```
./simv.vdb/snps/coverage/db/testdata/test1
```

To report any particular metric related information, use the `-metric` option with the specific metric ("+" separated arguments for more than one metric). The possible metric arguments are shown below:

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

An initial plus sign is not required, but is allowed.

By default, the coverage collection for covergroup and cover property is turned on in VCS. Hence, there is no need to specify any commands to collect the coverage for covergroup and cover property. However, when you run a design to report other metric such as line, toggle, assert, and so on, VCS reports the covergroup and cover property metric as well, by default.

For example, if `simv.vdb` contains coverage for line, covergroup, and cover property, but if you are interested to get only the report for `line` metric , then you can achieve it by using the following command:

```
urg -dir simv.vdb -metric line
```

If no `-metric` argument is specified, then URG reports all type of coverage in the indicated coverage directories.

URG generates the reports and places them in a directory `urgReport` , by default. Each time URG is run, the report directory is overwritten by the new report files. You can use `-report mydir` option to save the generated reports in `mydir`.

For example:

```
urg -dir simv.vdb -metric line+fsm -report mydir
```

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

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

Merging Coverage Results of Several Runs of the Same Executable

To merge the coverage result of several runs of the same executable, simulate the executable with different stimuli +number, or \$value\$plusarg option, and give a unique name to the test using the -cm_name option:

```
simv -cm line +number=2 -cm_name test_2  
simv -cm line +number=1 -cm_name test_1  
simv -cm line +number=0 -cm_name test_0
```

The results can be merged and displayed using URG as follows:

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

Now open the URG reports, select tests.html and the following text is displayed in the report:

```
Data from the following tests was used to generate this report  
simv/test_0  
simv/test_1  
simv/test_2
```

Merging Coverage Database Across Versions of VCS

A database created with a given major version of VCS can be read with any of the coverage tools from the same major version. For example, a database created with VCS 2012.09 (for any patch number) can be read by URG or DVE Coverage GUI from VCS 2012.09 (again, any patch number) or UCAPI-based application linked against a VCS 2012.09 UCAPI library. This is called as Patch-to-Patch compatibility.

Databases created with one major version are not guaranteed to be compatible with tools from a different major versions. For example, a database written from VCS 2012.09 is not guaranteed to be readable by tools from VCS 2011.12. Thus Release-to-Release compatibility is not guaranteed.

For more details about how versioning is done, see the section “Version Check” in the Coverage Technology Reference Manual, which discusses the major / minor version numbers maintained in the database's XML files. Note that DVE Coverage GUI and URG uses the UC API, thus the requirements are the same.

Format to Display Coverage Results

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

- Statistics table
- Table of coverable objects

Statistics tables are summaries of types of coverage elements. Each line in a statistics table reports the coverage for a class or category of object. [Figure 3-1](#) shows an example of a statistics table for line coverage:

Figure 3-1 Example of a Statistics Table

	Line No.	Total	Covered	Percent
TOTAL		29	26	89.66
INITIAL	24	3	3	100.00
ALWAYS	30	5	5	100.00
ALWAYS	41	11	11	100.00
ALWAYS	65	7	4	57.14
ALWAYS	111	3	3	100.00

Statistics tables are color-coded using the same color legend as for coverage tables shown in [Figure 3-42](#).

The table of coverable objects shows the coverage results for individual coverable objects. Coverable objects do not have percentages; they are either covered or uncovered. Coverable object tables show covered (and observed) objects in green and uncovered objects in red. [Figure 3-9](#) in the Condition Coverage Section shows a coverage data table for condition coverage.

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

Note that several metrics have options that change exactly what is covered and how to display it. For example, use the condition coverage option `-cm_cond allops` to control which vectors and conditions are monitored.

For more information, click this link [Coverage Technology Reference Manual](#) if you are using the VCS Online Documentation.

If you are using the PDF interface, click this link [cov_ref.pdf](#).

What is Covered in Each Coverage Metrics

URG, by default, shows detailed coverage reports for all the modules and instances for any given metric. URG shows the entire summary table for a metric it reports. For example, consider the following toggle coverage report.

Toggle Coverage for Module : jukebox

	Total	Covered	Percent
Totals	10	5	50.00
Total Bits	64	34	53.12
Total Bits 0->1	32	17	53.12
Total Bits 1->0	32	17	53.12
Ports	6	3	50.00
Port Bits	48	22	45.83
Port Bits 0->1	24	11	45.83
Port Bits 1->0	24	11	45.83
Signals	4	2	50.00
Signal Bits	16	12	75.00
Signal Bits 0->1	8	6	75.00
Signal Bits 1->0	8	6	75.00

The Line Coverage Report

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

Figure 3-2 Example of a Line Coverage Section Report:

Line Coverage for Module : GmSequence

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

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

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

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

Figure 3-3 Example of an Annotated Source Code File

```

35      always @(state or attention or full)
36      begin
37 1/1          wrt = 1'b0; <----- coverable lines are
38 1/1          oe = 0;                      shown in bold;
39 1/1          i = 0;                      covered are black
40 1/1          case (state)
41          idle:
42 1/1              if (!full && !x_not) && y_tot)
43 1/1                  if (attention || (y_tot ^ (!x_not))) begin
44 1/1<-- numbers are
45  /covered/coverable
46  /for each line
47          n_state = read;
48 1/1                  //last = 0;
49          end
50          else
51 0/1 ==>          n_state = <----- only first line of multi-
52          idle;                      line statements is
53 1/1          read: begin rd_loop
54 1/1              for( i=last; i < tables; i=i+1) // Round-robin structure
55                  if ((attention[i] == 1'b1) || (y_tot ^ (!x_not)))
56                  begin
57                      oe[i] = 1'b1;
58                      n_state = wr_fifo;
59                      last = i+1;
60                      disable rd_loop; <----- coverable lines with no
61 2/2          MISSING_ELSE
62                  last = 0; n_state = idle;                      uncovered statements are
63                  end
64 0/2 ==>          wr_fifo: begin
65                  wrt = 1; n_state = idle; <----- lines with uncovered
66                  end
67          MISSING_DEFAULT
68          endcase
69      end
70  endmodule

```

Note:

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

If the source code of your design is not available (for example, source code files have been moved to a new location, the files are not read-accessible, or the files are not visible on the machine/network on which you are running URG), then URG will generate a simplified report in place of the annotated source like [Figure 3-3](#). An example of the simplified report is as follows:

Figure 3-4 Example of a Simplified Report

Line Covered Statements

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

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

The Toggle Coverage Report

URG covers any toggle, be it 1->0 or 0->1.

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

Figure 3-5 Example of a Toggle Coverage Summary Report

Toggle Coverage for Module : jukebox

	Total	Covered	Percent
Totals	10	5	50.00
Total Bits	64	34	53.12
Total Bits 0->1	32	17	53.12
Total Bits 1->0	32	17	53.12
Ports	6	3	50.00
Port Bits	48	22	45.83
Port Bits 0->1	24	11	45.83
Port Bits 1->0	24	11	45.83
Signals	4	2	50.00
Signal Bits	16	12	75.00
Signal Bits 0->1	8	6	75.00
Signal Bits 1->0	8	6	75.00

The following figure shows a toggle coverage detailed report. In the detailed table, URG lists signals and ports that are not fully covered.

Figure 3-6 Example of a Toggle Coverage Detailed Report

Port Details				
	Toggle	Toggle 1->0	Toggle 0->1	Direction
clk	Yes	Yes	Yes	INPUT
rst	Yes	Yes	Yes	INPUT
full	No	No	No	INPUT
attention[3:0]	Yes	Yes	Yes	INPUT
attention[9:4]	No	No	No	INPUT
oe[3:0]	Yes	Yes	Yes	OUTPUT
oe[9:4]	No	No	No	OUTPUT
wrt	Yes	Yes	Yes	OUTPUT

Signal Details			
	Toggle	Toggle 1->0	Toggle 0->1
state[2:0]	Yes	Yes	Yes
n_state[2:0]	Yes	Yes	Yes
x_not	No	No	No
y_tot	No	No	No

Support for Interface Signals on Port Boundary

VCS now supports toggle coverage on port interface signals. In previous versions, VCS reported only the signals of the stand-alone interface and ignored interface signals declared in module or instance port lists.

Beginning with this release, VCS includes interface signals as part of the toggle coverage score for a given module or instance. You can use the toggle coverage compile-time option `-cm_tgl fullintf` to get toggle coverage reports for all signals of an interface located within the port boundary of a module or instance.

Use the `-cm_tgl fullintf` compile-time option to get toggle coverage on interface signals. Using this option enables full interface flattening at port boundaries. Therefore, this option reports all interface signals used in a module port list as port signals of the corresponding module or instance.

Consider the code shown in [Example 3-1](#).

Example 3-1 Toggle Coverage

```
interface intf(input bit clk);
    logic dvld;
    logic ack;
endinterface

module mod(intf intf1, input clk);
    always @(posedge clk)
        $display("\n Hello World ");
endmodule

module tb();
    logic clock;
    intf intf_0(clock);
    mod mod_0(intf_0, clock);
    initial begin
        clock = 0;
        forever #3 clock = ~clock;
    end

    initial
    #15 $finish;
endmodule
```

In previous versions, where VCS did not expand port interface signals, and just reported coverage at the source interface, a coverage report generated with `-cm_tgl` portsonly looked like [Figure 3-7](#).

Figure 3-7 Previous Version Coverage Report

```

Module Instance : tb
SCORE  TOGGLE
--      --
Module Instance : tb.intf_0
Port Details
    Toggle Toggle 1->0 Toggle 0->1 Direction
        clk          Yes      Yes           Yes      INPUT

Toggle Coverage for Instance : tb.intf_0
                                         Total     Covered   Percent
Totals                           1         1       100.00
Total Bits                      2         2       100.00
Total Bits 0->1                1         1       100.00
Total Bits 1->0                1         1       100.00

Module Instance : tb.mod_0
Port Details
    Toggle Toggle 1->0 Toggle 0->1 Direction
        clk          Yes      Yes           Yes      INPUT

Toggle Coverage for Instance : tb.mod_0
                                         Total     Covered   Percent
Totals                           1         1       100.00
Total Bits                      2         2       100.00
Total Bits 0->1                1         1       100.00
Total Bits 1->0                1         1       100.00

```

In [Figure 3-7](#), for instance `tb.mod_0`, only port signal `clk` is reported. Interface signal `intf1` is not reported.

From this release onwards, a coverage report generated with `-cm_tgl fullintf` looks like [Figure 3-8](#). Here, interface `intf1` declared at port list is expanded and shares coverage data from its source interface instantiation.

Figure 3-8 Current Version Coverage Report

```
Module Instance : tb.intf_0
Port Details
    Toggle Toggle 1->0 Toggle 0->1 Direction
        clk      Yes     Yes      Yes      INPUT
        dvld    No      No      No      INOUT
        ack     No      No      No      INOUT

Toggle Coverage for Instance : tb.intf_0
                    Total   Covered   Percent
Totals            3       1       33.33
Total Bits        6       2       33.33
Total Bits 0->1  3       1       33.33
Total Bits 1->0  3       1       33.33
```

```
Module Instance : tb.mod_0
Port Details
    Toggle          Toggle 1->0 Toggle 0->1 Direction
        clk          Yes     Yes      Yes      INPUT
        intf1.clk    No      No      No      INPUT
        intf1.dvld   No      No      No      INOUT
        intf1.ack    No      No      No      INOUT

Toggle Coverage for Instance : tb.mod_0
                    Total   Covered   Percent
Totals            3       1       33.33
Total Bits        6       2       33.33
Total Bits 0->1  3       1       33.33
Total Bits 1->0  3       1       33.33
```

Excluding Interface Signals from Design

Below is the syntax to exclude interface signals from a given design:

```
-node <instance pattern> <signal pattern>
```

Consider the code shown in [Example 3-2](#).

Example 3-2 Excluding interface signals

```
interface intf(input bit clk);
    logic          dvld;
    logic          ack;
endinterface
module mod(intf intf1, intf intf2, input clk);
    always @(posedge clk)
        $display("\n Hello World ");
endmodule

module tb();
    logic clock;
    intf intf_0(clock);

    mod mod_1(intf_0,clock);
    mod mod_2(intf_0,clock);
    initial begin
        clock = 0;
        forever #3 clock = ~clock;
    end
    initial
        #15 $finish;
Endmodule
```

You can use the following command to exclude signal `ack` from all instances of module `mod`:

```
-node tb.mod_*  intf*.ack
```

Note:

This solution also applies to structure, which has nesting similar to interface signals.

Interface Expansion Limitations

The following limitations apply for interface expansion at the port boundary using the `-cm_tgl fullintf` option:

- Enums are not supported within interfaces. You can infer real values of enum changes from FSM coverage, which shows transitions between two enum states. For example:

```
interface intf(input bit clk);
    typedef enum reg[1:0] {pink,red,yellow} color;
    color d;
    clocking master_cb @(posedge clk);
        inout d;
    endclocking
    modport master (import master_cb.d);
endinterface

module tb();
    logic clock;
    intf intf_0(clock);
endmodule
```

- Port interfaces which have an instance array in the path between the ref and actual handle are not supported. Consider the following example:

```
interface wbone;
    logic cyc_o;
    modport dut_slave( output cyc_o);
endinterface

module dut(wbone.dut_slave sifc_0);
    reg m_select;
```

```

endmodule

module dut_shell(wbone sifc[1] );
dut dut1(sifc[0].dut_slave);
endmodule

module top;
  wbone wbone_sifc[1] ();
  dut_shell mydut(wbone_sifc);
endmodule

```

In the above example, coverage for instance `top.mydut` and `top.mydut.dut1` is not reported because `top.mydut` has an instance array in its port list, and `top.mydut.dut1` has an instance array in the path between its `ref` instantiation and actual instantiation.

- Nested interfaces are not supported.

Interfaces instantiated within other interfaces are not reported at the port boundary of a given module. In previous releases, at compile time, VCS considered an instantiation of interface within interface as a separate instance, and did not count its interface signal list. Thus, coverage reports only interface self signals (that is, port signals + local signals) at any ref port. It does not report nested interface signals at the port boundary. For example, consider the following code:

```

interface intf(input bit clk);
endinterface

interface nst_intf(input bit nst_clk, intf s1);
  ntf s2(nst_clk);
endinterface
module mod(nst_intf intf1,input clk);
endmodule

module Top();

```

```

    logic clock;
    intf intf_0(clock);
    nst_intf nst_intf_1(clock,intf_0);
    mod mod_0(nst_intf_1,clock);
endmodule

```

Thus, in the above example, for interface `nst_intf`, interface `s1` is part of the port list of interface `nst_intf`, but interface `s2` is the physical instantiation inside the interface `nst_intf`. So, for instance `top.mod_0` of module `mod`, the following list of signals is reported.

```

Top.mod_0.clk
Top.mod_0.intf1.nst_clk
Top.mod_0.intf1.s1.clk

```

`s2` is a physical instantiation of an interface. In any case `s2` is reported as a separate instance:

```
Top.nst_intf_1.s2
```

In general, all nested interface data reported for a given interface at a module port boundary can be too much data, given that nesting of interfaces can be deep. Nested interfaces are always reported as separate instances anyway.

Therefore, for precise and clear reporting, it is better to report only one level of interface data for a given interface in a module.

The Condition Coverage Report

By default, condition coverage reports shows the conditions True or False for all the logical operators and its sensitized vectors for the expression and its sub-expressions, as shown in 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

Figure 3-9 Example of a Condition Coverage Report

Cond Coverage for Instance : test_jukebox.st1.kp1

	Total	Covered	Percent
Conditions	37	17	45.95
Logical	37	17	45.95
Non-Logical	0	0	
Event	0	0	

```
LINE      55
EXPRESSION (go && (press || x_not) && (!oe_s))
           -1-----2-----3-----
```

-1-	-2-	-3-	Status
-----	-----	-----	--------

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

```
LINE      55
SUB-EXPRESSION (press || x_not)
           -1--  -2--
```

-1-	-2-	Status
-----	-----	--------

0	0	Covered
0	1	Not Covered
1	0	Covered

When there are nested conditions, URG reports them hierarchically. For example, using the expression (go && (press || x_not) && (!oe_s)), the two terms for the operator && is (go) and (press || x_not).

The subexpression (press || x_not) is then broken down into its terms, press and ((x_not)) . These are reported separately in the second section.

Condition Coverage Report Splitting for Large Number of Conditions

A module may contain an extremely large number of conditions and at times the report page may be too large for a browser to handle.

Using URG, you can separate the condition report itself into its own page. But it can happen that the condition section by itself is still too large. In such cases with large expressions, the URG report will be split in a list format. You can control the splitting threshold using the URG option -split N. If page size of the line coverage annotation exceeds N, it is moved into a new page, and links are created for navigation.

Figure 3-10 List format for coverage text report with large expressions

```
Term  
additional_rd_port_num or  
hi_reg_0 or  
reg_0 or  
reg_1 or  
reg_10 or  
reg_11 or  
reg_12 or  
reg_13 or  
reg_14 or  
reg_15 or  
reg_16 or  
reg_17 or  
reg_18 or  
reg_19 or  
reg_2 or  
reg_20 or  
reg_21 or  
reg_22 or  
reg_23 or  
reg_24 or  
reg_25 or  
reg_26 or  
reg_27 or  
reg_28 or  
reg_29 or  
reg_3 or  
reg_30 or  
reg_31 or  
reg_32 or  
reg_33 or  
reg_34 or  
reg_35 or  
reg_36 or  
reg_37 or  
reg_38 or  
reg_39 or  
reg_4 or  
reg_40 or
```

Figure 3-11 HTML report to cover large expressions

The FSM Coverage Report

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

Figure 3-12 Example of an FSM Coverage Summary Report

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

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

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

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

The Branch Coverage Report

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

Figure 3-13 Original Source Code

```
1 always@(posedge clk)
2     if(rst)
3         chg_cnt <=#2 3'h0;
4     else
5         begin
6             if((chg_cnt > 3'h0)&&(y_tot || x_not))
7                 begin
8                     chg_cnt <=#2 chg_cnt - 1;
9                     nck_pulse <=#2 1'h1;
10                end
11            else
12                begin
13                    chg_cnt <=#2 change;
14                    nck_pulse <=#2 1'h0;
15                end
16        end
```

Figure 3-14 URG Branch Coverage Report

```

1   always@(posedge clk)
2       if(rst)
3           -1-
4           ==>
5               chg_cnt <=#2 3'h0;
6       else
7           begin
8               if((chg_cnt > 3'h0)&&(y_tot || x_not))
9                   -2-
10                  ==>
11                      begin
12                          chg_cnt <=#2 chg_cnt - 1;
13                          nck_pulse <=#2 1'h1;
14                      end
15                  else
16                      ==>
17                          begin
18                              chg_cnt <=#2 change;
19                              nck_pulse <=#2 1'h0;
20                          end
21
22      end

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

```

In Figure 3-14, the URG branch coverage report first shows the source code, which contains the branch alternatives for a given branch, with each branch control highlighted and indexed with a number. Subsequently, the source code is followed by a table showing the different combinations and the coverage status of the control branches.

One difference between URG and DVE Coverage GUI reports is that URG displays an arrow (==>) for each branch.

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

Figure 3-15 The Same Example Showing Covered Branches

```

1  always@(posedge clk)
2      if(rst)
3          -1-
4          ==>
5              chg_cnt <=#2 3'h0;
6      else
7          begin
8              if((chg_cnt > 3'h0)&&(y_tot || x_not))
9                  -2-
10                 ==>
11                     begin
12                         chg_cnt <=#2 chg_cnt - 1;
13                         nck_pulse <=#2 1'h1;
14                     end
15             end
16         end

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

```

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

[Figure 3-16](#) displays the same source code with different coverage. Note that the source code and index of control branch -2- are in red because it is not fully covered, that is, the else statement branch is not covered.

Figure 3-16 The Same Example Showing Different Coverage

```

1   always@(posedge clk)
2       if(rst)
3           -1-
4           ==>
5           begin
6               if((chg_cnt > 3'h0)&&(y_tot || x_not))
7                   -2-
8                   ==>
9                   begin
10                  chg_cnt <=#2 chg_cnt - 1;
11                  nck_pulse <=#2 1'h1;
12              end
13          else
14              ==>
15          begin
16              chg_cnt <=#2 change;
17              nck_pulse <=#2 1'h0;
18          end
19      end
20
21      BRANCH      -1-      -2-
22          1      -      | Covered
23          0      1      | Covered
24          0      0      | Not Covered

```

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

Figure 3-17 Coverage Example of a Ternary Operator

```
316      tri[7:0]dsko=oe_s ?dsko_n:8'hzz;  
          -1-  
  
BRANCH      -1-  
    1      | Not Covered  
    0      | Covered
```

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

Figure 3-18 Example of Multiple Branches on a Single Line

```
15 if(a) x <= 3'h0; else if(y) x <= 3'h1; else x <= 3'h2;  
-1-           -2-  
==>           ==>  
==>  
  
BRANCH      -1-      -2-      |  Covered  
    1        -          |  Covered  
    0        1          |  Covered  
    0        0          | Not Covered
```

Figure 3-19 Case Statements Example

```

327      case(state)
328          -1-
329      idle:
330
331      if(go &&(press || x_not)&&(! oe_s))
332          -2-
333          ==>
334      begin
335          kp_hold=1'h1;
336          n_state=hold;
337      end
338
339      hold:
340      if(oe_s &&(y_tot > 0))begin
341          -3-
342          ==>
343          begin
344              n_state=hold;
345              kp_hold=1'h1;
346          end
347      service:begin
348          ==>           this is the ‘service’ branch of ‘state’
349      end
350  endcase

```

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

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

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

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

The Assertion Coverage Report

URG reports the details of an assertion. It tells you how many times the assertion has succeeded and failed.

Summary Tables

The URG asserts.html page includes three summary tables, as shown in [Figure 3-20](#). Every first column of these summary tables provides a hyperlink to navigate from summary table to detail table (see [Figure 3-22](#)).

Figure 3-20 Summary tables in asserts.html page

Summary for Assertions

	NUMBER	PERCENT
Total Number	9	100.00
Not Covered	8	88.89
At Least 1 Real Success	1	11.11
At Least 1 Failure	3	33.33
At Least 1 Incomplete	0	0.00
Without Attempts	6	66.67

Summary for Cover Sequences

	NUMBER	PERCENT
Total Number	2	100.00
Not Covered	1	50.00
All Matches	1	50.00
First Matches	1	50.00

Summary for Cover Properties

	NUMBER	PERCENT
Total Number	4	100.00
Not Covered	4	100.00
Matches	0	0.00

Detail Report for Assertions

The Detail Report for Assertions includes the following tables, as shown in [Figure 3-22](#).

- Assertions Not Covered
- Assertions At Least 1 Real Success
- Assertions At Least 1 Failure
- Assertions At Least 1 Incomplete
- Assertions Without Attempts

- Assertions Excluded

Note:

The “Assertions Without Attempts” table will not be included in “Detail Report for Assertions”, if there are no incomplete assertions.

Information related to excluded assertions will be displayed in a separate row in “Summary for Assertions” table, as shown in [Figure 3-21](#).

Figure 3-21 Viewing excluded assertions

Summary for Assertions

	NUMBER	PERCENT
Total Number	9	100.00
Not Covered	5	55.56
At Least 1 Real Success	1	11.11
At Least 1 Failure	3	33.33
At Least 1 Incomplete	0	0.00
Without Attempts	3	33.33
Excluded	3	33.33

The list of all excluded assertions will be displayed in “Assertions Excluded” table, as shown in [Figure 3-22](#), in “Detail Report for Assertions”.

Figure 3-22 Tables in “Detail Report for Assertions”

Detail Report for Assertions

Assertions Not Covered:

ASSERTIONS	CATEGORY	SEVERITY	ATTEMPTS	REAL SUCCESSES	FAILURES	INCOMPLETE
dummy.inst1.c2	30	0	0	0	0	0
dummy.inst1.c3	2	0	0	0	0	0
dummy.inst1.c4	3	0	0	0	0	0
top.ins1.c3	2	0	20	0	20	0
top.ins1.c4	3	0	20	0	20	0

Assertions At Least 1 Real Success:

ASSERTIONS	CATEGORY	SEVERITY	ATTEMPTS	REAL SUCCESSES	FAILURES	INCOMPLETE
top.ins1.c2	1	0	20	2	18	0

Assertions At Least 1 Failure:

ASSERTIONS	CATEGORY	SEVERITY	ATTEMPTS	REAL SUCCESSES	FAILURES	INCOMPLETE
top.ins1.c2	1	0	20	2	18	0
top.ins1.c3	2	0	20	0	20	0
top.ins1.c4	3	0	20	0	20	0

Assertions Without Attempts:

ASSERTIONS	CATEGORY	SEVERITY	ATTEMPTS	REAL SUCCESSES	FAILURES	INCOMPLETE
dummy.inst1.c2	30	0	0	0	0	0
dummy.inst1.c3	2	0	0	0	0	0
dummy.inst1.c4	3	0	0	0	0	0

Assertions Excluded:

ASSERTIONS	CATEGORY	SEVERITY	EXCLUSION	EXCLUDE ANNOTATION
dummy.c10	30	0	Excluded	
dummy.c8	10	0	Excluded	user excluded anno
dummy.c9	20	0	Excluded	

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

The ATTEMPTS column lists the number of times VCS began 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 the assert statement succeeded. A real success is when the entire expression succeeds or matches without the vacuous successes. This column is color-coded. A cell with 0 value shows there are no real success and is colored in red and the success values are shown in green.

The MATCHES column lists the total number of times the `cover` statement matched. The MATCHES column is color-coded according to its content. A cell with a 0 value is considered not covered and is displayed in red, while a cell with a non-zero value is considered covered and is displayed in green.

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

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

Detail Report for Cover Sequence

The “Detail Report for Cover Sequence” section now includes the following tables, as shown in [Figure 3-23](#).

- Cover Sequences Not Covered
- Cover Sequences All Matches
- Cover Sequences First Matches
- Cover Sequences Excluded

Figure 3-23 Tables in “Detail Report for Cover Sequence”

Detail Report for Cover Sequences

Cover Sequences Not Covered:

COVER SEQUENCES	CATEGORY	SEVERITY	ATTEMPTS	ALL MATCHES	FIRST MATCHES	INCOMPLETE
dummy.inst1.c5		30	0	0	0	0

Cover Sequences All Matches:

COVER SEQUENCES	CATEGORY	SEVERITY	ATTEMPTS	ALL MATCHES	FIRST MATCHES	INCOMPLETE
top.ins1.c5		1	0	20	2	0

Cover Sequences First Matches:

COVER SEQUENCES	CATEGORY	SEVERITY	ATTEMPTS	ALL MATCHES	FIRST MATCHES	INCOMPLETE
top.ins1.c5		1	0	20	2	0

Detail Report for Cover Properties

The Detail Report for Cover Properties includes the following tables:

- Cover Property Not Covered
- Cover Property with Matches
- Cover Properties Excluded

The Covergroup Report

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

Figure 3-24 Example of a Covergroup Report

Summary for Variable RD

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	121	116	5	4.13

User Defined Bins for RD

Uncovered bins

NAME	COUNT	AT LEAST	NUMBER
SUPERHIGH_2a	0	1	1
SUPERHIGH_2b	0	1	1
SUPERHIGH_2c	0	1	1
SUPERHIGH_2d	0	1	1
SUPERHIGH_2e	0	1	1
SUPERHIGH_2f	0	1	1
SUPERHIGH_30	0	1	1
SUPERHIGH_31	0	1	1
SUPERHIGH_32	0	1	1
SUPERHIGH_33	0	1	1

Covered bins

NAME	COUNT	AT LEAST
WAYHIGH_10	80	1
WAYHIGH_20	60	1
HIGH	140	1
MED	40	1
LOW	60500	1

In [Figure 3-25](#), the VCS coverage feature hole analysis compresses 5 bins into a single row for the UNCOVERED BINS table. The following is an example of cross details:

Figure 3-25 Example of a Detailed Cross Table

Summary for Cross RDxWD

Samples crossed: RD WD

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT	MISSING
Automatically Generated Cross Bins	484	474	10	2.07	474

Automatically Generated Cross Bins for RDxWD

Element holes

RD	WD	COUNT	AT LEAST	NUMBER
[SUPERHIGH_2a, SUPERHIGH_2b, SUPERHIGH_2c, SUPERHIGH_2d, SUPERHIGH_2e, SUPERHIGH_2f, SUPERHIGH_30, SUPERHIGH_31, SUPERHIGH_32, SUPERHIGH_33, SUPERHIGH_34, SUPERHIGH_35, SUPERHIGH_36, SUPERHIGH_37, SUPERHIGH_38, SUPERHIGH_39, SUPERHIGH_3a, SUPERHIGH_3b, SUPERHIGH_3c, SUPERHIGH_3d, SUPERHIGH_3e, SUPERHIGH_3f, SUPERHIGH_40, SUPERHIGH_41, SUPERHIGH_42, SUPERHIGH_43, SUPERHIGH_44, SUPERHIGH_45, SUPERHIGH_46, SUPERHIGH_47, SUPERHIGH_48, SUPERHIGH_49, SUPERHIGH_4a, SUPERHIGH_4b, SUPERHIGH_4c, SUPERHIGH_4d, SUPERHIGH_4e, SUPERHIGH_4f, SUPERHIGH_50, SUPERHIGH_51, SUPERHIGH_52, SUPERHIGH_53, SUPERHIGH_54, SUPERHIGH_55, SUPERHIGH_56]				

Uncovered bins

RD	WD	COUNT	AT LEAST	NUMBER
[WAYHIGH_10]	[MED, LOW, TR]	--	--	3
[WAYHIGH_20]	[MED]	0	1	1
[WAYHIGH_20]	[TR]	0	1	1
[HIGH]	[MED]	0	1	1
[HIGH]	[TR]	0	1	1
[MED]	[LOW, TR]	--	--	2
[LOW]	[TR]	0	1	1

Excluded/Illegal bins

RD	WD	COUNT
*	[ig_bins]	--
*	[il_bins]	--

Excluded (121 bins)

Illegal (121 bins)

Covered bins

RD	WD	COUNT	AT LEAST
WAYHIGH_20	HIGH	30	1
WAYHIGH_20	LOW	30	1
HIGH	HIGH	110	1
HIGH	LOW	30	1
MED	HIGH	20	1
MED	MED	20	1

There is also a section for each point and cross showing the

individual coverage percentage, information about the point or cross, and so on.

Viewing Results for Coverage Group Variants

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

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

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

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

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

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

```

        }
    }

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

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

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

    @(posedge CLOCK);
}

```

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

```

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

```

```

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

```

Understanding Covergroup Page Splitting

In some situations, the detailed covergroup report can become quite large, which can cause difficulties when you load and view the report. You can split such large reports into meaningful smaller pages using the option `-split N`.

The `-split N` option controls the size of all files before being split. The argument is an integer specifying the maximum size in kilobytes (KB) for any generated file. This number is used as a guideline, not an absolute limit. The default value is 200KB.

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

You can split covergroup using three methods: Instance splitting, Bin table splitting, and Variable/Cross splitting.

Instance splitting

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

URG never splits a group report.

Page name: `original_page_name + "_i" + instance_id`

For example, `grp1_i1.html`, `grp1_i2.html`, `grp2_i1.html`, ...

Bin table splitting

The bin table splitting has two modes, single page mode and multiple page mode.

single page mode

This mode can be used for bin tables that are big enough to have their own pages, but not big enough for further splitting. If a bin table has more than "split_bin", but less than "bin_part" rows, it will be moved into an individual bin table page. In this situation, the

covergroup page displays a note alerting you to the multiple-page splitting that URG has performed. The covergroup page also contains links to the various pages.

Page name: original_page_name + "_bin" + bin_id

For example, grp1_bin1.html, grp1_bin2.html,
grp2_1_bin1.html, ...

In the bin table page, there are:

- Ordinary page head and foot.
- The bin table.
- Links "Go back" pointing to exactly where the table used to be in the covergroup page.

In the following figures, [Figure 3-26](#) shows the original report and [Figure 3-27](#) shows how the report is split.

Figure 3-26 Single page mode

Summary for Variable cp1

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Automatically Generated Bins	64	5	59	92.19

Automatically Generated Bins for cp1

Uncovered bins

NAME	COUNT	AT LEAST	NUMBER
[auto[56:59]]	0	1	1
[auto[116:119]]	0	1	1
[auto[144:147]]	0	1	1
[auto[176:179]]	0	1	1
[auto[188:191]]	0	1	1

Covered bins

[Click here to see the table](#)

[Go to top](#)

Figure 3-27 Single Page Splitted Report

Covered bins for cp1

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

[Go back](#)

Automatically Generated Bins for cp1

Covered bins

NAME	COUNT	AT LEAST
------	-------	----------

auto[0:3]	4	1
auto[4:7]	1	1
auto[8:11]	3	1
auto[12:15]	4	1
auto[16:19]	4	1
auto[20:23]	5	1
auto[24:27]	5	1
auto[28:31]	6	1
auto[32:35]	4	1

[Go back](#)

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



multiple pages mode

If a bin table has more than "split_bin" and "bin_part" rows, it will be split into several pages, each has at most "bin_part" rows. This mode is used for bin tables that are so huge that further splitting is a must.

Notice that in this mode sort works only in the current bin table sub-page.

Page name: original_page_name + "_bin" + bin_id + "_" + page_id

For example, grp1_bin1_1.html, grp1_bin1_2.html,
grp2_1_bin1_1.html, ...

In the covergroup page, the original bin table is substituted by the following elements:

- A note "This cross/variable has %d bins so the list has been split into multiple HTML pages. Click on the page number below to see each subset of bins."
- A summary table (page link / expected / uncovered / covered / percent).

In the following figures, [Figure 3-28](#) shows the original report and [Figure 3-29](#) shows how the report is split.

Figure 3-28 Multiple page

Summary for Variable cp1

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Automatically Generated Bins	64	5	59	92.19

Automatically Generated Bins for cp1

Uncovered bins

NAME	COUNT	AT LEAST	NUMBER
[auto[56:59]]	0	1	1
[auto[116:119]]	0	1	1
[auto[144:147]]	0	1	1
[auto[176:179]]	0	1	1
[auto[188:191]]	0	1	1

Covered bins

This variable has 59 bins so the list has been split into multiple HTML pages.

Click on the page number below to see each subset of bins.

PAGE	EXPECTED	UNCOVERED	COVERED	PERCENT
1	10	0	10	100.00
2	10	0	10	100.00
3	10	0	10	100.00
4	10	0	10	100.00
5	10	0	10	100.00
6	9	0	9	100.00

[Go to top](#)

Figure 3-29 Multiple Page Splitted Report

Covered bins for cp1

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

[Go back](#)

< 1 2 [3] 4 5 6 >

Notice: sort works only in the current subset of bins.

Automatically Generated Bins for cp1

Covered bins

NAME	COUNT	AT LEAST
auto[84:87]	3	1
auto[88:91]	1	1
auto[92:95]	2	1
auto[96:99]	4	1
auto[100:103]	1	1
auto[104:107]	1	1
auto[108:111]	5	1
auto[112:115]	3	1
auto[120:123]	8	1
auto[124:127]	4	1

[Go back](#)

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

Variable or Cross Page Splitting

In variable/cross splitting method, instead of reporting each variable/

cross after the summary table on the same page, they are split into sub-pages linked by the summary table. The splitting is controlled by the URG option -split N.

If the covergroup page name is "grp4.html", the sub-page names are "grp4_1.html", "grp4_2.html"

Covergroup instance splitting works together with variable/cross splitting.

If the covergroup page name is "grp4.html", the split instance page names are "grp4_i1.html", "grp4_i2.html"...., then sub-page names become "grp4_i1_1.html", "grp4_i1_2.html"

For example in the following [Figure 3-30](#), RD/WD/RDXWD has been split into multiple sub-pages.

Figure 3-30 Variable/Cross Splitting



Correlation Report: Which Tests Covered Which Coverage Bins

When you have multiple test files (intermediate results files) from multiple simulations, URG merges the coverage results so that if

something is covered in one, but not all the test files, URG reports it as covered. In covergroup and assertion coverage, you can have URG indicate in the `modinfo.txt`/`mod*.html` pages which test covered each assertion or bin. This feature is enabled with the `-show tests` option.

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 -tests file -show tests  
urg -dir mydir.vdb -tests file -show maxtests N
```

In test correlation mode, only the functional coverage and assertion coverage matrices 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 3-31](#):

Figure 3-31 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 3-32](#):

Figure 3-32 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 3-33](#) shows that the test name for test T2 is `simv2.vdb/test`.

Figure 3-33 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/	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

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 3-34](#):

Figure 3-34 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.

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 (Diff 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 as follows:

```
urg -dir mydir1.vdb mydir2.vdb -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 and 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 functional and assertion 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 is 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 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.

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 with the `-diff` flag, an error is reported and URG exits.

- `-grade`
- `-annotate`
- `-trend`
- `-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.

Reporting Only Uncovered Objects (`-show brief`)

This section describes how URG allows you to create reports only for uncovered objects (instead of creating reports for all coverable objects).

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 brief -format text
```

Brief Report for Line Coverage

For line coverage reports (which provides annotated source code that shows coverable objects and which objects are not covered), URG shows only the lines that contain uncovered statements (“uncovered lines”), 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
```

Brief Report for Condition Coverage:

```
LINE      510
EXPRESSION (a || (b && c))
           1   -----2---
```

-1- -2- Status

0	0	Covered
0	1	Covered
1	0	Covered

```
LINE      510
EXPRESSION (b && c)
           1     2
```

-1- -2- Status

0	1	Not Covered
1	0	Covered
1	1	Covered

The condition that is fully covered is omitted in the brief report because that has 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.

Brief Report for 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 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.

URG Support for Uncovered Only (`-show brief`) Feature on Selected Metrics

In prior versions of the software, the uncovered only (`-show brief` option) feature had a global effect on all metrics. The module list and hierarchy were also adjusted to form a consistent report.

URG now supports the uncovered only (`-show brief` option) feature on selected metrics, as shown below:

```
% urg -dir simv.vdb -show brief [metrics...]
```

In the above command, `metrics` is an optional argument. It specifies the metrics that should be reported in uncovered only mode, and uses the same format as the `urg -metric` option (such as `line+cond+group`). If you do not specify `metrics`, URG defaults to the original uncovered only mode.

If you specify `metrics`, URG enables uncovered mode only on the selected metrics. Other metrics, together with the module list and hierarchy, remain in their default mode. For example, if you use the `-show brief assert` option, the module list looks like the default mode, as shown in [Figure 3-35](#).

Figure 3-35 Show Brief Assert Output

Total Module Definition Coverage Summary			
SCORE	FSM	ASSERT	
88.10	76.19	100.00	
Total modules in report: 4			
SCORE	FSM	ASSERT	NAME
85.29	70.59	100.00	coin_fsm
100.00	100.00		jukebox
			test_jukebox
			station

But the difference is that since assert metric is selected as -show brief, the assert score box for the module `coin_fsm` has no link. Because it is fully covered, there is no detailed report for it. FSM detailed reports exists for both `coin_fsm` and `jukebox`.

As a comparison, in the original -show brief mode, fully covered and empty modules are not displayed (see [Figure 3-36](#)).

Figure 3-36 Fully Covered and Empty Modules Not Displayed

Total Module Definition Coverage Summary			
SCORE	FSM	ASSERT	
88.10	76.19	100.00	
Total modules in report: 1			
SCORE	FSM	ASSERT	NAME
85.29	70.59	100.00	coin_fsm

Similar to the module list, if you turn on selected metrics for -show brief, the hierarchy looks the same as the default mode (see [Figure 3-37](#)).

Figure 3-37 Selected Metrics Turned On

SCORE	FSM	ASSERT	
27.42	34.83	20.00	test_jukebox
SCORE	FSM	ASSERT	
100.00	100.00		jb1
17.65	35.29	0.00	st0
SCORE	FSM	ASSERT	
17.65	35.29	0.00	coin1

The jb1 shown in [Figure 3-37](#) is not displayed in the original -show brief mode.

In the detailed report, the detailed module and instance coverage report are reported only for metrics selected by -show brief [metrics...].

Uncovered Only Examples

Consider the examples shown in [Figure 3-38](#), [Figure 3-39](#), and [Figure 3-40](#).

Figure 3-38 Example-1: Line Coverage for Module: fifo

	Line No.	Total	Covered	Percent
TOTAL		17	15	88.24
INITIAL	25	3	3	100.00
ALWAYS	32	7	6	85.71
ALWAYS	42	7	6	85.71

```

38      else if (write & (full != 0) )
39      0/1 (1 unreachable)    ==> $display ($stime, " tried
to write a full fifo");
40                                /**
41                                always @ (posedge clk)
42                                end

```

```
48          else if (read & empty)
49      0/1 (1 unreachable)  ==> $display ($stime, " tried
to read a full fifo");
```

Figure 3-39 Example-2: Toggle Coverage for Module: fifo

	Total	Covered	Percent	
Totals	9	6	66.67	
Total Bits	90	68	75.56	
Total Bits 0->1	45	36	80.00	
Total Bits 1->0	45	32	71.11	
Ports	7	4	57.14	
Port Bits	74	52	70.27	
Port Bits 0->1	37	28	75.68	
Port Bits 1->0	37	24	64.86	
Signals	2	2	100.00	
Signal Bits	16	16	100.00	
Signal Bits 0->1	8	8	100.00	
Signal Bits 1->0	8	8	100.00	
Port Details				
	Toggle	Toggle 1->0	Toggle 0->1	Direction
data_in[5]	No	No	Yes	INPUT
data_in[7:6]	No	No	No	INPUT
data_in[13]	No	No	Yes	INPUT
data_in[15:14]	No	No	No	INPUT
full	No	No	No	OUTPUT

Figure 3-40 Example-3: Branch Coverage for Module: fifo

	Line No.	Total	Covered	Percent
Branches		6	4	66.67
IF	32	3	2	66.67
IF	42	3	2	66.67

```
32      if ((write > 0) & !full)
-1-
```

```

33      begin
34          dfifo [head]    <= # `DELAY data_in;
35          ==>
36          # `DELAY head = head + 1;
37          e_count = e_count + 1;
38      end
39      else if (write & (full !=0) )
-2-
39      $display ($stime, " tried to write a full fifo");
        ==>
        MISSING_ELSE
        ==>

```

Branches:

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

Summary for Variable WD

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	4	1	3	75.00

User Defined Bins for WD

Uncovered bins

NAME	COUNT	AT LEAST	NUMBER
TR	0	1	1

Excluded/Illegal bins

NAME	COUNT
il_bins	0 Illegal
ig_bins	0 Excluded

Covered bins

NAME	COUNT	AT LEAST
HIGH	140	1
MED	40	1
LOW	60510	1

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 shown with 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 do not have 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	HNWOOVISDIPV_0
<hr/>				
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 modules and instances 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.

Summary Only URG Reports

The generation of detailed URG reports for very large coverage models can require a large memory footprint and be time consuming to load and display in your browser. To reduce this overhead, you can view a URG summary report. Since this feature does not generate the detailed report, it improves the performance of URG.

Use Model

URG provides the following command-line option for generating the summary of reports:

`urg –show summary <n>`

where, *n* is optional value to specify how many levels of hierarchy of reports should be generated.

Regardless of the value of *n*, or by default, URG generates the following report pages:

- dashboard
- modlist (code or assertion coverage)
- asserts (assertion coverage)
- groups (group coverage)
- tests

URG disables the following while generating the summary only reports:

- `modinfo.txt`, `modN.html`, `modN-1.html`, and so on for code or assertion coverage.
- `grpinfo.txt`, `grpN.html`, `grpN-1.html`, and so on for group coverage.
- Links to the detailed report pages in HTML reports.

If the value of `n` is:

- greater than zero and less than the maximum level of hierarchy, then URG generates the hierarchy until the specified level.
- greater than the maximum level of hierarchy, then URG generates the entire hierarchy.
- zero, then URG does not generate any hierarchy.
- less than zero, then URG generates an error to mention that no hierarchy is generated.

Coverage Report Files

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

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

Common Report Elements

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

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

Figure 3-41 Example of a Coverage Table

SCORE	LINE	COND	TOGGLE	FSM	BRANCH
82.76	96.39	79.17	76.08		79.41 Csx0d3

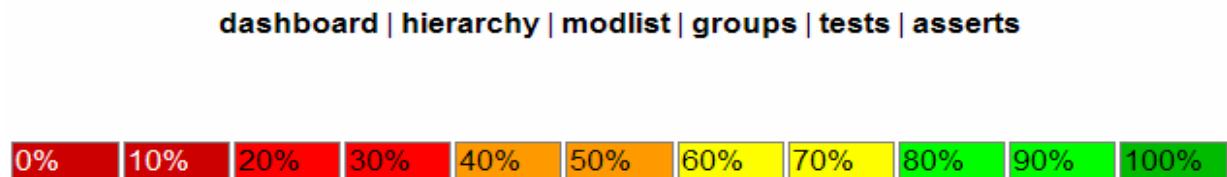
In this example, the first box shows the overall score of all metrics. By default, this is the simple average of all the metric percentages. You can control the way the score is computed with the `-scorefile` option. For more information, see “Grading and the `-scorefile` Option” in the Coverage Technology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

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

The LINE box is green because it falls in the upper range of target values. As shown in [Figure 3-41](#), values display in a range of 11 colors from red (low) to green (high). These colors are graduated every 10 percentage points (with 100 being the 11th class).

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

Figure 3-42 Color Legend for Coverage Tables



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

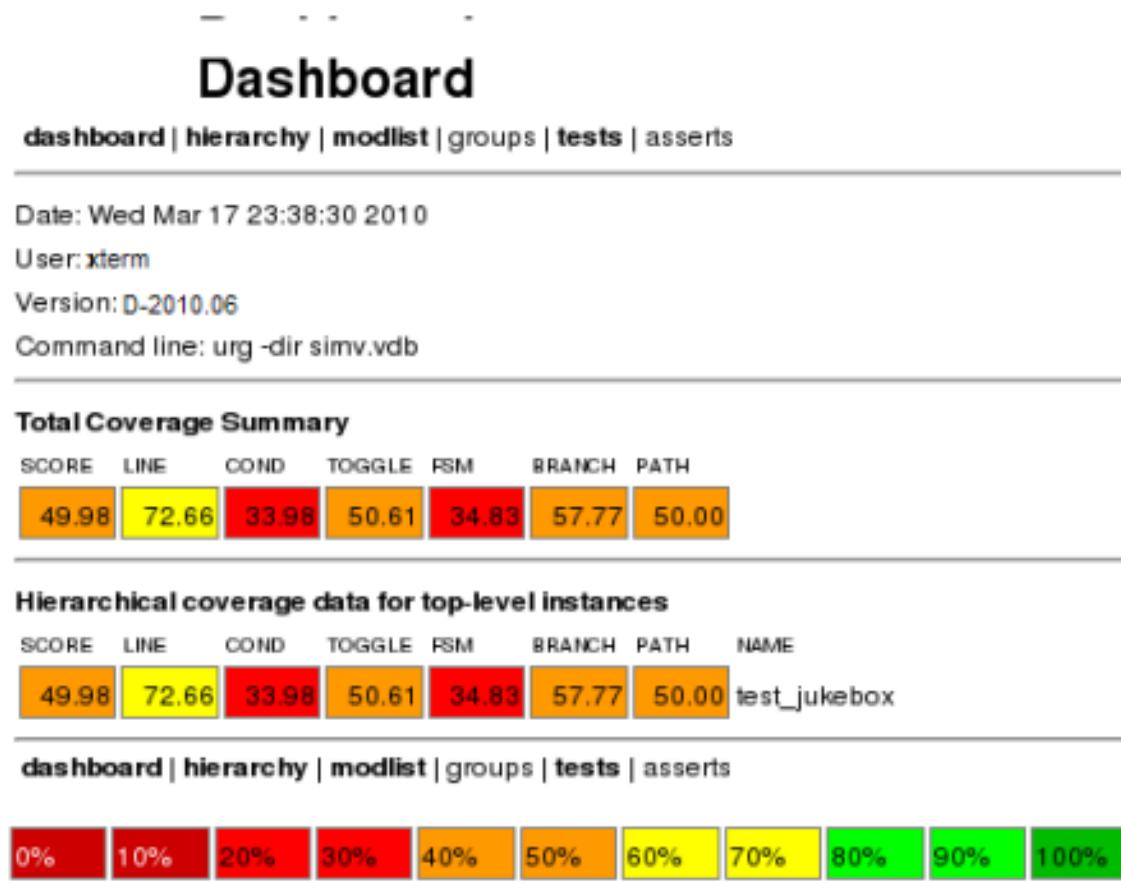
A text version of the report shown in [Figure 3-41](#) is as follows:

SCORE	LINE	COND	TOGGLE	FSM	Branch
82.76	96.39	79.17	76.08		79.41

The Dashboard File

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

Figure 3-43 Example of a Dashboard Report



Note:

- In this example report, the boldface words: **dashboard**, **hierarchy**, **modlist**, **tests**, **groups**, **asserts**, are hyperlinked to the corresponding top-level files.
- If there is no group coverage information, the **groups** will not appear in boldface.

The Hierarchy File

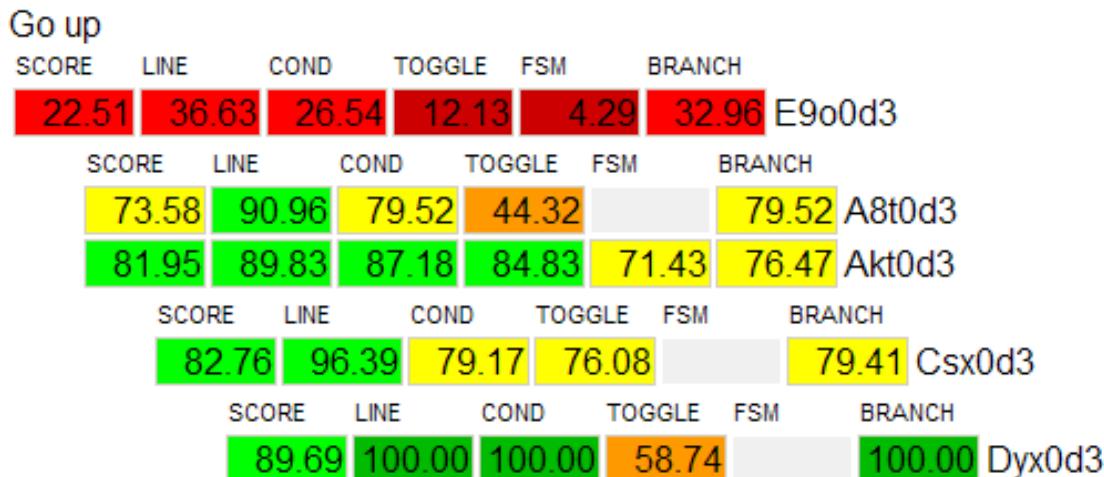
The hierarchy file (`hierarchy.html` or `hierarchy.txt`) contains indented lists of all modules, interfaces, and component instances in the design. The indentation corresponds to the design hierarchy, that is, the child modules are indented underneath their parent modules.

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

Note: In hierarchy view, the instances for code coverage and assertion coverage are shown but the instances for group coverage in `hierarchy.html` file is not shown.

[Figure 3-44](#) displays a partial section of a `hierarchy.html` page. The data shown is the cumulative coverage information of the entire subtrees at each instance. For example, the coverage for `Akt0d3` is the cumulative coverage of that instance plus the coverage of `Csx0d3` and `Dyx0d3`. To see the coverage of instance `Akt0d3`, click `Akt0d3` to open the coverage report. Notice that a mouseover will change the color of a hypertext link to red and hovering over a score turns the score red.

Figure 3-44 Example of a Hierarchy File



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

Figure 3-45 Example of a Hierarchy Broken into Multiple Pages

SCORE	LINE	COND	TOGGLE	FSM	BRANCH	PATH	ASSERT NAME
77.49	90.12	42.59	83.87	70.59	79.49	75.76	100.00 coin_fsm

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

If a particular instance itself has no coverage data, but one of its children does, the instance has a link to a `modN.html` page.

Therefore, you can still traverse through the coverage data in the `modN.html` page to the children or parents.

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

The Modlist File

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

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

Figure 3-46 Example of a Modlist File

SCORE	LINE	COND	TOGGLE	FSM ↑	BRANCH	NAME
86.95	100.00		87.50	71.43	88.89	Ydy0d3
72.50	86.24	64.29	80.57	60.00	71.43	Wsgje3
73.04	95.38	42.86	84.10	57.14	85.71	Joh9i3
83.77	97.96	88.89	84.55	57.14	90.32	Pph9i3
74.39	100.00	60.00	61.74	56.67	93.55	Hrqwf3
81.12	100.00	88.89	64.74	56.67	95.31	Osqwf3

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

The Groups File

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

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

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

Figure 3-47 Example of a Groups File

Total groups in report: 10

SCORE	INSTANCES	WEIGHT	GOAL	NAME
17.08	17.08	1	100	test_jukebox.st4.coin1::Cvr
17.91	17.91	1	100	test_jukebox.st2.coin1::Cvr
27.07	27.07	1	100	test_jukebox.st0.coin1::Cvr
27.07	27.07	1	100	test_jukebox.st1.coin1::Cvr
27.07	27.07	1	100	test_jukebox.st3.coin1::Cvr
80.11		1	100	test_jukebox.st0.coin1::atm::packet
80.11		1	100	test_jukebox.st1.coin1::atm::packet
80.11		1	100	test_jukebox.st2.coin1::atm::packet
80.11		1	100	test_jukebox.st3.coin1::atm::packet
80.11		1	100	test_jukebox.st4.coin1::atm::packet

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

The modN File

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

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

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

Figure 3-48 Example of a Module File

Module : kp_fsm

SCORE	LINE	COND	TOGGLE	FSM	BRANCH	PATH	ASSERT
74.00	89.66	54.05	60.27		88.24	77.78	

Source File(s) :

`kp_fsm.v`

Module self-instances :

SCORE	LINE	COND	TOGGLE	FSM	BRANCH	PATH	ASSERT	NAME
35.31	51.72	27.03	23.29		41.18	33.33		test_jukebox.st4.kp1
68.54	89.66	45.95	41.10		88.24	77.78		test_jukebox.st1.kp1
69.62	89.66	51.35	41.10		88.24	77.78		test_jukebox.st2.kp1
70.72	89.66	51.35	46.58		88.24	77.78		test_jukebox.st3.kp1
74.00	89.66	54.05	60.27		88.24	77.78		test_jukebox.st0.kp1

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

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

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

Figure 3-49 Example of a Module Instance File

Module Instance : vitv.HNTO_0.HNJOOVISp_0

Instance :

SCORE	LINE	COND	TOGGLE	FSM	ASSERT
80.65			80.65		

Instance's subtree :

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

Module :

SCORE	LINE	COND	TOGGLE	FSM	ASSERT	NAME
80.65			80.65			HnJOpvisp

Parent :

SCORE	LINE	COND	TOGGLE	FSM	ASSERT	NAME
97.73	100.00		95.45			HNTO_0

Subtrees :

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

As for all report pages, the coverage data boxes column headings are linked to the corresponding coverage reports. In the above example, clicking on TOGGLE in the column headings opens the toggle coverage report for module instance

[vitv.HNTO_0.HNJOOVISp_0](#). These hypertext links provide a convenient way to navigate within a modN.html page, and the only way to view coverage data reports if the modN.html page has been split for size.

Module with Parameters

URG gives cumulative coverage for all the parameterized instances of a module. If a module has been instantiated twice and parameters are different for each instantiation, then the module coverage gives the overall coverage for all the two instances.

The grp*N* Files

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

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

Figure 3-50 Example of a grpN File

Summary for Group test_jukebox.st4.coin1::Cvr

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Variables	125	122	3	25.41
Crosses	484	482	2	0.41

Variables for Group test_jukebox.st4.coin1::Cvr

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT
RD	121	120	1	0.83	100	1
WD	4	2	2	50.00	100	1

Crosses for Group test_jukebox.st4.coin1::Cvr

CROSS	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT
RDxWD	484	482	2	0.41	100	1

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

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

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

The Asserts File

The asserts report file (`asserts.html` or `asserts.txt`) displays assertions by category and severity as follows:

Assertions by Category

	ASSERT	PROPERTIES	SEQUENCES
Total	5	5	0
Category 0	5	5	0

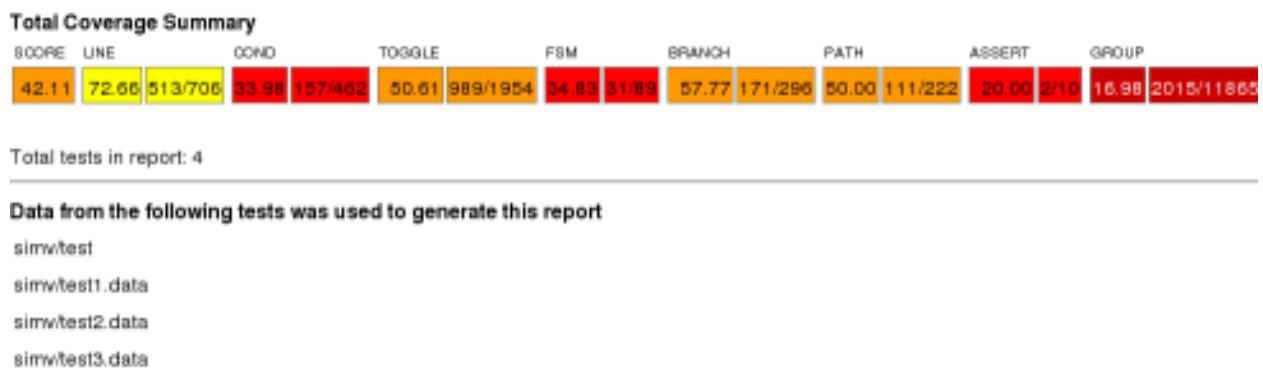
Assertions by Severity

	ASSERT	PROPERTIES	SEQUENCES
Total	5	5	0
Severity 0	5	5	0

The Tests File

The tests report file (`tests.html` or `tests.txt`) has several different formats depending on if the grading option is applied and what argument is used. The default file format of a `tests.html` is shown in [Figure 3-51](#).

Figure 3-51 Example of a Tests File



Analyzing Trend Charts

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

Quick Overview

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

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

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

This overview covers the following three topics:

- “[Generating Trend Charts](#)”
- “[Customizing Trend Charts](#)”
- “[Navigating Trend Charts](#)”

Generating Trend Charts

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

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

`-report someUniqueName`

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

`-trend root path`

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

`-trend root path rootdepth N`

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

`-trend rootfile txtfile`

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

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

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

`-trend srcfile txtfile`

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

Trend Plot

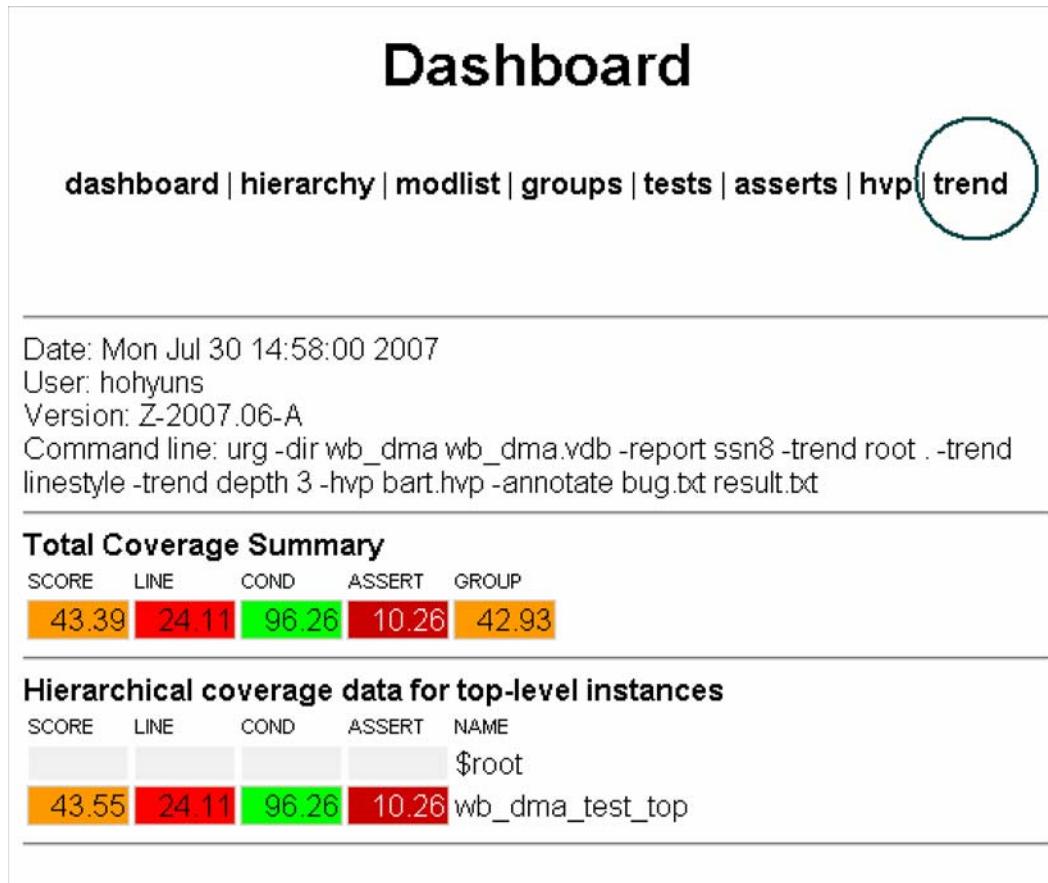
URG automatically generates trend charts by scanning previously saved URG reports to extract metrics data for trend plot. You must specify the path where URG can locate the previous saved reports. Once URG finds the reports, it will extract all the metrics to create a set of trend charts.

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

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

When you specify the `-trend` option on the command-line, URG reports will include a **trend** tab in the menu bar of trend chart dashboard. See [Figure 3-52](#).

Figure 3-52 URG Dashboard with the -trend Option



Customizing Trend Charts

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

`-trend depth number`

Sets the depth *number* of the DUT instance/HVP feature hierarchy for which URG generates trend charts. The default depth is 1, that is, only the top-level chart is generated. The more levels that you specify, the more you can drill down into the trend reports to find finer-grained details of information. The larger values of *number* also result in larger URG report sizes and longer runtimes. The report sizes and runtimes grow exponentially with respect to the depth *number*. See “[Trend Chart Linkage](#)” on page [104](#) for more details. A command-line example follows:

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

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

`-trend linestyle`

Displays a different line style (solid line, dashed line, and so on) for each curve on the trend chart, particularly useful for black-and-white printing. Refer to [Figure 3-54](#) for a chart with various line styles as compared to [Figure 3-53](#) without different line styles. In [Figure 3-54](#), not only has each line a distinct color, but also a unique line style. The line styles are automatically selected by internal line style palette. A command-line example follows:

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

`-trend offbasicavg`

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

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

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

Note:

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

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

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

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

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

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

```
setenv URG_NO_SESSION_XML
```

Figure 3-53 Trend Chart Without the linestyle Option

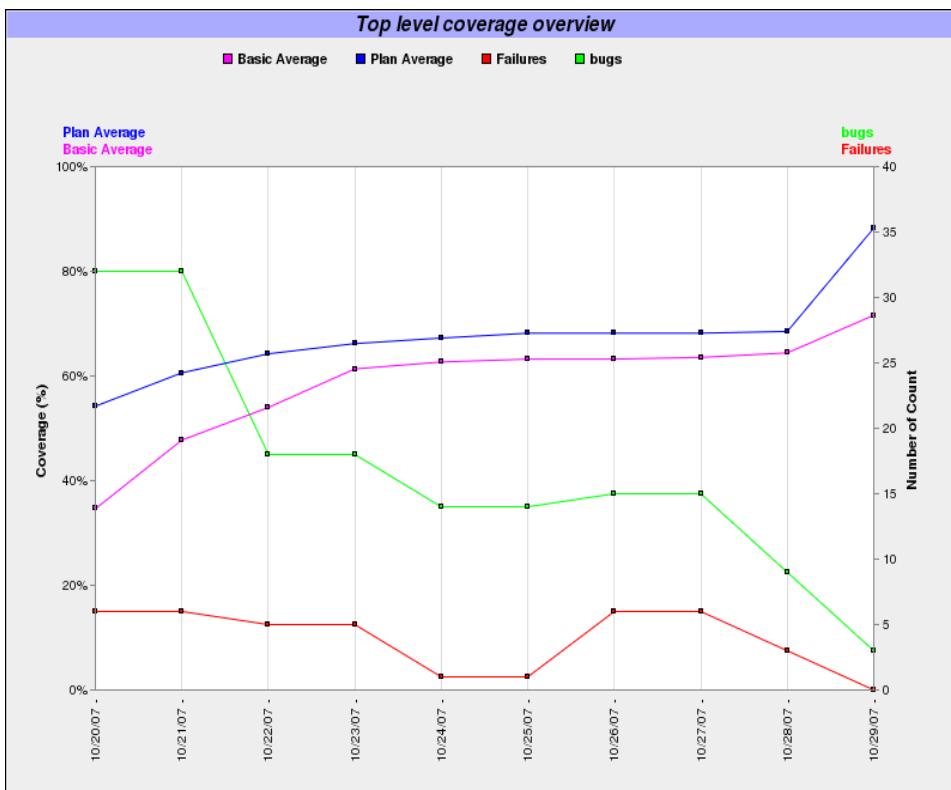


Figure 3-54 Trend Chart with the linestyle Option

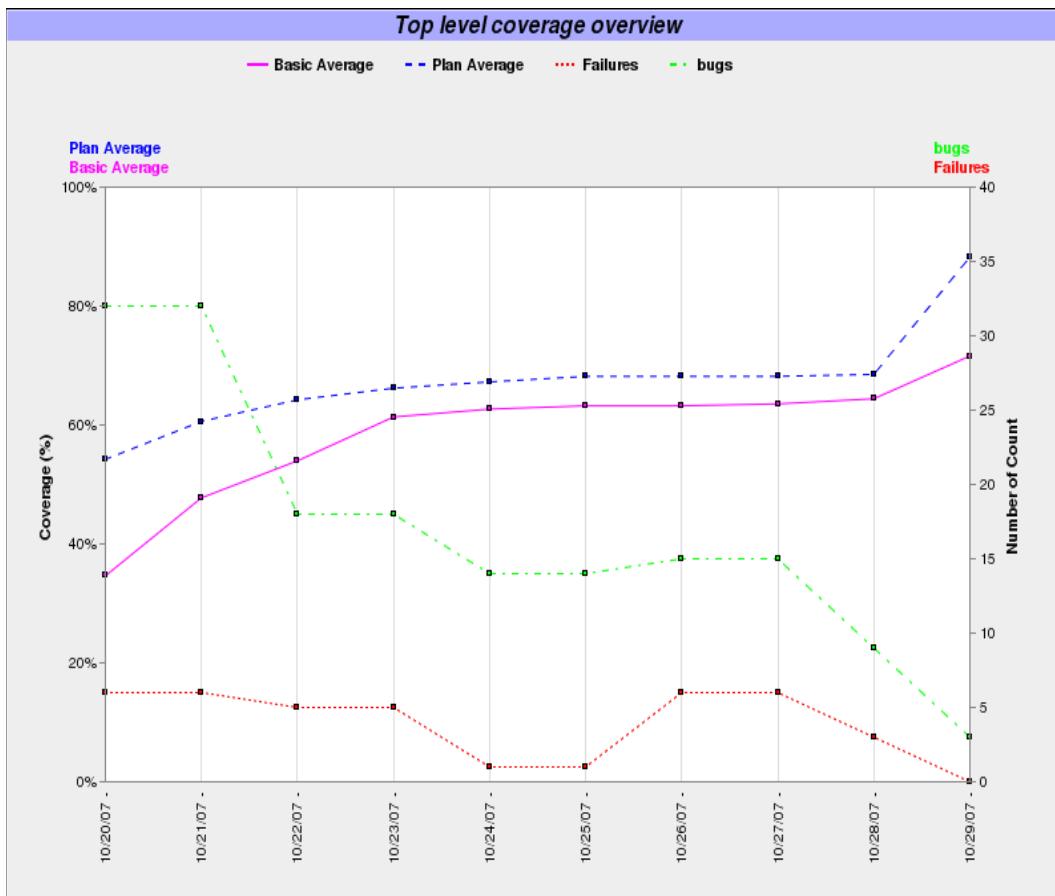
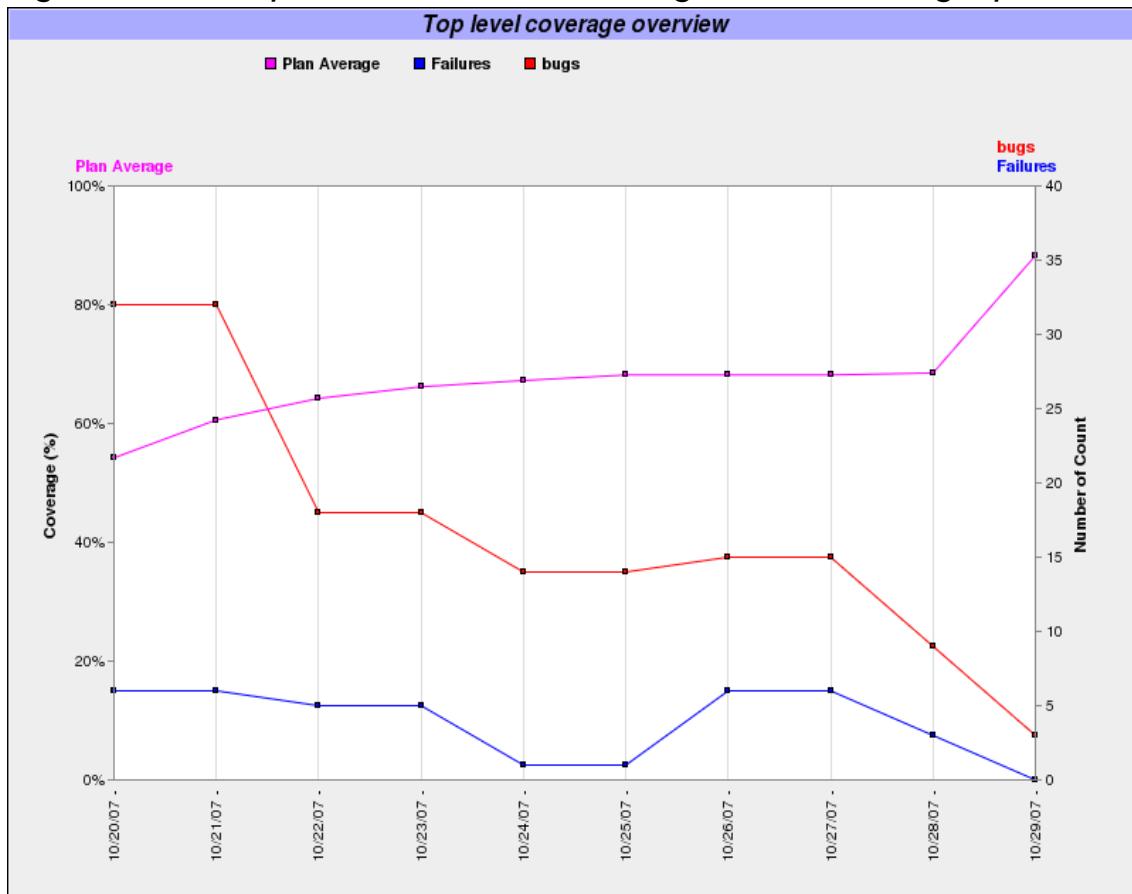


Figure 3-55 Top-level Trend Chart Using the offbasicavg Option



Navigating Trend Charts

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

URG trend charts provide the following three types of hyperlinks:

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

2. Curve hyperlinks on a trend chart link to instance-wise or feature-wise hierarchical sub-level charts.
3. Session date hyperlinks of the x-axis link to older URG reports used to generate the data points on a trend charts.

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

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

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

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

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

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

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

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

Trend Chart Linkage

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

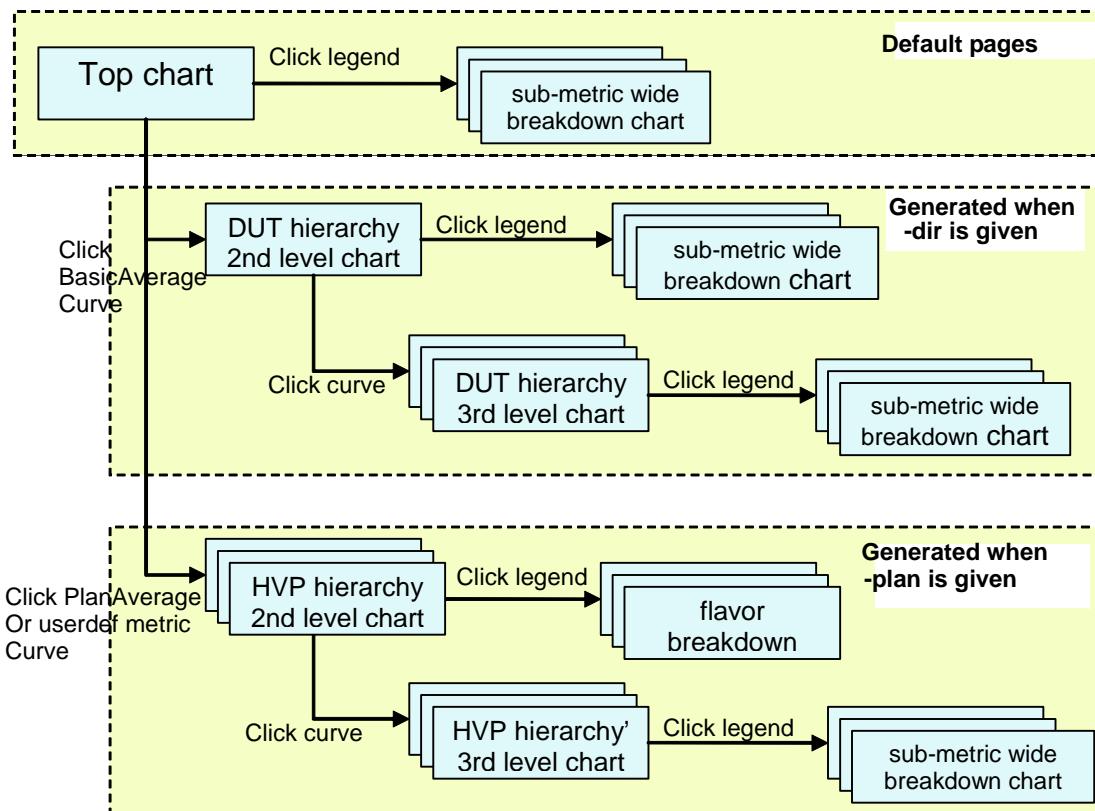
Many of the linkage paths are hierarchical, although the chart linkage does not need to be strictly hierarchical. There are two main hierarchies to consider: metric-wide and design-wide. The metric-wide hierarchy breakdown expands metrics into their sub-metrics. The design-wide hierarchy is simply the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

- [“Organization of Trend Charts”](#)
- [“Top-Level Chart”](#)
- [“Metric-Wide Breakdown Linkage”](#)
- [“Hierarchical Linkage”](#)
- [“Links to Previous Sessions”](#)

Organization of Trend Charts

One HTML page displays one or more chart images. All other charts can be accessed from the top-level chart by clicking on curves or legend labels to drill down on the metric-wide or design-wide hierarchy. [Figure 3-56](#) shows a diagram that illustrates the URG trend chart organization:

Figure 3-56 Trend Chart Hierarchy



Top-Level Chart

URG displays a top-level chart when you select the `trend` tab in the menu bar from a URG report (see [Figure 3-52](#)). It shows the raw coverage metric score which is called the Basic Average score by

default. If you use the `-plan` option, URG will display the top-level VMM Plan Average score, the Failures for tests.`.fail` metric, and other user-defined metrics such as bug count.

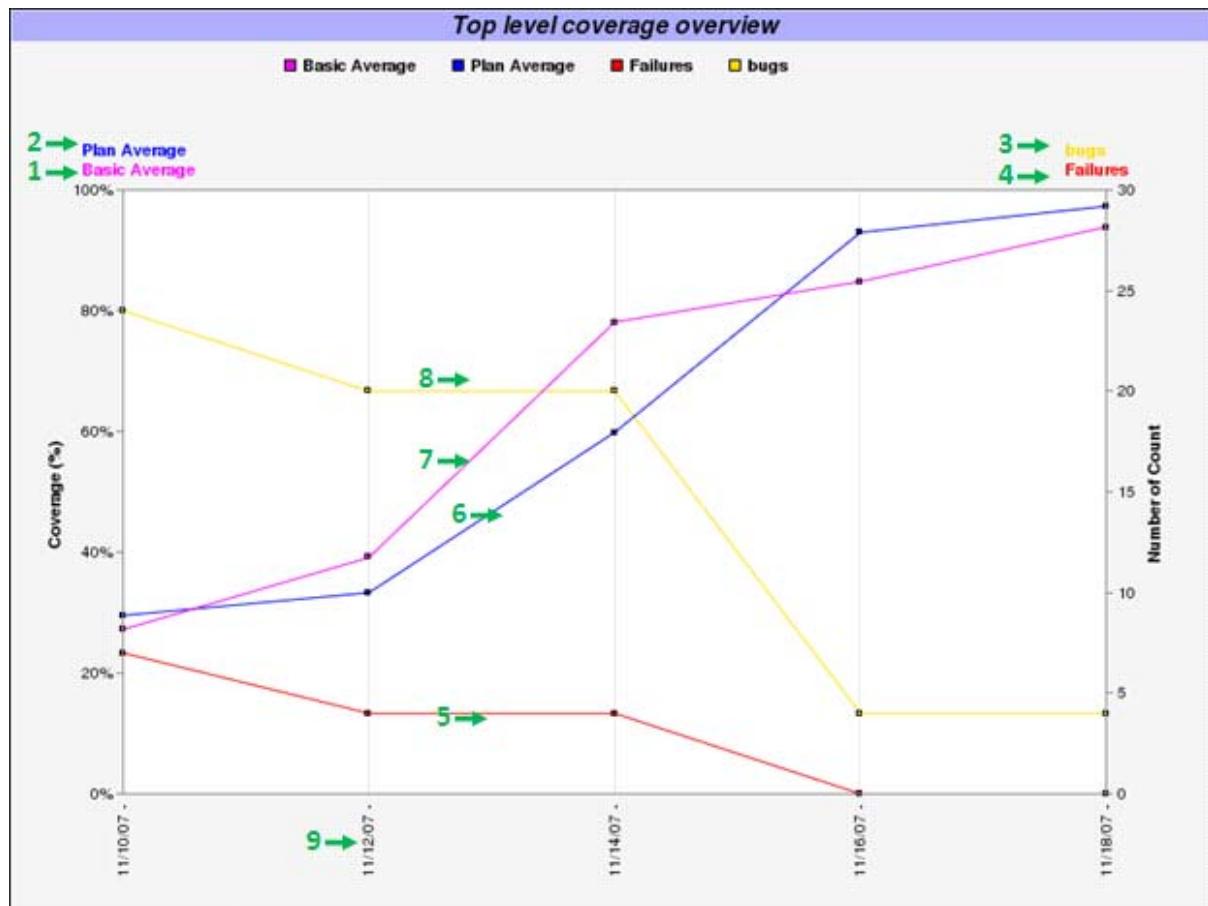
If you do not use the `-plan` option, the top-level chart will simply display a single line charting of the basic raw coverage average.

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

In [Figure 3-57](#), the top-level trend chart contains four metric legends:

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

Figure 3-57 Top-Level Chart with Linkage Description



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

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

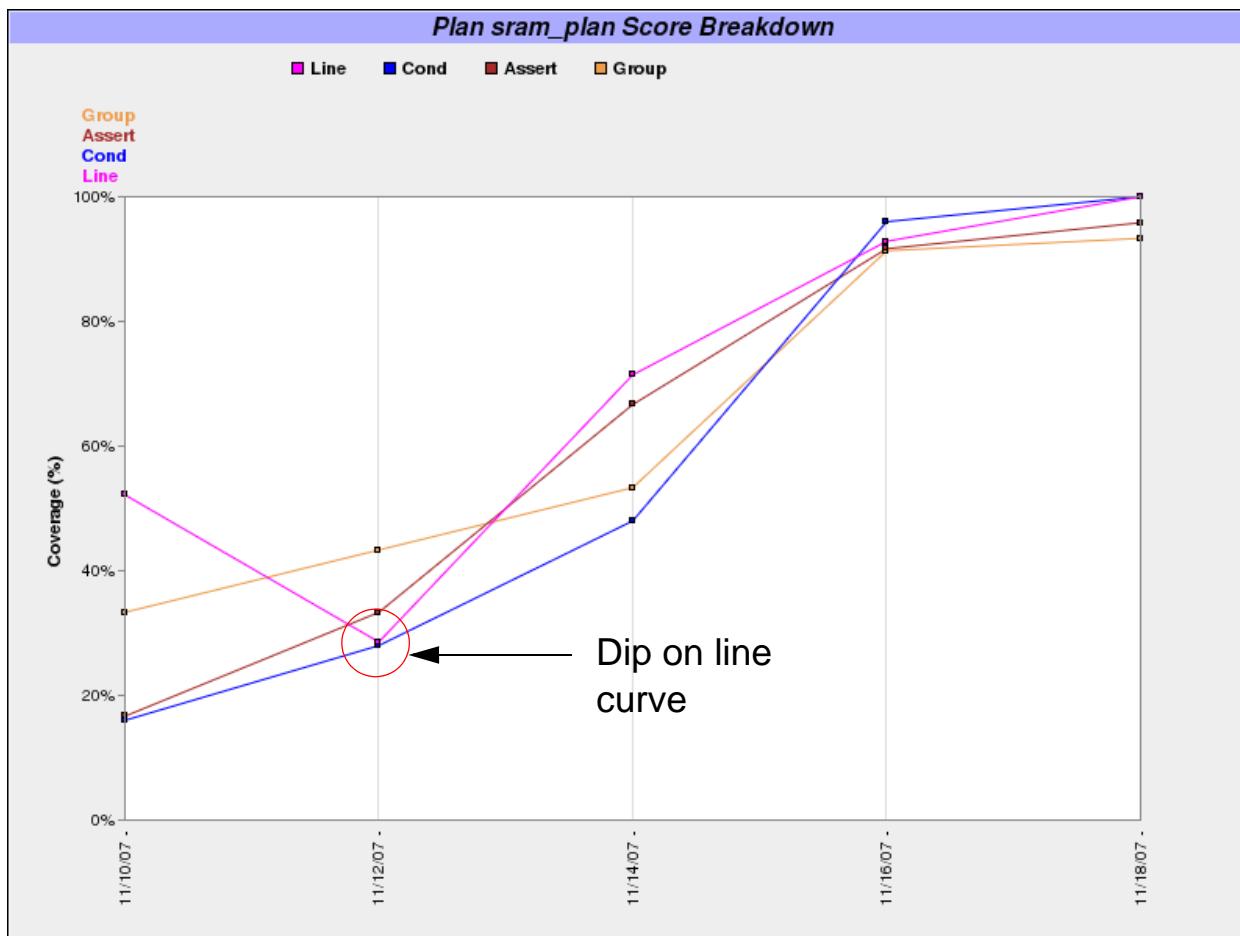
The Failures legend in [Figure 3-57](#) represents the `test.fail` count. The Failures legend links to other enumerated scores in the test metric, such as total test count, `test.pass`, `test.warn`, and so on.

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

Metric-Wide Breakdown Linkage

To view the complete breakdown chart of a metric, click a metric legend label above either of the y-axes. In [Figure 3-57](#), the legend labels at the top of the y-axis are metric-wide breakdown chart links (Group, Assert, Cond, and Line). The Basic Average and Plan Average breakdown charts show the components that comprise the average score.

Figure 3-58 Plan Average Score Breakdown Page



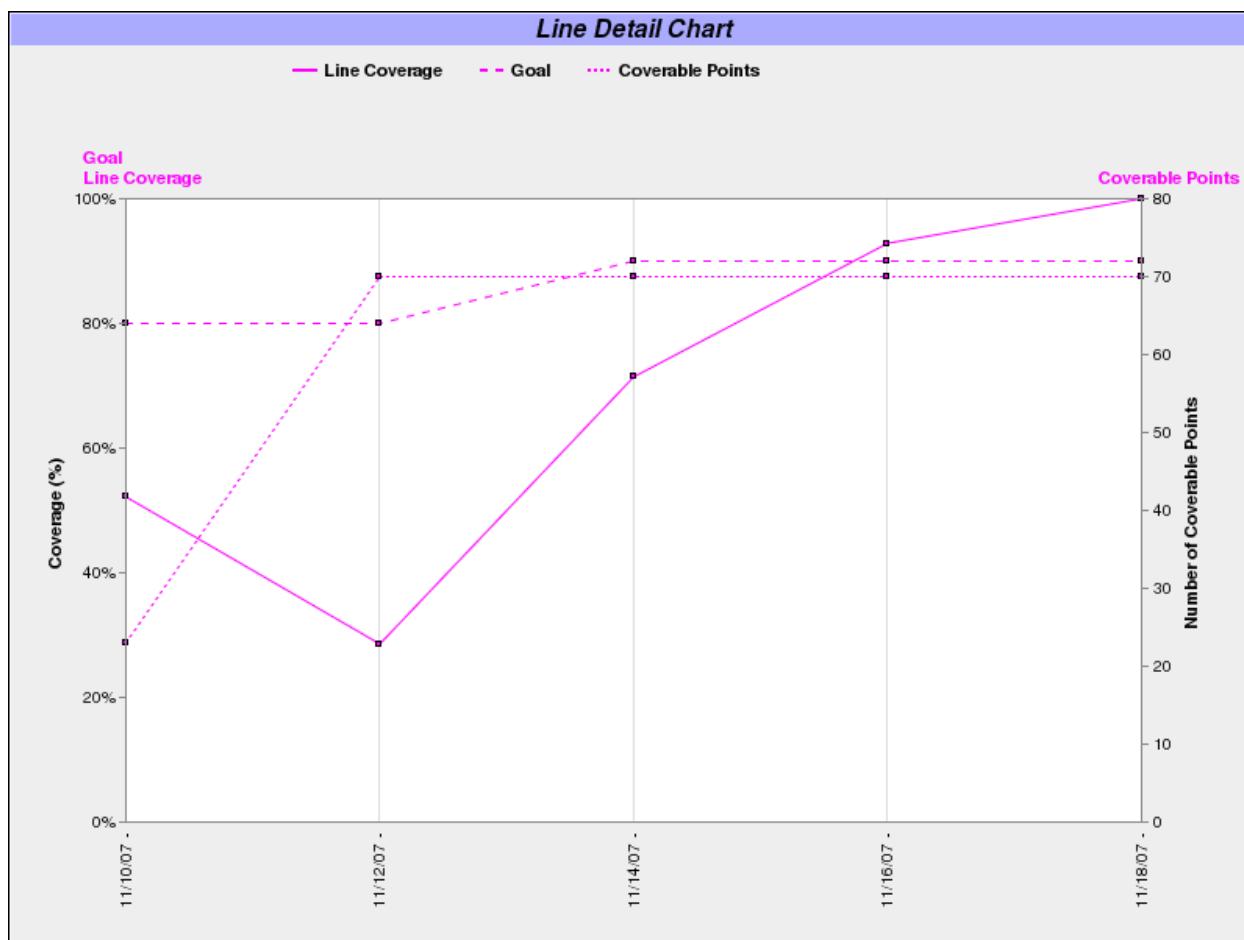
When you click the green arrow in [Figure 3-57](#), you see the Plan Average score breakdown page. This page displays multiple charts. [Figure 3-58](#) shows coverage metric curves that comprise the Plan Average score.

The metric detail curve is useful in many cases. For instance, In [Figure 3-58](#), there is a dip on the Line curve for the 11/12/2007 session. This dip could be caused the introduction of new code. If you inspect the Line detail chart, you see that total number of coverable objects suddenly increased on 11/12/2007. This increase

in coverable objects caused the drop of Line coverage score. The metric detail curve allows you to compare the goal and the real score in one chart. According to the chart, the Line score started meeting the goal on 11/16/2007.

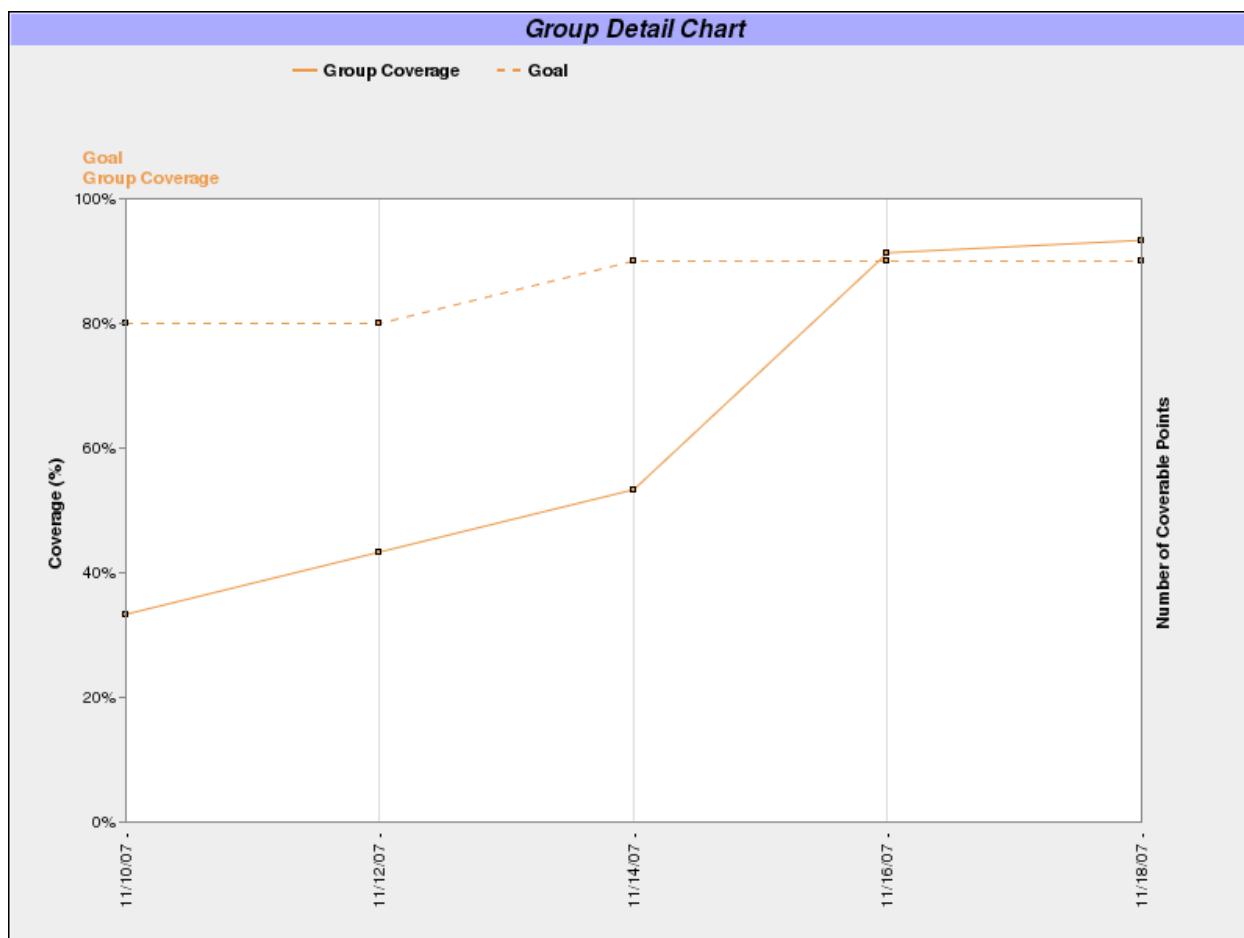
[Figure 3-59](#) and [Figure 3-60](#) show each metric's details, including the total coverable points curve and goal curve (if they exist).

Figure 3-59 Line Metric Detail for Plan Average Breakdown



In [Figure 3-59](#), the three curves represent the Line Coverage score, the Goal of the line metric, and total number of Coverable Points, respectively.

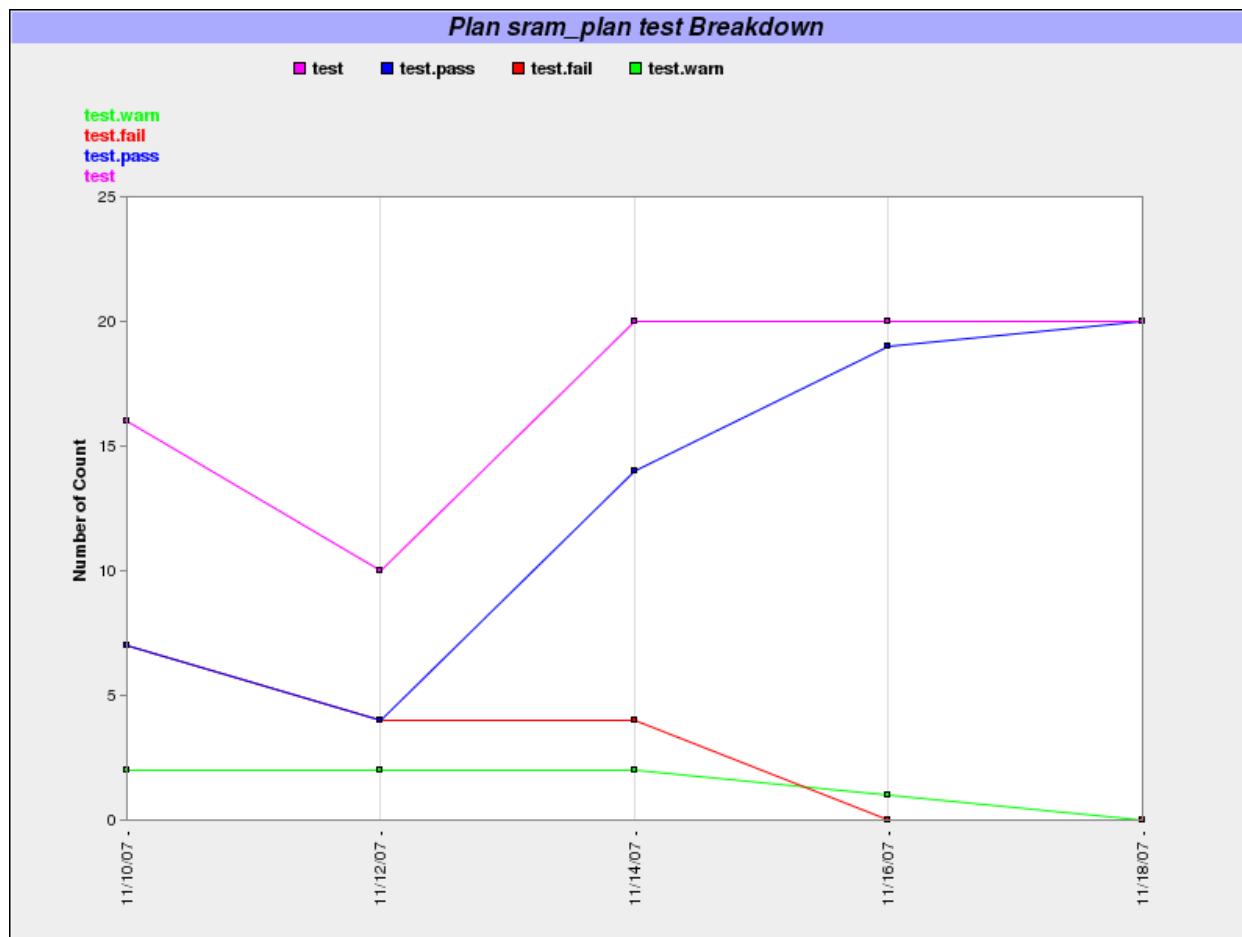
Figure 3-60 Group Metric Detail for Plan Average Breakdown



In Figure 3-60, there is no curve for the total number of coverable objects because the group coverage metric is not a ratio type, but rather is a percentage type. If you have not specified a goal for the metric, URG does not create goal curve.

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

Figure 3-61 Failure Breakdown Chart



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

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

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

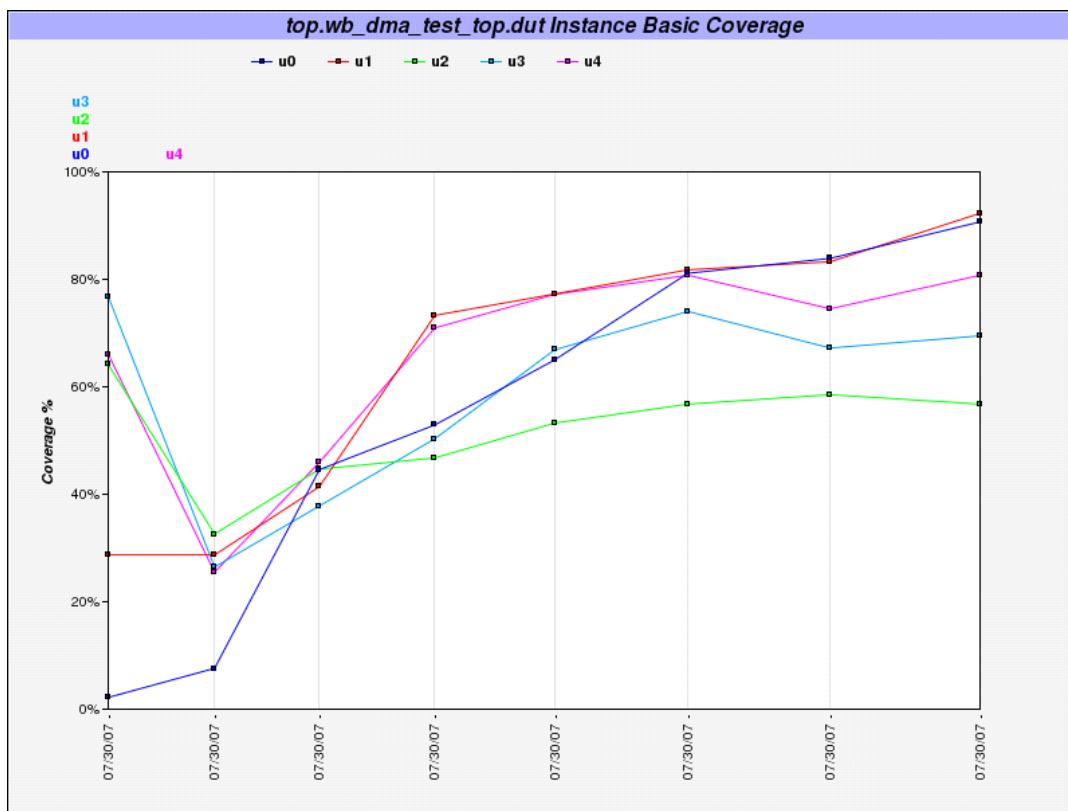
Hierarchical Linkage

You can view subhierarchy charts by clicking individual curve on a top-level chart. Take [Figure 3-57](#) for an example, the green arrows 5, 6, 7, and 8 are subhierarchy chart hyperlinks. Each curve displays the trend of a metric and is color coded to match the metric legend it represents. [Figure 3-57](#) displays:

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

For example:

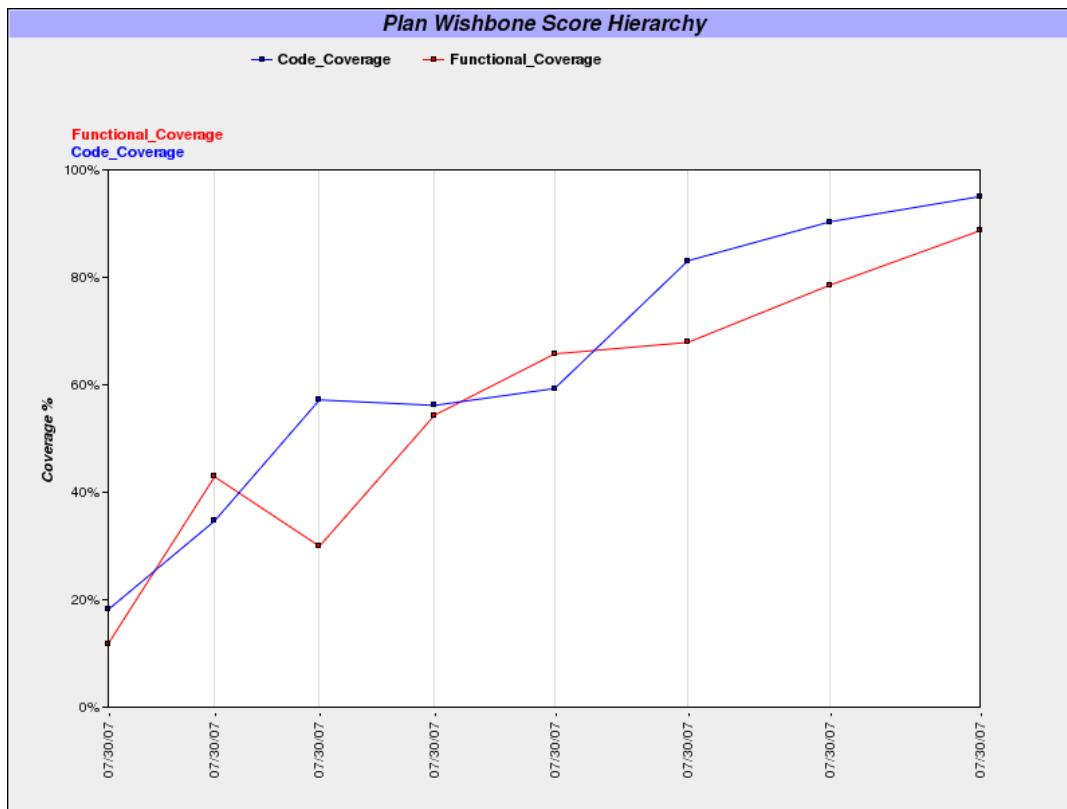
Figure 3-62 Plan Average Feature Hierarchy Chart



Click the Basic Average legend (the green arrow 1 in [Figure 3-57](#)), which links to the chart of raw coverage score as shown in [Figure 3-62](#). The top.wb_dma_test_top.dut instance contains five sub-instances. You can drill down one more level by clicking any individual curve to view the sub-instance chart. Further, the legend labels are also clickable to view the flavor breakdown charts of this average coverage score.

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

Figure 3-63 VMM Plan Hierarchy Chart for Plan Average Score



Click the Plan Average legend (the green arrow 2 in [Figure 3-57](#)), which links to the chart of coverage score curves for each sub-feature as shown in [Figure 3-63](#). Each curve is clickable if you specify the `-trend depth <number>` argument with a value of 2 or higher and there are indeed VMM Plan subhierarchies available. The legend labels above the y-axis are also clickable for flavor breakdown charts.

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

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

Links to Previous Sessions

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

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

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

4

Viewing Coverage Reports Using the DVE Coverage GUI

This chapter contains the following section:

- “Starting the DVE in Coverage Mode”
- “Using the Coverage GUI”
- “Displaying Code Coverage”
- “Displaying Assertion Coverage”
- “Displaying Testbench Coverage”
- “Filtering Instances with No Coverable Objects”
- “Working with HVP Files”
- “Working with Coverage Results”
- “Generating URG Report from DVE Coverage GUI”

Starting the DVE in Coverage Mode

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

`-cov`

Opens the DVE Coverage GUI.

`-dir <dir>`

Opens the coverage database in the given directory.

`-f <file>`

Opens the coverage directories listed in the given file.

`-tests <file>`

Opens coverage tests listed in the given file.

`-show availabletests`

Lists the available tests for the given design.

`-assert minimal`

Reports only modules and instances which have assertion in assertion coverage. Code coverage database can not be loaded with this option.

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.vdb by VCS).
- 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.

Note:

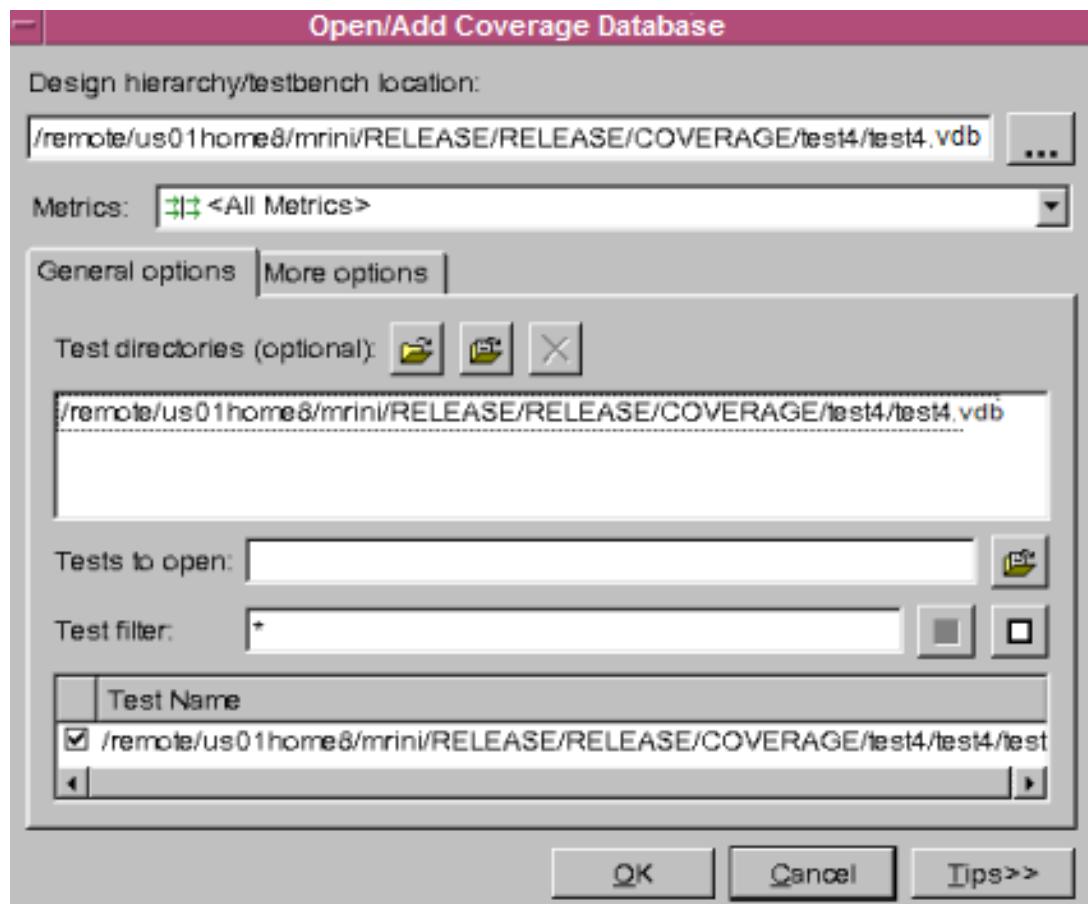
DVE can load only one design database directory at a time.

To open a coverage database

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

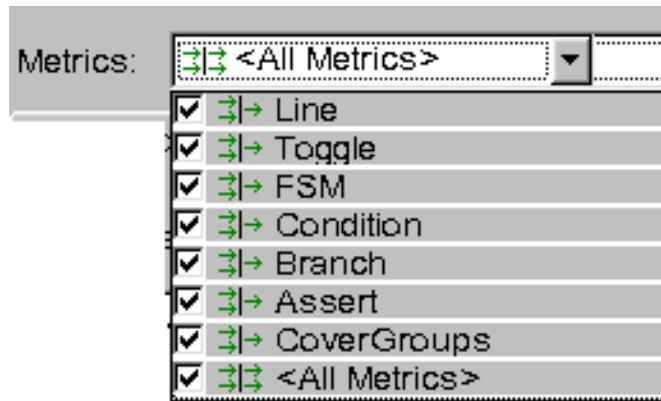
The Open/Add Coverage Database dialog box appears.

Figure 4-1

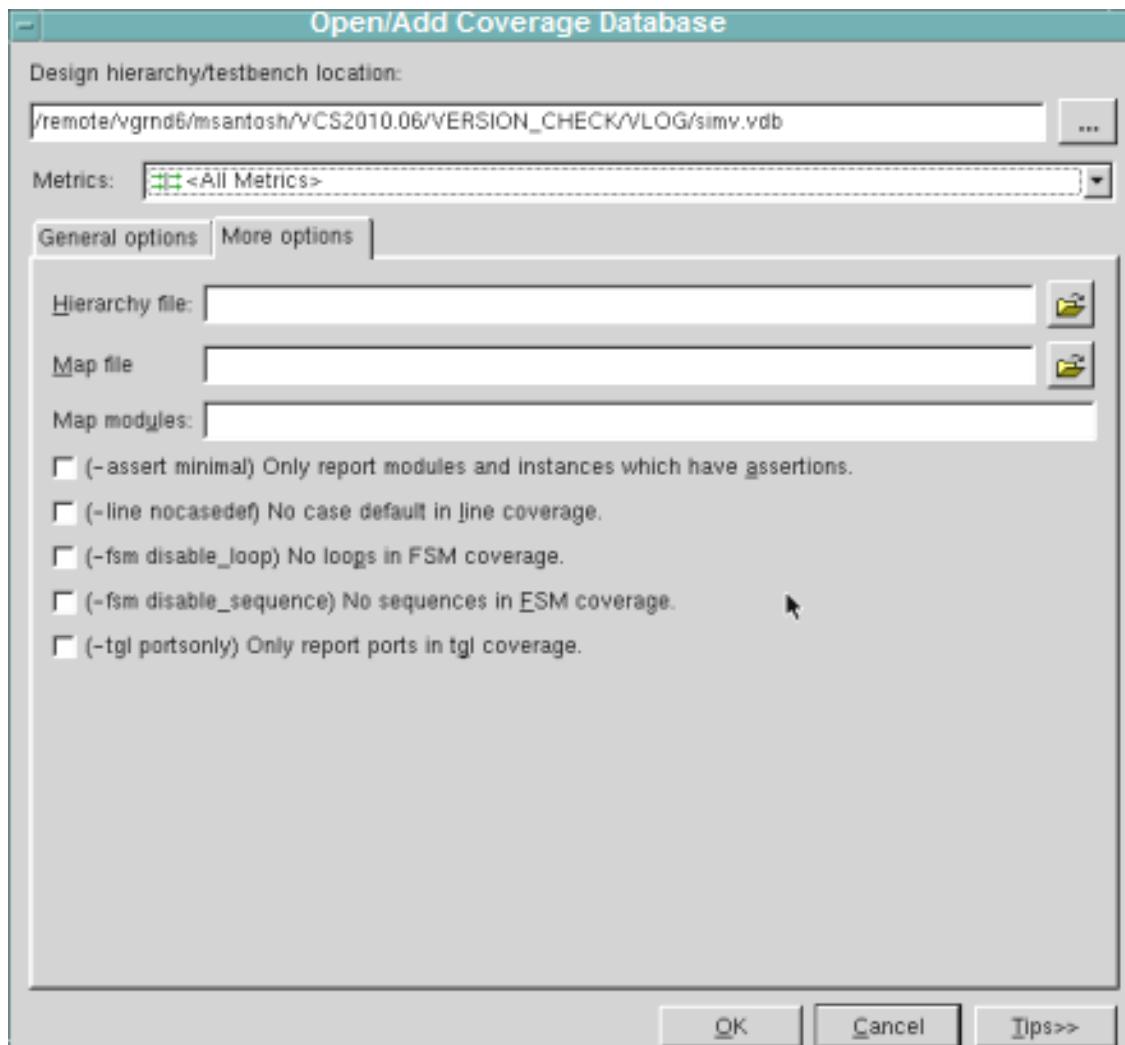


2. Click the **Browse** button below the Design hierarchy/testbench location field to locate and specify the design hierarchy/testbench location.

3. Click the **Metrics** drop-down list to select the metrics you want to include.



4. Click the **More Options** tab and select the desired coverage reporting options.



You can click the **Tips>>** button to read the description of each of these options.

5. (Optional) Click any of 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.

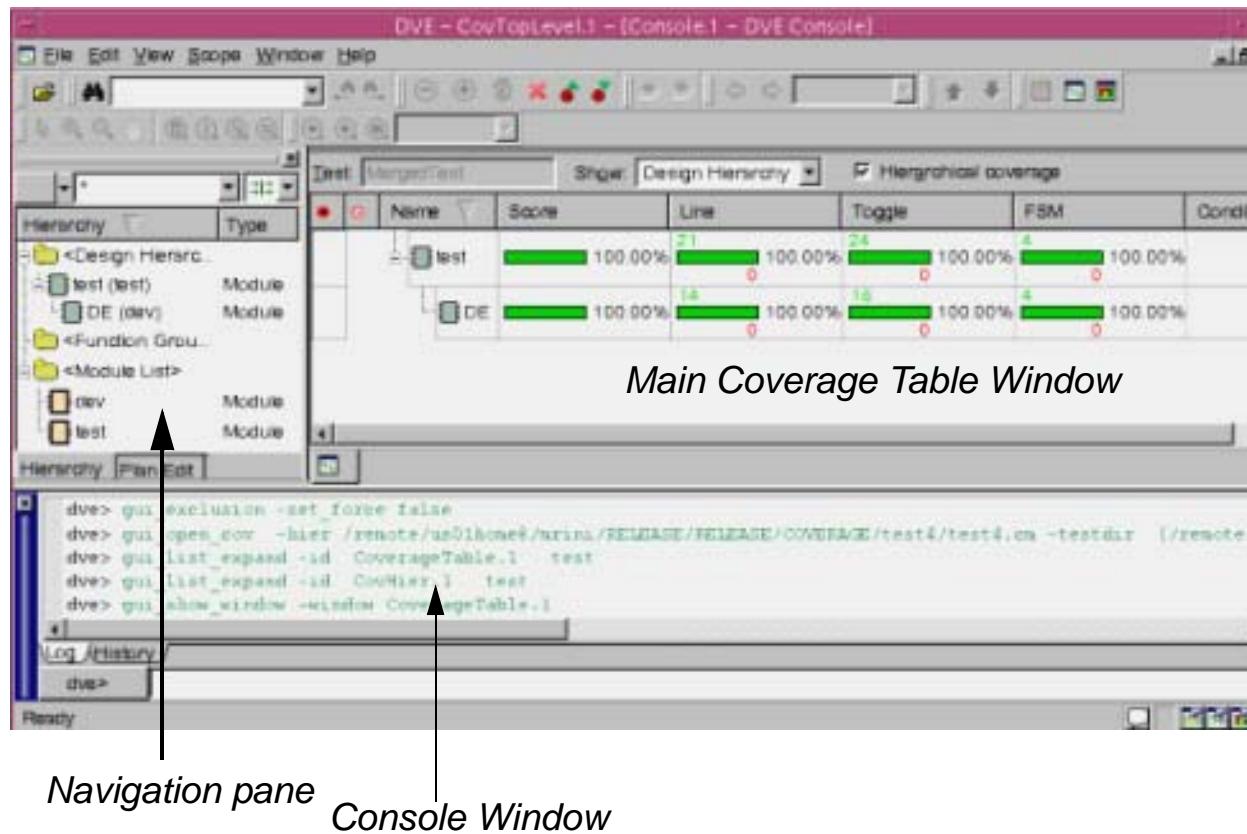
-  Displays a dialog box for selecting a filter file.
-  Toggles enabling and disabling of a filter file.
-  Selects all tests in the list. If you select more than one test file, the results of the tests are merged.
-  Clears all files in the list.

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 4-2



Loading Multiple Coverage Tests Incrementally

You can interactively load many coverage tests and merge them interactively in the DVE Coverage GUI.

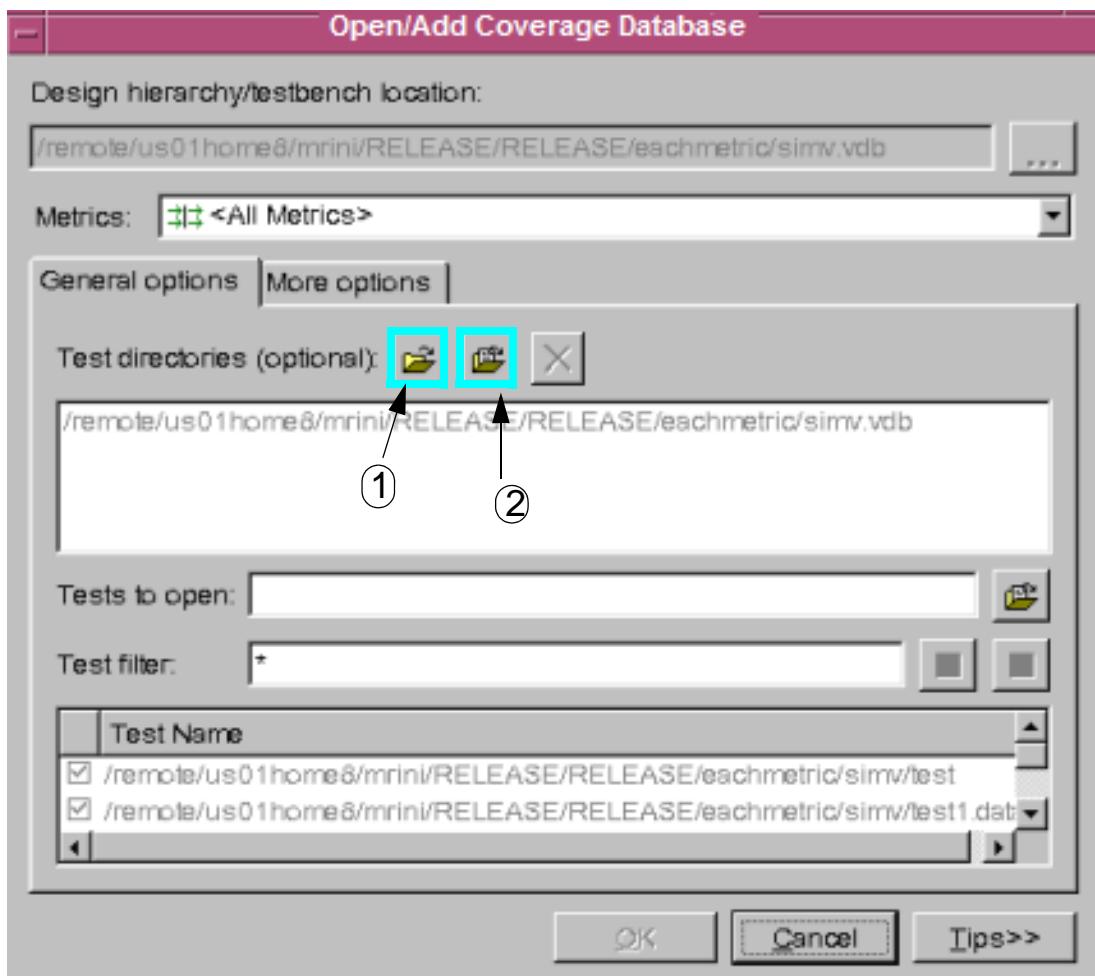
As per the previous behavior, if you select two tests and load them in the DVE Coverage GUI, the two tests are merged into a single test. If you want to load another test (3rd test), you will have to close the currently loaded database, then reopen the database and select all the three tests to load them.

With the incremental loading feature, you can re-open the Open/Add Coverage Database dialog box and select the third test, load it, and merge it together with the previously merged tests. Thus, you save your time to close, re-open, and merge the database.

To load multiple tests

1. Select **File > Open/Add database**.

The Open/Add Coverage Database dialog box appears.



2. Click the left icon from the **Test directories** field (indicated as 1 in the figure) to add a test path.

3. Repeat Step 2 to add multiple test, if required.
4. Click the right side icon from the **Test directories** field (indicated as 2 in the figure) if you have a file with test paths.

You can either perform step 2 or 3, or both to add the test directories.

5. Specify the test path in the **Tests to Open** field.

The tests are listed under the Test Name area.

6. Select or clear the check boxes against the test you want to open for the given design.
7. (Optional) Type the strings to filter the test in the **Test filter** field.
8. Click **OK** to open the databases.

Note:

- This feature is supported only for XML-based coverage database.
- While loading the test incrementally, you can not change the Design Hierarchy and Metrics.

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.

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.

Note:

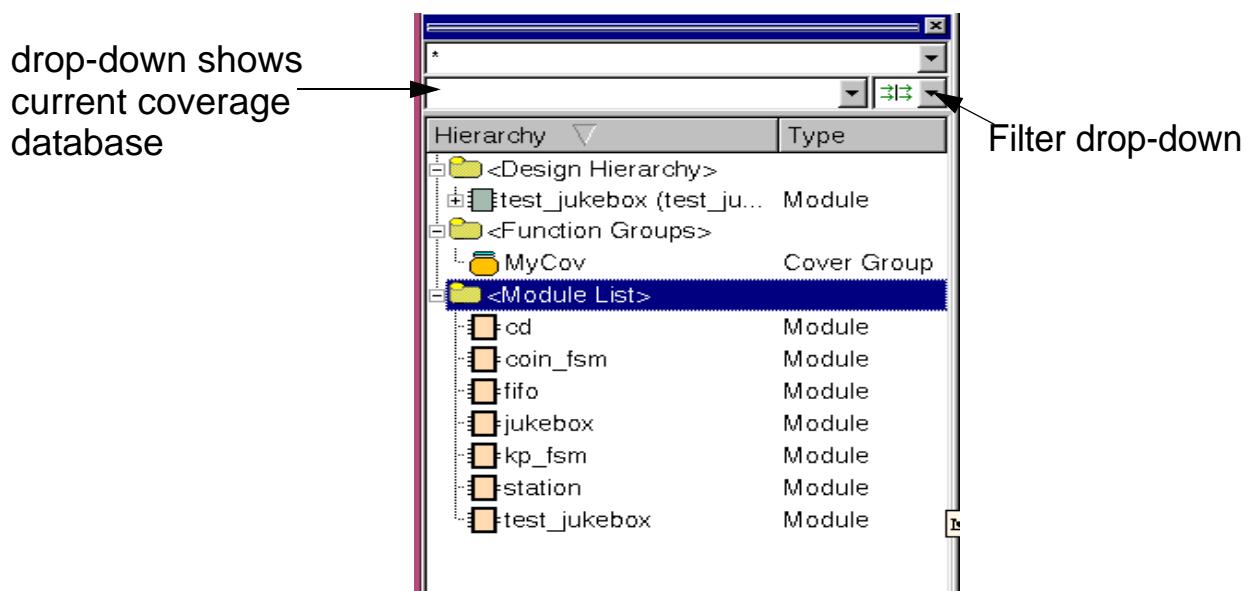
DVE can only remember one root location directory at a time. DVE replaces any earlier root location directory with whichever new location you specify.

Using the Coverage GUI

This section describes how to use the DVE Coverage GUI.

The Navigation Pane

Figure 4-3



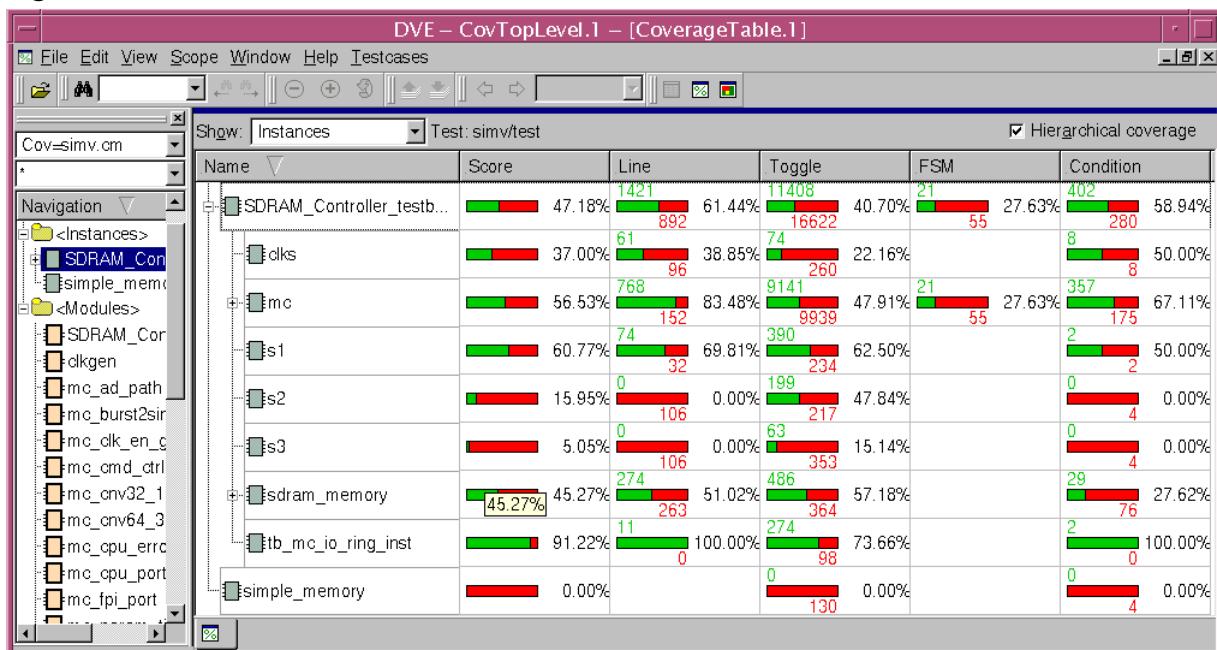
The Navigation pane is to navigate to the required database. It consists of the following items:

- 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 4-4

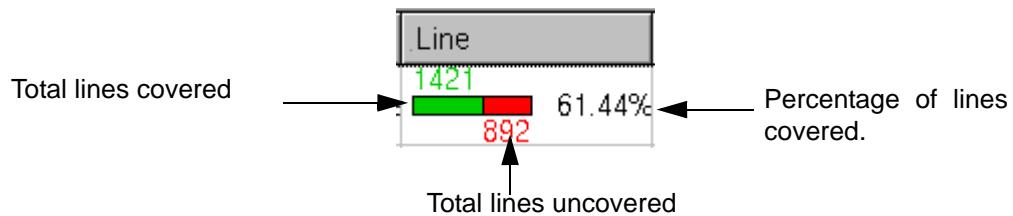


The Main Coverage Table window gives you summary information for each type of coverage. For example, [Figure 4-5](#) shows the details for lines covered.

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

The coverage detail appears as follows;

Figure 4-5



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 and their 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 4-1 on page 4](#).

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.

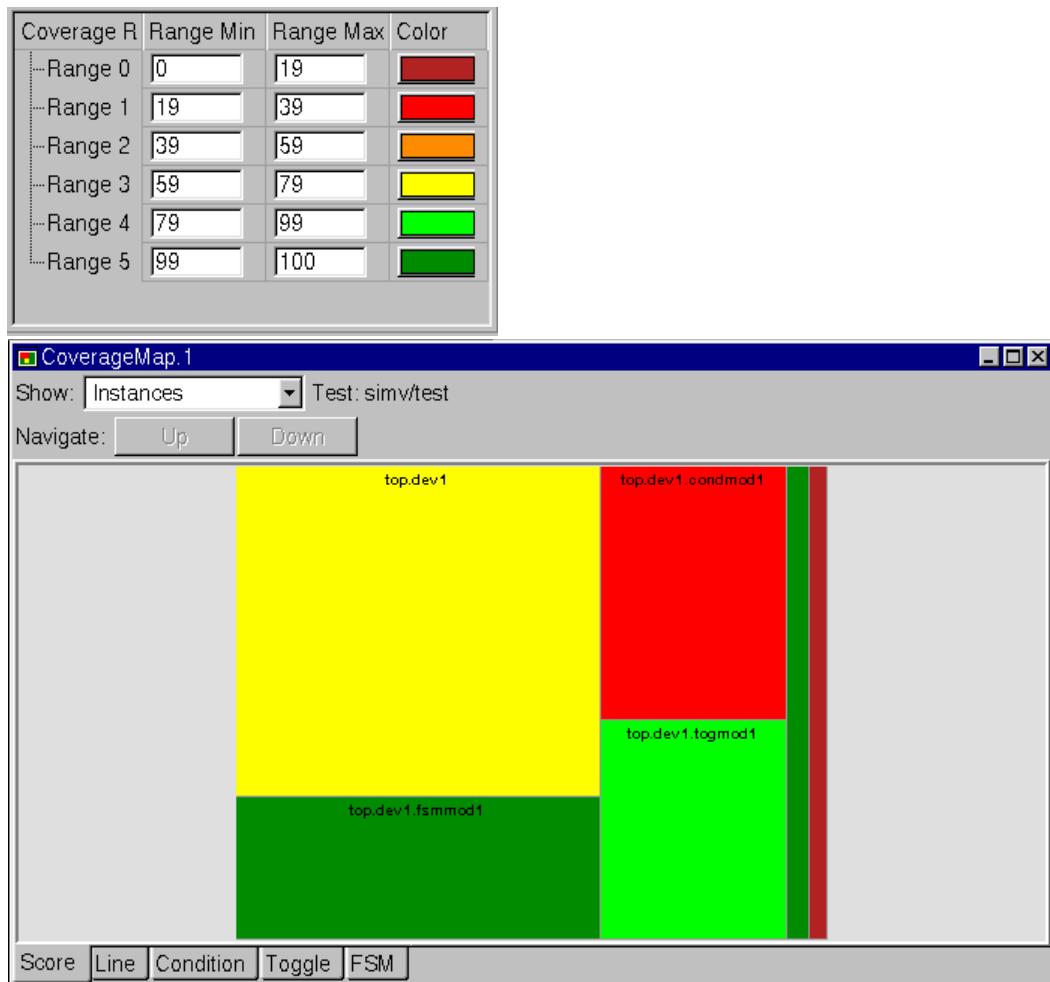
Detail Window

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 4-6](#) shows a coverage map and the default color key and range.

Figure 4-6



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

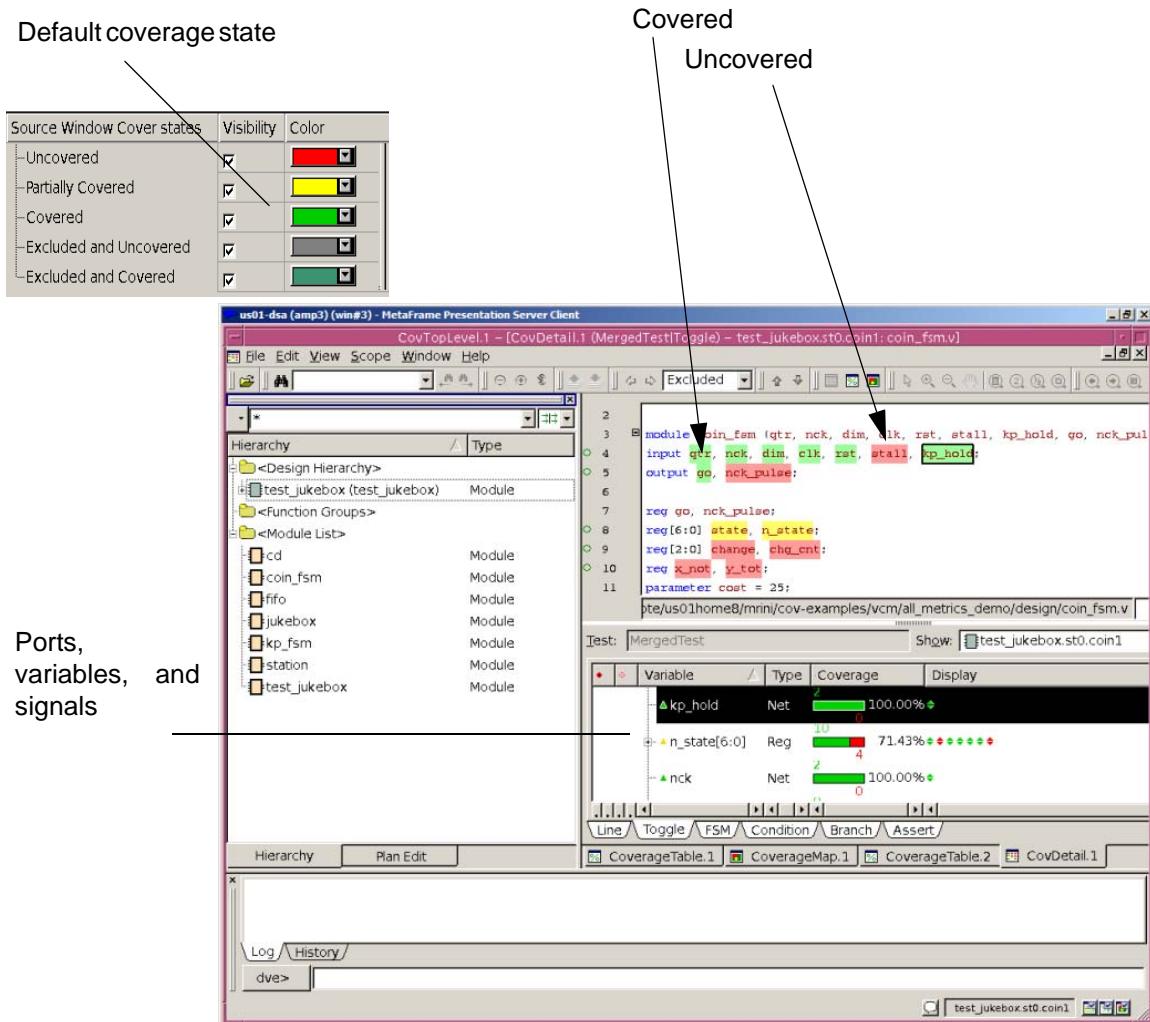
Drill down into the view by selecting a rectangle. Double-clicking the selected rectangle changes the view to display its children.

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 4-7



Navigating in 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 4-8

The screenshot shows a window titled "CovDetail.1 (MergedTest|Line)". On the left is a text editor displaying Verilog code with line numbers from 109 to 125. The code includes several "log" statements and control logic. To the right of the code is a vertical bar divided into colored segments (green, yellow, red, grey) representing coverage data. A callout bubble with the text "Click a band to display corresponding code." points to the top green segment of the bar. The window has standard operating system window controls at the top.

```
109 begin
110 log3 = log3++;
111 log4 = log4++;
112 end
113
114 always @ clk
115 begin
116 log5++;
117 log6++;
118 end
119
120 always @ (posedge
121 begin
122 log8++;
123 log9++;
124 end
125
```

Creating User-Defined Groups

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

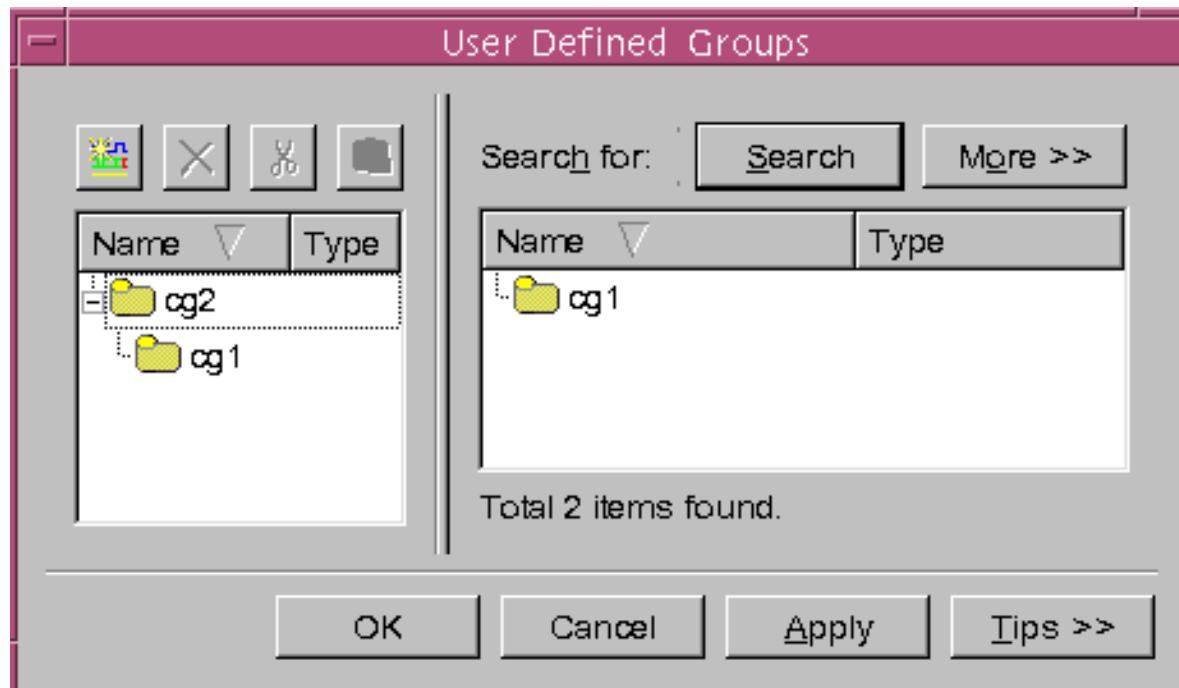
To create a user-defined covergroup

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

The User Defined Group dialog box appears.

Covergroups and assertions in your project are listed in the left pane.

Figure 4-9



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**.

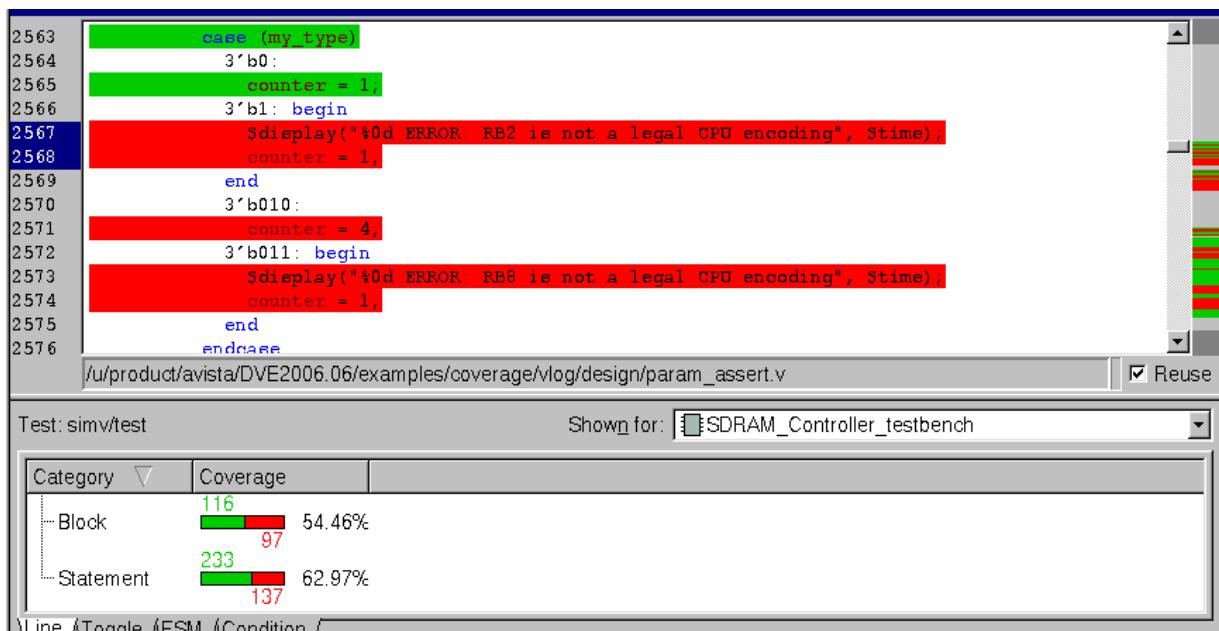
Your group gets created and the dialog box closes. You can click **Apply** to save the group. The dialog box remains open and you can make more groups.

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 4-10



- 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. Color banding adjacent to the scroll bar aid in navigating the code. [Figure 4-7](#) 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 **Show** drop-down to look at the coverage of other instances or module variants corresponding to the current instance or module.

Displaying Toggle Coverage

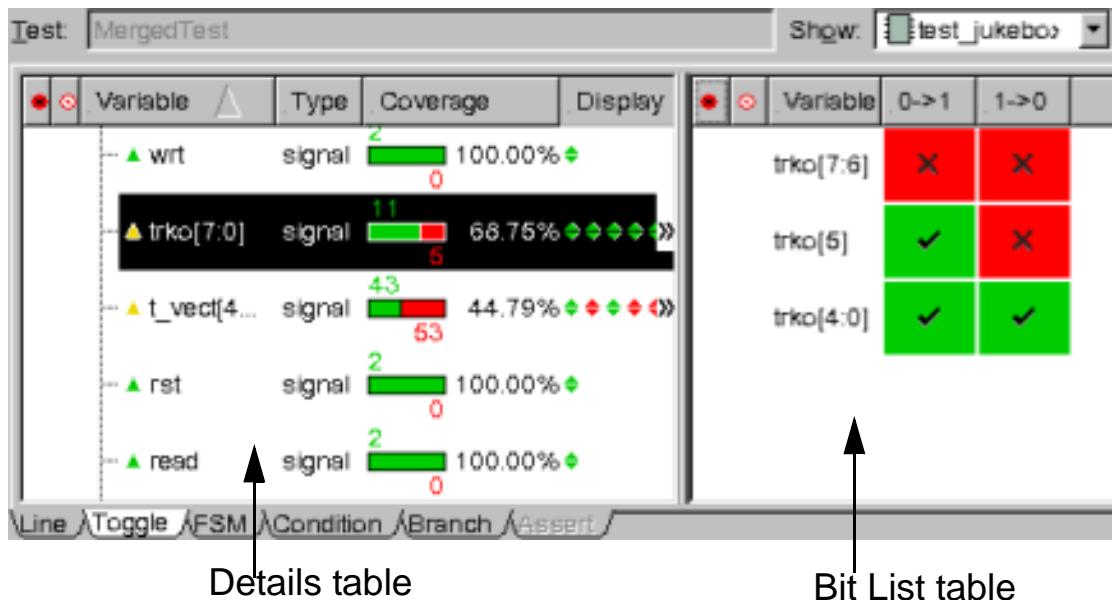
You can view the summary of toggle coverage in the coverage Summary Table. In the Summary Table, double-click on a row under the Toggle column to view the coverage Details view.

Displaying Nets and Registers

The toggle coverage **Details view** displays the value transitions from 0 to 1 and 1 to 0 for each net and register.

You can also view the nets and registers highlighted with their coverage range in the Source view. Use the **Show** drop-down list to look at the coverage of other instances or module variants corresponding to the current instance or module.

Figure 4-11 Toggle Coverage Detail view



The toggle coverage Details view contains the following:

- **Variable** column - Identifies the nets and registers. An up arrow indicates toggle from 0 to 1, and a down arrow indicates toggle from 1 to 0. The arrow is colored green if the toggle is covered, red if it is uncovered, and yellow when the signal has both covered and uncovered bits.
- **Type** column - Displays the type, signal or MDA.
- **Coverage** column - Displays the total coverage of nets and registers that 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 Coverage column indicates grey color when you exclude the signal.
- **Display** column - Indicates all the possible transitions for the register with pairs of up-down arrows and displays two transitions for each bits.

You can view the bit list of each signal in the Bit List table at the right-side of the Details table. To see the bits, select a row in the Details table. You can compress and uncompress the bits in the Bits List table using either the CSM options **Compress Bits** and **Uncompress Bits** or by double-clicking on the bit.

Displaying 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. Similar to 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.

You can exclude the top level MDA, any mda layer in the middle, or the word and bits.

The **Type** column in the coverage Details view indicates MDA. You can view the MDA bits in two modes, compressed and uncompressed, in the Bit List table.

Variable	Type	Coverage	Display	Variable	0->1	1->0
a[3:0][4:0][0:5]	MDA	1.67%	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	a[2][3][3]	✓	✗
firstBit[3:0][4:0]...	MDA	0.42%	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	a[2][1][3]	✓	✗
firstVector[3:0]...	MDA	4.17%	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	a[1][2][3]	✓	✗
lastBit[3:0][4:0]...	MDA	5.42%	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	a[0][2][3]	✗	✓
oneBitFirst[2:1]...	MDA	0.21%	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Other bits of a[3:0][4:0][0:5]	✗	✗
oneBit[2:1][3:0]...	MDA	0.21%	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			
shortBitFirst[2]...	MDA	3.12%	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			
wholeVector[3]...	MDA	41.67%	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			
zeroCovered[3]...	MDA	0.00%	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			

To view compressed or uncompressed MDA bits, select the option **Compress MDA bits** from **Edit > Preferences > Detail View**.

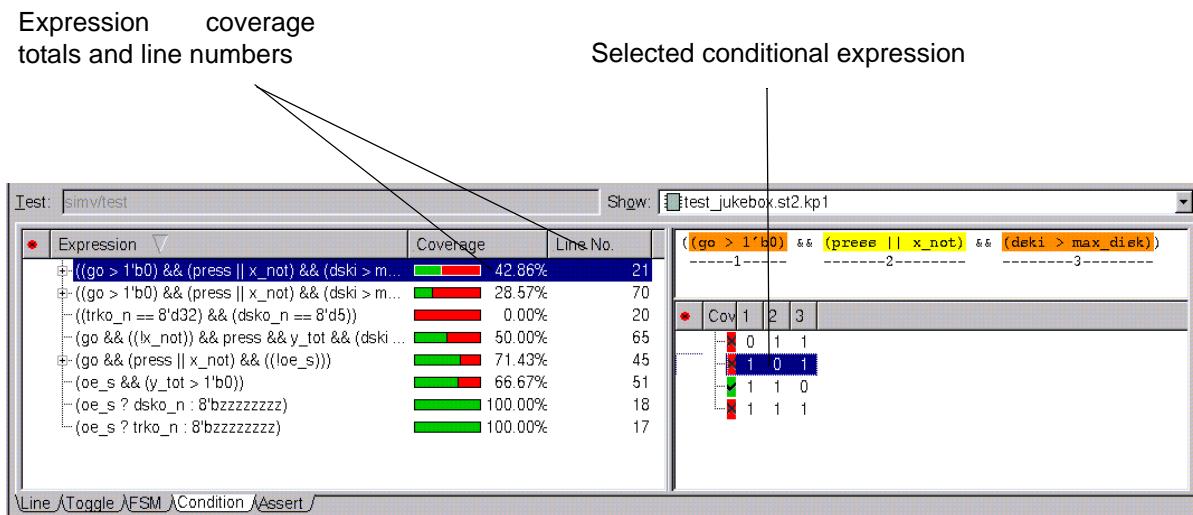
Displaying Condition Coverage

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 [Coverage Technology Reference Manual](#).
- 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.

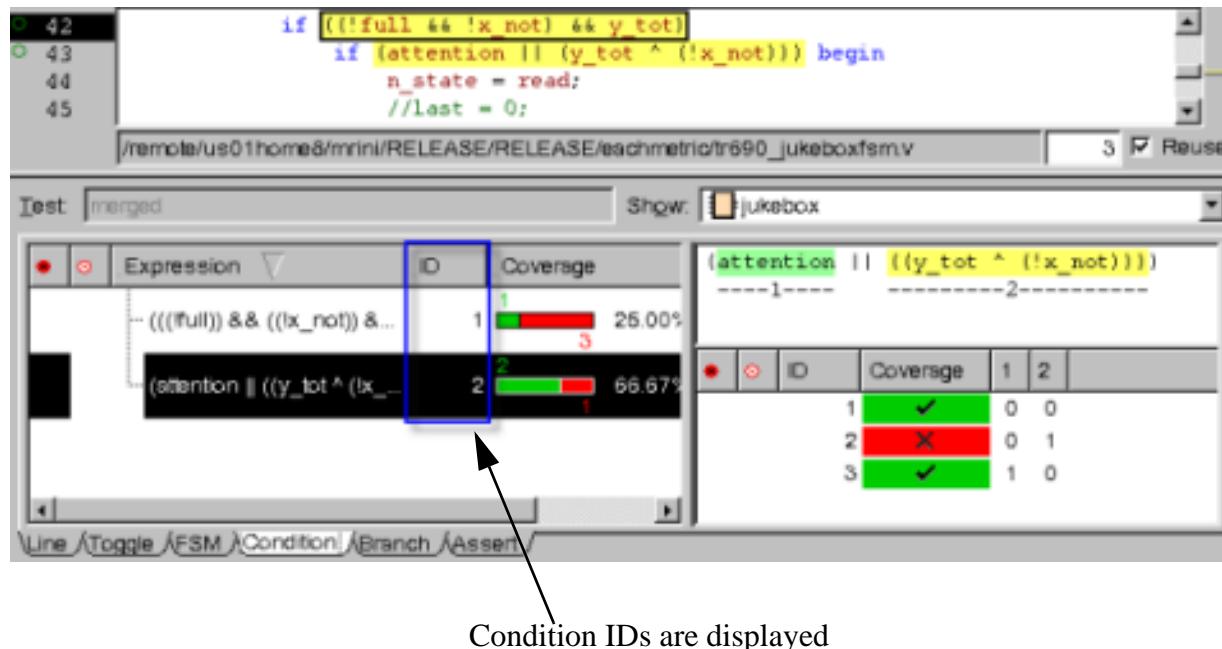
[Figure 4-12](#) shows condition results for a simple expression.

Figure 4-12 Condition Detail Window



Viewing Condition IDs

To view the condition IDs, select **View > Show Condition IDs**. The condition IDs are displayed in the Detail window as follows:



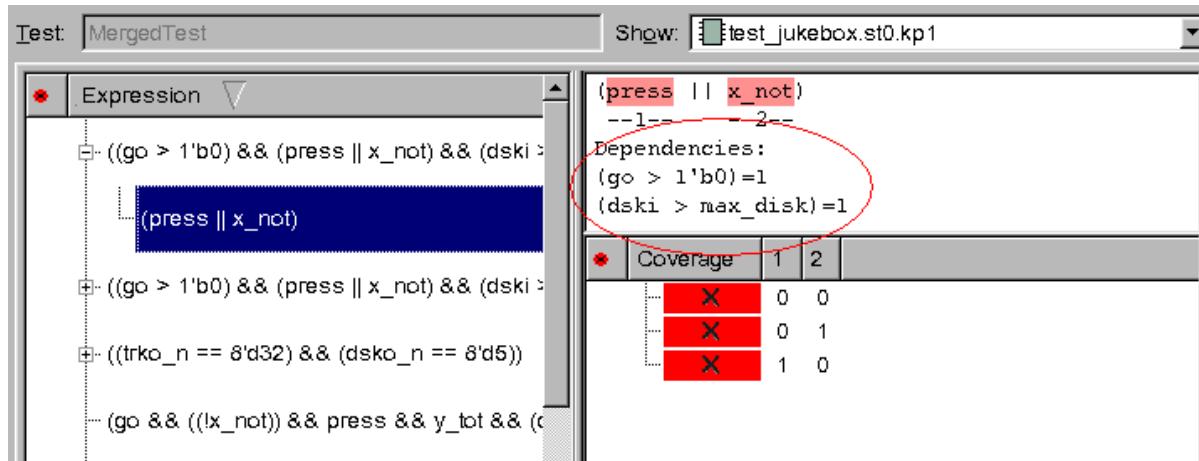
The display of condition ID is supported only with the New Unified Database flow.

DVE Coverage with Condition Coverage Observability

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

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

In the DVE Coverage GUI, you can view the observability expression in the Expression Display pane. The observability expression and its values are displayed in the format: [Expression]=[Value].



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 4-13 Default FSM Detail Display

```

6562 module me_port_mux(pld_cmd_ack, pld_wd_ack, p2d_cmd_ack, p2d_wd_ack,
6563     p3d_cmd_ack, p3d_wd_ack, px_cmd_id, px_cmd_true_2w, px_cmd_valid,
6564     px_cmd_addr, px_cmd_type, px_cmd_sbe, px_wd_valid, px_wd_greset, gclk,
6565     p1d_cmd_valid, p1d_cmd_addr, p1d_cmd_type, p1d_cmd_true_2w, p1d_cmd_sbe,
6566     p1d_wd_valid, p1d_wd, p2d_cmd_valid, p2d_cmd_addr, p2d_cmd_type,
6567     p2d_cmd_sbe, p2d_wd_valid, p2d_wd, p3d_cmd_valid, p3d_cmd_addr,
6568     p3d_cmd_type, p3d_cmd_sbe, p3d_wd_valid, p3d_wd, mcd_cmd_ack, mcd_wd_ack
6569 );
6570

```

Test: simv/test Shown for: SDRAM_Controller_tb.m.port_mux

FSM	State	Transition	Sequence
selected_port	3 75.00%	3 30.00%	0 0.00%

Mode: States & Transitions

	0	1	2	3
0'h0	-	✓	-	-
1'h1	✗	-	✓	✗
2'h2	✗	✓	-	✗
3'h3	✗	✗	✗	-

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 4-14](#) 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 4-14 FSM Transition Detail Table

	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 4-15 shows transitions with covered transitions displayed in green and uncovered in red. Not-possible transitions are not listed in the table.

Figure 4-15 FSM Transition List

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 4-16 FSM Sequence Table

	0	1	2	3	4	5
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 twnty	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 4-17](#) 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 by using `-cm_fsmopt` sequence switch.

Figure 4-17 FSM Sequence List

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

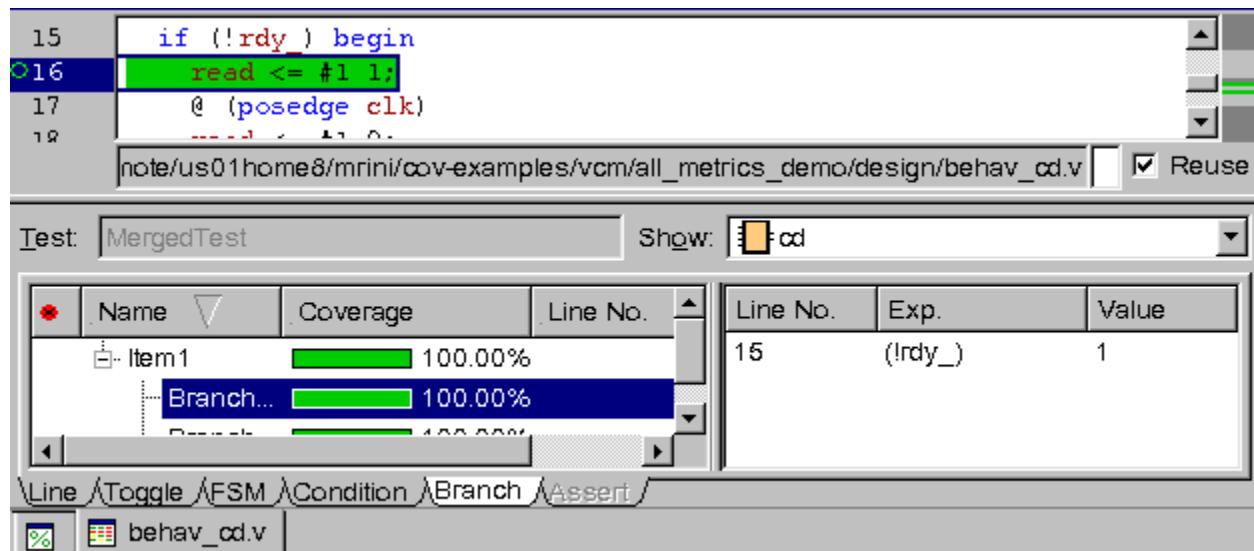
This section describes branch coverage. You can view the branch coverage information in the following windows:

- 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 4-18



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 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 Assertion Coverage

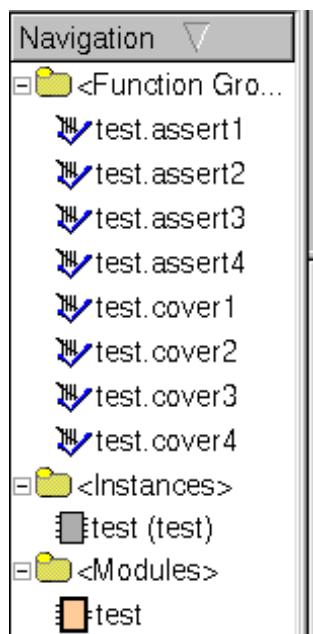
Currently, DVE can display limited information about SystemVerilog Assertion (SVA) and OpenVera Assertion (OVA) coverage.

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 4-19 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 Block identifier of the statement
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 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.ova_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 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 4-20 SystemVerilog assert and cover statements in the Covers Tab

Instance	Cover Name	Attempts	Real Successes	Failures	Incompletes
<Function Groups>	User defined Group	745	321	422	2
operator_preced_and_delay1_testbench.d0	operator_preced_and_delay1	5	1	4	0
operator_preced_and_delay2_testbench.d0	operator_preced_and_delay2	5	1	4	0
operator_preced_and_delay_define1_testbench.d0	operator_preced_and_delay_define1	5	1	4	0
operator_preced_and_or1_testbench.d0	operator_preced_and_or1	5	3	2	0
operator_preced_and_or_delay1_testbench.d0	operator_preced_and_or_delay1	5	2	2	1
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	4	1	0
operator_preced_or_delay2_testbench.d0	operator_preced_or_delay2	5	4	1	0
operator_preced_or_thoroughout_delay_1_testbench.d0	operator_preced_throughout_delay	100	0	100	0
operator_preced_or_thoroughout_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	27	3	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	28	2	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 4-21 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 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 started keeping track of the statement or directive, but simulation ended before the statement could succeed or match.

Displaying Testbench Coverage

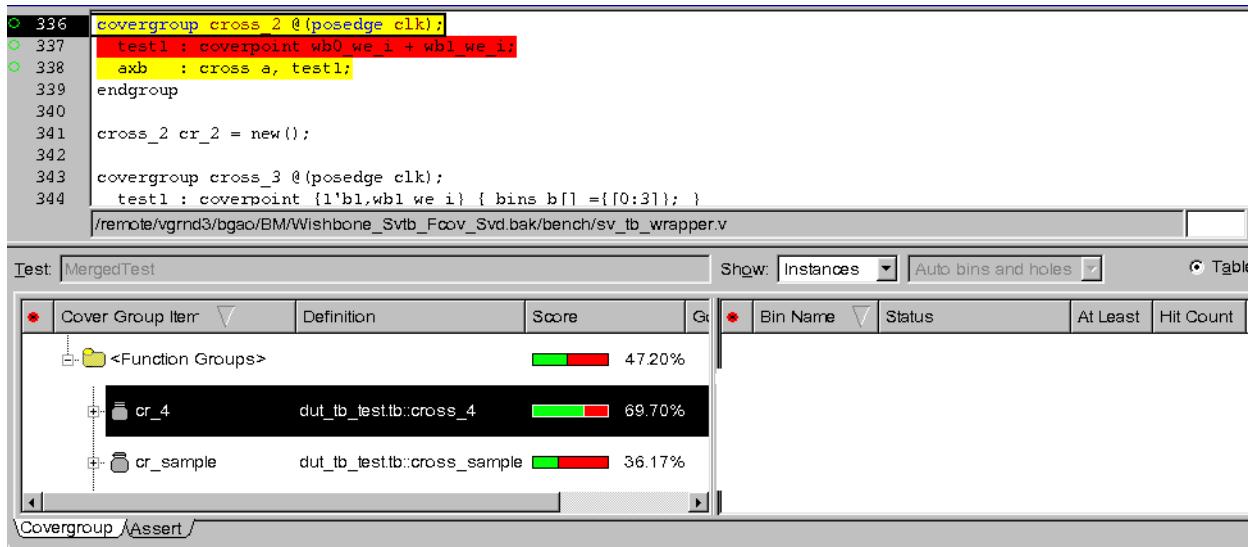
DVE displays testbench coverage information as specified in SystemVerilog `covergroup` constructs in SystemVerilog program blocks and the OVA `covergroup` construct in the OpenVera code.

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 covergroup 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 4-22](#). 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 shows the DVE Coverage GUI interface. At the top, there is a code editor window displaying Verilog-like code for covergroups:

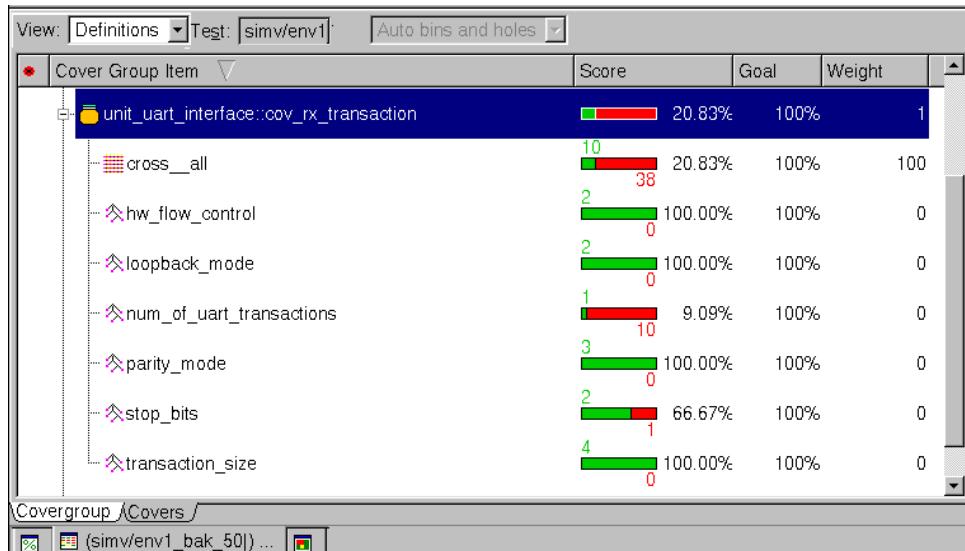
```
336  covergroup cross_2 @ (posedge clk);
337    test1 : coverpoint wb0_we_i + wb1_we_i;
338    axb   : cross a, test1;
339  endgroup
340
341  cross_2 cr_2 = new();
342
343  covergroup cross_3 @ (posedge clk);
344    test1 : coverpoint {1'b1,wb1_we_i} { bins b[] ={[0:31]}; }
```

The file path is listed as /remote/vgrnd3/bgao/BM/Wishbone_Svtb_Scov_Svd.bak/bench/sv_tb_wrapper.v

Below the code editor is a detailed covergroup report window. The title bar says "Test: MergedTest". The "Show" dropdown is set to "Instances". The report table has columns: Cover Group Item, Definition, Score, and Bin Name, Status, At Least, Hit Count. The data in the table is:

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%				

Figure 4-22 Covergroup Item Pane



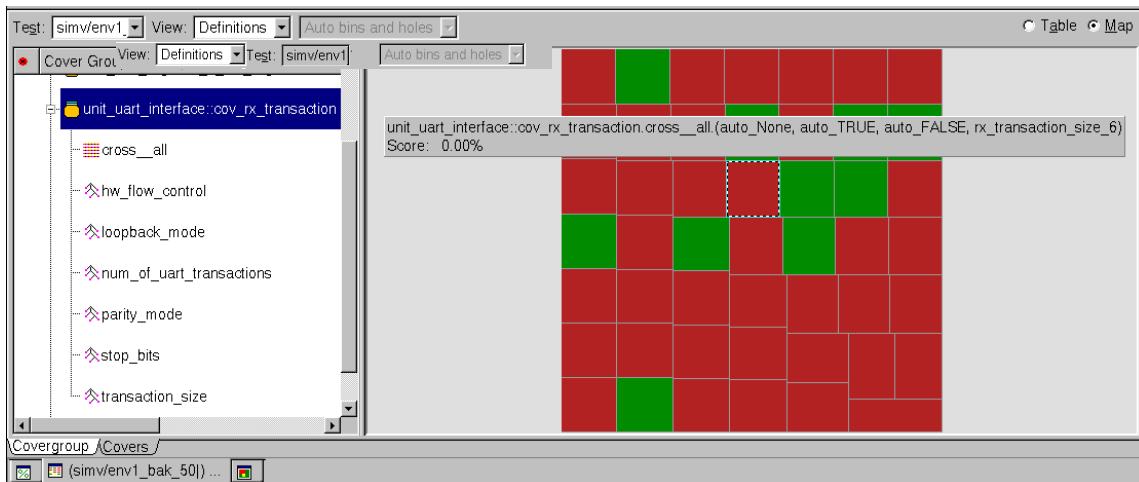
[Figure 4-23](#) shows the map view results for covergroup unit_uart_interface::cov_rx_transaction. The cumulative score for the cross below the mouse pointer in the map view is 20.83, as displayed in the tooltip.

Figure 4-23 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 4-24 Results for Crosses



To view results in table format, select **Table**. [Figure 4-25](#) 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 4-25 Coverpoint Table View

The screenshot shows the DVE Coverage GUI interface. The top bar includes 'Test: MergedTest', 'View: Definitions', 'Auto bins and holes', and radio buttons for 'Table' (selected) and 'Map'. The left pane, titled 'Cover Group Item', shows a tree structure with nodes like 'wb_dma_cfg_cov::config_c' expanded, revealing 'active_xpriority', 'active_cnt', 'priority_cnt', and 'traffic_mode'. The 'active_xpriority' node is highlighted with a blue background. The right pane is a table with the following data:

active_cnt	priority_cnt	At Least	Size	Hit Count
[CH_CNT_13, CH_CNT_14]	[PR_CNT_1, PR_CNT_2, PR_CNT_3]	1	104	0
[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

Filtering Instances with No Coverable Objects

Some instances or modules in your design can have no coverage data. If your designs are large, and contain multiple such instances, it will be difficult to find instances or modules with actual coverage data you are interested in.

In previous releases, DVE Coverage GUI displays the full hierarchy, including instances/modules with no coverable objects, as shown in [Figure 4-26](#), [Figure 4-27](#), and [Figure 4-28](#).

Figure 4-26 Dummy Instances and Modules in the Hierarchy Pane

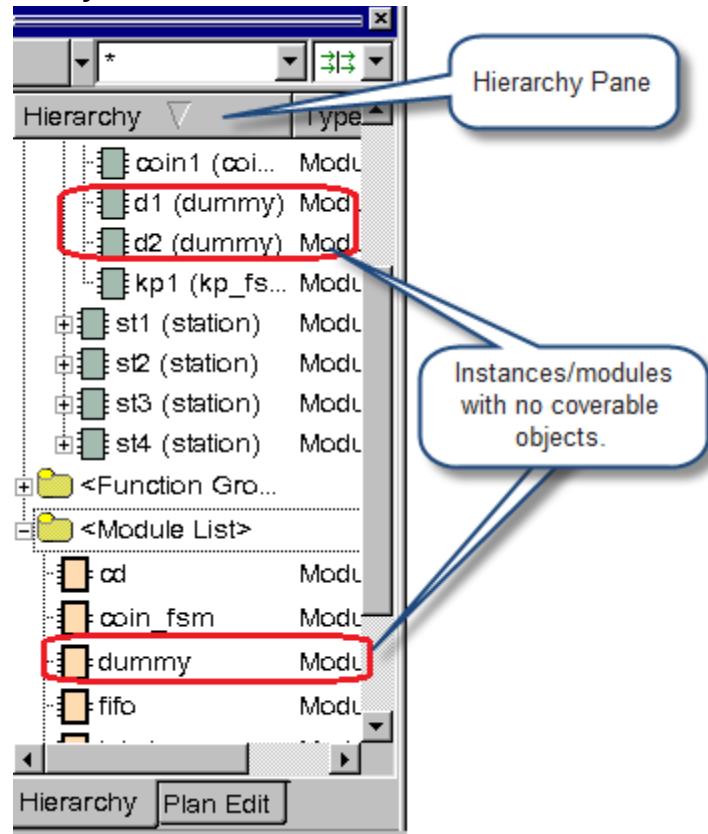


Figure 4-27 Dummy Instances in the Coverage Table Window

Name	Score	Line	Assert
test_jukebox			
cd1	100.00%	100.00%	
fifo1	88.24%	88.24%	
jb1	96.00%	96.00%	
st0			
coin1	67.42%	67.42%	
kp1	89.66%	89.66%	
d1			
d2			
st1			
coin1	89.69%	89.69%	
kp1	89.66%	89.66%	
d1			
d2			
st2			
st3			
..			

Figure 4-28 Dummy Modules in Coverage Table Window

Name	Score	Line	Assert
cd	100.00%	100.00%	
coin_fsm	91.01%	91.01%	
dummy			
fifo	88.24%	88.24%	
jukebox	96.00%	96.00%	
kp_fsm	89.66%	89.66%	
station			
test_jukebox			

By default, DVE Coverage GUI now displays only the part of design hierarchy where the coverage data exists, and hides the instances/modules with no coverable objects (instances/modules that have a hierarchical coverable count of zero). This helps you to view only instances/modules with actual coverage data, thereby saving your debug time. This feature hides dummy instances/modules in Hierarchy Pane and Coverage Table (see [Figure 4-26](#), [Figure 4-27](#), and [Figure 4-28](#)).

To view full design hierarchy, including instances/modules with no coverable objects

Use the `-show fullhier` option or the DVE Coverage Application Preferences dialog box.

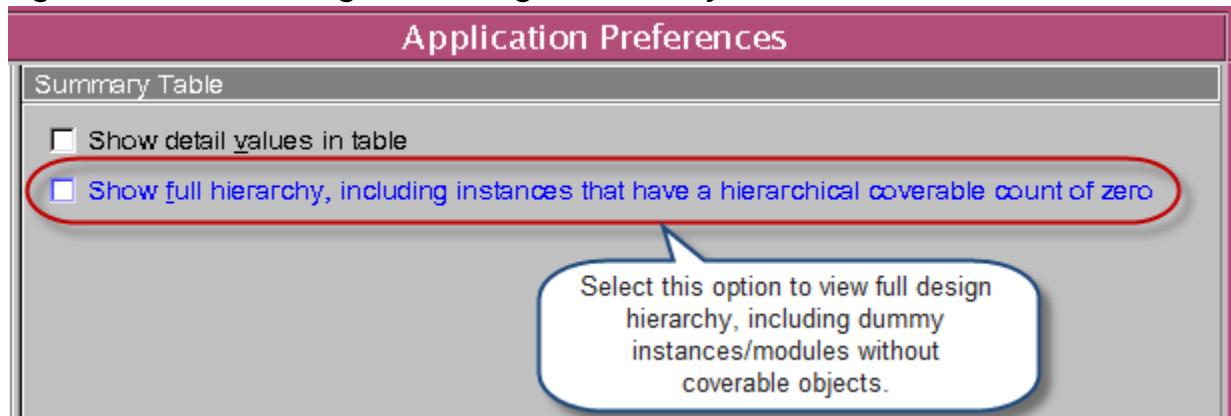
To disable this feature using the Application Preferences dialog box:

1. Select **Edit > Preferences**.

The Application Preferences dialog box appears.

2. In the **Summary Table** category, select the **Show full hierarchy, including instances that have a hierarchical coverable count of zero** check box.

Figure 4-29 Viewing Full Design Hierarchy



3. Click **OK**.

DVE makes this mode as default until you again enable this feature by clearing the **Show full hierarchy, including instances that have a hierarchical coverable count of zero** check box.

Note:

Command-line option takes precedence over preference setting.

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, see the *LCA category* 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

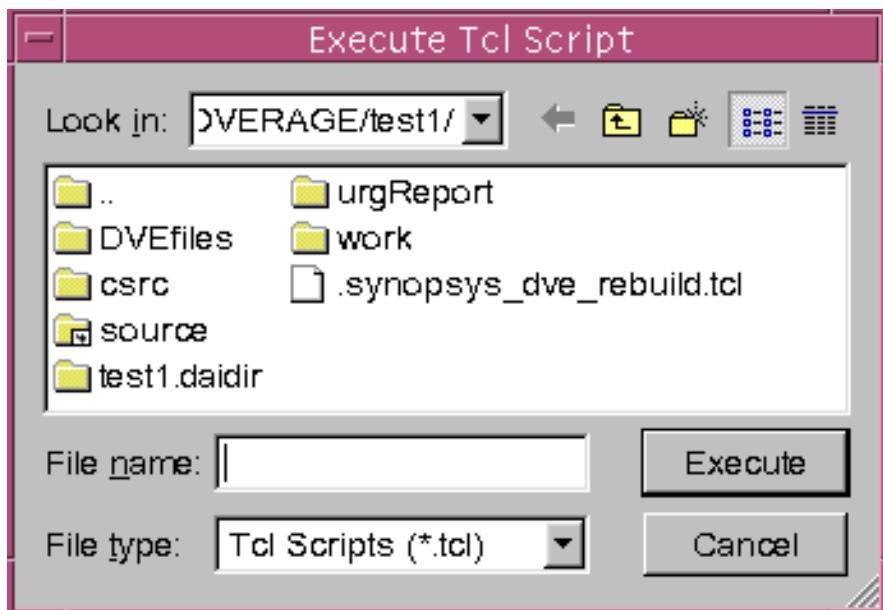
1. Select **File > Execute TCL Script**.

The Execute TCL Script dialog box appears.

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

The Tcl script is executed.

Figure 4-30



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.

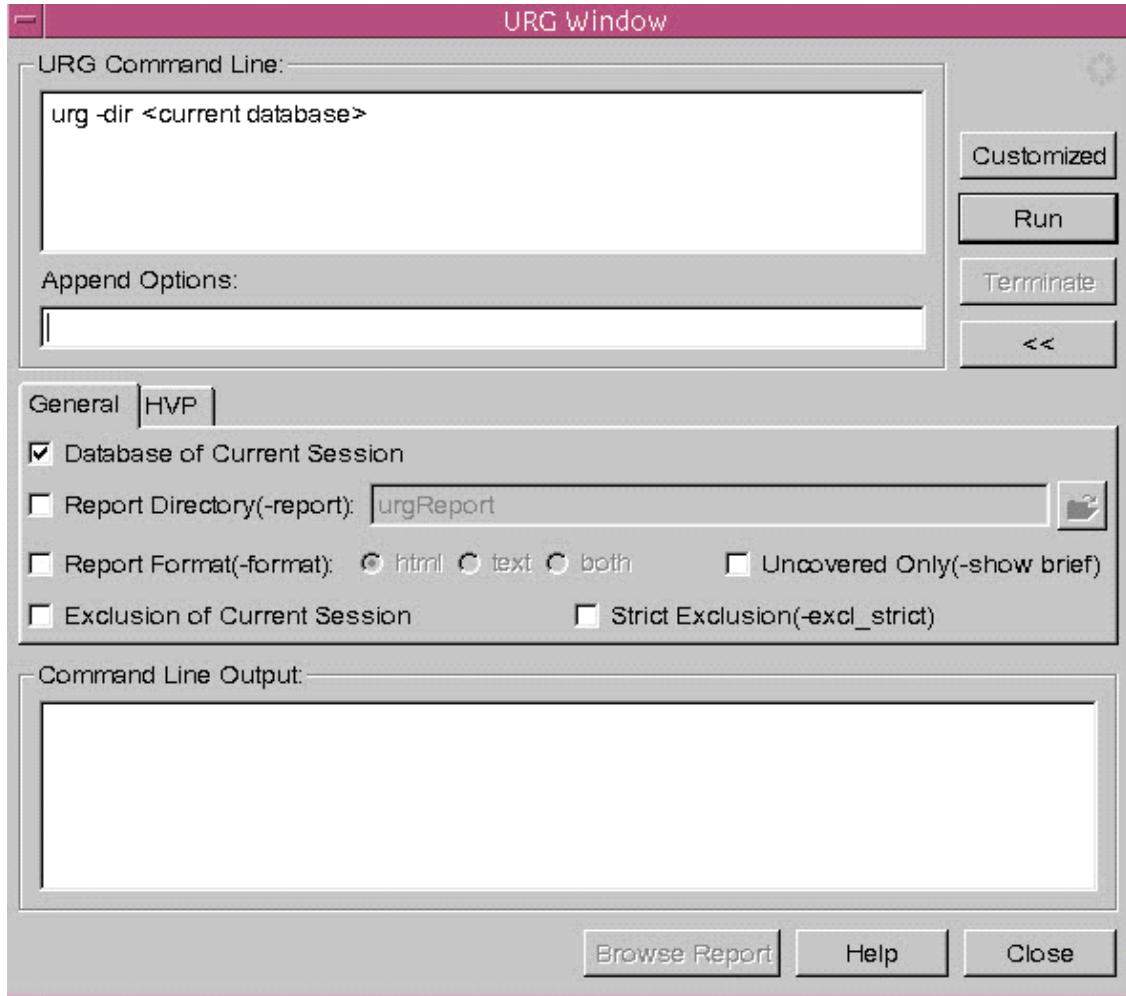
Generating URG Report from DVE Coverage GUI

You can now generate a URG report from the DVE coverage GUI. URG Window is the interface you use to generate the URG report. You need to set the configuration options in the URG Window and run URG to generate the report.

To generate a URG report

1. Load the coverage database in DVE.
2. Click **Generate URG Report** from the **File** menu.

The URG Window dialog box opens.



3. (Optional) Enter your command in the **Append Options** text box.

4. Click the **Show/hide URG option tabs** button “>>”.

The option tabs General and HVP are displayed below the Append Options text box.

5. Select the following options under the **General** tab. The options that you select in the configuration tabs appear in the URG Command Line text area.

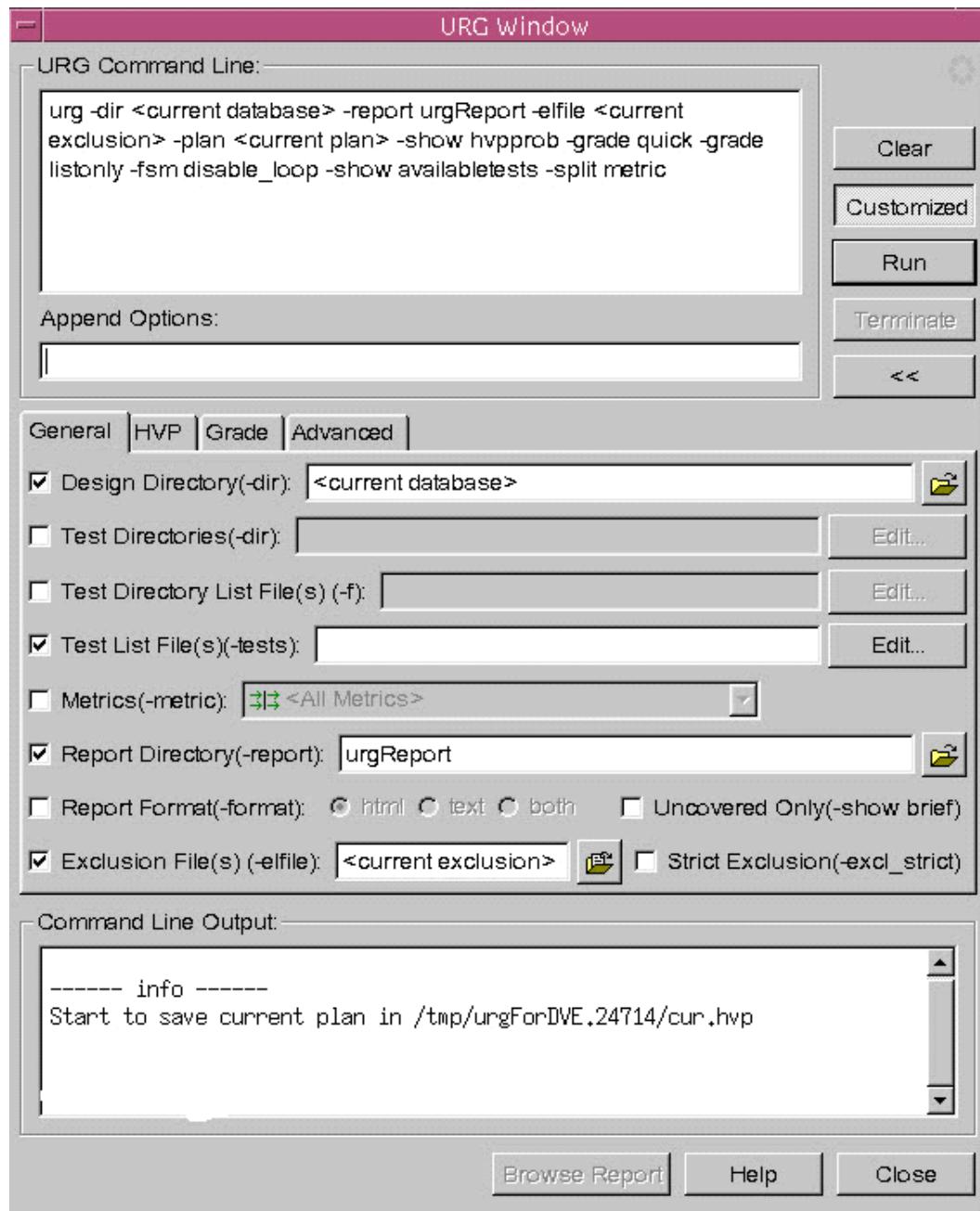
- Database of Current Session — Specifies database of the current session.
 - Report Directory(-report) — Generates report in the default directory (urgReport). Click the **Set URG report directory** button to select the directory where you want to save the generated report.
 - Report Format(-format) — Specifies the report format, html, text, or both, in which you want to view the report.
 - Uncovered Only (-show brief) — Specifies only the uncovered items.
 - Exclusion of Current Design — Specifies the exclusion performed in the current session. You can select this check box if you want to include the current exclusion.
 - Strict Exclusion (-excl.strict) — Specifies exclusion in strict mode.
6. Click the **HVP** tab and select the check box HVP Plan of Current Session to include the HVP plan of the current session.
7. Click **Run** to execute the URG report.

The URG report is generated. If there is an error in the report generation, the errors are printed in the Command Line Output text area.

8. Click the **Browse Report** button to view the generated URG report.

The report is displayed in a standard browser.

- Click the **Customized** button to display additional configuration options and tabs (Grade and Advanced). The available customized configuration options are as follows:



- General tab
 - Design Directory(-dir) — Specifies the base design/test directory. You can browse and select the desired base design/test directory.
 - Test Directories(-dir) — Identifies the test directories for source data. Click the **Edit** button to display the Choose test directories dialog box. You can add or delete the directories, or move the selected directories up/down.
 - Test Directory List File(s)(-f) — Specifies multiple directories for source data in a file.
 - Test List File(s)(-tests) — Specifies a file containing names of tests to report from specified directories.
 - Metrics — Specifies the metrics for Line, FSM, Condition, Toggle, Branch, and Assertion.
 - You can select your own exclusion file in the customized mode.
- HVP tab
 - Top Plan File(-plan) — Reports the Hierarchical Verification Plan (HVP) given in the specified file.
 - Filter/Override(s)(-mod) — Reads filtered/overridden HVP data from the specified file. The `-plan` options must also be given.
 - Userdata List File(s)(-userdatafile) — Specifies a file containing HVP data file names for annotation. The `-plan` option must also be given.
 - Userdata File(s)(-userdata) — Reads HVP data from the given file for annotation. The `-plan` option must also be given.

- Show full HVP tree(-show hvfullhier) — Lets you see the full hierarchy tree, including the plans and features which are filtered out.
- Show problem HVP tree(-show hvpprob) — Lets you see the failed plan that was set for one or more metrics.
- **Grade tab**
 - Grade Algorithm(-grade) — Specifies which algorithm to use: quick, greedy, or score.
 - Goal(-grade goal) — Stops grading when the cumulative score reaches the specified goal. It has no effect on the score algorithm.
 - Timelimit(-grade timelimit) — Specifies the time limit for the report generator to run before exiting. Only those tests that are graded before the time limit is hit are included in the graded list.
 - Maxtest(-grade maxtests) — Specifies the maximum number of tests to include in the report.
 - Minincr(-grade minincr) — The score improvement for each metric that the test contributed to the cumulative value when it was merged.
 - Reqtests(-grade requests) — Specifies reading a list of test names from `file_name` for inclusion in the grading report. (used only for greedy algorithm).
 - Listonly(-grade listonly) — Generates only a grading report.
- **Advanced tab** — Lists various URG commands with description specified against each command.

To run URG and view the customized report, repeat Steps 7 and 8.

5

Coverage Post-processing Techniques

This chapter contains the following sections:

- “[Merging](#)”
- “[Grading and Coverage Analysis](#)”
- “[Mapping Coverage](#)”
- “[Exclusion](#)”
- “[Editing the Covergroup Coverage Database](#)”

Merging

After running multiple simulation, you can merge all the coverage database to get an overall coverage report for all of your design files. You can also create a single set of merged data that contains all the

information from your simulation runs. After merging, you can load these files to see the overall coverage results instead of reloading all the original test data files.

You can create a merged coverage data file using the `-dbname` flag to URG. For example:

```
% urg -dir simv.vdb -dbname merged_dir/merged_test -format text
```

This command creates `merged_dir.vdb/snps/coverage/db/testdata/merged_test/`

```
% urg -dir simv.vdb -dbname merged_dir2/ -format text
```

This command gives error because the testname after dirname is missing.

```
% urg -dir simv.vdb -dbname merged_dir3 -format text
```

This creates `merged_dir3.vdb` and testname is default (test).

```
% urg -dir simv.vdb -dbname simv.vdb/merged_test2 -format text
```

This creates a test subdir called `merged_test2` inside `simv.vdb` (the original directory containing all the test data).

```
simv.vdb/snps/coverage/db/testdata/merged_test2
```

There are three techniques for merging:

- Serial Merging
- Parallel Merging
- Flexible Merging

Serial Merging

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

Parallel Merging

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

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

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

When you enable parallel merging, `urg_parallel.elog` file gets created. The `urg_parallel.elog` file contains warnings and errors that occur during one of the sub-processes. If you specify the `-verbose` option, then the following message is generated in the log file created by the `-log` option:

```
Status: 13 Mergings, 1 remain (12 Done, 0 Running, 1 on
```

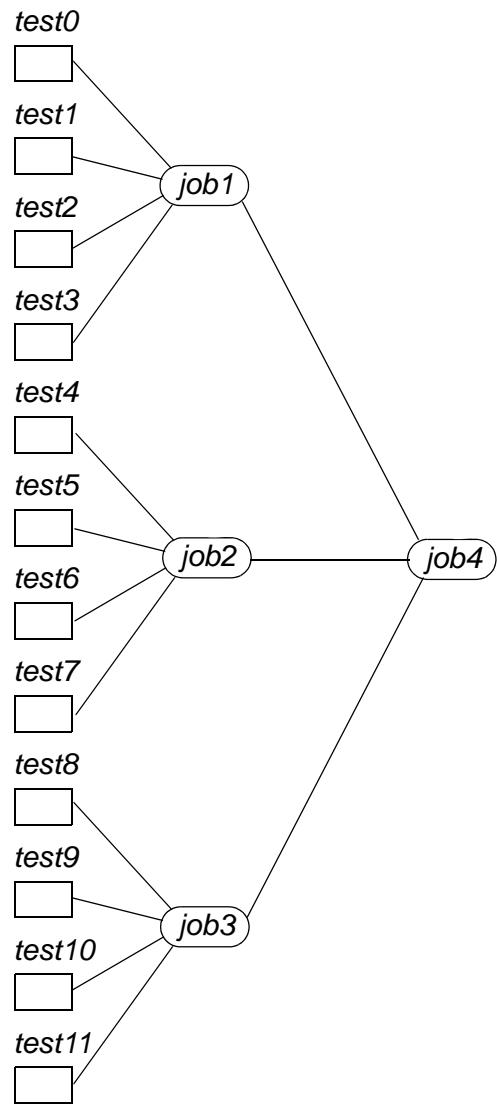
```
queue, 0 waiting)
Merging the Final One
The error messages of sub-jobs are saved in
urg_parallel.elog.
Distributed Merging SUCCESSFUL!!!
```

Note:

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

[Figure 5-1](#) shows the parallel merging of 12 coverage databases. URG merges test0, test1, test2, and test3 into one merging; test4, test5, test6, and test7 into a second merging; and test8, test9, test10, and test11 into a third merging.

Figure 5-1 Parallel Merging of 12 Tests



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

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

You can control the number of tests that URG merges into a merging, and the number of lower level mergings that go into higher level mergings. The smaller the number of tests (and mergings) the more jobs and levels in the parallel merging. For additional information, see the section entitled, “[Specifying the Number of Tests in a Merging](#)” on page 8.

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

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

Specifying the Machines that Perform the Jobs

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

The following is an example of a machine file:

```
RHEL32_machine1  
RHEL32_machine2  
RHEL32_machine3  
RHEL32_machine4
```

The following displays an example using this argument:

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

Using a GRID Computing Engine

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

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

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

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

Using LSF

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

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

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

```
bsub -del bkill  
...
```

Specifying the Number of Tests in a Merging

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

If for 12 test, by default URG divides the process into four jobs on two levels (see [Figure 5-1](#)).

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

Flexible Merging

This type of merging technique is available only for covergroup coverage.

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

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

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

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

Example

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

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

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

Merge Equivalence

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

Merge Equivalence Requirements for Autobin Coverpoints

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

Merge Equivalence Requirements for User-defined Coverpoints

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

Merge Equivalence Requirements for Crosspoints

The crosspoint C_1 is said to be merge equivalent to a crosspoint C_2 , if the crosspoints have the same number of coverpoints and their corresponding coverpoints are merge equivalent.

Rules for Flexible Merging Databases

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

Rules for Merging Coverpoints

The coverpoints $P(T_1)$ in first test run T_1 and $P(T_2)$ in the second test run T_2 are merged according to the following rules:

- If the coverpoints are merge equivalent. The merged coverpoints will contain a union of all the coverpoint bins in $P(T_1)$ and $P(T_2)$, but URG will drop the coverpoint bins that are defined only in the earlier coverage model.
- If the coverpoints are not merge equivalent. The merged coverpoint will contain all the coverpoint bins in the most recent test run and the older test run data is not considered and dropped.

Rules for Merging Crosspoints

The crosspoint $C(T_1)$ in test T_1 and $C(T_2)$ in test T_2 are merged according to the following rules:

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

Example

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

Example 5-1 Test01

```

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

```

Example 5-2 Test02

```

cp1: coverpoint firstsig;
      option.auto_bin_max = 32;
cp2: coverpoint secondsig {
      bins first = { [0:63] };
      bins second = { [65:128] };

```

```

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

```

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

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

Coverpoint Analysis

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

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

Crosspoint Analysis

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

Flexible Merge Database

Coverage model for test01	Coverage model for test02	Flexible merge Database Profile for test01 and test02 (test02 is the recent database and test01 is the old database.)
cp1: coverpoint firstsig; option.auto_bin_max = 64;	cp1: coverpoint firstsig; option.auto_bin_max = 32;	//auto_bin_max differs and is not merge equivalent. cp1: coverpoint firstsig; auto_bin_max = 32; // (From test02)
cp2: coverpoint secondsig{ bins first = {[0:63]}; bins mid = {[71:82]};}	cp2: coverpoint secondsig{ bins first = {[0:63]}; bins second = {[65:128]};}	// (cp2 is merged across test01 and test02) cp2: coverpoint secondsig { bins first = {[0:63]}; bins second = {[65:128]};}
cp3: coverpoint thirdsig;	cp3: coverpoint thirdsig;	//cp3 is merged across test01 and test02 cp3: coverpoint thirdsig;
bit[7:0]signal; cp4: coverpoint signal;	bit[15:0]signal; cp4: coverpoint signal;	//width of signal has changed and not merge equivalent cp4: coverpoint signal; // (From test02)
cc1: cross cp1, cp2;	cc1: cross cp1, cp2;	// cc1 is not merged equivalent because cp1 is not merge equivalent cc1: cross cp1, cp2; // (From test02)
cc2: cross cp2, cp3 { bins mybin = binsof(cp2) intersect [0:255]; }	cc2: cross cp2, cp3 { bins mybin=binsof(cp2) intersect [0:255]; bins yourbin=binsof(cp2) intersect [256:511];}	// cc2 is merged across test01 and test02) cc2: cross cp2, cp3 { bins mybin = binsof(cp2) intersect [0:255]; bins yourbin = binsof(cp2) intersect [256:511];}
cc3: cross cp2, cp3 { bins my_st = binsof(cp2) intersect [0:255];}	cc3: cross cp2, cp3 { bins my_st = binsof(cp2) intersect [0:8191];}	cc3: cross cp2, cp3 { bins my_st = binsof(cp2) intersect [0:8191];} The autocrosses in test01 will be merged with the autocrosses of test 02.
	cc4: cross cp1, cp2, cp3;	cc4: cross cp1, cp2, cp3; // (From test02)

Grading and Coverage Analysis

Grading, also known as autograding, means comparing the effectiveness of tests. Grading is useful, for example, when you want to eliminate redundant tests from a test suite. Use the `-grade` option to specify and run test grading and generate test grading reports.

To invoke grading in URG, use the following syntax:

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

For more information on options for grading, see the *Coverage Technology Reference Manual*.

Examples Using the Grading Option

Scoring

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

Note:

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

Figure 5-2 Example of a Scoring Report

Total Coverage Summary

SCORE	LINE	COND	TOGGLE	ASSERT
64.80	95.20	45.83	18.18	100.00

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

Scores are the standalone scores for each test.

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

Quick Grading

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

Figure 5-3 Example of a Quick-grading Report

Total Coverage Summary

SCORE	LINE	COND	TOGGLE	ASSERT
64.80	95.20	45.83	18.18	100.00

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

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

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

Greedy Grading

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

TOTAL boxes. Applying the greedy argument provides a thorough report, but it is more time-consuming than using the score or quick argument.

Figure 5-4 Example of a Full-grading Report

Total Coverage Summary																																																																																																																																																																																															
SCORE	LINE	COND	TOGGLE	ASSERT																																																																																																																																																																																											
64.80	95.20	45.83	18.18	100.00																																																																																																																																																																																											
Tests are in graded order																																																																																																																																																																																															
Shows the accumulated (Total), standalone (Test), and incremental (Incr) score overall and for each metric																																																																																																																																																																																															
<table border="1"> <thead> <tr> <th colspan="3">SCORE</th> <th colspan="3">LINE</th> <th colspan="3">COND</th> <th colspan="3">TOGGLE</th> <th colspan="3">ASSERT</th> </tr> <tr> <th>TOTAL</th><th>TEST</th><th>INCR</th><th>TOTAL</th><th>TEST</th><th>INCR</th><th>TOTAL</th><th>TEST</th><th>INCR</th><th>TOTAL</th><th>TEST</th><th>INCR</th><th>TOTAL</th><th>TEST</th><th>INCR</th><th>NAME</th></tr> </thead> <tbody> <tr> <td>18.16</td><td>18.16</td><td>18.16</td><td>31.20</td><td>31.20</td><td>31.20</td><td>8.33</td><td>8.33</td><td>8.33</td><td>4.55</td><td>4.55</td><td>4.55</td><td>28.57</td><td>28.57</td><td>28.57</td><td>mysimv/test2</td></tr> <tr> <td>25.01</td><td>14.01</td><td>6.85</td><td>41.60</td><td>36.00</td><td>10.40</td><td>16.67</td><td>8.33</td><td>8.33</td><td>6.06</td><td>4.55</td><td>1.52</td><td>35.71</td><td>7.14</td><td>7.14</td><td>mysimv/test7</td></tr> <tr> <td>31.86</td><td>14.01</td><td>6.85</td><td>52.00</td><td>36.00</td><td>10.40</td><td>25.00</td><td>8.33</td><td>8.33</td><td>7.58</td><td>4.55</td><td>1.52</td><td>42.86</td><td>7.14</td><td>7.14</td><td>mysimv/test9</td></tr> <tr> <td>37.79</td><td>13.61</td><td>5.93</td><td>60.80</td><td>34.40</td><td>8.80</td><td>31.25</td><td>8.33</td><td>6.25</td><td>9.09</td><td>4.55</td><td>1.52</td><td>50.00</td><td>7.14</td><td>7.14</td><td>mysimv/test6</td></tr> <tr> <td>43.27</td><td>14.21</td><td>5.49</td><td>72.00</td><td>36.80</td><td>11.20</td><td>33.33</td><td>8.33</td><td>2.08</td><td>10.61</td><td>4.55</td><td>1.52</td><td>57.14</td><td>7.14</td><td>7.14</td><td>mysimv/test8</td></tr> <tr> <td>48.28</td><td>13.21</td><td>5.01</td><td>79.20</td><td>32.80</td><td>7.20</td><td>37.50</td><td>8.33</td><td>4.17</td><td>12.12</td><td>4.55</td><td>1.52</td><td>64.29</td><td>7.14</td><td>7.14</td><td>mysimv/test5</td></tr> <tr> <td>53.08</td><td>16.78</td><td>4.81</td><td>85.60</td><td>32.80</td><td>6.40</td><td>41.67</td><td>8.33</td><td>4.17</td><td>13.64</td><td>4.55</td><td>1.52</td><td>71.43</td><td>21.43</td><td>7.14</td><td>mysimv/test3</td></tr> <tr> <td>57.89</td><td>14.79</td><td>4.81</td><td>92.00</td><td>32.00</td><td>6.40</td><td>45.83</td><td>8.33</td><td>4.17</td><td>15.15</td><td>4.55</td><td>1.52</td><td>78.57</td><td>14.29</td><td>7.14</td><td>mysimv/test4</td></tr> <tr> <td>62.24</td><td>16.38</td><td>4.35</td><td>93.60</td><td>31.20</td><td>1.60</td><td>45.83</td><td>8.33</td><td>0.00</td><td>16.67</td><td>4.55</td><td>1.52</td><td>92.86</td><td>21.43</td><td>14.29</td><td>mysimv/test1</td></tr> <tr> <td>64.80</td><td>14.59</td><td>2.56</td><td>95.20</td><td>31.20</td><td>1.60</td><td>45.83</td><td>8.33</td><td>0.00</td><td>18.18</td><td>4.55</td><td>1.52</td><td>100.00</td><td>14.29</td><td>7.14</td><td>mysimv/test0</td></tr> </tbody> </table>	SCORE			LINE			COND			TOGGLE			ASSERT			TOTAL	TEST	INCR	NAME	18.16	18.16	18.16	31.20	31.20	31.20	8.33	8.33	8.33	4.55	4.55	4.55	28.57	28.57	28.57	mysimv/test2	25.01	14.01	6.85	41.60	36.00	10.40	16.67	8.33	8.33	6.06	4.55	1.52	35.71	7.14	7.14	mysimv/test7	31.86	14.01	6.85	52.00	36.00	10.40	25.00	8.33	8.33	7.58	4.55	1.52	42.86	7.14	7.14	mysimv/test9	37.79	13.61	5.93	60.80	34.40	8.80	31.25	8.33	6.25	9.09	4.55	1.52	50.00	7.14	7.14	mysimv/test6	43.27	14.21	5.49	72.00	36.80	11.20	33.33	8.33	2.08	10.61	4.55	1.52	57.14	7.14	7.14	mysimv/test8	48.28	13.21	5.01	79.20	32.80	7.20	37.50	8.33	4.17	12.12	4.55	1.52	64.29	7.14	7.14	mysimv/test5	53.08	16.78	4.81	85.60	32.80	6.40	41.67	8.33	4.17	13.64	4.55	1.52	71.43	21.43	7.14	mysimv/test3	57.89	14.79	4.81	92.00	32.00	6.40	45.83	8.33	4.17	15.15	4.55	1.52	78.57	14.29	7.14	mysimv/test4	62.24	16.38	4.35	93.60	31.20	1.60	45.83	8.33	0.00	16.67	4.55	1.52	92.86	21.43	14.29	mysimv/test1	64.80	14.59	2.56	95.20	31.20	1.60	45.83	8.33	0.00	18.18	4.55	1.52	100.00	14.29	7.14	mysimv/test0												
SCORE			LINE			COND			TOGGLE			ASSERT																																																																																																																																																																																			
TOTAL	TEST	INCR	TOTAL	TEST	INCR	TOTAL	TEST	INCR	TOTAL	TEST	INCR	TOTAL	TEST	INCR	NAME																																																																																																																																																																																
18.16	18.16	18.16	31.20	31.20	31.20	8.33	8.33	8.33	4.55	4.55	4.55	28.57	28.57	28.57	mysimv/test2																																																																																																																																																																																
25.01	14.01	6.85	41.60	36.00	10.40	16.67	8.33	8.33	6.06	4.55	1.52	35.71	7.14	7.14	mysimv/test7																																																																																																																																																																																
31.86	14.01	6.85	52.00	36.00	10.40	25.00	8.33	8.33	7.58	4.55	1.52	42.86	7.14	7.14	mysimv/test9																																																																																																																																																																																
37.79	13.61	5.93	60.80	34.40	8.80	31.25	8.33	6.25	9.09	4.55	1.52	50.00	7.14	7.14	mysimv/test6																																																																																																																																																																																
43.27	14.21	5.49	72.00	36.80	11.20	33.33	8.33	2.08	10.61	4.55	1.52	57.14	7.14	7.14	mysimv/test8																																																																																																																																																																																
48.28	13.21	5.01	79.20	32.80	7.20	37.50	8.33	4.17	12.12	4.55	1.52	64.29	7.14	7.14	mysimv/test5																																																																																																																																																																																
53.08	16.78	4.81	85.60	32.80	6.40	41.67	8.33	4.17	13.64	4.55	1.52	71.43	21.43	7.14	mysimv/test3																																																																																																																																																																																
57.89	14.79	4.81	92.00	32.00	6.40	45.83	8.33	4.17	15.15	4.55	1.52	78.57	14.29	7.14	mysimv/test4																																																																																																																																																																																
62.24	16.38	4.35	93.60	31.20	1.60	45.83	8.33	0.00	16.67	4.55	1.52	92.86	21.43	14.29	mysimv/test1																																																																																																																																																																																
64.80	14.59	2.56	95.20	31.20	1.60	45.83	8.33	0.00	18.18	4.55	1.52	100.00	14.29	7.14	mysimv/test0																																																																																																																																																																																

Grading and the -scorefile Option

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

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

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

Note:

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

The score file has the following simple format:

```
metric1 weight1  
metric2 weight2  
...  
metricN weightN
```

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

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

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

```
line 1  
group 2
```

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

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

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

Using Index-Based Test Grading for Code Coverage

URG now includes a faster index-based grading feature that you can use with code coverage. The index grading algorithm is significantly faster than greedy grading and produces similar quality results.

Use index-based grading instead of greedy grading when you want to get the optimizations of greedy grading in less time. Index-based grading only works with line, condition, toggle, branch, and FSM metrics. The syntax is:

```
% urg -grade index
```

You cannot specify the `index` suboption with `greedy`, `quick`, or `score`, since they are mutually exclusive grading algorithms.

The output format for the grading results is the same as for greedy grading, showing the cumulative total and incremental value added by each test for all metrics and each metric.

[Figure 5-5](#) shows an excerpt from an index-based grading report.

Figure 5-5 Excerpt From an Index-Based Grading Report

Total Coverage Summary																																																									
SCORE	LINE	COND	TOGGLE																																																						
66.64	99.92	100.00	68.27																																																						
Total tests in report: 1000																																																									
Tests are in graded order																																																									
Shows the accumulated (Total), standalone (Test), and incremental (Incr) score overall and for each metric																																																									
<table border="1"> <thead> <tr> <th colspan="3">SCORE</th> <th colspan="3">LINE</th> <th colspan="3">COND</th> <th colspan="3">TOGGLE</th> </tr> <tr> <th>TOTAL</th><th>TEST</th><th>INCR</th><th>TOTAL</th><th>TEST</th><th>INCR</th><th>TOTAL</th><th>TEST</th><th>INCR</th><th>TOTAL</th><th>TEST</th><th>INCR</th> </tr> </thead> </table>				SCORE			LINE			COND			TOGGLE			TOTAL	TEST	INCR	<table border="1"> <thead> <tr> <th colspan="3">SCORE</th> <th colspan="3">LINE</th> <th colspan="3">COND</th> <th colspan="3">TOGGLE</th> <th>NAME</th> </tr> <tr> <th>TOTAL</th><th>TEST</th><th>INCR</th><th>TOTAL</th><th>TEST</th><th>INCR</th><th>TOTAL</th><th>TEST</th><th>INCR</th><th>TOTAL</th><th>TEST</th><th>INCR</th><th>NAME</th> </tr> </thead> </table>				SCORE			LINE			COND			TOGGLE			NAME	TOTAL	TEST	INCR	NAME																		
SCORE			LINE			COND			TOGGLE																																																
TOTAL	TEST	INCR	TOTAL	TEST	INCR	TOTAL	TEST	INCR	TOTAL	TEST	INCR																																														
SCORE			LINE			COND			TOGGLE			NAME																																													
TOTAL	TEST	INCR	TOTAL	TEST	INCR	TOTAL	TEST	INCR	TOTAL	TEST	INCR	NAME																																													
16.05	16.05	16.05	33.67	33.67	33.67	3.12	3.12	3.12	11.36	11.36	11.36	simv.tmp/test000																																													
22.35	15.71	6.30	44.90	32.65	11.22	6.25	3.12	3.12	15.91	11.36	4.55	simv.tmp/test112																																													
27.97	15.03	5.62	54.08	30.61	9.18	9.38	3.12	3.12	20.45	11.36	4.55	simv.tmp/test036																																													
33.59	15.03	5.62	63.27	30.61	9.18	12.50	3.12	3.12	25.00	11.36	4.55	simv.tmp/test041																																													
38.53	14.35	4.94	70.41	28.57	7.14	15.62	3.12	3.12	29.55	11.36	4.55	simv.tmp/test063																																													
43.12	14.01	4.60	76.53	27.55	6.12	18.75	3.12	3.12	34.09	11.36	4.55	simv.tmp/test124																																													
47.72	14.01	4.60	82.65	27.55	6.12	21.88	3.12	3.12	38.64	11.36	4.55	simv.tmp/test155																																													
51.98	13.67	4.26	87.76	26.53	5.10	25.00	3.12	3.12	43.18	11.36	4.55	simv.tmp/test017																																													
53.49	13.67	1.52	87.76	26.53	0.00	25.00	3.12	0.00	47.73	11.36	4.55	simv.tmp/test138																																													
55.01	13.67	1.52	87.76	26.53	0.00	25.00	3.12	0.00	52.27	11.36	4.55	simv.tmp/test138																																													

Index-based grading is faster in part because it sets a limit on how many covering tests to track for each coverable object. You can adjust this limit higher or lower. If you raise the limit, grading results improve, but more memory and time are used to grade the tests. If you lower the limit, grading results quality may suffer, but the grading finishes faster.

You control the `indexlimit` value using a suboption to `-grade`, as shown below:

```
% urg -grade index indexlimit N
```

where `N` can be any positive integer.

The value of `N` controls the maximum amount of space used per coverable object during the grading process. The default value for `N` is 20.

Note:

All other suboptions of the `-grade` switch (`goal`, `timelimit`, `maxtests`, `minincr`, `requests`, `listonly`, and `precision`) work with the `-grade index` option as well.

Even at the default limit value of 20, index grading consumes more memory than greedy grading (part of the speed comes from doing operations in memory to avoid extra disk IO). Higher values result in more accurate results, whereas lower values use less memory and time.

If you are grading a large design and use a high `indexlimit` in 32-bit mode, URG generates the following warning to inform you that your URG run might run out of memory:

Warning- [URG-LTW] Memory use warning

The design has a very large number of objects and, combined with the `indexlimit` value of 500, urg might run out of memory while grading. If it does, the best option is to use urg in 64-bit mode by using the `-mode64` flag. You can use 64-bit urg even if you used 32-bit VCS.

Alternatively, you can lower the `indexlimit` value using the '`-grade index indexlimit N'` flag, although setting the value too low will reduce the quality of the graded test list results.

Limitations of Index-Based Grading Feature

Index-based grading is not supported for tests containing path, covergroup, or assertions data.

If you specify the `-grade index` option, and a test is found to contain path, covergroup, or assertions data, URG generates the following error message:

Error- [URG-GCGNS] Not supported

The index grading option is not yet supported with path

coverage, covergroup or assertion/property coverage.
Rerun URG but omit the 'index' suboption from the -grade flag.

Difference Reports

The difference (diff) report is used to generate a report that shows the difference in objects covered between two tests, such as test1 and test2. If covered (t) is the set of objects covered by a test t , then URG -diff reports shows the report for covered (test1) – covered (test2).

A new flag –diff is added to URG as follows:

```
urg -dir ... -tests myfile -diff
```

Only two tests may be specified and they must be given in the "-tests" file. If more or less than two tests are specified or the "-tests" flag is not given at all, URG reports an error and exit. The order in which the two tests are specified matters. The first test specified is called the diff test. The second test is the base test. The order is as given in the "-tests" file; the diff test is the first test and the base test is the second test. The operation that URG does is "diff test – base test."

When the flag "-diff" is specified, URG generates the following report that differs from the normal report. Else, the report is exactly like any other URG report. An optional subflag may be specified to change the type of diff from object-based to count-based. If no subflag is given, "-diff object" is implied. The two forms are:

```
urg -dir ... -tests myfile -diff [object]
urg -dir ... -tests myfile -diff count
```

If the "--diff" count flag is given, then the differences between the hit counts is shown for any metrics that have counting data in tests, such that 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 (by default this is 1), then the object is shown as covered.

Only the assert and group metrics are supported with the "--diff" flag. If data from other metrics is present, URG prints a warning message, and the data is ignored. No scores, objects, etc. from other metrics appears in a report generated with "--diff".

Showing the diff Tests in Dashboard and Test Pages

At the top of the dashboard and test pages, the details about the diff being processed is shown. For example, if the "--tests" file contains:

- simv/test2
- simv/test1

Then the dashboard page contains the following details:

Date: Fri Sep 14 16:02:24 2007

User: vernon

Version: Z-2007.06-A

Command line: urg -diff -dir simv.cm 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 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 should be no list of tests in the tests.html/txt file, since the only two tests are the base test and diff test and they are already called out explicitly.

Objects Covered in the diff Report

Default Mode

When you look at a report generated with the "-diff 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 shows only what was covered in the diff test but not covered in the base test.

For example,:

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	vity

A dashboard like this means that, of all line coverage objects in the design, 15.11% of them were covered in the diff test, but were not covered in the base 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:

Base test	Diff test	In Diff Report
Covered	Covered	Not Covered
Covered	Not Covered	Not Covered
Not Covered	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.

Count Mode

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	Base Test	Diff Test	In Diff Report	Reasons
1	10	7	Not covered, count 0	Covered more times in the base test

Hit Count Goal	Base Test	Diff Test	In Diff Report	Reasons
1	7	10	Covered, count 3	Covered three times in the diff test than in the base test
4	7	10	Not covered, count 0	Covered more times in the base test
1	10	10	Not covered, count 0	Covered same number of times in the base test
1	10	0	Not covered, count 0	Covered more times in the diff test than in the base test.
1	0	1	Covered, count 1	Covered one more time in the diff test than in the base test.

The metrics for which counting is not enabled, you can use the same reporting rules as in default mode, even when "--diff" count was given.

Differences in diff and Regular Report

There are no other differences between a diff report and a regular URG report. You cannot see tables with test names in the column headers, etc.

Unsupported flags

The following flags are not compatible with the "--diff" flag. If 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

Support for Exclusion

All types of exclusion is supported with the “–diff” flag, including the “–hier” file and the loading of exclude files.

Determining Which Test Covered an Object Report

You can use the URG “-show tests” option to display, next to each covered object in the report, the names of some of the tests that covered the object. This feature is enabled using the “-show tests” flag or the “-show maxtests N” flag.

When using the “-show tests” flag, only up to three tests is shown for each coverable object. You can change that limit by using the “-show maxtests N” flag, where N is the maximum number of tests you want to show for each object. For example,

```
urg -dir ... -show tests
urg -dir ... -show maxtests 5
```

In which test cover which object mode, URG generates a report that differs from the normal report. Currently, functional covergroups and assertions are supported and are discussed below.

Covered Objects

When reporting which tests covered which object for group coverage, the tables showing the covered bins have additional columns. The “TEST” column gives the ID number assigned to each test, and the COUNT column shows the hit count for that bin. If the maximum number of tests being reported per object is N, then N additional TEST/COUNT columns is shown as follows:

Covered bins

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 assertion, additional tables are added after original assertion tables. Every assertion with non-zero attempt has its own detailed table, with the name as header. First row of a detailed table is the total count, followed by rows of contributing tests. Test numbers (of format T1, T2, ...) are also used here.

For example, an assertion table may look like this:

Cover Directives for Properties: Details

Name	Attempts	Matches	Vacuous Matches	Incomplete Matches
T1C1	40	36	0	4

T1C1

Name	Attempts	Matches	Vacuous Matches	Incomplete Matches
Total	40	36	0	0
T1	20	18	0	0
T2	20	18	0	0

When pointing device hovers over any test number, a pop-up text of the test name appears.

The following illustration is an example of pop-up text:

Covered bins

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	T1	simv2.vdb/test	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

Tests Page

In which test cover which object mode, list of tests in the tests page (tests.html/txt) will 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:

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 following command-line arguments are not compatible with which test cover which object mode. If they are given in this mode, an error will be reported and URG will exit.

- -diff

Resetting or Deleting Covergroup in the Database

As functional coverage model evolves, or changes, the coverage model from the older databases need to be deleted or their coverage should be set to zero. To delete or reset the covergroup in the database, do the following:

1. Create a file that contains a list of commands.
2. Give the file created using "-group db_edit_file <filename>."

The syntax for covergroup Reset and Delete commands are as follows:

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

begin ( funccov|assert) (reset|delete) (module|tree)
(module_name|instance_name) covergroup_name|assert_name
{optional coverpoint crosspoints} end
```

Where:

funccov|assert:

identifies the metric type. ‘funccov’ means you are trying to edit a covergroup. For an assertion you would have used the keyword ‘assert’.

reset|delete:

Identifies what operation you want to perform. ‘reset’ means you want to reset the hit counts to zero. ‘delete’ means you want to delete the specified covergroup or assertion.

module|tree:

Identifies the scope of the operation. ‘module’ means that the operation applies on all instances of the module. ‘tree’ means that the operation applies only on a specific instance.

covergroup_name:

Identifies the name of the covergroup, which is the target of the operation.

optional coverpoint crosspoints:

In case of ‘reset’ operation for ‘funccov’, you can specify a list of coverpoint/crosspoints whose hit counts should be reset to zero.

Using the Reset Command

The Reset command resets the hit counts of all bins under a covergroup to zero.

- To reset covergroup gc under module instance top.i1, use the command:

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

- To reset covergroup gc under all instances of module definition my_mod, use the command:

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

- To reset coverpoint/crosspoint 'ra' under covergroup gc with all instances of module definition my_mod, use the command:

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

- To reset coverpoint/crosspoint 'ra' under covergroup gc with all instance top.i1, use the command:

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

Using the Delete Command

The delete command deletes a covergroup from the coverage database.

- To delete covergroup 'gc' under all instances of module definition my_mod, use the command:

```
begin funcov delete module my_mod gc end
```

- To delete covergroup 'gc' under all instance top.i1, use the command.

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

Resetting and Deleting Assertion Coverage

As design changes, assertions evolve and changes. In the older coverage databases, their hit counts should be set to zero or they should be deleted.

The syntax for resetting and deleting assertion coverage is as follows:

```
begin assert (reset|delete) (module|tree)  
(module_name|instance_name) {optional assertions} end
```

Using the Reset Command

The syntax for using the assertion Reset command is as follows:

- To reset the hit counts assert mid_first from instance top.i1, use the command:

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

- To reset hit counts of all asserts under instance top.i1, use the command.

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

Using the Delete Command

The syntax for using the assertion Delete command is as follows:

- To delete assert mid_first from instance top.i1, use the command:

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

- To delete all asserts under instance top.i1, use the command:

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

- To delete assert mid_first from all instances of module mid, use the command:

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

- To delete all asserts from all instances of module mid, use the command:

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

Mapping Coverage

Code coverage data is collected on a module and instance basis. If the design hierarchy changes, the list of modules or instances changes, and you can no longer directly merge code coverage data.

It is possible to merge coverage data from some non-identical designs using mapping. You can compile a design (or part of a design) with multiple different contexts, and still merge coverage data, but only for subtrees of the design that are identical across all

versions of the design that you're merging. This is useful when some coverage data is collected at the block level and other data for that block is collected in system-level simulation, or when the verification process involves swapping out different top-level testbenches that each instantiate the design.

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

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

The `-map` option syntax is as follows:

```
urg -dir <base_design>.vdb -dir <input_design>.vdb -map  
<module name>
```

Where:

`<base_design>`

The first directory mentioned is the base design and will report the design coverage directory after the merge.

`<input_design>`

The second directory provided to the URG command line is the Input design. The input design coverage is merged into the base design.

The cumulative coverage will be available with the base design coverage directory.

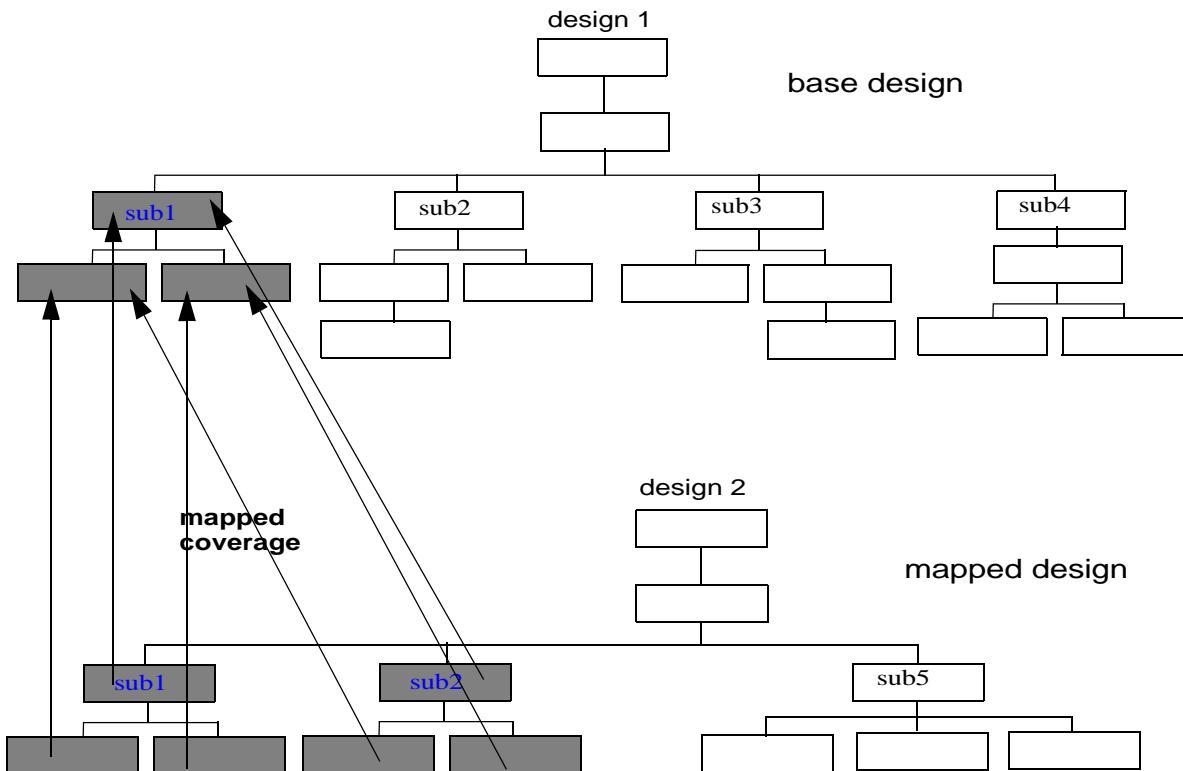
```
<module name>
```

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

Note:

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

Figure 5-6 Mapping Coverage



The illustration shown in [Figure 5-6 on page 37](#), two designs instantiate a common subhierarchy, labeled `sub1`. The code for the subhierarchy, in both designs, is in the source file named `sub1.v`. The module name (identifier) of the top-level module in the

subhierarchy is sub1. This illustration shows mapping coverage information for that subhierarchy from the simulation of design 2 to the coverage information for that subhierarchy from the simulation of design 1. There can be multiple instances of the subhierarchy in the design from which coverage information is mapped (mapped design). However, there can only be one instance of the subhierarchy in the design *to* which the coverage information is mapped (base design).

The following procedure explains how to map coverage information:

1. Compile the base design for coverage and then simulate that design while monitoring for coverage. For example:

```
cd /net/design1  
vcs -cm line sub1.v sub2.v sub3.v sub4.v main.v test.v  
simv -cm line
```

2. Compile the mapped design for coverage and then simulate that design while monitoring for coverage. For example:

```
cd /net/design2  
vcs -cm line sub1.v sub5.v main.v test.v  
simv -cm line
```

3. Run URG specifying the name of the top-level module in the subhierarchy. Also, specify the coverage directory for the base design, then specify the mapped design. For example:

```
urg -dir /net/design1/simv.vdb /net/design2/simv.vdb -  
map sub1
```

Using the above command, the generated report files only contain sections for the module instances in the subhierarchy. These module instances are identified by their hierarchical names in the first (or base) design. The coverage information in these sections is the coverage information from both designs.

If you also include the `-hier` compile-time option to specify a larger subhierarchy that includes the subhierarchy for which you want mapped coverage, the resulting report files contain sections for the module instances in this larger hierarchy, but the sections for the instances in the smaller subhierarchy, that is the subhierarchy with mapped coverage, also contain coverage information from the other design.

Using `-map` for Assertion Coverage Mapping

Syntax

```
urg -dir <base_design>.vdb -dir <input_design>.vdb -map  
<module name>
```

Example

```
urg -dir coverage_chip.vdb -dir coverage_block.vdb  
-map block
```

Scenario-1: Assertions Present in a Separate Bind Module

If assertions are present in a separate bind module for the input design, then `-map` use model is somewhat different when compared to the input design for which assertion is present within the module itself. For example:

```
bind block block_sva bind_My_block1(....); (Assertion  
mentioned in block_sva module for the input design block,  
bind_My_block1 is bind instance here).
```

Syntax:

```
urg -dir <base_design.vdb> -dir <input_design.vdb>  
-map <bind module name>
```

Examples:

```
urg -dir coverage_chip.vdb -dir coverage_block.vdb  
-map block_sva
```

```
urg -dir coverage_chip.vdb -dir coverage_block.vdb  
-map block_sva -map block (This is required if user is  
interested in mapping line, toggle, code coverage for same block).
```

Known issue:

URG generates below warning during report generation time for this case; however mapping is correctly happening.

```
Warning [UCAPI-MAP-INSMISMATCH] Instance mismatch  
in mapping
```

Mapping Subhierarchy Coverage in VHDL

You can also map a VHDL subhierarchy's coverage from one design to the other using the `-map` URG command-line option.

The procedure for VHDL is as follows:

1. Compile for coverage and simulate the base design while monitoring for coverage.
2. Compile for coverage and simulate the mapped design while monitoring for coverage.
3. Run URG specifying the subhierarchy with the `-map` option and the coverage directory for both designs as arguments to the `-dir` option.

4. You specify the name of the entity at the top of the subhierarchy that you use in both designs. You can specify this two ways:

- Specify the entity name:

```
-map entity_name
```

- Specify the library name and entity name:

```
-map library_name.entity_name
```

An example command line for URG is as follows:

```
urg -dir /net/design2/simv.vdb -map sub1
```

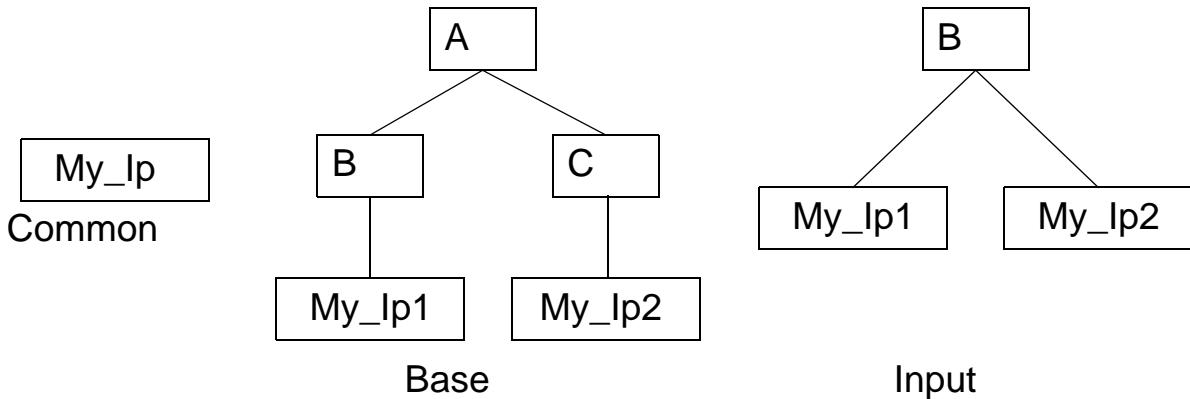
By default, the report contains data only for sections for the entity instances in the subhierarchy. These entity instances are identified by their hierarchical names in the first or base design. The coverage information in these sections is the coverage information from both designs.

Understanding Instance-Based Mapping

Instance-based mapping allows you to specify an instance in your “base design” for which you want to merge coverage data for two different designs.

Assume that you have the same module instantiated in different places in two different designs. (For example, one of the designs could be a system-level design, and the other, an RTL design.)

Consider the following illustration:



Here, you are able to map instances `B.My_Ip1` and `B.My_Ip2` to instance `A.B.My_Ip1`.

The mapping configuration file for the above illustration would contain the following information:

```
MODULE: My_Ip
INSTANCE:
  SRC: B.My_Ip1, B.My_Ip2
  DST: A.B.My_Ip1
```

The syntax is composed of the following elements:

- **MODULE:** Specifies the name of the common module.
- **INSTANCE:** Provides source and destination information for MODULE.
- **SRC:** Represents the list of one or more module instances in the input design.

You can specify multiple instances in a comma-separated list after the `SRC` keyword. If an instance does not appear in the specified input design, VCS ignores that instance.

- DST: Represents a list of one or more module instances in the base design to which the source module instances are to be mapped.

Note:

Both SRC and DST can contain either a full hierarchical path or the * wildcard.

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

The syntax is as follows:

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

Where, `file_name` is the mapping configuration file.

Note the following guidelines:

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

Examples of Instance-Based Mapping

The following examples illustrate how to use the mapping configuration file syntax.

Example 1

If you want to merge all instances of a specified module in the input design, use the * wildcard following SRC:

```
MODULE: My_Ip
INSTANCE:
    SRC: *
    DST: A.B.My_Ip1
```

This syntax merges all the instances (B.My_Ip1, B.My_Ip2) of the input design into A.B.My_Ip1.

Example 2

If you want to merge one instance of the input design into all the instances of the base design, use the * wildcard after DST:

```
MODULE: My_Ip
INSTANCE:
    SRC: B.My_Ip1
    DST: *
```

This syntax merges the data from B.My_Ip1 into both A.B.My_Ip1 and A.C.My_Ip2.

Using -mapfile for Assertion Coverage Mapping

Syntax

```
urg -dir <base_design>.vdb -dir <input_design>.vdb  
-mapfile <file_name>
```

Where, file_name is the mapping configuration file.

Example

```
urg -dir coverage_chip.vdb -dir coverage_block.vdb  
-mapfile block
```

Scenario-1: Assertions Present in a Separate Bind Module

If assertions are present in a separate bind module for the input design, then `-mapfile` use model is somewhat different when compared to the input design for which assertion is present within the module itself. For example:

```
bind block block_sva bind_My_block1(. . .);  
(Assertion mentioned in block_sva module for the input design  
block, bind_My_block1 is bind instance here).
```

Syntax:

```
urg -dir <base_design.vdb> -dir <input_design.vdb>  
-mapfile <block>
```

`-mapfile` Example:

MODULE: block

INSTANCE:

SRC: B.My_block1.bind_My_block_1,
B.My_block2.bind_My_block2

DST: A.B.My_block.bind_My_block_base

MODULE: block (This will require if user is also interested mapping
line, toggle etc code coverage for same block).

INSTANCE:

SRC: B.My_block1, B.My_block2

DST: A.B.My_block

Note:

URG issues below warning during report generation time for this case; however mapping is correctly happening.

Warning [UCAPI-MAP-INSMISMATCH] Instance mismatch
in mapping

Scenario-2: Swapping the Module Name with Bind Module Name

Currently, URG does not issue any warning or error messages if you swap the module name with bind module name using the `-mapfile` option, as mentioned below. In both cases, you will be able to see correct mapping.

-mapfile:

MODULE: block_sva

INSTANCE:

SRC: B.My_block1.bind_My_block_1,
B.My_block2.bind_My_block2

DST: A.B.My_block.bind_My_block_base

Using Multiple `-map` Options

You can specify multiple `-map` options as long as you also specify the respective `-dir` options when using URG. For example:

```
urg -dir chip.vdb t1.vdb t2.vdb -map t1 -map t2
```

The first directory specified with `-cm_dir` must have both `t1` and `t2` instantiated.

In addition, you can use URG to merge intermediate coverage data files that were generated on different platforms. In other words, the intermediate information in the coverage directories is platform-independent.

Mapping Relocated Source File

URG determines the location of the source directory or source file using information built into the coverage database.

However, if you change the location of a source file, you can map the path to the relocated file using the `-pathmap` option.

By default, the source directory, which is provided at compile time, is assumed to be constant. Using this feature, you can change the location of the source file after compile time.

Syntax for relocating the source file

```
urg -pathmap <file>
```

where,

`<file>` — contains one or more mapping rules in the format of `<from> : <to>`

`<from>` — is an exact match of the path prefix of the original source directory.

`<to>` — refers to the path of the relocated directory.

For example:

```
urg -dir simv.vdb -pathmap pmap
```

The file “pmap” contains the following rule:

```
/home/work/src : local/files/source
```

Thus, for the source file /home/work/src/test1.v, URG will try local/files/source/test1.v first and then /home/work/src/test1.v.

Exclusion

Any complex design will probably have code or combinations that can never happen. For example, certain combinations of values may never occur together. When you're trying to verify the design, you want to exclude these impossible coverage targets, so you can focus testing on the parts of the design that can happen. This section explains some ways to remove impossible or uninteresting parts of the design from coverage monitoring.

There are two complementary types of exclusion. One is the exclusion of general areas that are not of interest - "don't care" exclusion. When you exclude a module, instance, covergroup, coverpoint, or cross you are saying that you don't want the scores from those regions taken into account when computing your coverage score. From the point of view of your verification plan, it doesn't matter if they are covered or not.

The other type of exclusion is done when you are asserting that a given coverable object - for example, a signal bit, a cross bin, a line of code - cannot be covered, and therefore should not be counted

when computing your coverage score. When you exclude at the object level, you are saying not to count that object because it can't be covered. In fact, it is not legal to exclude an object if it is covered, and if it becomes covered an error will be flagged.

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).

Exclusion Using DVE Coverage GUI

Note:

Exclusion for condition sop is not supported in DVE, because it is not supported in UCAPI.

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 -dir simv.vdb -excl.strict
```

or choose the option "Do not allow Covered Objects to be excluded" in the DVE Application Preferences window accessed from the **Edit > Preferences** menu > **Exclusion** category. 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.

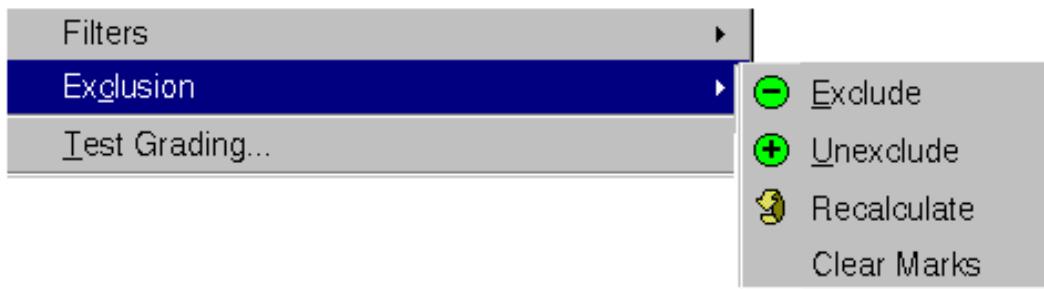
After you have excluded objects in DVE, you can save those exclusions to a file. The mode you are in (Default or Strict) when you create an exclude file is saved with the rest of that file's contents. If you load an exclude file in a different mode than the mode in which it was created, a warning message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

Exclude

Unexclude

Recalculate



Clear Exclusion



Load Exclusion



Save Exclusion

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.

In DVE detailed pane, you can do multiple select in List pane using the **Ctrl** key for all coverage metric. In case of line metric, the multiple select is possible only in Source pane. The same procedure can be followed for all metrics.

Following is an example for condition metric:

1. Start DVE in coverage mode.
2. Load the coverage database.
3. Open the condition metric coverage detailed pane.
4. Click on any expression or sub-expression in the expression list pane.
5. Press the **Ctrl** key and select the desired expressions or sub-expressions.
6. Right-click and select **Exclude** to exclude all the selected items.
7. Re-calculate.

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.

Adding Exclusion Annotation

You can add an annotation to the excluded coverable objects, such as mentioning the reason behind the exclusion to an exclusion file. Upon loading the elfile, the annotation can be displayed in URG reports and the DVE coverage GUI. Consider the following exclusion file as an example:

```
MODULE : top
ANNOTATION: "Unreachable signal"
Toggle a
```

In this example, you add the annotation "Unreachable signal" to signal `a` of module `top`, which you want to exclude. You can add, edit, and delete exclusion annotation to an excluded object or scope in DVE.

Note:

The annotation is not added for the scope, container, or object that is not excluded/partially excluded, even if you have it in the elfile or edit it in the GUI before recalculation. The annotation will only be set for the fully excluded object, scope, or container after recalculation.

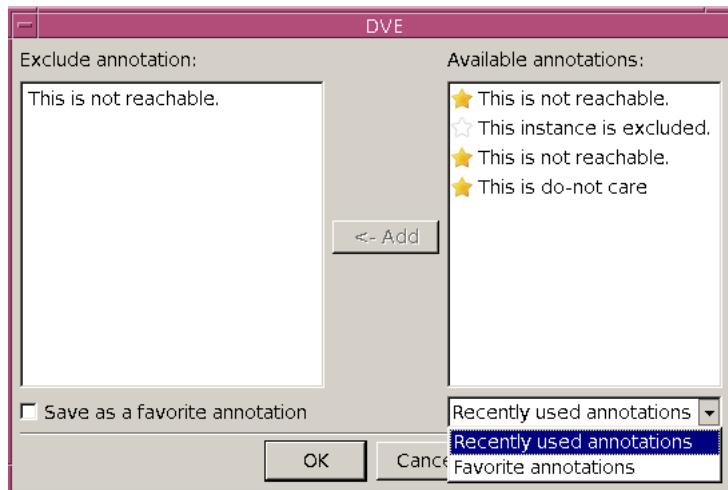
To add and delete exclusion annotation

1. Start DVE in coverage mode.
2. Load the coverage database.

3. Select the object or scope in the Hierarchy pane, Detail view, or Summary view.
4. Right-click on a object, select **Exclude** to mark for exclusion.
5. Right-click on the excluded object, scope, or container and select **Edit Exclude Annotation**.

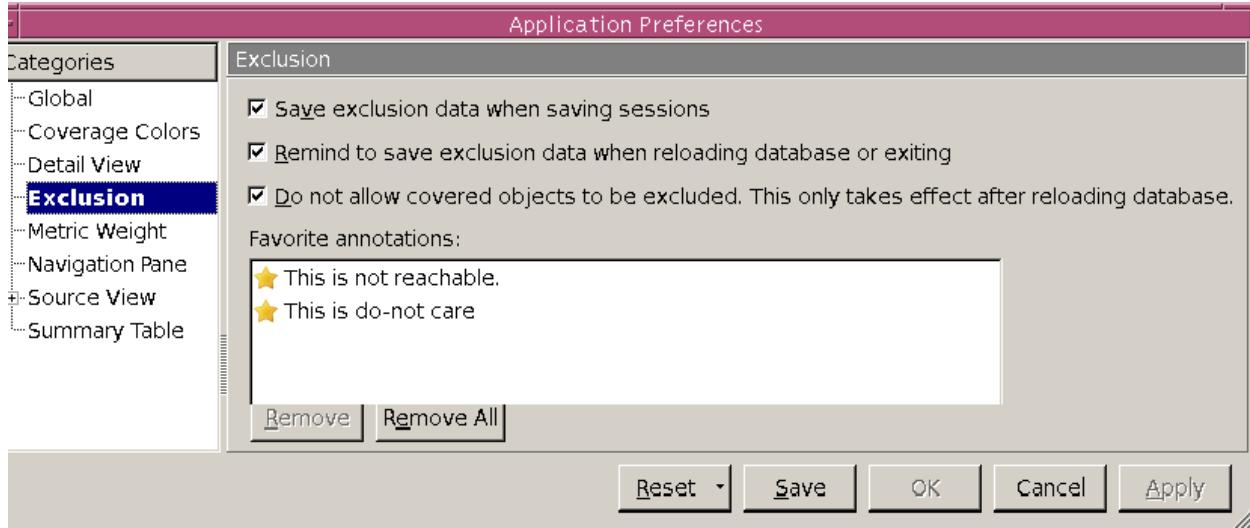


The Add Exclude Annotation dialog box appears that contains the following fields.



- Exclude annotation — Identifies a text area where you can type your exclusion annotation.
- Available annotations — Lists all the annotations that you have created.

- Save as a favorite annotation — Saves the added annotation as your favorite, if selected. Favorite annotations are saved in user preferences, as shown in the following figure:



If you save any annotation as your favorite, the next time when you start DVE, you can use the favorite annotations. Recently used annotations are not saved

- Add — Adds annotation to Exclude annotation column from the list of available annotations.
- Recently used annotations — Contains two options, recently used annotations and favorite annotations. The selected option will be effective next time when you open the Add Exclusion Annotation dialog box.

6. Type the exclusion comment under the Exclude annotation text area.

You can add annotation up to 16K characters.

7. Select the **Save as a favorite annotation** check box and click **OK**.

The exclusion annotation is saved and is displayed in the tooltip, when you point your mouse over the object, container, or scope, in the Summary view.

The following icon indicates the item is annotated.



8. Select the object in the Hierarchy pane, right-click and select **Delete Exclude Annotation**.

The exclusion annotation is deleted.

Note:

- You can select multiple objects to add the same annotation.
- The annotation on module is not propagated to its instances.

You cannot add annotations to unexcluded objects.

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 exclusion symbols:



- 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	<input checked="" type="checkbox"/> 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%
•	est0					
•	est1	64.68%	89.09%	55.10%	70.59%	43.96%
•	est2	27.77%	40.74%	41.89%	11.76%	16.67%
	coin1	24.55%	40.74%	29.03%	11.76%	16.67%
	kp1					
•	est3	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%
•	est4	20.54%	37.27%	30.61%	0.00%	14.29%

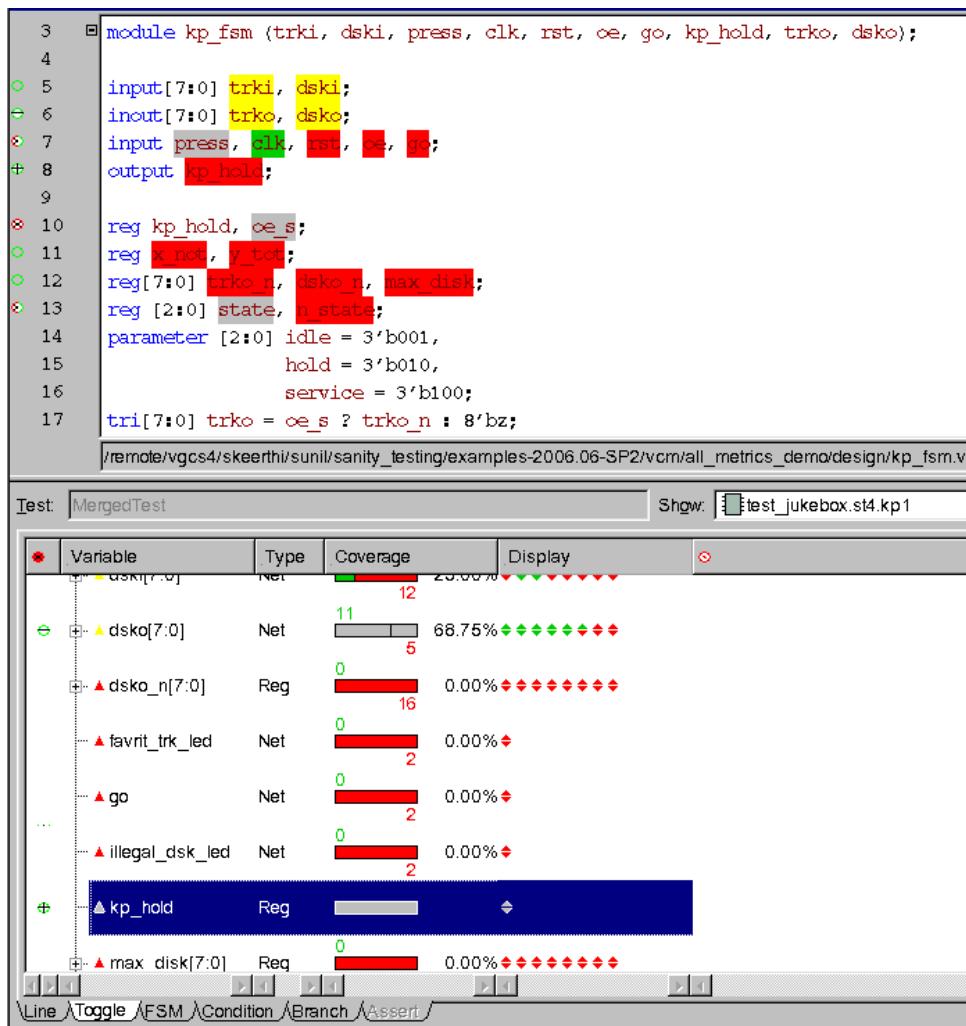
Interactive Marking within Detail Views

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 5-7 Detail view showing markers in Source and List windows

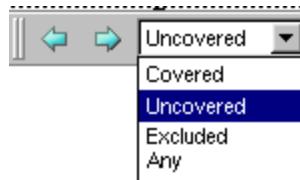


Within the list views, you can toggle the exclusion markers by clicking them. The markers toggle as follows:

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

Exclusion Browser

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



Saving and Loading Saved Exclusions

There are two ways to save exclusions:

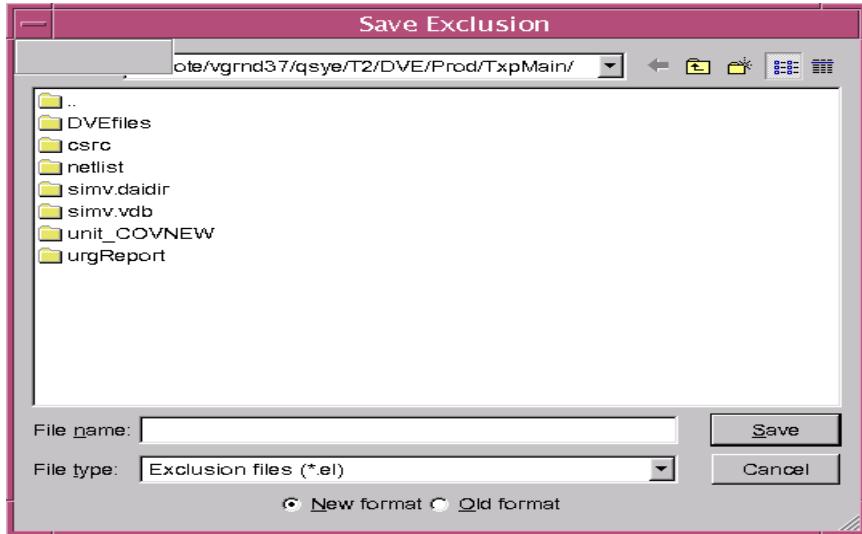
- Using the **File > Save Exclusions** menu.
- Using the **Edit > Preferences > Exclusion**, then selecting the option **Save exclusion data when saving sessions**.

To save an exclusion state using the Save Exclusions menu

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

The Save Exclusion File dialog box appears.

2. Enter a name for the exclude file.



3. Select the **New Format** radio button, if not already selected.

If you select the **Old format** radio button, a warning message appears.

4. Click **OK**.

The exclusion state is saved.

Exclusion File with the Unified Database

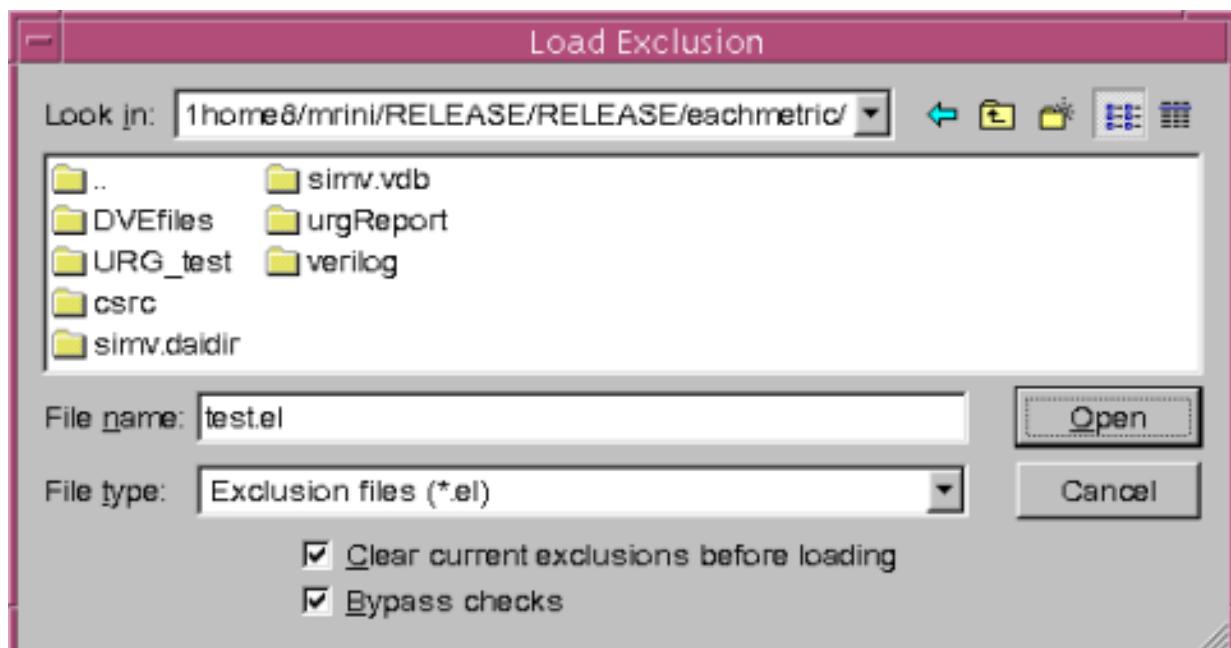
The elfiles generated with the unified database doesn't work with the old database, therefore to distinguish the elfiles generated using the old database format, you should add a string //TIMESTAMP to the elfiles.

To convert the old database format elfiles to the new format, load the coverage database and the old format elfile, then select the **New Format** radio button and save the elfile with a different name in the Save Exclusion dialog box. You can also save the new format elfile in the old database format, using the **Old format** radio button in the Save Exclusion dialog box.

To load the exclusion file

1. From the **File** menu, select **Load Exclusions > From File....**

The Load Exclusion dialog box appears.



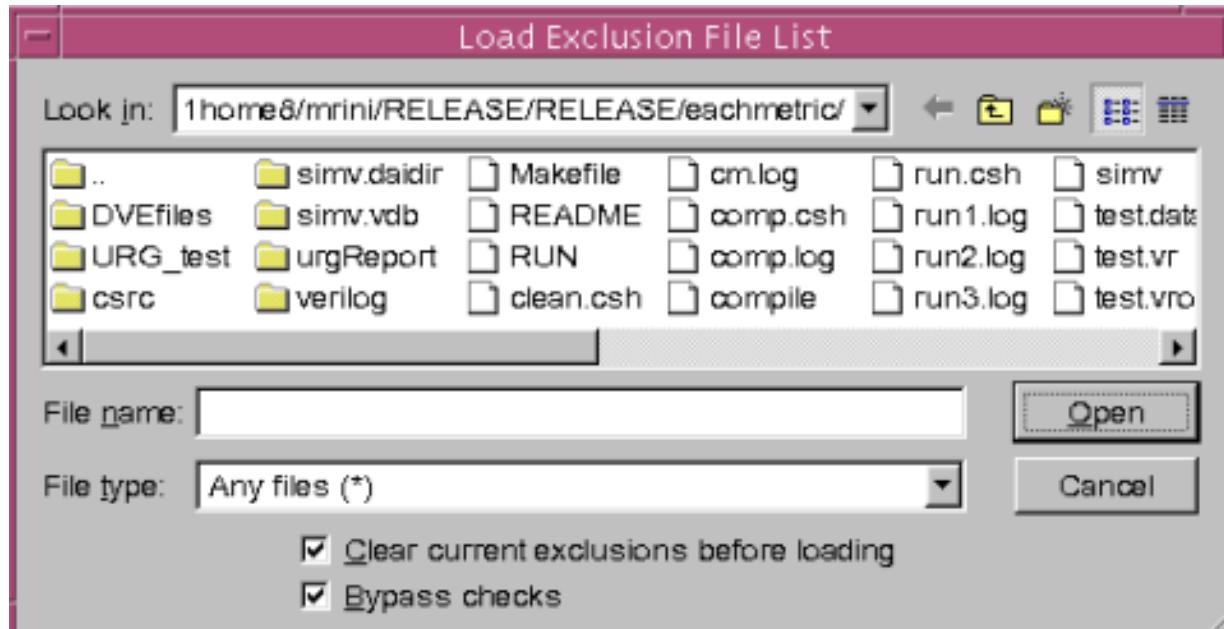
2. Select an exclude file from the directory.
3. (Optional) Select the **Clear current exclusions before loading** check box to clear all the exclusions.
4. Select the **Bypass checks** check box to bypass the checksum checks.
5. Click **Open**.

The exclusion file is loaded bypassing the checksum validation.

To load multiple exclusion file from a file list

1. From the **File** menu in the DVE Coverage GUI, select **Load Exclusions > From File List.**

The Load Exclusion File List dialog box appears.

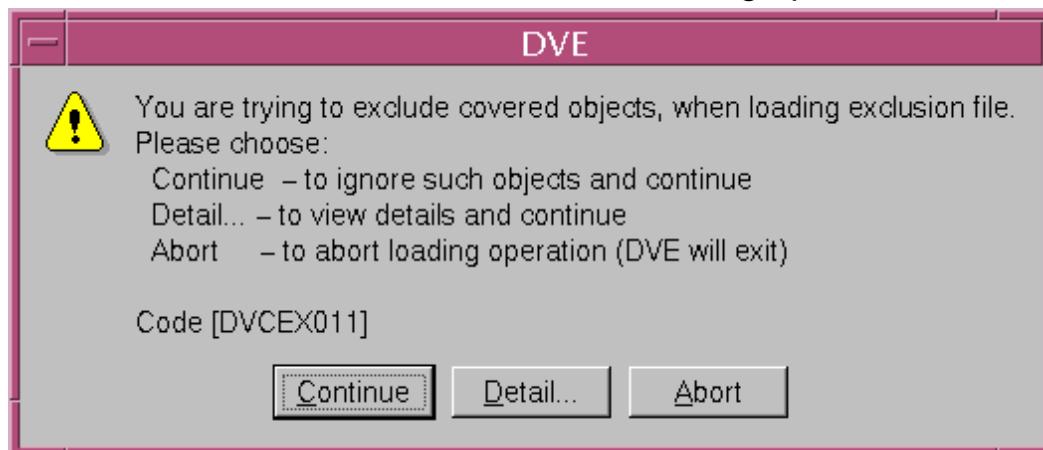


2. Select the file list in which you have stored multiple exclusion files and click **Open**.

The exclusion files are 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 view.

- Right-click the green exclusion marker beside a particular line number and select the exclude/include option.

The screenshot shows the DVE (Digital Verification Environment) interface. The title bar reads "DVE – CovTopLevel.1 – [CovDetail.1 (MergedTest|Line) – test_jukebo". The menu bar includes File, Edit, View, Scope, Window, Help. The toolbar has various icons for file operations and analysis. The left pane is a "Hierarchy" tree showing the design structure: "Design Hierarchy" > "test_jukebox" (Module) > "cd1 (cd)" > "fifo1 (fifo)" > "jb1 (jukebox)" > "st0 (station)" > "st1 (station)" > "st2 (station)" > "st3 (station)" > "st4 (station)". The right pane displays Verilog code with line numbers 29 to 45.1. Lines 32, 34, 35, 36, 38, 40, 42, 43, 44, 45, and 45.1 have green markers next to them, indicating they are excluded or included. Line 39 has a red marker and contains the error message "\$display (\$stime, " tried to write a full fifc");".

```

29
30     /*** the write ***/
31     always @ (posedge clk)
32         if ((write > 0) & !full)
33             begin
34                 dfifc[head] <= #`DELAY data_in;
35                 #`DELAY head = head + 1;
36                 e_count = e_count + 1;
37             end
38         else if (write & (full != 0))
39             $display ($stime, " tried to write a full fifc");
40     /*** the read ***/
41     always @ (posedge clk)
42         if (read & !empty)
43             begin
44                 data_out <= #`DELAY dfifc [tail];
45                 #`DELAY tail = tail + 1;
45.1                e_count = e_count - 1;
46             end

```

The object is excluded/included.

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.

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.

Excluding Half Toggle Transitions

Toggle Coverage for a bit is combination of two sequences 0 -> 1 and 1 -> 0, which gets counted in the final score of the toggle coverage. The total count of toggle coverable objects for signal is the number of bits in that signal times two, where two is number of sequence per bit.

You can exclude either sequence of a given bit in a signal. For example, you can exclude only the 0 to 1 transition for a bit.

Consider the below example, where p[0] goes from 0 to 1 and p[1] goes from 1 to 0:

```
initial begin
reg [1:0] p;

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

You can exclude the 1 to 0 transition for p[0] and the 0 to 1 transition for p[1], since neither can occur in this code.

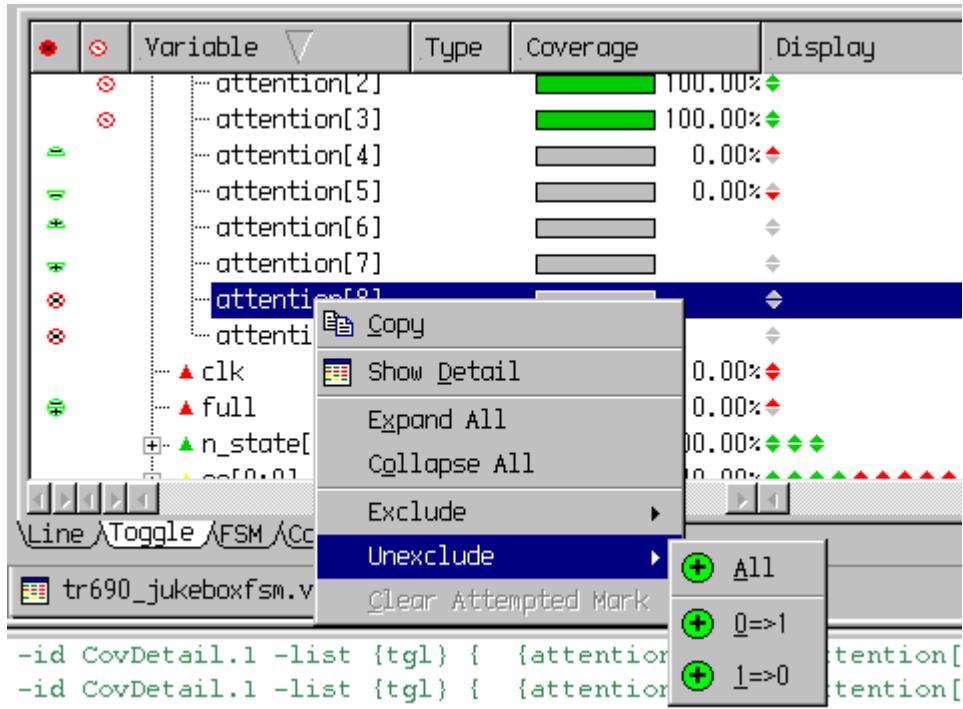
For the above example, the toggle coverage exclude file will be as follows:

```
== elfile ==
Module: top
Toggle 0to1 p[1]
```

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.

Toggle Coverage MDA Exclusion

DVE Coverage supports the Verilog MDA exclusion. You can exclude the entire MDA range items or individual bits of an MDA range items, in the Detail window.

By default, the VCS coverage engine does not monitor MDAs as a part of toggle coverage. You can use the `-cm_tgl` and `-cm_tgl_mda` options, as shown below, to enable monitoring of MDAs.

```
%vcs -cm_tgl -cm_tgl_mda
```

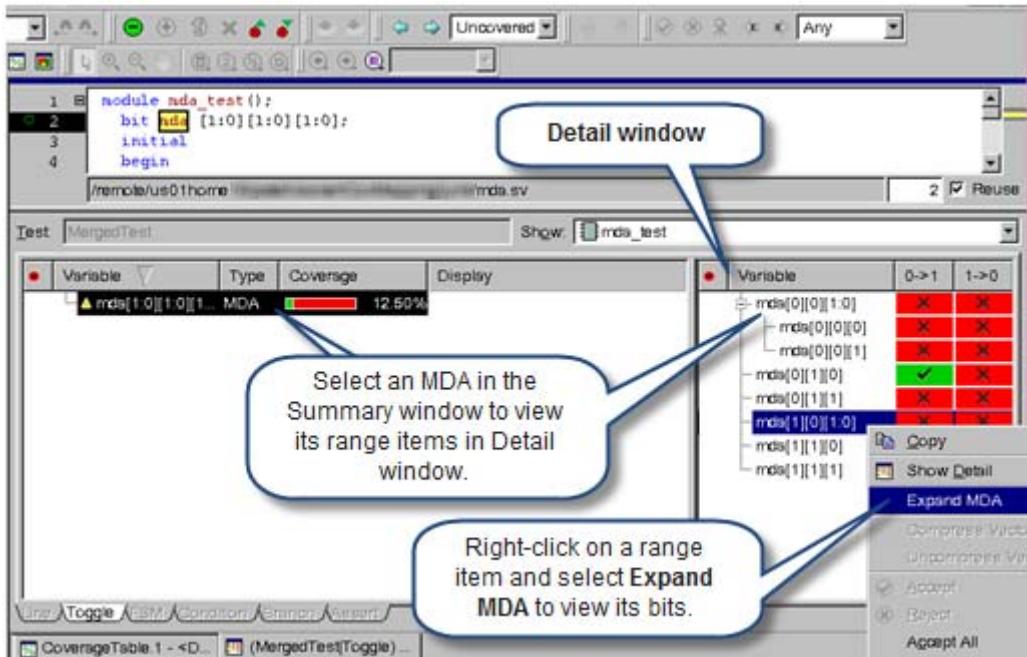
To exclude bits of an MDA range item, perform the following steps:

1. Select an MDA range item in the Summary window to view its range items in the Detail window.
2. Right-click and select **Expand MDA** to show the compressed MDA bits, as shown in [Figure 5-8](#).

The compressed MDA bits are visible.

3. Select **Exclude/Unexclude** icons to exclude/include the MDA bits.

Figure 5-8 MDA Exclusion



Note:

If there are multiple dimensions (for example, A [1 : 3] [4 : 7]), DVE expands them, dimension by dimension and from left to right.

Elf file Syntax for MDA Exclusion

You can use elf file to exclude MDA completely or to specify a list of ranges to be excluded. The following examples shows the syntax to exclude MDAs.

Elf file 1 excludes the MDA as a whole, as follows:

```
MODULE: test
Toggle mda1
```

Elf file 2 excludes a specific range of the MDA, as follows:

```
MODULE: test
Toggle mda1[1:16] [7:4]
```

Limitations

- Compile-time MDA exclusion by the `-cm_hier` option for bit-select and part-select is not supported.
- Exclusion of signal transitions from "0->1" and "1->0" in the left window are not supported.
Session reloading that restores the state of expansion is not supported.

Excluding Covergroups

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

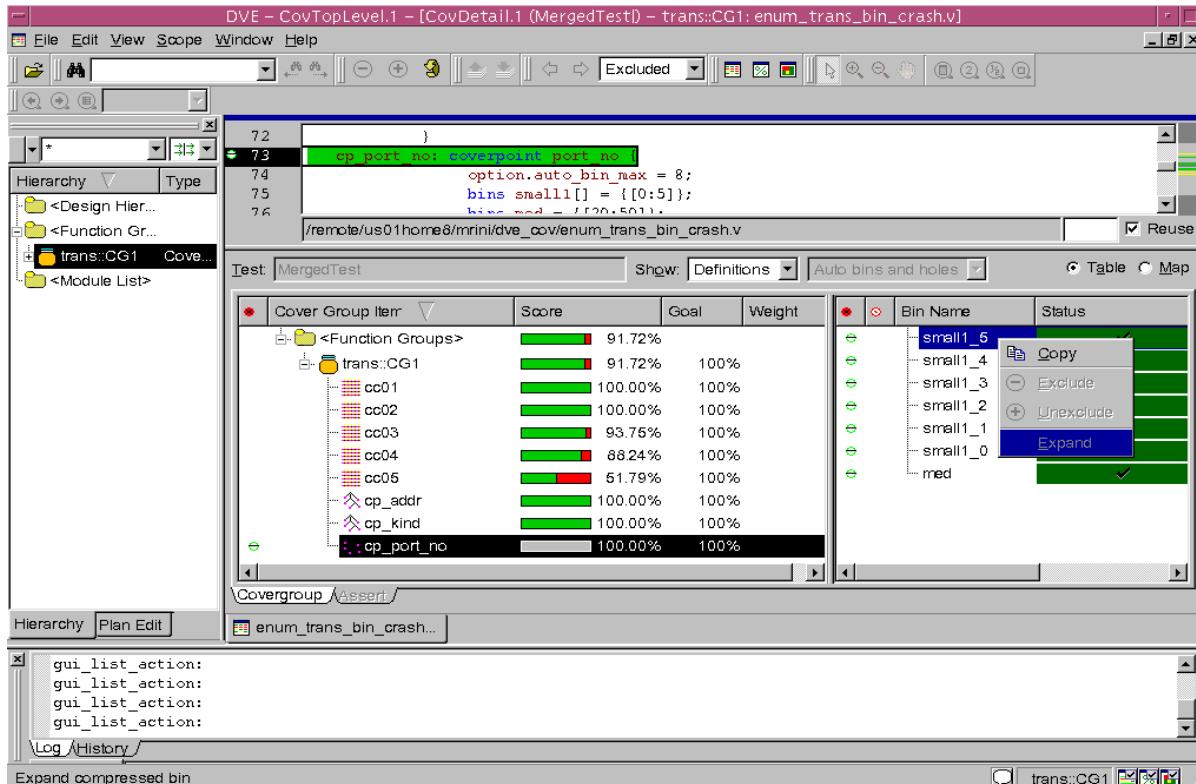
To exclude covergroups in DVE

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 Exclusions](#)” on page 63.

Note:

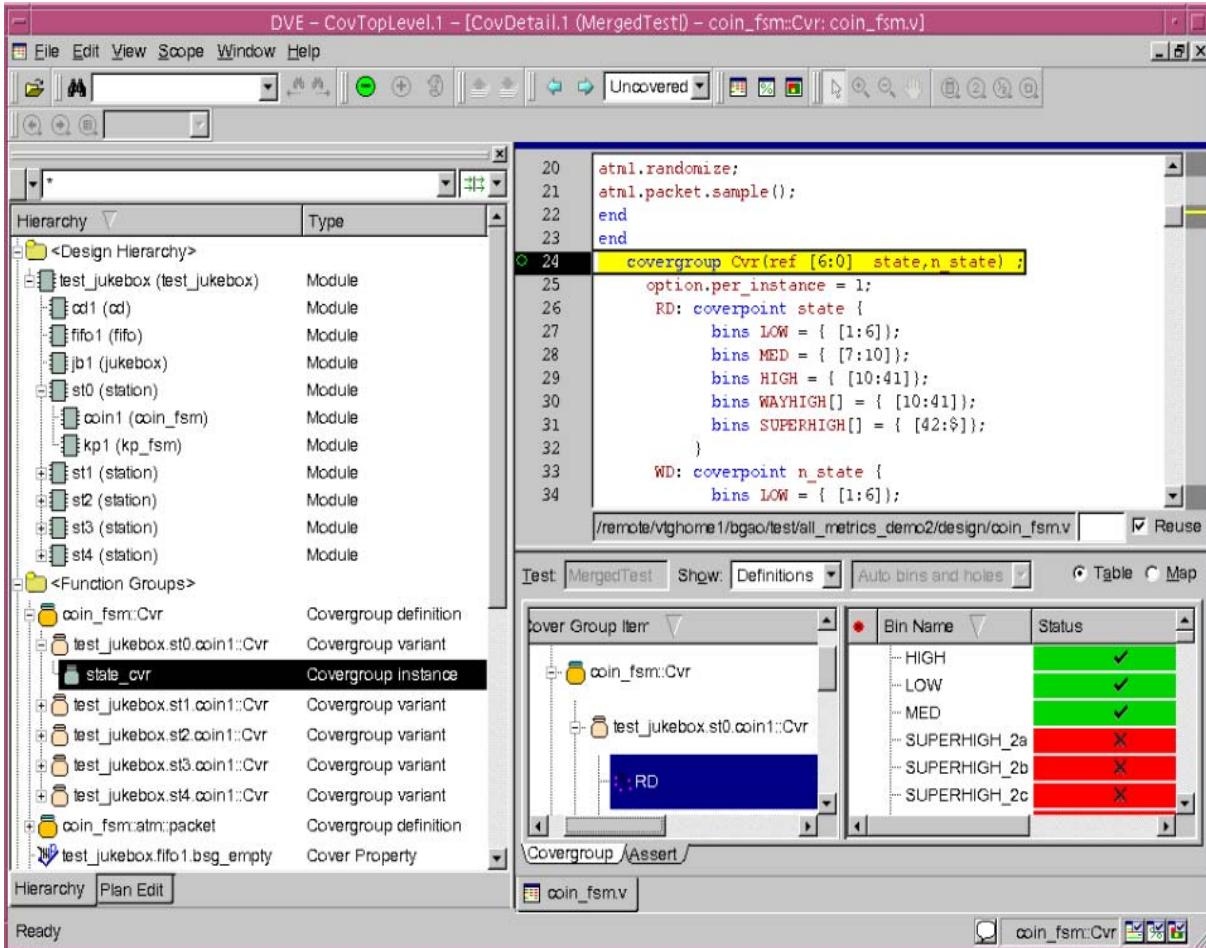
In Strict mode, you cannot exclude a cover point/cross once the score is 100%. When the score is greater than 0 and less than 100, its uncovered children are excluded and covered children are marked as attempted.

Exclusion Propagation

Coverpoint and coverpoint bins exclusion are propagated to the crosses automatically. If you exclude coverpoint or coverpoint bins, 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.



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 bins 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.

Cross-of-a-Cross Exclusion Support

The exclusion of a cross-of-a-cross or its bins in the URG-based exclusion flow. This feature helps you achieve your coverage targets by eliminating unreachable or uninteresting coverage goals.

Excluding Cross-of-a-Cross

You can exclude a cross-of-a-cross either by selecting it from DVE or by specifying the cross-of-a-cross or its bins in an exclusion file.

Using Exclusion File to Exclude Cross-of-a-Cross

To exclude a cross-of-a-cross, specify the name of the cross-of-a-cross with keyword `coveritem` under the covergroup in which it is defined. Following is the syntax:

```
covergroup <scope_name>::<covergroup_name>
    coveritem <cross-of-a-cross_name>
```

To exclude bins of a cross-of-a-cross, first specify the cross-of-a-cross and then specify its bins. Following is the syntax:

```
covergroup <scope_name>::<covergroup_name>
    coveritem <cross-of-a-cross_name>
    {bins_defn}
```

Consider the covergroup shown in [Example 5-3](#):

Example 5-3 Excluding Cross-of-a-Cross #1

```
covergroup cov1  @(posedge clk);
    cp1: coverpoint a;
    cp2: coverpoint c;
    cp3: coverpoint b;
```

```

    cr1: cross cp1, cp2;
    cr2: cross cr1, cp3;
endgroup

```

Based on [Example 5-3](#), you can use the following elfile to exclude cross-of-a-cross cr2 as a whole:

```

covergroup top::cov1
    coveritem cr2

```

Or, to exclude a specific bin of cross-of-a-cross cr2, you can use the following elfile. The bin is the cross of auto[1] of coverpoint a, auto[0] of coverpoint c and auto[0] of coverpoint b.

```

covergroup top::cov1
    coveritem cr2
    bins {{auto[1]}, {auto[0]}, {auto[0]}}

```

Now, consider the covergroup shown in [Example 5-4](#):

Example 5-4 Excluding Cross-of-a-Cross #2

```

covergroup cov1 @(posedge clk);
    cp1: coverpoint a;
    cp2: coverpoint c
    {
        bins B1= {0};
        bins B2= {1};
    }
    cp3: coverpoint b
    {
        bins b1 = {[0:7]};
        bins b2 = {[8:15]};
    }
    cp4: coverpoint d;
    crbd : cross cp3, cp4;
    cr1: cross cp1, cp2;
    cr2: cross crbd, cp1;
    cr3: cross cr2, cp2;

```

```
endgroup
```

Based on [Example 5-4](#), you can use the following elfile to exclude the bin of cross-of-a-cross cr2, which is cross of user-defined bin b1 of coverpoint b, autocross bin range of auto[0] – auto[15] of coverpoint d, and auto-bin auto[0] of coverpoint a.

```
covergroup top::cov1
    coveritem cr2
        bins {{b1}, {auto[1]-auto[15]}}, {auto[0]}
```

Also based on [Example 5-4](#), you can exclude a bin of cross-of-a-cross cr3. Note that cr3 is made up of a cross-of-a-cross (cr2) and coverpoint cp2.

```
coveritem cr3
    bins {{{{b1}, {auto[1]-auto[15]}}, {auto[0]}}, {B1}}
```

The exclusion status of a coverpoint, coverpoint bin, cross, cross bin, cross-of-a-cross, and cross-of-a-cross bin is propagated to the container cross-of-a-cross.

In [Example 5-4](#), if the bin b1 of coverpoint b is excluded, the status is automatically propagated to the bins of cr2 which contain b1 (for example, {{{b1}, {auto[1]-auto[15]}}, {auto[0]}}). Similarly, the bin bins {{{{b1}, {auto[1]-auto[15]}}, {auto[0]}}, {B1}} of cr3 are also excluded.

Enhanced Syntax of Covergroup Exclusion

For manually written exclusion files, you must specify a cross-of-a-cross and its bins using the following syntax. In this syntax description, the line highlighted in bold shows the enhancement made to the syntax. The line highlighted in gray is replaced by the line highlighted in bold from this version forwards.

```
covergroup_spec ::= cg_excl_list
| cg_inst_excl_list
| cg_excl_spec
cg_excl_spec ::= covergroup [ex_identifier]
{coveritem_exclusion}
cg_excl_list ::= covergroup ex_identifier_list
ex_identifier_list ::= ex_mod_identifier [,
ex_mod_identifier]
cg_inst_excl_list ::= covergroup ex_inst_identifier_list
ex_inst_identifier_list ::= ex_inst_identifier [,
ex_inst_identifier]
ex_mod_identifier:: hierarchy::covergroup_name_identifier
ex_inst_identifier:::
hierarchy.covergroup_inst_name_identifier
coveritem_exclusion ::= coveritem_excl_list
| coveritem_excl_spec
coveritem_excl_list ::= coveritem ex_identifier_list
cp_excl_spec ::= coveritem ex_identifier
{bin_exclusion}
bin_exclusion ::= bin bin_list
bin_list ::= bin_spec [, bin_spec]
bin_spec ::= cp_bin_spec
| auto_cross_bin_spec
cp_bin_spec ::= ex_identifier
| auto_cp_bin_spec
auto_cp_bin_spec ::= auto[bin_range]
| auto[bin_range]-auto[bin_range]
bin_range ::= ex_identifier
| ex_identifier:ex_identifier
auto_cross_bin_spec ::= {cross_bin_dim_list}
cross_bin_dim_list ::= cross_bin_dim [, cross_bin_dim]
```

```
cross_bin_dim_list ::= cross_bin_dim | auto_cross_bin_spec  
[, cross_bin_dim | auto_cross_bin_spec ]  
cross_bin_dim ::= {} | {cp_bin_list}  
cp_bin_list ::= cp_bin_spec [, cp_bin_spec]  
ex_identifier ::= simple_identifier | escaped_identifier
```

Generating URG Reports Using Exclude Files

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.

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

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

The figure [Figure 5-9](#) 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 5-9 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

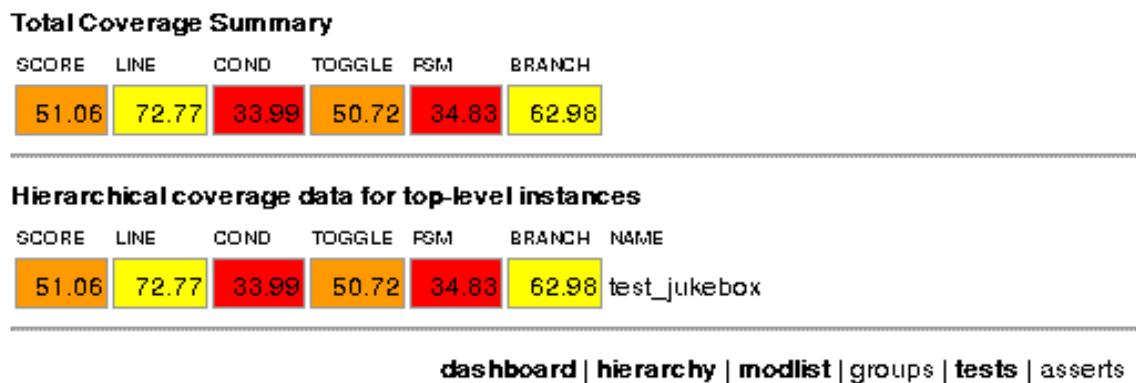
Hierarchical coverage data for top-level instances						
SCORE	LINE	COND	TOGGLE	FSM	BRANCH	NAME
49.97	72.66	33.98	50.61	34.83	57.77	test_jukebox

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

When the exclude file is specified using the switch `elfile`, URG generates the following report [Figure 5-10](#). 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.vdb -elfile filename.el
```

Figure 5-10 Example of a Report generated with an exclude file



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.vdb -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 bins level exclusion in terms of its name.
- 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 bins 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 bins 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.

Elfile Checksum

When exclusions are saved into a file, DVE generates a checksum for the exclusions and dumps it into the file. This checksum is obtained from the metric variant checksum information and names of the modules (or covergroups for testbench metric). The purpose of the checksum is to synchronize the checks.

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 the check_sum, the synchronization checks are not done.

For code coverage metrics, one line per excluded module is dumped and this line consists of checksums of all the metrics of which objects are excluded. This line starts with the prefix \\ MOD_CHKSUM. Here is an example:

```
\\" MOD_CHKSUM: 3 mod v11 L1411939641 v11 T1074138296 v11  
C2409237268 v10 B301527767
```

For covergroup metric, the checksum information starts with the prefix // Checksum and consists of the names of the excluded covergroups and its instances. An example is shown below:

```
// Checksum: 300200840 3953929699 cg4_i | 3562878031  
3865826166 cg3_i | 1282958077 2393771040 cg2_i
```

Line Exclusion Format

The line exclusion format example is shown below. It contains information on the checksum, excluded block_ids, and the module or instance scope for which the exclusion is carried out.

Exclusion Format

```
CHECKSUM: "2216186502 3722401086"  
INSTANCE: test.inst1  
Block 7  
Block 9  
Block 10
```

Verilog File

```
20 always @(data,state) begin  
21 case(state)  
22  
23 3'b000:begin
```

```

24      if(data)
25          next_state=3'b001;
26      else
27          next_state=3'b000;
28      end
29 3'b001:begin
30      if(!data)
31          next_state=3'b010;
32      else
33          next_state=3'b001;
34      end

```

URG Report

```

20                               always @(data,state) begin
21                               case(state)
22                               1/1
23                               1/1      3'b000:begin
24                               excluded      if(data)
25                               excluded      next_state=3'b001;
26                               excluded      else
27                               excluded      next_state=3'b000;
28                               excluded      end
29                               excluded      3'b001:begin
30                               excluded      if(!data)
31                               excluded      next_state=3'b010;
32                               excluded      else
33                               excluded      next_state=3'b001;
34                               1/1

```

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
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 grouped in a single comma-separated list.

Example elfile3

```

covergroup top.i1::cov1
    coveritem cp1
    bins auto[8], auto[10], auto[11], auto[12], auto[13]
    coveritem cp2
    bins b3, b5
    coveritem cp3
    bins zb2
    coveritem cc1
    bins cb1
    coveritem cc2
        bins {{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 bins specification $\{\{\text{auto}[1], \text{auto}[4]\}, \{\text{b}2\}, \{\text{zb}1[13], \text{zb}1[14], \text{zb}2\}\}$ 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
        bins {{auto[1], auto[ 8]}, {}, {zb1[18]}} // {} -
            Consider all bins of cp2 in cross exclusion
    coveritem cc2

        bins {{}, {}, {}} //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
bins {{}, {b3,b4,b5}, {}}
```

Thus, URG will automatically exclude these uncovered bins while generating URG report.

Exclusion Annotations in URG

You can view the exclusion annotation in URG, in the last column of each coverable object, in the table for Condition, FSM, Toggle, and Branch. For Line coverage, you can see the annotation in the line below the excluded line in the Line Coverage Report.

Following is an example of exclusion annotation in a Branch coverage report:

Branches:

-1-	Status	Exclusion Annotation
3'b000	Covered	
3'b001	Covered	
3'b010	Covered	
3'b011	Covered	
3'b100	Covered	
default	Excluded	This is do-not care.

Note:

The existing annotation is overwritten when loading multiple exclusion files if there are annotations in each file for a scope/object.

Exclusion Annotation Syntax

The annotation in the Exclude file is added before the coverable object.

The keywords - ANNOTATION, ANNOTATION_BEGIN, and ANNOTATION_END are used to describe annotations.

The ANNOTATION keyword is used for adding annotation to a single scope/object.

The ANNOTATION_BEGIN/ANNOTATION_END keywords are used for adding annotations for scopes or containers.

Example 1

MODULE: top

```
ANNOTATION: "it is unreachable."
Signal a
signal b
```

In the above example only signal "a" is excluded and annotated.

Example 2

INSTANCE: TOP.CPU.ALU

```
ANNOTATION_BEGIN: "unreachable"
Condition 1 (1)
Condition 1 (2)
Condition 1.1 (1)
Condition 1.1 (2)
```

ANNOTATION_END

Exclusion Mark in URG

URG marks the partially or fully excluded modules or instances with exclusion symbols in hierarchy, modlist and modinfo html/txt based pages in urg report directory generated.

In the HTML reports, the marks are similar to DVE. Partially excluded scopes(modules or instances) are marked with and fully excluded scopes are marked with .

In the text reports, partially excluded scopes are annotated with (x) and fully excluded scopes are annotated with (X).

Examples

Example-1: Module and Instance Partially Excluded

HTML Reports:

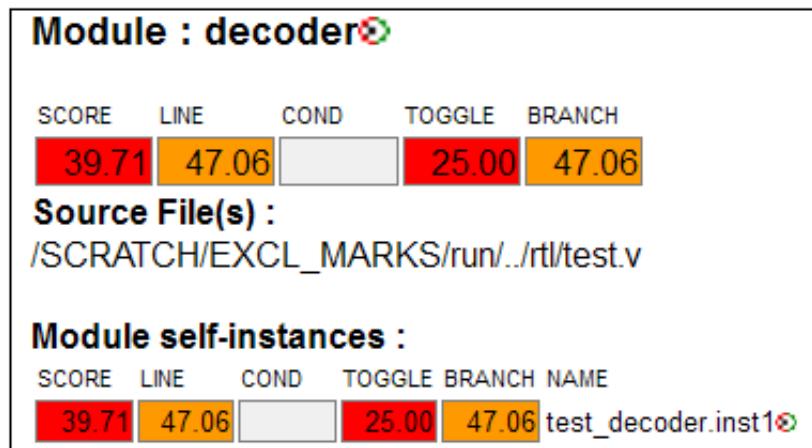
Modlist.html

Total modules in report: 3					
SCORE	LINE	COND	TOGGLE	BRANCH	NAME
39.71	47.06		25.00	47.06	decoder
71.89	97.83		25.00	92.86	test_decoder
100.00			100.00		\$unit:../rtl/test.v:../tb/tb.v

Hierarchy.html

SCORE	LINE	COND	TOGGLE	BRANCH	
53.89	76.25		25.00	60.42	test_decoder
SCORE	LINE	COND	TOGGLE	BRANCH	
39.71	47.06		25.00	47.06	inst1

Modinfo.html



Text Reports:

```
=====
Module : decoder(x)
=====

SCORE LINE COND TOGGLE BRANCH
39.71 47.06 -- 25.00 47.06

Source File(s) :

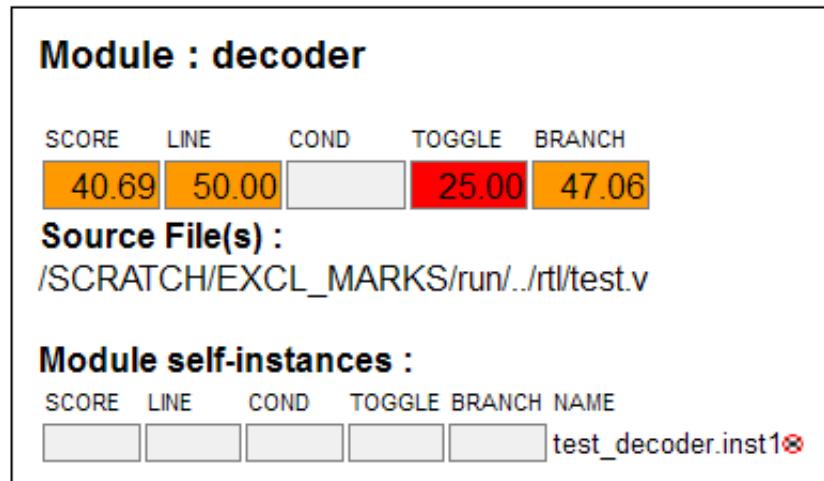
/SCRATCH/EXCL_MARKS/run/..../rtl/test.v

Module self-instances :

SCORE LINE COND TOGGLE BRANCH NAME
39.71 47.06 -- 25.00 47.06 test_decoder.inst1(x)
```

Example-2: Instance Completely Excluded

HTML Reports:



Text Reports:

```
=====
Module : decoder
=====
SCORE LINE COND TOGGLE BRANCH
40.69 50.00 -- 25.00 47.06

Source File(s) :

/SCRATCH/DOC_EXMP/EXCL_MARKS/run/..../rtl/test.v

Module self-instances :

SCORE LINE COND TOGGLE BRANCH NAME
-- -- -- -- -- test_decoder.inst1(X)
```

Exclusion in UCAPI

UCAPI allows you to mark some objects as excluded, which means they are ignored when computing the `covdbCoverable` and `covdbCovered` properties of handles - they are excluded from the computation of the coverage score. This section describes how objects and regions can be excluded.

Hierarchy Exclusion Files

UCAPI supports hierarchy exclusion files, which allow you to specify source regions, design hierarchies, and some individual objects for exclusion or explicit inclusion for coverage. For example, a hierarchy exclusion file with the following content would exclude the sub-hierarchy of the design rooted at the path `top.d1`:

```
-tree top.d1
```

See the *VCS Coverage Metrics User Guide* for the complete description of the hierarchy exclusion file syntax.

Hierarchy exclusion files are loaded using the `covdb_loadmerge` function with an already-loaded design handle:

```
covdb_loadmerge(covdbHierFile, designHdl, filename);
```

Exclusion by Object Handle

Applications may set the `covdbExcludedAtReportTime` flag using the `covdb_set` function:

```
status = covdb_get(obj, region, test, covdbCovStatus);
status |= covdbStatusExcludedAtReportTime;
covdb_set(obj, region, test, covdbCovStatus, status);
```

When an object is marked excluded, it no longer contributes its covdbCoverable (or covdbCovered) count to the count of any containing object or region. For example, if you exclude a statement in a basic block, it lowers the covdbCoverable count of the basic block by 1.

Exclusion bits set using the covdb_set function are not saved to the main UCAPI database or to test files – they are saved to a separate exclusion file. Exclusion is saved using the covdb_save_exclude_file function:

```
covdb_save_exclude_file(testHdl, "myexcludes.el", "w");
```

Either a design handle or a test handle must be given as the first argument. If the first argument is a test handle, then all exclusions of non-test-qualified objects are saved to the file, along with any exclusions of objects in test-qualified regions that appear in that test. If the first argument is a design handle, then only exclusions of non-test-qualified objects are saved.

Exclude files can be saved with mode “w” (write a new file, overwriting if the file already exists) or mode “a” (append the exclusion data onto an existing file).

You can load a previously-saved exclude file using covdb_load_exclude_file:

```
covdb_load_exclude_file(testHdl, "projectexcl.el");
```

As for the covdb_save_exclude_file function, a design handle or a test handle may be given as the first argument. If a test handle is given, exclusions will apply to all non-test-qualified objects as well as to any test-qualified objects in the specified test. If a design handle is given, exclusions will be loaded only for non-test-qualified objects.

The `covdb_unload_exclusion` function removes all `covdbExcludedAtReportTime` bits from all coverable objects in the design.

```
covdb_unload_exclusion(designHdl);
```

Default vs. Strict Exclusion

UCAPI has two exclusion modes - default and strict. In default mode, any object may be excluded. The exclusion mode can be selected using the `covdb_configure` function. To change the mode from default to strict:

```
covdb_configure(designHdl, covdbExcludeMode, "strict");
```

In strict mode, excluding a covered object is illegal. If a container is excluded, then only the uncovered objects in that container will be excluded. If a region, such as a module, is excluded, then only the uncovered objects throughout that region will be excluded.

In strict mode, UCAPI keeps track of the covered objects that were in regions or containers that were excluded - it calls these "attempted exclude" objects. The list of such objects can be saved to a file using the `covdb_save_attempted_file` function. For example:

```
covdb_save_attempted_file(mytestHdl, "attempts.txt", "w");
```

The third argument specifies that the file should be overwritten.

Applications can also use 'a' to append the attempts to the specified file.

Editing the Covergroup Coverage Database

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

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

The `db_edit_filename` is a user-written file that contains `funccov` and `assert` statements in the format described in “[db_edit_file Syntax](#)” .

A `funccov` statement is used to reset or delete covergroup coverage data. An `assert` statement is used to reset or delete assertion coverage data. Examples of those statements are shown in “[db_edit_file Syntax](#)” .

Note:

You cannot edit the code coverage database.

This section contains the following topics:

- “[db_edit_file Syntax](#)”
- “[Resetting Covergroups or Coverpoints](#)”
- “[Removing Covergroups from the Database](#)”
- “[Editing the Assertion Coverage Database](#)”

db_edit_file Syntax

The syntax for editing covergroups in the coverage database is:

```
begin funccov (reset|delete) (module|tree)
  (module-name|instance-name)
  covergroup-name [optional coverpoints or crosses]
end
```

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

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

The syntax for editing assertions in the coverage database is:

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

Resetting Covergroups or Coverpoints

[Example 5-5](#) resets the covergroup named `gc` in the `top.i1` module instance.

Example 5-5

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

[Example 5-6](#) resets the covergroup named `gc` under all instances of the `my_mod` module definition.

Example 5-6

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

[Example 5-7](#) resets the coverpoint or crosspoint named `ra` under the covergroup named `gc` under all instances of the `my_mod` module definition.

Example 5-7

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

Example 5-8 resets the coverpoint or crosspoint named `ra` under the covergroup named `gc` under all instances of `top.i1`.

Example 5-8

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

Removing Covergroups from the Database

Example 5-9 deletes the covergroup named `gc` under all instances of the `my_mod` module definition.

Example 5-9

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

Example 5-10 deletes the covergroup named `gc` under all instances of `top.i1`.

Example 5-10

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

Editing the Assertion Coverage Database

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

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

This section contains the following topics:

- “Resetting the Coverage Scores for Assertions”
- “Removing Assertions from the Coverage Database”

Resetting the Coverage Scores for Assertions

[Example 5-11](#) resets the hit counts for the `mid_first` assertion in the `top.i1` instance.

Example 5-11

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

[Example 5-12](#) resets hit counts of all assertions under the `top.i1` instance.

Example 5-12

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

Removing Assertions from the Coverage Database

[Example 5-13](#) deletes the `mid_first` assertion from the `top.i1` instance.

Example 5-13

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

[Example 5-14](#) deletes all assertions under the `top.i1` instance.

Example 5-14

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

Example 5-15 deletes the `mid_first` assertion from all instances of the `mid` module.

Example 5-15

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

Example 5-16 deletes all assertions from all instances of the `mid` module.

Example 5-16

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

6

Unified Coverage API

This chapter contains the following sections:

- “Overview”
- “Data Model”
- “Predefined Coverage Metrics”
- “Loading a Design”

Overview

The Unified Coverage Application Programming Interface, or UCAPI, provides a single, consistent, and extensible interface to access coverage data generated by Synopsys tools. With this API, you can create custom reports and tools for displaying and analyzing coverage data.

The UCAPI library provides a set of C functions that can be called to get handles to objects in the coverage database. Handles have properties that may be retrieved, such as name, coverage status, and source location information. Handles are connected with 1-to-1 and 1-to-many relations that allow you to traverse all of the data in the database.

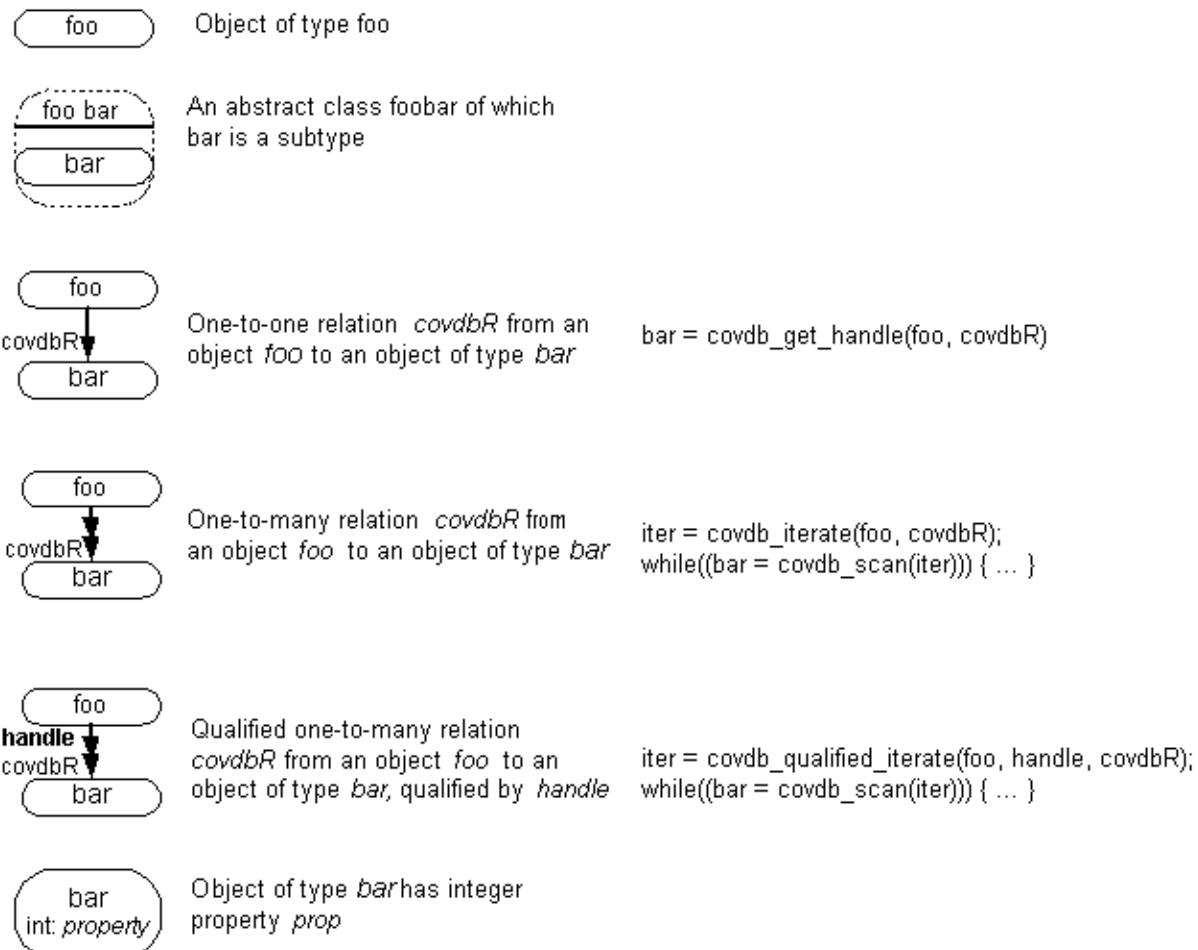
UCAPI coverage data is represented using a small set of simple building blocks that can be used to model any type of coverage data. The objects in the database are the same regardless of the design language (for example, Verilog or VHDL), functional coverage language, or coverage metric type. Different coverage metrics arrange the objects in different ways, but the same basic building blocks are used through UCAPI. This means you can write applications that can understand any type of coverage data without having to follow a set of rules for each different metric.

UCAPI does not provide a complete representation of the design – only what is necessary to present coverage information in proper context. For example, there is no representation of the control structure of code – only the aspects of the design that are monitored by each different metric. Data about the design that is independent of any specific coverage metric is called *unqualified* data.

Inside each region in the design, the coverage data for a given metric is made of UCAPI building blocks. These represent the objects that were monitored for coverage in that region. Since each metric has its own separate collection of such objects, this is called *metric-qualified* data.

Whether or not an object is covered depends on which test or collection of tests you have loaded. For example, a cross bin may have been covered in one simulation run, but not in another. Data that depends on a particular test or set of tests is called *test-qualified data*.

UCAPI uses a series of diagrams to represent object relationships. The following is a summary of the relation types.



Database Contents

A UCAPI database is loaded from one or more directories. The first directory loaded must always contain the unqualified design information - this is typically the data directory created when the design was compiled. Subsequent directories can be full directories (containing unqualified data) or test-only directories (containing only information produced by one or more test runs).

Persistence

UCAPI handles are either volatile or persistent. A persistent handle is guaranteed to remain valid until the application explicitly releases it. Volatile handles are guaranteed to be valid only until the next UCAPI operation is performed.

Some UCAPI functions automatically create persistent handles. But for performance reasons, most UCAPI functions return volatile handles. This results in significantly faster performance since new handles are not allocated for each call - the previous handle is reused “in place”. Applications can get a persistent handle for any volatile handle on demand.

The following rules explain in which conditions handles are persistent or remain valid:

1. Handles returned by `covdb_iterate` and `covdb_qualified_iterate` are persistent and must be freed using `covdb_release_handle` or memory leaks will occur.
2. A handle returned by `covdb_scan` on an iterator is guaranteed to be valid only until the next call of `covdb_scan`, `covdb_iterate`, or `covdb_qualified_iterate`.

3. Handles returned by `covdb_load`, `covdb_merge`, and `covdb_loadmerge` are persistent and should be freed using `covdb_unload` when the application is done with them.
4. Handles returned by `covdb_get_handle` and `covdb_get_qualified_handle` are not persistent and may be invalidated by the next call to either `covdb_get_handle` or `covdb_get_qualified_handle`.
5. All test-qualified handles become invalid when the test is unloaded even if they have been made persistent by the application.
6. All handles associated with a design become invalid when the design is unloaded even if they have been made persistent by the application.
7. Handles passed to an error callback function are persistent and remain valid until released by the application using `covdb_release_handle`.

UCAPI Setup and Compilation Requirements

UCAPI is supported on RedHat, Solaris, and SuSE. Depending upon the OS, different compilation options will need to be passed to the compiler. We recommend that you use a setup script similar to the following:

```
#!/bin/csh

#
# Common env var setup script for UCAPI examples
#

if (-f /etc/SuSE-release) then
  #SuSE
```

```

        set OTHER_FLAGS = -m32
        set ARCH = suse9
else if (-f /etc/redhat-release) then
# REDHAT
        set ARCH = linux
        set OTHER_FLAGS = -m32
else
# Solaris
        set OTHER_FLAGS = "-lsocket -lnsl"
        set ARCH = sparcOS5
endif

#Make sure VCS_HOME is set to a directory
if (! -d $VCS_HOME) then
    echo Error: VCS_HOME does not point to a valid directory
endif

# Make sure the current ARCH subdirectory exists
if (! -d $VCS_HOME/$ARCH) then
    echo Error: this is an $ARCH machine but there is no VCS
installation for $ARCH at $VCS_HOME
endif

# Setup the LD_LIBRARY_PATH
if(! $?LD_LIBRARY_PATH) then
    setenv LD_LIBRARY_PATH ${VCS_HOME}/${ARCH}/lib
else
    setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}: ${VCS_HOME}/
${ARCH}/lib
endif

# A Summary
echo UCAPI Setup Complete
echo "    VCS_HOME          : " $VCS_HOME
echo "    ARCH              : " $ARCH
echo "    LD_LIBRARY_PATH   : " $LD_LIBRARY_PATH

```

As shown in the script, Redhat and SuSE require the `-m32` option to be passed to the C compiler, while Solaris requires the `-lsocket` and `-lnsl` libraries to be included. The script also verifies that

`VCS_HOME` and the OS specific install of VCS is present on the system before attempting to compile. This is required as the UCAPI library is precompiled for different OS's and included within the VCS install.

Once the setup is complete, the c compiler, `cc`, is called with the following arguments:

```
cc -g -I${VCS_HOME}/include -o ucapi_example ucapi_example.c \
\\
$VCS_HOME/$ARCH/lib/libucapi.a -ldl -lm $OTHER_FLAGS
| & tee compile.out
```

Note that `$VCS_HOME/include` will include necessary UCAPI include files and `$VCS_HOME/$ARCH/lib` includes the precompiled UCAPI library file. Any operating system specific flags will be supplied by the setup script.

Enabling Error Reporting

By default, UCAPI will not print error messages to `stdout`. However, this option can be enabled using the `covdb_configure` call:

```
covdb_configure(covdbDisplayErrors, "true");
// setting to "false" disables / default
```

It is recommended that the above line of code be added to the beginning of your UCAPI program.

Dynamic UCAPI Library

You can link UCAPI library dynamically, instead of linking it statically with each version of VCS, and select different versions of UCAPI depending on the version of VCS that created the coverage database. Thus, you need not recompile your application and you can maintain one executable.

Usage Model

To link the UCAPI library dynamically, use the following commands:

```
setenv VCS_HOME <vcs_release_build>
set path = ($VCS_HOME/bin $path)
cc -I${VCS_HOME}/include -L${VCS_HOME}/bin/vcs -location`/
lib -lucapi <CC_FLAGS> <LD_FLAGS> -o <EXE> <C_FILES>
```

Data Model

This section describes the UCAPI objects used to model the contents of a coverage database.

The basic concept of UCAPI is that any coverage metric's objects can be represented using a small set of building blocks. By defining relations between these objects and setting properties (such as name, relative weight, covered/not-covered) on them, the full set of coverage data can be represented, while keeping the interface simple, and the same for all metrics.

UCAPI Object

This section describes properties and relations that apply to all UCAPI handles.

The `covdbType` property can be read from any UCAPI object handle and it gives the UCAPI type of the object.

Any UCAPI object may have string or integer annotations on it, which can be retrieved by using the `covdb_get_annotation`, `covdb_get_integer_annotation`, and `covdb_get_qualified_annotation` functions described in the “Reading Annotations” section of the Coverage Technology Reference Manual.

The supported annotations are described in the relevant sections of this document.

Common Properties

This section describes properties that are supported on many UCAPI handle types.

These properties can be read without a test handle:

- `covdbLineNo` – the line in the source file where the object’s text begins.
- `covdbName` – the name of the object
- `covdbFullName` – the full name (including hierarchical name, if applicable)
- `covdbFileName` – the source file in which this object is defined

The following properties may only be read from metric-qualified objects, but don't require a test handle:

- covdbWeight – the relative weight of this coverable object.
- covdbCoverable – this property represents the number of objects that are coverable. For a simple coverable object such as a toggle sequence, the covdbCoverable value will be one. This value may be greater than one for non-coverable objects, such as covdbContainers, covdbSourceDefinitions, etc. It may also be greater than one for a coverable object handle that represents multiple distinct coverable objects.
- covdbCovGoal – the coverage percentage to be reached for this coverable object. By default, this is 100.0.
- covdbCovCountGoal – the number of times an object must be hit to count it as covered. By default, this is 1.
- covdbAutomatic – non-zero if the object contains only automatically-created sub-objects (for containers, crosses, and sequences)

The following properties require a test handle to specify for which test the data is being queried:

- covdbCovered – if the value of the covdbCovered property is equal to the value of the covdbCoverable property, the handle's contents are fully covered.
- covdbCovCount – the number of times the object has been covered.

Note that covdbCovCount property can only be read if the coverage tool that monitored the data had counting enabled.

The `covdbCovStatus` property may be read from metric-qualified objects with or without a test handle. If it is read without a test handle, then the `covdbStatusCovered` flag will never be set, since only a test can mark an object covered.

The value of `covdbCovStatus` can be any combination of the following:

- `covdbStatusCovered` – covered
- `covdbStatusUnreachable` – can never be covered, as when a formal tool has proven it is impossible to cover
- `covdbStatusIllegal` – should be considered an error if the object was covered
- `covdbStatusExcludedAtCompileTime` – the object was marked excluded at compile time
- `covdbStatusExcludedAtReportTime` – the object was marked excluded at report time
- `covdbStatusExcluded` – the object was marked excluded either at report time or compile time – can be checked by applications when it doesn't matter when the object was marked excluded

Coverable objects with `covdbStatusExcluded` set (or `covdbStatusExcludedAtReportTime` or `covdbStatusExcludedAtCompileTime`) are ignored when computing coverage scores – the `covdbCoverable` property will not include them in its count, for example. However, they may still be useful information. For example, some coverage tools will collect expression coverage information for debugging or verification analysis purposes that is not counted against a project's coverage goals. When coverable objects inside a container (or cross, or

sequence) are marked `covdbStatusExcluded`, they will be excluded from the `covdbCovered` and `covdbCoverable` values for that container (or cross or sequence).

Note that, while all of these properties may be retrieved from any coverable object handle, all objects may not have useful information. For example, a container object may have no associated file name or line number, but that information may be retrieved from the objects it contains. Whether or not this information is present for a given object depends on the metric and the coverage tool that created that object.

Coverable Objects

Coverable objects are entities in the database that can be covered or not covered. This small set of object types is sufficient to model any type of coverage. All coverable objects are metric-qualified. That means that each coverable object is associated with exactly one metric.

Coverable objects are assigned names that are unique within the region or container in which they are defined. Named objects, such as signal bits, may contain a list of lower-level objects - for example, a signal bit "x[1]" can have toggle objects representing the toggle from 0 to 1 and the toggle from 1 to 0. In this example, the names of those objects in UCAPI are "0 > 1" and "1 > 0".

Variant names are assigned by UCAPI using the name of the module plus a parenthesized list of its parameter/generic values. For example, if a module M has two variants for a given metric, its variant names might be "M(p = 1)" and another variant "M(p = 2)". The name of the unqualified module is "M". See the section titled "[Source Definition and Source Instance](#)" for more information on variants.

In the diagrams for coverable objects in this section, only the special properties and relations for each type are shown below.

Block

A block represents an atomic coverable object. The `covdbType` property for a block object is `covdbBlock`. Block objects have no additional relations on them. Depending on the metric, they may have `covdbFileName`, `covdbLineNo`, and `covdbName` properties.



`covdbBlock`

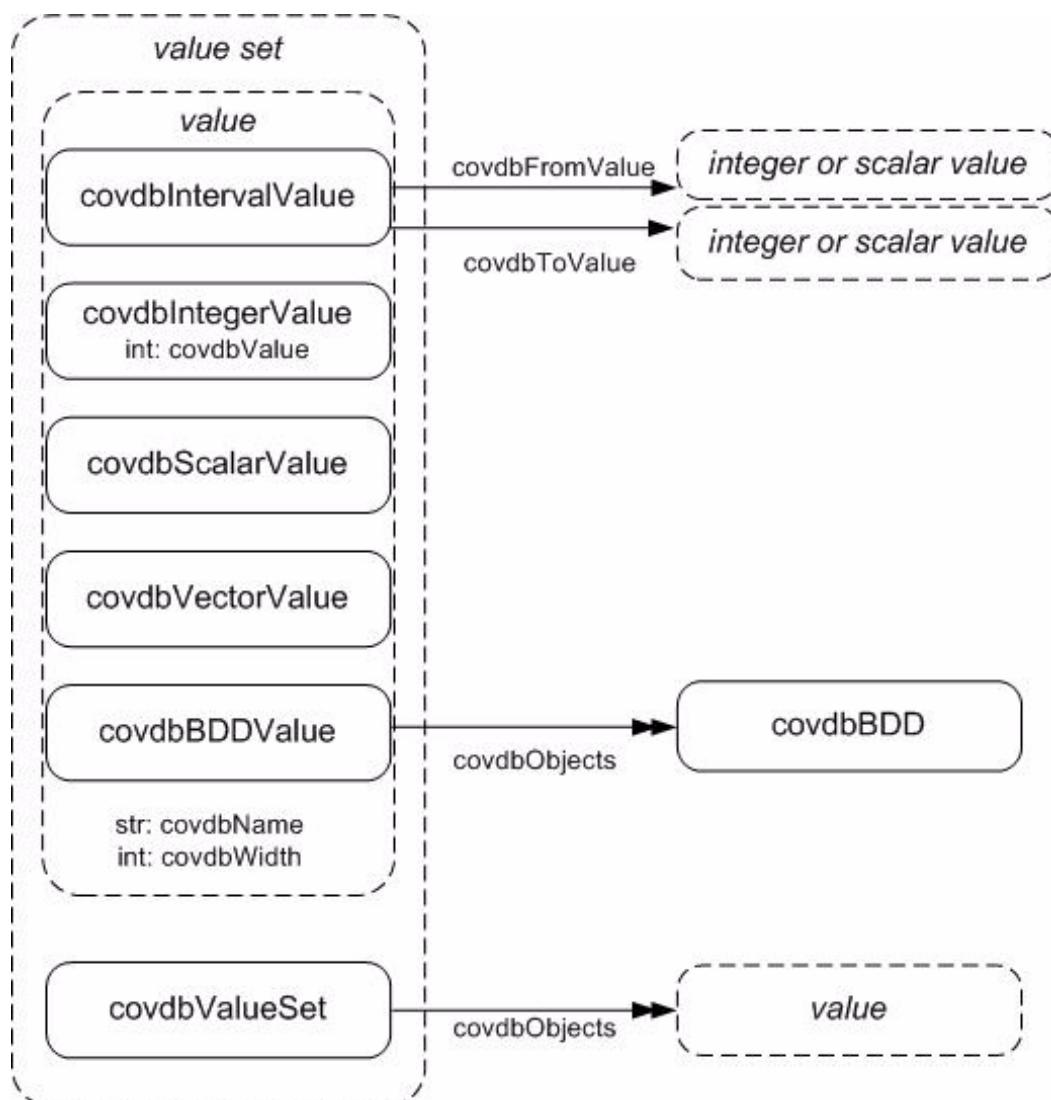
Value Set

A value set represents a set of values taken on by an expression. The value may be represented as an exact value, an interval (range) of values, a Binary-Decision Diagram (BDD), or a set of values that is any combination of these values. These coverable object handle types are collectively called "value sets".

The `covdbType` property of a value set object is one of `covdbIntervalValue`, `covdbIntegerValue`, `covdbScalarValue`, `covdbVectorValue`, `covdbBDDValue`, or `covdbValueSet`. The following diagram shows their structures. This section explains how the properties and value(s) for each type of handle are retrieved.

All value set objects support these properties:

- covdbName - a string representation of the value. This may be simply the value as a string, or it could be a name for the value, such as the name of an enumerated type.
- covdbWidth - the bit-width of the expression.



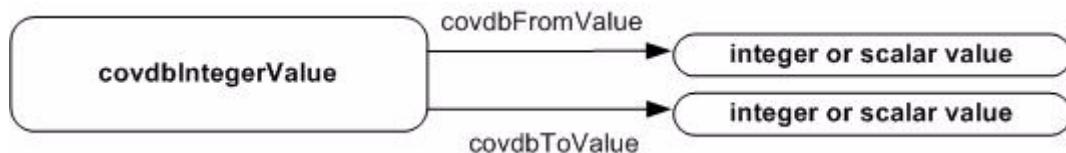
Integer and Scalar values

The value of an object of type `covdbIntegerValue` or `covdbScalarValue` is read using the `covdbValue` property. For integers, the value is the integer; for `covdbScalarValues`, the value is one of the enumerated types `covdbScalar0`, `covdbScalar1`, or `covdbScalarX` (see section 6.5).

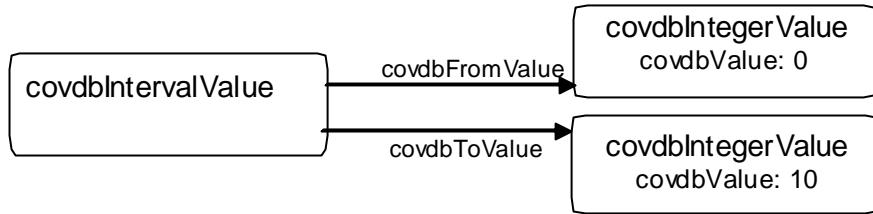
For example:

```
covdbTypesT objty = covdb_get(objHdl, regHdl, NULL,  
                               covdbType);  
if (covdbIntegerValue == objty) {  
    int val = covdb_get(objHdl, regHdl, NULL, covdbValue);  
} else if (covdbScalarValue == objty) {  
    covdbScalarValueT val =  
        covdb_get(objHdl, regHdl, NULL, covdbValue);  
}
```

Interval values



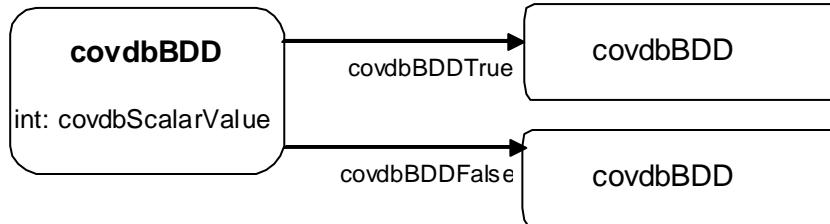
A handle of type `covdbIntervalValue` contains two `covdbIntegerValue` handles - one is the "from" value and the other is the "to" value of the interval. For example, the interval [0, 10] would be represented as shown follows:



BDD values

The `covdbBDD` type is used to represent BDD values - ordered decision diagrams. Whether a BDD object is a terminal or not can be determined from the `covdbScalarValue` property. This property will return:

`covdbValue0`: for 0 valued terminals
`covdbValue1`: for 1 valued terminals
`covdbValueX`: for non-terminal objects



BDD values are currently not supported.

Vector values

An object of type `covdbVectorValue` has two properties:

- `covdbSigned` - non-zero if the vector value is a signed value.

- covdbTwoState - non-zero if the vector value is in two-state representation. If zero, the value is in four-state representation.
- covdbWidth - the width of the value in bits.

If the vector value's covdbTwoState property is non-zero, the values will be represented as an array of unsigned ints. The value is retrieved with the function `covdb_get_vec_2state_value`.

If covdbTwoState is zero, the values will be represented as an array of `covdbVec32ValueT` structs. Each struct has a 32-bit control word and a 32-bit data word. Each bit of the control word and each corresponding bit of the data word represents a single scalar value (0, 1, X, or Z). The value for each combination is shown follows:

Table 6-1

control	data	value
0	0	0
0	1	1
1	0	Z
1	1	X

For example, the following is the declaration of `covdbVec32ValueT` from the `covdb_user.h` header file:

```
typedef struct covdbVec32Value_s {
    unsigned int c;
    unsigned int d;
} covdbVec32ValueT;
```

Also, the following is an example of code that reads and prints out `covdbVectorValueT` handles:

```
if (covdbVectorValue == objty) {
    int signp = covdb_get(objHdl, regHdl, NULL, covdbSigned);
    int width = covdb_get(objHdl, regHdl, NULL, covdbWidth);
```

```

    if (covdb_get(objHdl, regHdl, NULL, covdbTwoState)) {
        unsigned *vals =
            covdb_get_vec_2state_value(objHdl, regHdl);

        for(i = 0; i < width; ) {
            unsigned wd = vals[i / 32];
            for(j = 0; j < 32 && i < width; i++, j++) {
                printf("%d", (wd >> j) & 0x1);
            }
        }
        printf("\n");
    } else {
        covdb4stateValT *vals =
            covdb_get_vec_4state_value(objHdl, regHdl);

        for(i = 0; i < width; ) {
            unsigned wd = vals[i / 32];
            for(j = 0; j < 32 && i < width; i++, j++) {
                int c = vals[i].c;
                int d = vals[i].d;
                char rep;

                if (!c && !d) rep = '0';
                else if (!c && d) rep = '1';
                else if (!d) rep = 'Z';
                else rep = 'X';

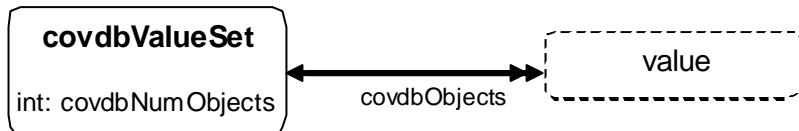
                printf("%c", rep);
            }
        }
        printf("\n");
    }
}

```

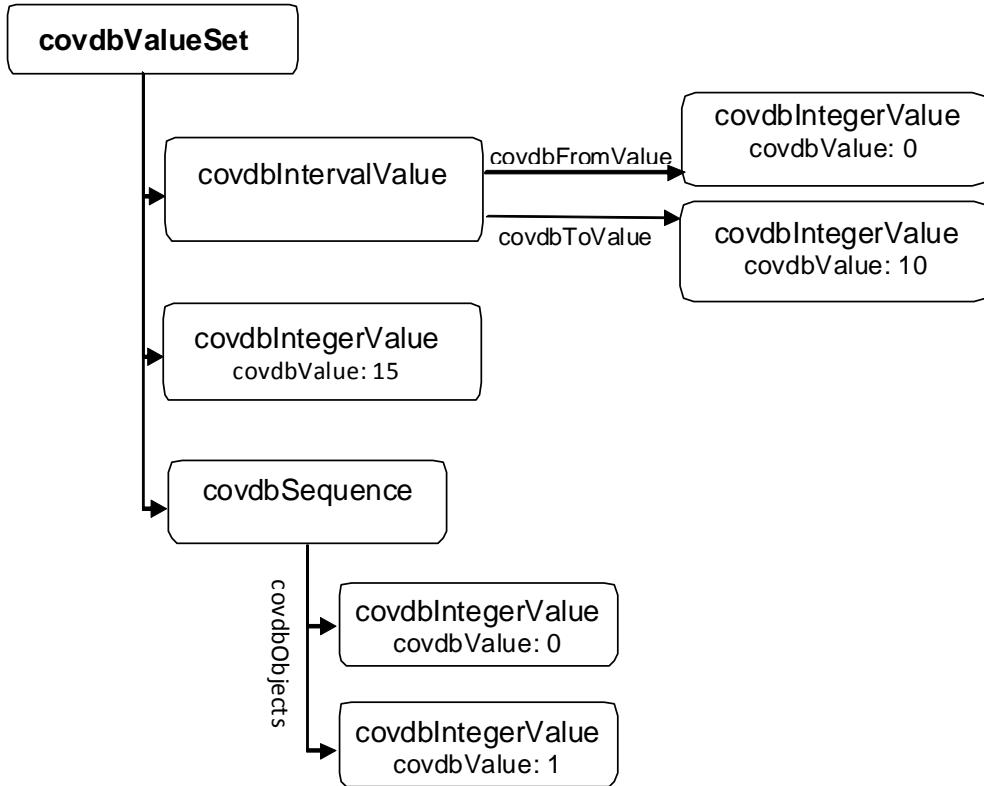
Value Set

A `covdbValueSet` handle is a collection of value sets or sequences. It represents a given expression taking on one of the values in the value sets or going through one of the sequences of values specified by the sequence objects.

A `covdbValueSet` handle is itself a coverable object. It is either covered or not covered. The objects inside it are there only to describe its structure. For example, the `covdbCovCount` of the `covdbValueSet` handle tells how many times any one of the values or sequences occurred - applications cannot query each object inside the bin to get their individual `covdbCovCount` values.



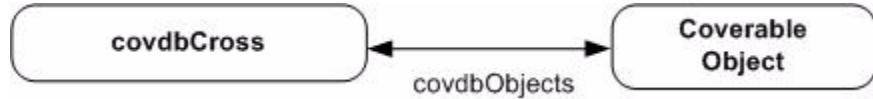
For example, a coverable object that represented an expression taking a value between 0 and 10 (inclusive), the value 15, or the value of the expression changing from 0 to 1 would be represented as follows:



Sequence

An object of type `covdbSequence` contains an ordered collection of other coverable objects and represents the *sequential* coverage of each member object. For example, a sequence of value sets might represent a signal taking on the specified sequence of values.

Note that a `covdbSequence` object itself is either covered or not covered. The coverage statuses of its contents do not contribute to its coverage status, because the sequence itself is an atomic coverable object.

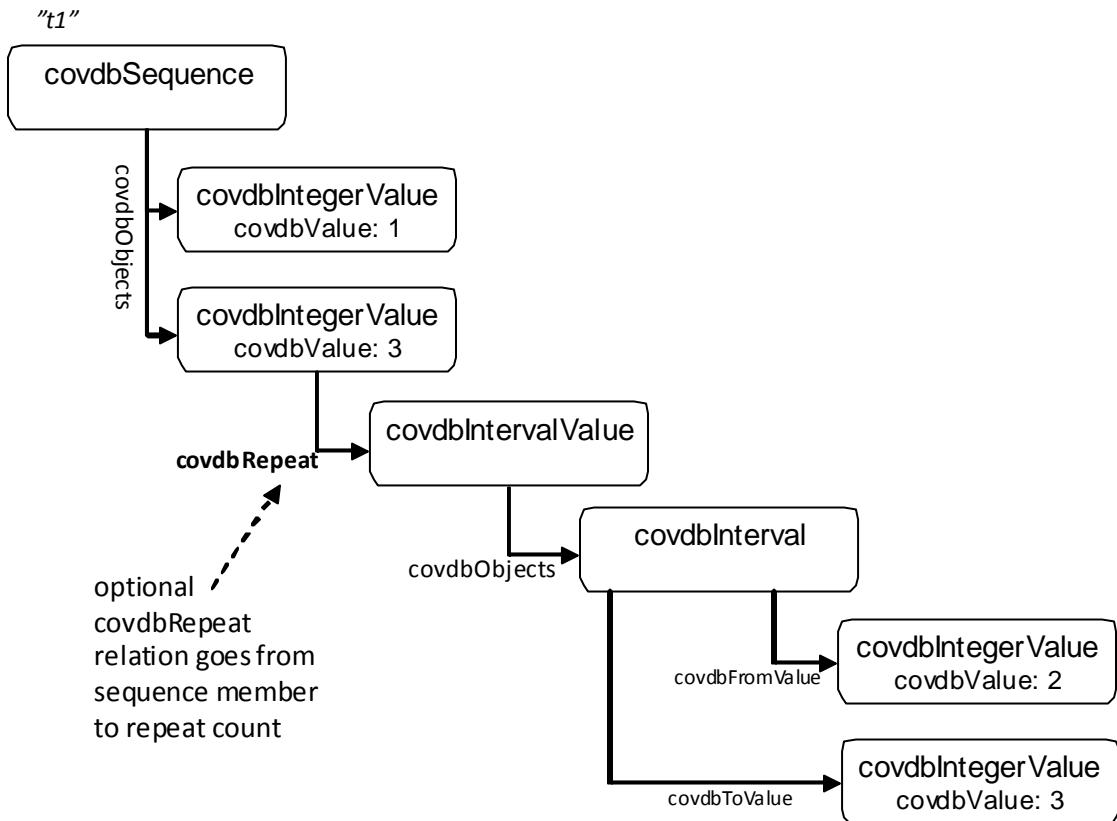


A coverable object in a covdbSequence's objects list has an optional repeat count. This indicates that the object is repeated the specified number of times in the sequence. The repeat count may be a single integer or it may be a set of values, indicating the repetition may be any of the specified numbers.

For example, if a testbench coverage transition bin is defined as:

```
coverpoint a
    bin t1 : 1 -> 3 * (2..3)
```

This indicates that the value of a should transition from '1' to '3', then from '3' to '3' again 1 or 2 additional times. In UCAPI, this is represented using the covdbRepeat relation, as follows:



When a value set is used as a repeat, it will have a value for covdbRepeatType that is one of:

- covdbGoto
- covdbConsecutive
- covdbNonconsecutive

When the value set is not a repeat value, the value of covdbRepeatType will be -1.

Cross

An object of type `covdbCross` has the same structure as a sequence, but the objects in the `covdbObjects` iterator are in no particular order. A `covdbCross` object represents the *simultaneous* coverage of its member objects.

Like a `covdbSequence` object, a `covdbCross` object is either covered or not covered. The coverage statuses of its `covdbObjects` do not contribute to its coverage status, because the sequence itself is the coverage event.



Stable Names for Coverable Objects

Code coverage automatically finds coverable objects in a design, and monitors whether they are covered in simulation runs. These objects are identified in an ad hoc way, depending on the type of object. For example, signals in toggle coverage and machines in finite-state machine coverage can be identified by signal names. Statements, branches, and conditions are identified by line numbers. Conditions are identified by a sequence number within a given module (for example, 1st, 2nd, and 3rd). A path is named by the sequence of basic blocks through which it flows.

Following are the two categories of coverable objects:

Coverable objects with stable names — An object name is stable, if it is not sensitive to unrelated changes. For example, the name of a signal for toggle or FSM coverage is stable. The name of the coverable object can be changed by changing the signal name itself.

Coverable objects with unstable names — Object names that are sensitive to unrelated changes in the design are unstable. For example, any coverable object whose name depends on a source line number has an unstable name, since even adding a comment can change the name.

The stable names for coverable objects feature allows you to specify stable names for coverable objects, whose names are unstable by default.

Important:The support for stable names for coverable objects is not currently available for VHDL.

Naming the Next Statement in a Module

A new pragma is introduced to allow you to name the next statement in a module. For example:

```
//coverage name a1  
if (a || (b && c))
```

In the example, the name `a1` is assigned to the statement `if (a || (b && c))`. Any conditions found in the statement can be addressed starting with `a1`, rather than with a sequence number in the module or source line number. Pragma should be just inserted immediately before the condition.

You can attach a tag to a statement, but that statement may contain multiple conditions. Each condition may also contain subconditions, and each condition or subcondition will have a list of vectors.

Using Stable Names for Coverable Objects

Assigning stable names for coverable objects is useful for excluding continuous conditions and vectors specified in el files for condition coverage.

This is accomplished by providing tags in the source and generating an el file using DVE for condition coverage metric. This metric will contain the stable names and excluded vectors in the generated file.

Specifying Tags in the Source

This section explains how tags are specified in the source, and how coverable object names are derived from those tags.

Condition and Subcondition

Consider the following example:

```
//coverage name a1
if (a || (b && c)) {
    ...
}
```

The top-level condition is `a || (b && c)`. It is broken down by code coverage into two terms. Its name is `a1`:

```
a || (b && c) Condition a1
1      --2---
```

There is also a subcondition with two terms, named a1.1:

```
b && c      Condition a1.1  
1      2
```

Multiple Conditions in a Statement

Statements may contain more than one independent condition. For example:

```
//coverage name mycond  
x <= (a || (b && c)) ? ( (d && (e || f)) ? 1'b1 : 1'b0 ) : 1'b1;
```

There are two conditions, (a || (b && c)) and (d && (e || f)) in the example. Both the conditions are part of the same statement, but there is only one name pragma.

VCS will assign names based on the name. The first condition is named as mycond:

```
a || (b && c)      Condition mycond  
1      --2---  
b && c              Condition mycond.1  
1      2
```

The second condition is named as mycond.2:

```
d && (e || f)      Condition mycond.2  
1      --2---  
e || f              Condition mycond.3  
1      2
```

If there were additional conditions, then they would be named as mycond.<serial number of condition on the line>. For example, mycond.2, mycond.3, and so on.

Note: This additional naming only holds for one statement.

Multiple Statements on a Line

If a single source line contains multiple statements, then only the first statement will be named by a preceding pragma. For example:

```
//coverage name a2
x <= (a || b) ? 1'b0 : 1'b1;  y = (c && d) ? 1'b0 : 1'b1;
a || b    Condition a2
1      2
```

There is no name for the second condition `c && d`, since it occurs in another statement that has no pragma preceding it. If you want to assign a name to `c && d`, then you should put the statement on a separate line:

```
//coverage name a2
x <= (a || b) ? 1'b0 : 1'b1;
//coverage name a3
y = (c && d) ? 1'b0 : 1'b1;
a || b    Condition a2
1      2
c && d    Condition a3
1      2
```

Names in the Generated Reports

Currently in the URG reports, condition ID numbers are shown if the `-cond_ids` option is specified. These appear inlined with the condition coverage reports. For example, in URG, you can view the following:

```

LINE    12
CONDITION_ID   1
STATEMENT      if ((a || (b && c)))
               1      --2---
EXPRESSION     -1-      -2-
                  0      0 | Not Covered
                  0      1 | Not Covered
                  1      0 | Not Covered

```

If the condition was named with a pragma in the source, then the name will appear instead:

```

LINE    12
CONDITION_ID   a1
STATEMENT      if ((a || (b && c)))
               1      --2---
EXPRESSION     -1-      -2-
                  0      0 | Not Covered
                  0      1 | Not Covered
                  1      0 | Not Covered

```

Subconditions will be named as described above. For example, the subcondition b && c is labeled as a1.1:

```

LINE    12
CONDITION_ID   a1.1
STATEMENT      if ((a || (b && c)))
               1      2
EXPRESSION     -1-      -2-
                  1      1 | Not Covered
                  0      1 | Not Covered
                  1      0 | Not Covered

```

El File for Condition Coverage

If you use DVE to generate the el file, then the dumped file contains the stable name for tagged conditions instead of the Integer IDs. Expression IDs for subexpression in a new database may differ with respect to the old database.

While applying the el file, if the module or instance checksum has changed, then only the conditions with given stable names are considered for exclusion. The remaining non-stable IDs are ignored.

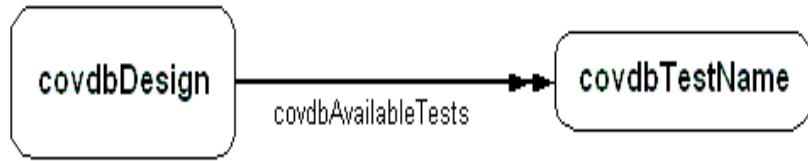
Design Objects

Design objects are used to give context for coverable objects. Design object handle types include covdbDesign, covdbTest, covdbMetric, covdbTestName, covdbSourceDefinition, and covdbSourceInstance. They do not have coverage state themselves. They can be considered to be the framework on which the coverable objects are distributed. The covdbName property can be read from any design object handle.



Design and Test Name

A covdbDesign handle indicates a specific design in the coverage database. A design contains a list of test names for which coverage data exists.



The `covdbTestHandle` handles you get from `covdbAvailableTests` are not the same as `covdbTest` handles – applications cannot use them to get coverage data. But applications can use their `covdbName` properties to load real `covdbTest` handles.

Tests and Metrics

A `covdbTest` handle is a test whose data has been loaded into the UCAPI interface. A test is the result of a single execution of a tool, or the merged results of two or more executions.

Test handles are used to get test-qualified handles and data as described in the introduction to this section. Handles to `covdbTest` objects are accessed using the `covdb_load`, `covdb_merge`, and `covdb_loadmerge` functions.

Metric handles have the `covdbType covdbMetric`. The only property that can be read from a `covdbMetric` handle is its name, using `covdbName`.

The 1-to-many relation `covdbMetrics` may be iterated from a `covdbTest` handle to get the list of metrics that have coverage data in that test.

After tests are loaded, applications can iterate over the list of loaded tests from a design handle using the `covdbLoadedTests` relation:



Source Definition and Source Instance

A `covdbSourceDefinition` handle represents a non-instantiated design region, such as a Verilog module or a testbench coverage group. A `covdbSourceInstance` represents a specific instance of a `covdbSourceDefinition`.

Source regions represent the declarative regions from the design or testbench - modules, entity-architectures, covergroups, packages - and instances of those regions. Region handles can be unqualified, metric-qualified, or test-qualified.

Unqualified regions are the skeleton of the design. You cannot see any coverable objects in them because the list and shape of coverable objects depend on a particular metric.

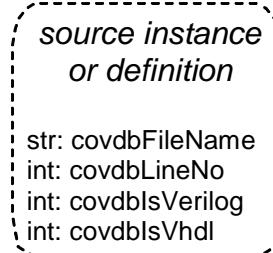
Metric-qualified region handles contain the coverable objects for one specific metric. For example, a condition-qualified handle to a module M will contain the coverable objects for conditions and vectors found and monitored in M.

Test-qualified region handles are used for regions that only exist within a specific test. For example, covergroup handles are always test-qualified, since covergroups are used on a per-test basis - they don't necessarily exist in every test.

The following properties may be read from any source definition or instance:

- covdbFileName – the source file name of this definition or instance.
- covdbLineNo – the line number where this definition or instance is defined in its source file.
- covdbIsVerilog – non-zero if the definition or instance was defined in Verilog.
- covdbIsVhdl – non-zero if the definition or instance was defined in VHDL.

Source definitions may represent modules, entity-architectures, or testbench coverage groups. Source instances may be instances of any of these objects.

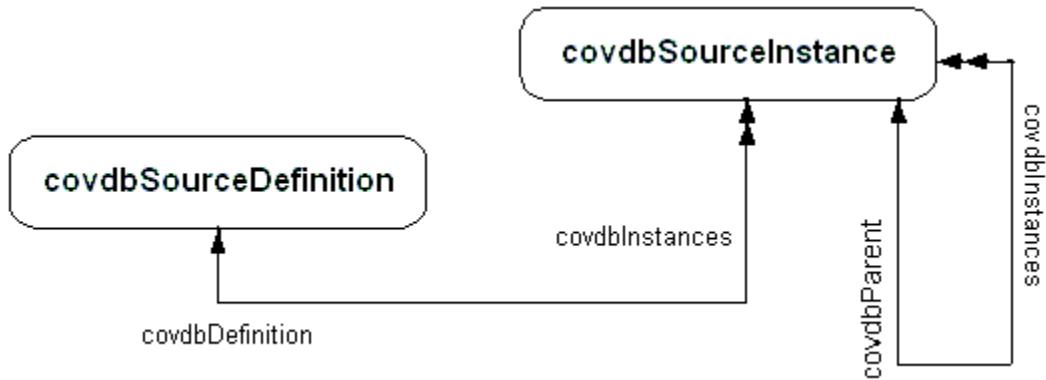


Source definitions have a 1-to-many relation `covdbInstances` to get an iterator of all instances of that definition. Similarly, source instances have a 1-to-1 relation `covdbDefinition` to get the corresponding definition of an instance. Source instances also have a `covdbInstances` relation from them; each instance in this list is a child instance in the design hierarchy. The `covdbParent` relation is the instance's parent in the design hierarchy.

Properties such as `covdbCovered` and `covdbCoverable` may be read directly from metric-qualified source region handles (`covdbCovered` requires that a test handle be given) to compute the coverage score for that metric. Any property that can be read from an unqualified source definition or instance (such as `covdbFileName`) can also be read from a metric-qualified source definition or instance handle.

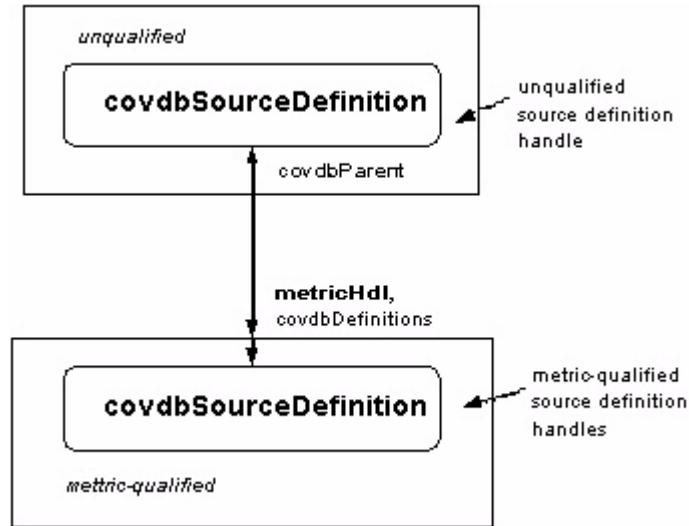
The `covdbDefinitions` and `covdbInstances` 1-to-many relations may be used from a design handle to iterate over all definitions or over the top-level instances in the design, respectively. When these are qualified with a metric, the list of metric-qualified definitions and top-level instances for that specified metric are obtained.

The figures below examine different subsets of the relations between different design region handles. First, is the structure of the unqualified design. Without using any metric handles, an application can walk through the base design – all definitions and all instances of those definitions. Note that the `covdbInstances` relation from a source definition gives its list of self-instances, whereas the `covdbInstances` relation from a source instance gives its instance children in the design hierarchy.

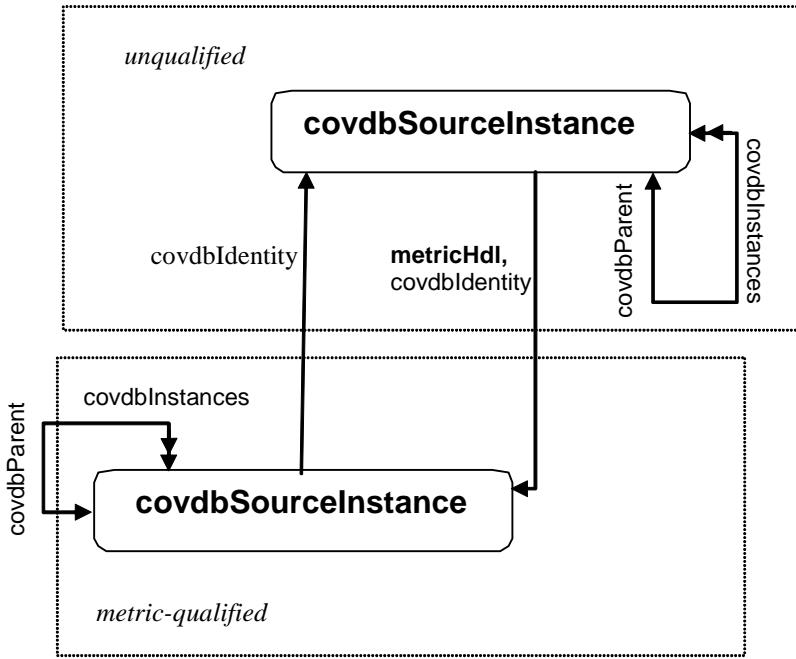


To get coverage information about a given metric in a given region, an application must get a metric-qualified handle for that region. For example, to get the list of FSMs in a module M, applications first get the FSM metric-qualified handle to M.

There may be more than one FSM-qualified handle for M in a design, due to parameter/generic values. In other words, the module M may have been *split* by the FSM metric. Therefore, there is not necessarily only a single FSM-qualified handle for M, and applications use the metric-qualified 1-to-many relation `covdbDefinitions` to walk through those handles, as shown in the following figure:



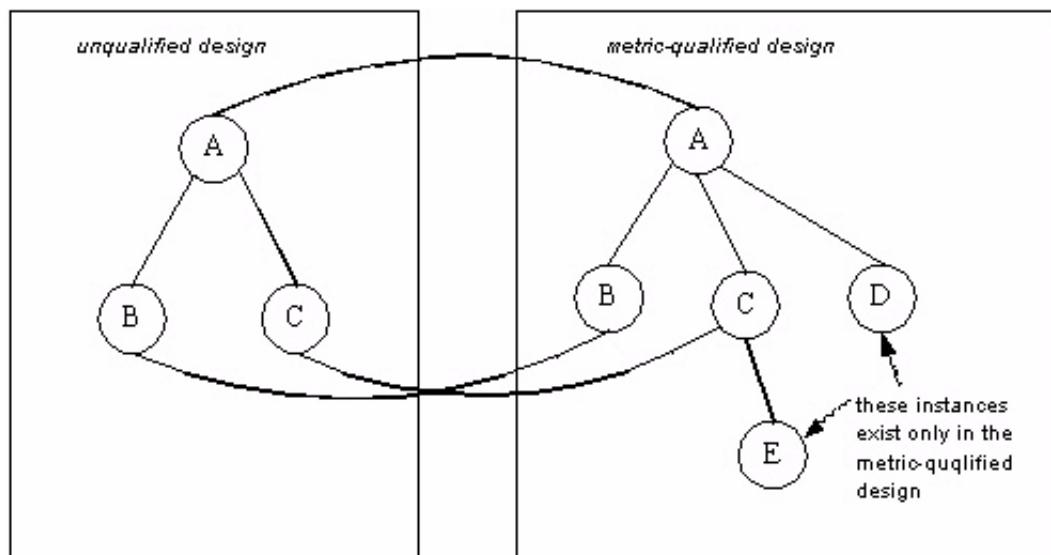
Similarly, unqualified `covdbSourceInstance` handles have metric-qualified versions. However, since each instance is unique in the design – it does not potentially represent multiple versions, like `covdbSourceDefinitions` – there is always only one metric-qualified source instance for each unqualified source instance handle for a given metric. The metric-qualified source instance handle is accessed using the metric-qualified 1-to-1 relation `covdbIdentity`, and the unqualified instance handle is accessed from the metric-qualified instance handle using `covdbIdentity` with no qualifier:



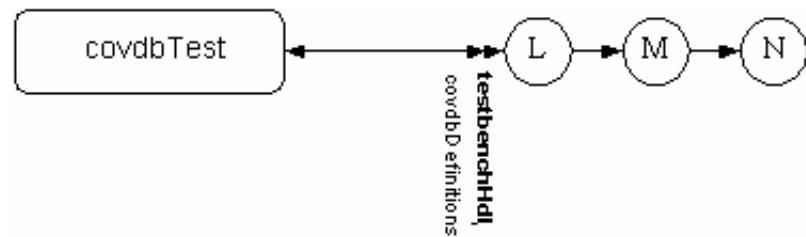
There may be metric-qualified source instances or definitions with no corresponding unqualified handle. For example, some instances exist only for the assertions metric (bound-in OVA units, for example). An application traversing only over the unqualified design will not see these source regions. To distinguish them from metric-qualified handles of unqualified source regions, we call these regions *metric-only* regions.

If a metric-only region is attached to the unqualified design, it will be visible only when iterating over the `covdbInstances` children of the appropriate metric-qualified instance parent.

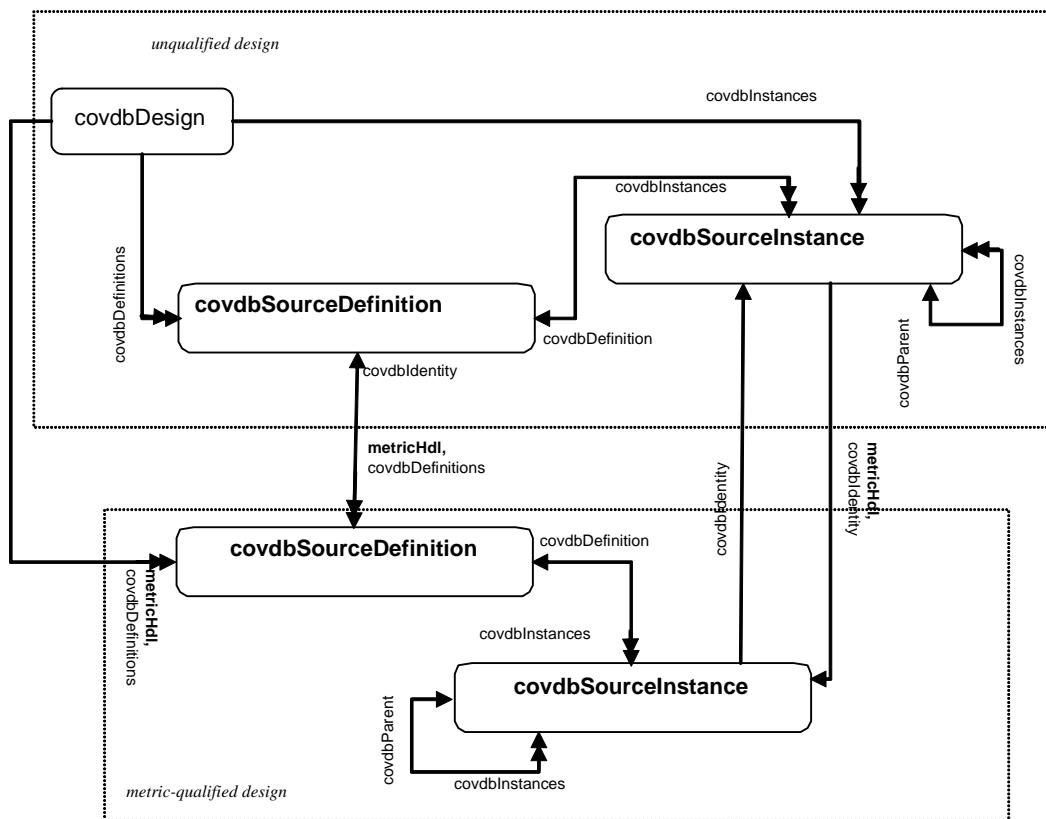
For example, the following figure shows the unqualified and metric qualified design hierarchies for an example design. The source instances D and E are only visible in the metric-qualified design and may not be directly accessed from any unqualified source handle.



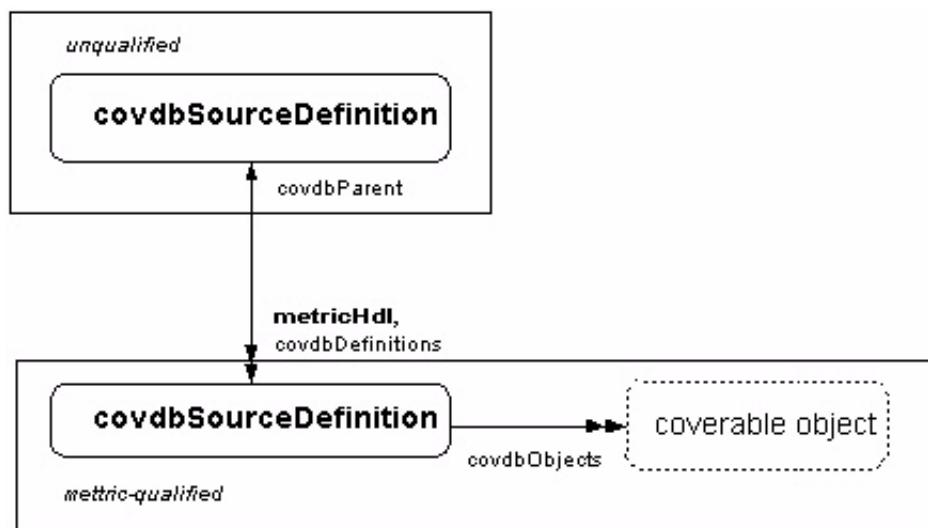
If a metric-only region is separate from the design (such as a testbench coverage covergroup declared outside of any module), applications must use the metric-qualified 1-to-many relations `covdbDefinitions` or `covdbInstances` from a test or design handle – this will give access to the complete list of definitions or the list of metric-qualified top-level instances. For example, in the following figure, L, M, and N are Vera covergroups, accessed from a test handle.



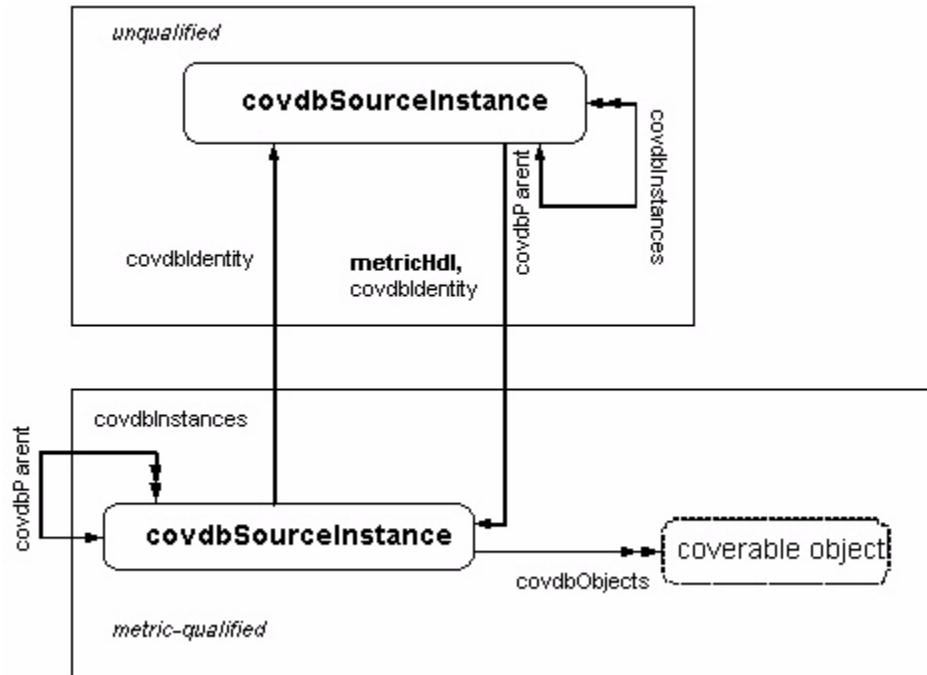
The following figure shows the complete set of relations used to traverse the design.



There is no metric-qualified covdbObjects list off of source definitions or instances. All coverable objects are accessed from the metric-qualified source definitions and instances. For example, to get to the coverable objects list for a source definition:



Similarly, to get to the metric-qualified objects from a covdbSourceInstance handle:



The `covdbDeepCoverable` and `covdbDeepCovered` properties apply only to metric-qualified `SourceInstance` handles, and return the coverable (covered) counts for the source instance and its entire design subtree.

Test-qualified Source Regions

Some metrics create test-qualified source regions that only exist within a given test. For example, testbench coverage groups do not exist in the unqualified design or even in the metric-qualified design - they only exist in the test-qualified data.

Test-qualified regions are accessed in two ways:

- Directly from a test handle using the metric handle as a qualifier

- From within metric-qualified source definitions and instances in the main design

Like unqualified source definitions, test-qualified source definitions may have multiple versions, due to variations between instances of the groups. These versions are accessed with the test-qualified `covdbDefinitions` relation. As for HDL source definitions, coverable objects are only accessed from the versions of definitions, not from the master definition.

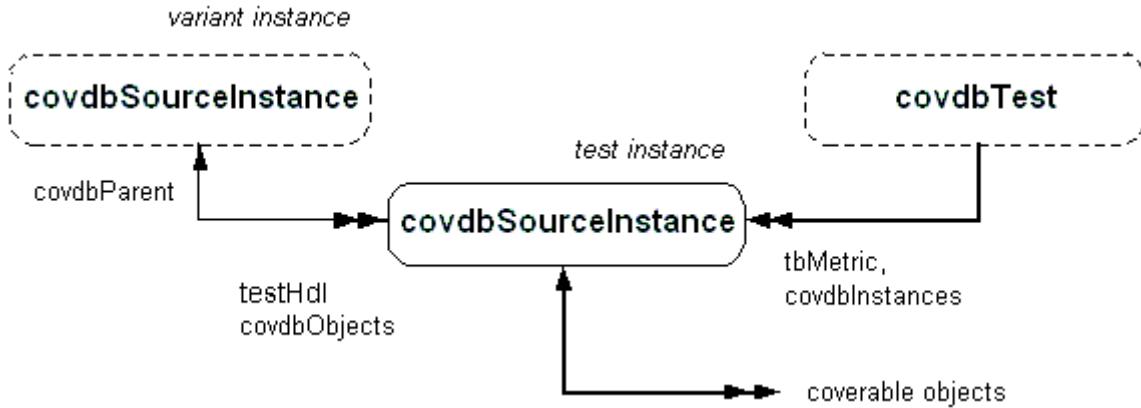
The following figure shows how test-qualified definitions are accessed. Access is either from a testbench-qualified module definition to the list of its test-qualified definitions, or from a test handle giving the list of all test-qualified definitions in the design.

Test-qualified instances may occur inside any HDL module instance, or entirely outside the design hierarchy. They may occur inside an HDL module instance even if they are not defined within the corresponding HDL module¹.

Test-qualified instances that are inside HDL module instances are accessed by the test-qualified `covdbObjects` relation from the metric-qualified HDL module instance handle.

Test-qualified instances that are not within the HDL design hierarchy are accessed from a `covdbTest` handle, using the metric-qualified `covdbInstances` relation. The following figure shows how test-qualified instances are accessed in UCAPI.

1. Such as an SVTB covergroup defined in one module but instanced by another module - UCAPI supports a covergroup defined in \$root and instanced in the design, for example.



Covergroup definitions and instances have the same `covdbInstances/covdbParent` relationship as HDL definitions and instances.

Other Objects

Container

Objects of type `covdbContainer` are abstractions that represent a collection of coverable objects that logically belong together. Containers may have two different lists in them:

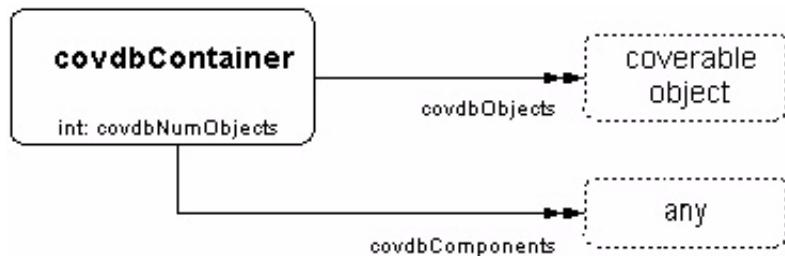
- `covdbObjects` – a list of coverable objects or more `covdbContainers`. The covered/coverable numbers for the container is the sum of the covered/coverable numbers for all of the contained objects.
- `covdbComponents` – a list of objects making up the structure of the container. For example, a container representing a cross declaration will have the list of coverpoints being crossed in its `covdbComponents` list.

Containers whose objects are `covdbCrosses` or `covdbSequences` may also have a `covdbComponents` list. If a container has a `covdbComponents` list, it means that:

- Each cross or sequence in the container's `covdbObjects` list has the same number of objects in it.
- The container will have the same number of items in its `covdbComponents` list as are in each item in its `covdbObjects` list.
- Each item in each cross or sequence will correspond to the same numbered item in the `covdbComponents` list.

The objects in the `covdbComponents` list do not contribute to the coverage values for the container – they are provided only for reference.

Containers are themselves not coverable objects – they contain coverable objects in their `covdbObjects` lists. This is distinct from Sequences and Crosses, each of which represent a single coverable object.



In general, objects in containers are in source declaration order. This is tool-dependent and metric-dependent.

Predefined Coverage Metrics

This section describes how UCAPI models each of the predefined coverage metrics. Note that while the various metrics described below use terms such as “basic block”, “condition” and “signal,” these are not UCAPI object types, they are just describing what the UCAPI types are being used to model. *All object types are defined in “Data Model” on page 8*— there are no special object types for any metric.

Line Coverage

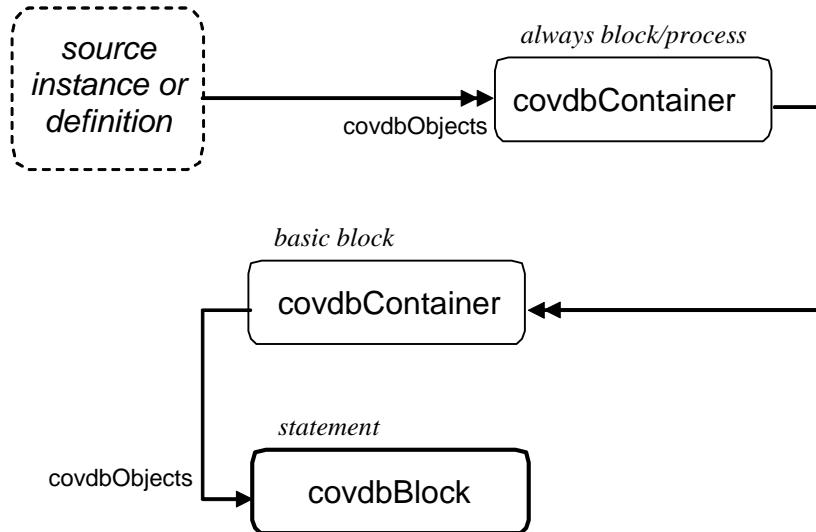
Line or Statement coverage tracks which statements in the design code were executed during simulation.

Statement coverage is organized in two levels, with always blocks/processes at the top level. The type of an always block or process is covdbContainer.

Within each always block/process container, statements are organized into *basic blocks*, which are straight-line sections of code. Basic blocks are also covdbContainer objects.

Within the basic block containers, each statement is modeled as a covdbBlock object.

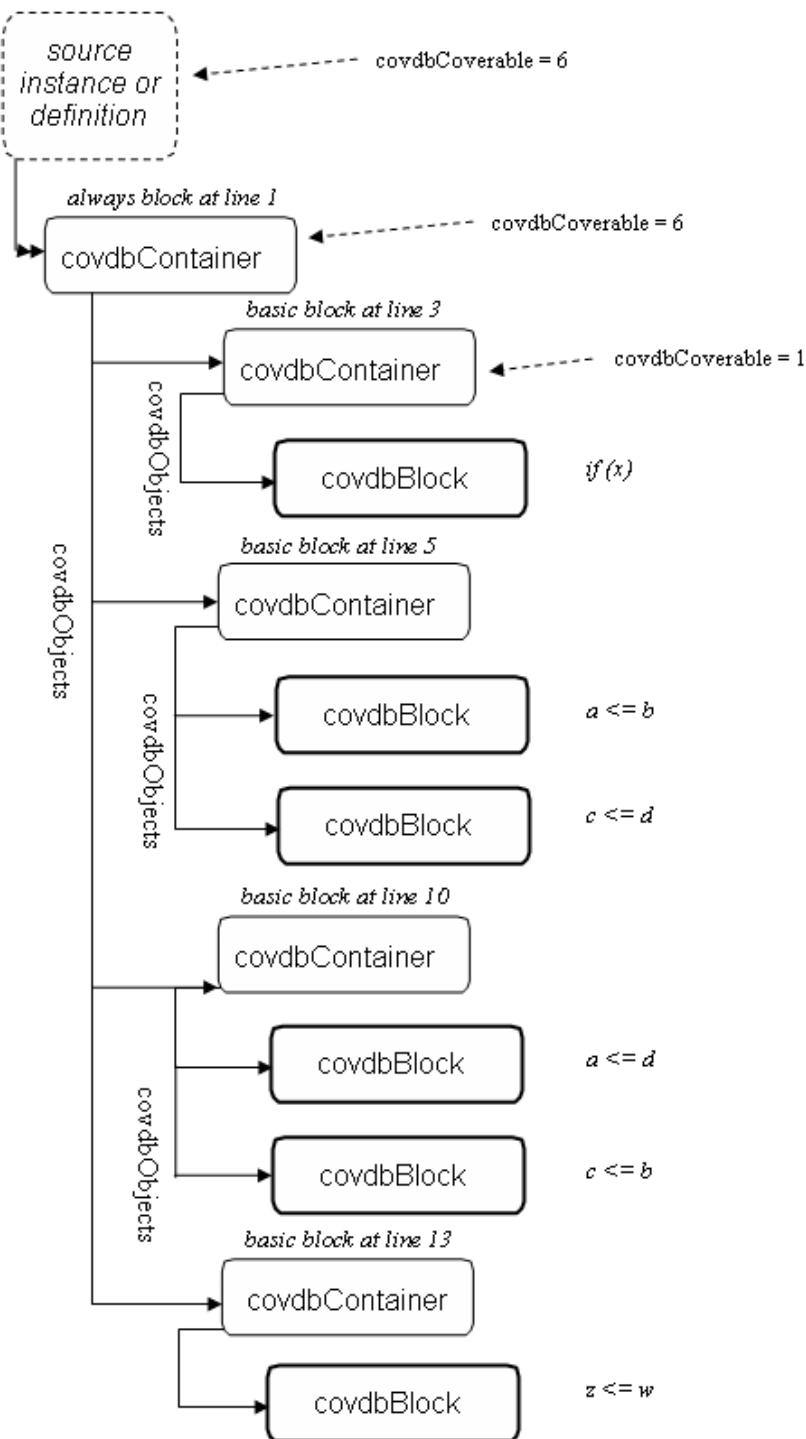
The organization of data for statement coverage is shown in the following figure.



For example, consider the following code:

```
1 always@(posedge clk)
2 begin
3     if (x)
4         begin
5             a <= b;
6             c <= d;
7         end
8     else
9         begin
10        a <= d;
11        c <= b;
12    end
13    z <= w;
14 end
```

This is represented in UCAPI as follows:

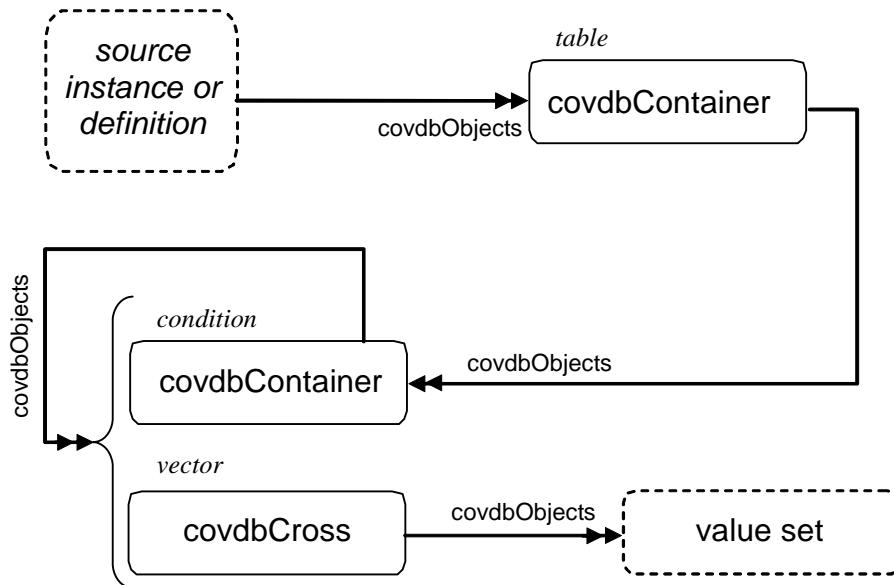


There is no representation of source *lines* as objects in UCAPI. There may be multiple statements on a source line, or a statement may extend across several lines. Source line information is available using the `covdbLineNo` property on the `covdbContainer` and `covdbBlock` handles.

Condition Coverage

Condition coverage tracks which values occurred in logical or bitwise expressions in the design. A *condition* is that logical or bitwise expression, and a *vector* is distinct set of values taken on by the terms in the expression. Conditions are modeled as `covdbContainer` objects, and vectors are modeled as `covdbCross` objects containing value sets.

Conditions are organized into tables, such as logical conditions, non-logical conditions, and event conditions. The `covdbName` of the table container gives the type of conditions in that table.



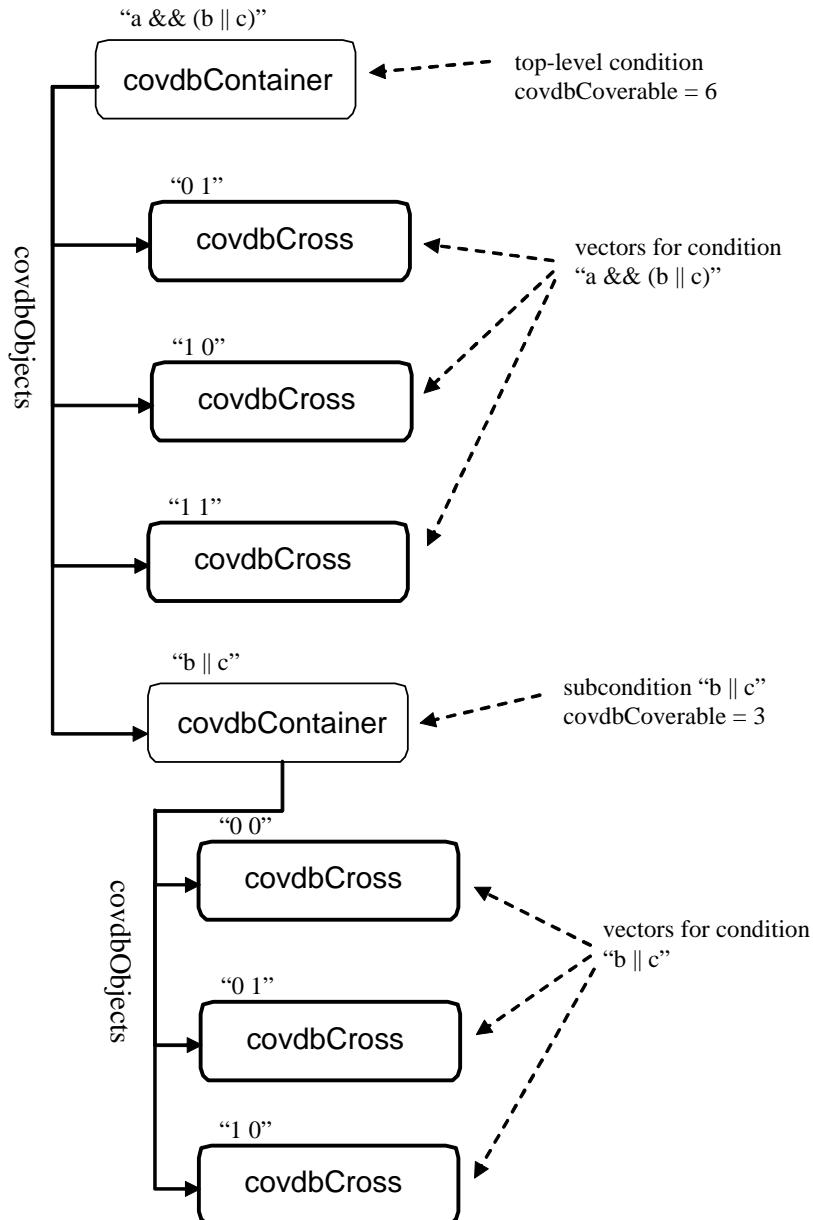
The exact vectors in a given database depend on the compile-time options used with VCS. For example, using the `-cm_cond` full flag results in full truth tables (vectors) for every condition, and using `-cm_cond sop` results in sum-of-products-style vectors.

Note that there is no representation of which conditions are nested *textually* within other conditions (such as the condition for an “`if`” statement that is nested inside a “`case`” statement). However, conditions can have sub-conditions within the same expression, and in this case, the sub-condition will be in the list of objects for the parent condition.

For example, if the following code is compiled in default condition coverage in VCS:

```
if (a && (b || c))
```

The parent condition consists of two terms in a “`&&`” expression, “`a`” and “`(b || c)`”. The sub-condition is the expression “`b || c`”, which has two terms in a “`||`” expression, “`b`” and “`c`”. In this case, UCAPI would represent the condition coverage objects as shown in the following diagram (these conditions would be in the “logical” conditions table, which is not shown in the diagram).



Some conditions are expanded into multiple conditions. A bitwise expression such as:

```
x = y & z;
```

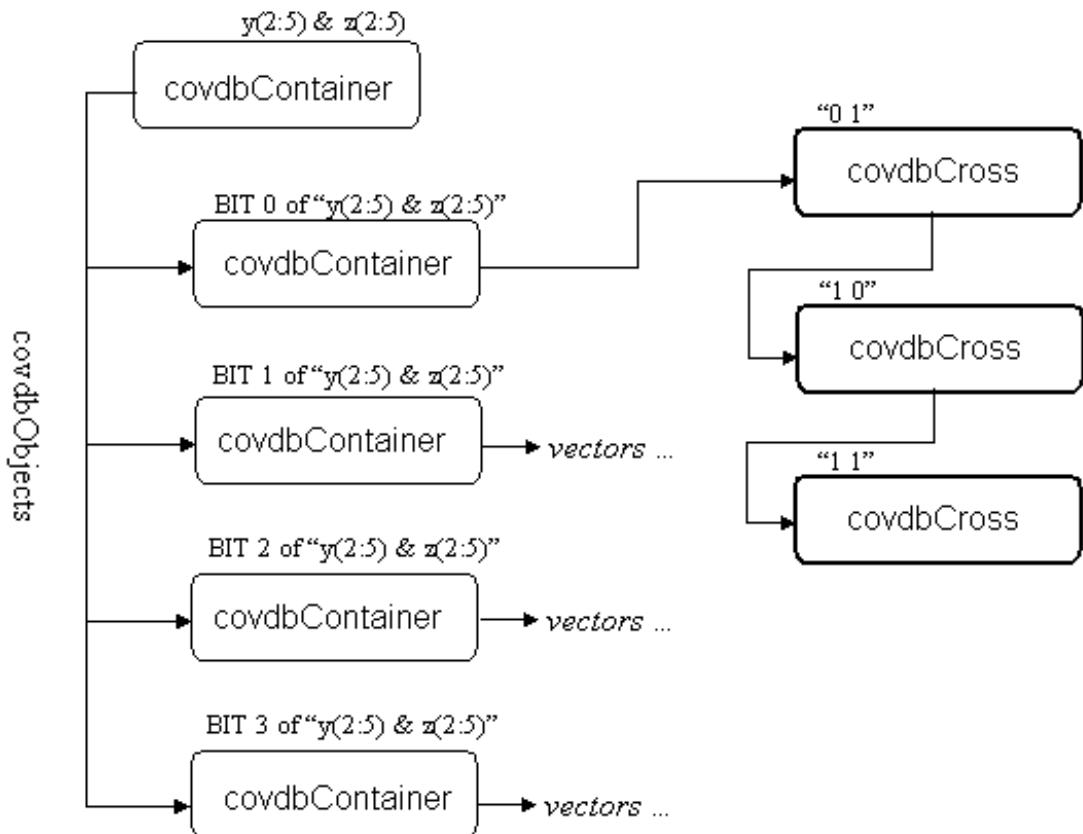
If x , y and z are 4-bit vectors, this represents four different conditions:

```

y(0) & z(0)
y(1) & z(1)
y(2) & z(2)
y(3) & z(3)

```

This is represented in UCAPI as additional conditions inside the main condition, as shown in the following figure. Note that these “non-logical” conditions themselves do not have any vectors of their own – only the single-bit conditions have vectors.



Note that “BIT 0 of $y(2:5) \& z(2:5)$ ” is the expression for $y(2) \& z(2)$, but UCAPI does not provide the original indices.

Another type of condition coverage, called “sum-of-products” or SOP condition coverage, is modeled similarly. For example, consider the expression:

```
((a == 2) || b && c) || (d && e)
```

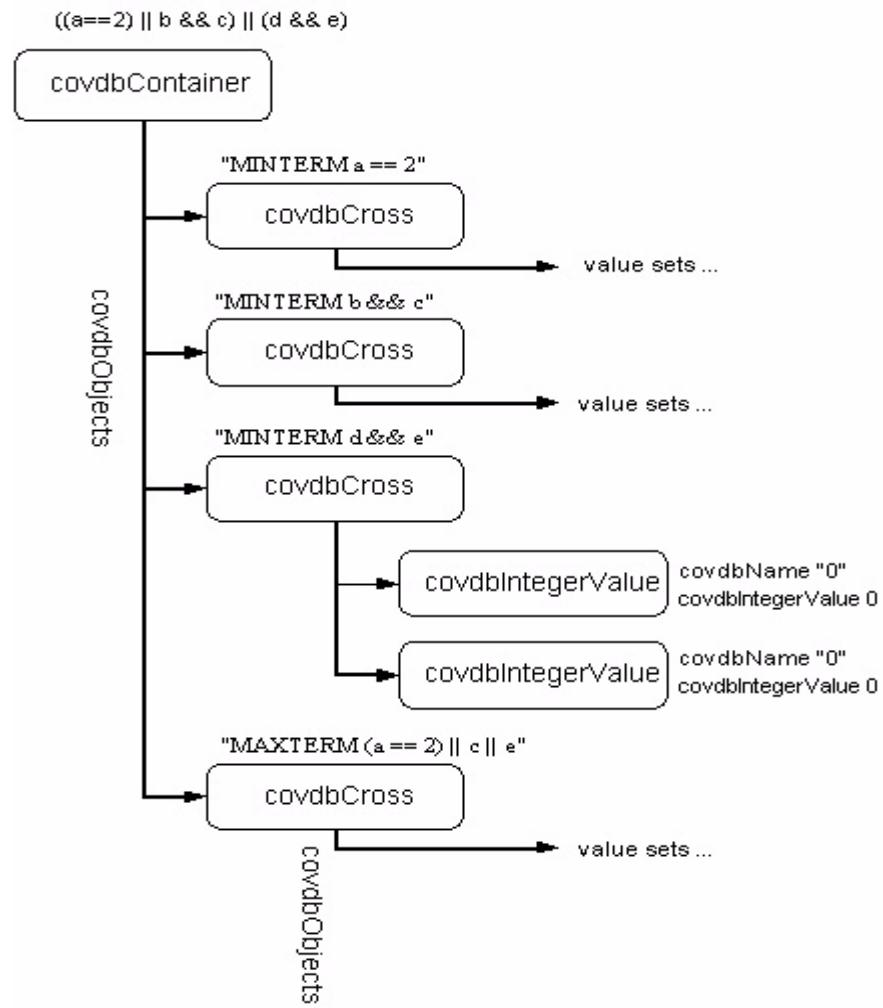
The SOP format of this expression is the disjunction of the following minterms:

```
(a == 2) ||  
(b && c) ||  
(d && e)
```

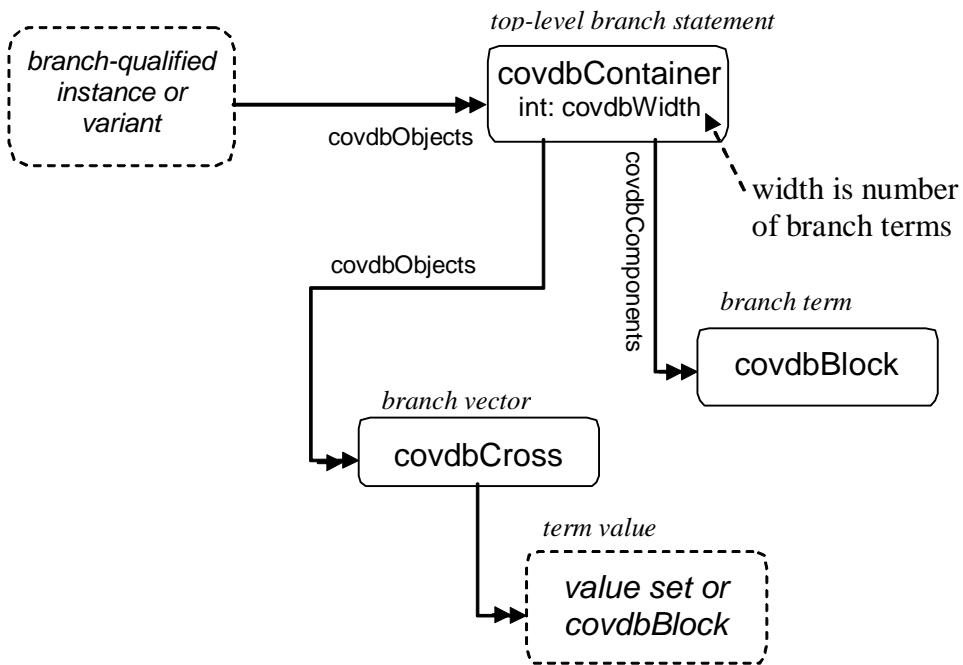
Similarly, the product-of-sums (POS) format of this expression is the conjunction of the following maxterms:

```
((a == 2) || c || e) &&  
((a == 2) || b || e) &&  
((a == 2) || c || d) &&  
((a == 2) || b || d)
```

UCAPI organizes the SOP (minterms) and POS (maxterms) expressions into containers under the expression, as shown in the following figure.



As described in “[Value Set](#)” on page 13, values are not always integers. In condition coverage, “don’t care” values may be present in vectors, and will have the value type `covdbScalarValue` and a `covdbValue` of `covdbValueX`.



Branch Coverage

Branch coverage monitors the execution of conditional statements in your design. Conditional statements include `if/else` statements, `case` statements, and the ternary operator “`? :`”

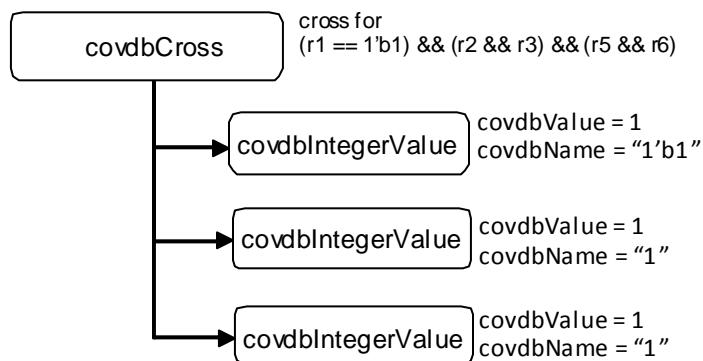
Branch coverage differs from condition coverage in that branch coverage does not track all the ways in which a Boolean condition can be true or false – it only tracks whether it is true or false. Branch coverage also tracks all conditions, rather than the subset of

complex conditions that condition coverage monitors. Branch coverage also tracks case statements, which condition coverage ignores.

The coverable objects for branch coverage are all crosses of value sets. For example, consider a branch that is covered when the following expression is true:

```
(r1 == 1'b1) && (r2 && r3) && (r5 && r6)
```

That branch would be modeled as a cross of these three terms, each of which is a covdbIntegerValue value set:



Branch coverage is organized into disjoint sections of each always/initial block or process by the top-level branch statements. For example, the following code has two top-level branch statements:

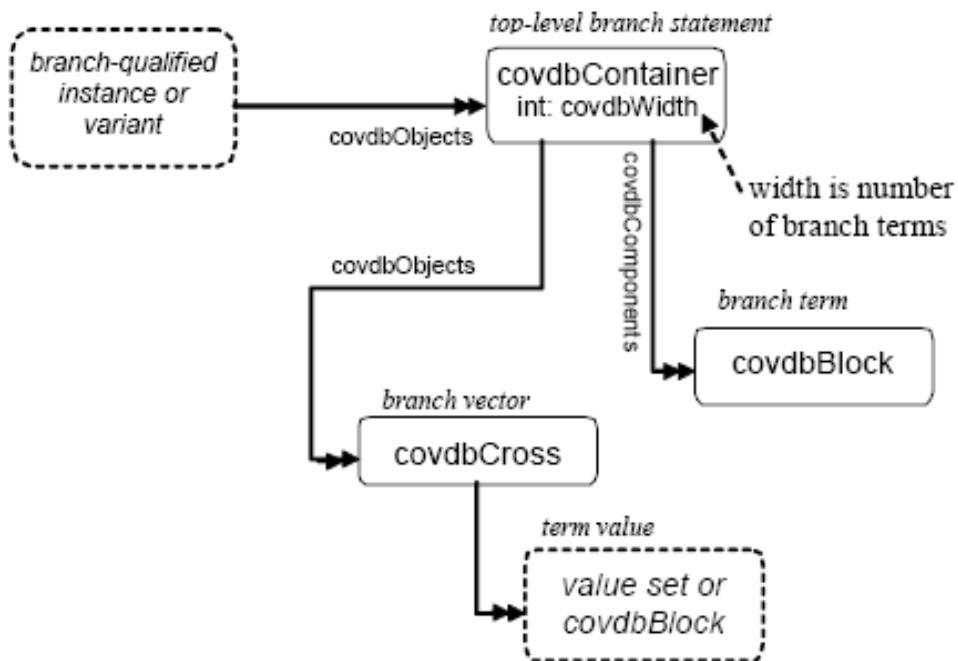
```
1 initial
2 begin
3   case (r1)           First top-level statement
4     1'b1 : if (r2 && r3)
5       r4 = 1'b1;
6     1'b0 : if (r7 && r8)
7       r9 = r10 ? 1'b0 : 1'b1;
```

```

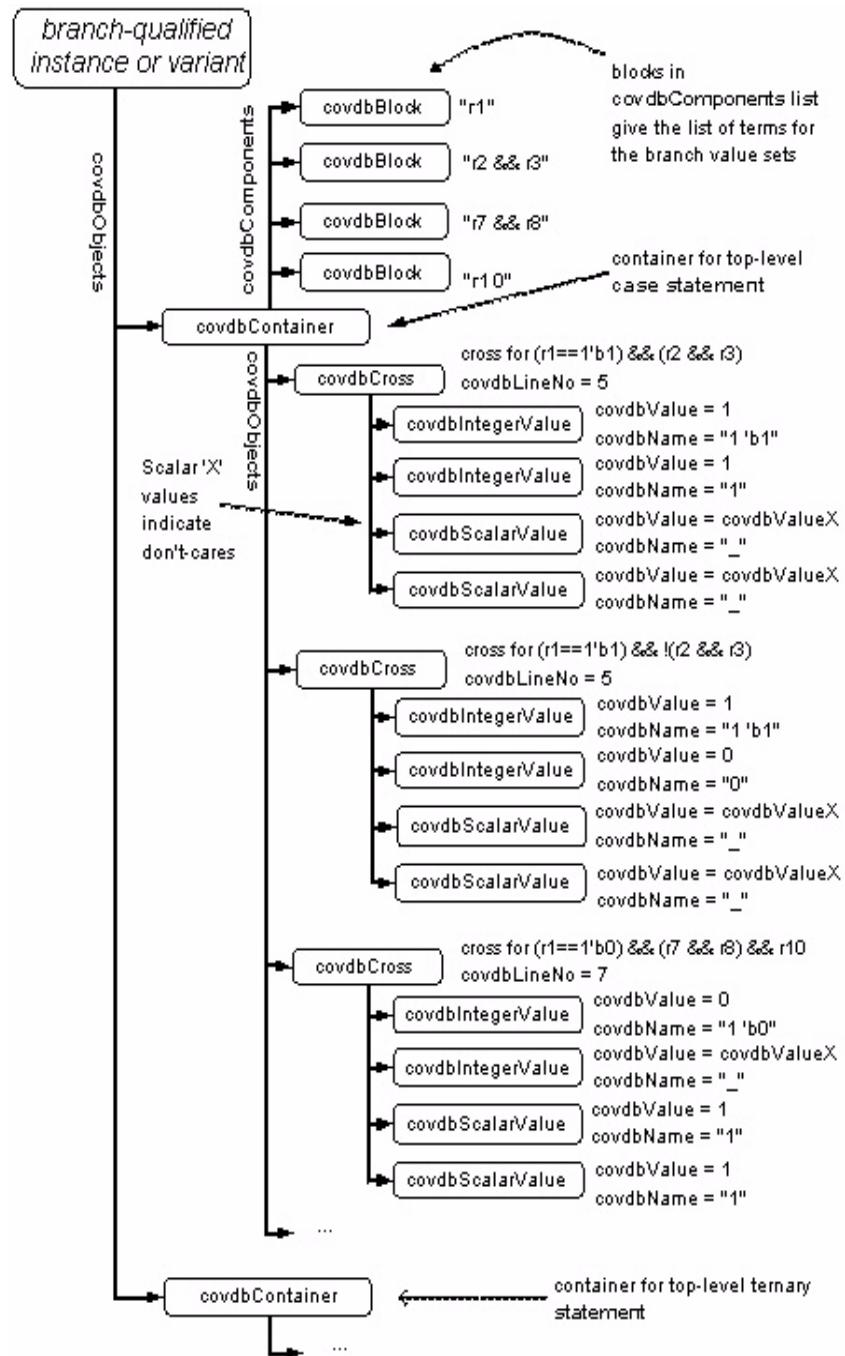
8          1'bx : $display("x");
9          default : $display("no op");
10         endcase
11
12         r9 = (r10 && r11) ? 1'b0 : 1'b1; Second top-level
13         statement

```

Since the branches in each top-level statement do not interact, UCAPI collects their branches into separate containers underneath the source region handle. The model for objects inside a branch coverage region is as follows:



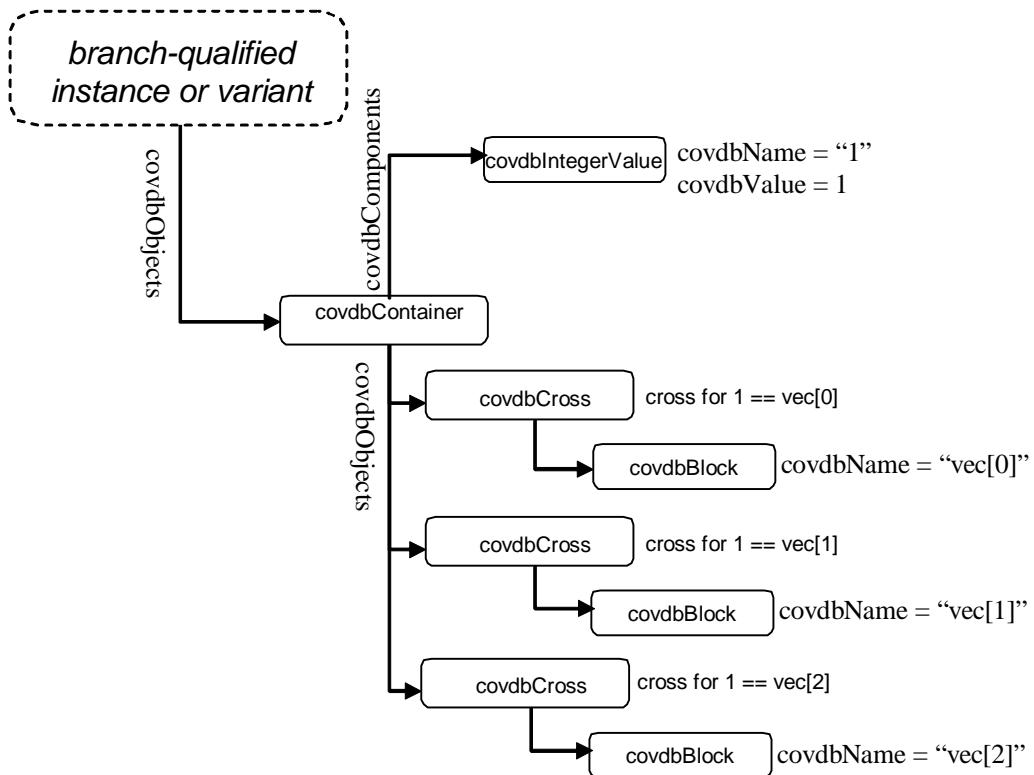
For the sample code shown above, the UCAPI model is:



Note that in Verilog, case alternatives do not have to be constant values. For example, the following is legal:

```
case (1)
  vec [0] : ...
  vec [1] : ...
  vec [2] : ...
endcase
```

In this example, the case alternatives are not constant values, and therefore cannot be modeled as value sets. In this case, we use covdbBlock handles for the case alternatives. Note that the argument to the case conditional ("1") is when possible modeled as a value set:



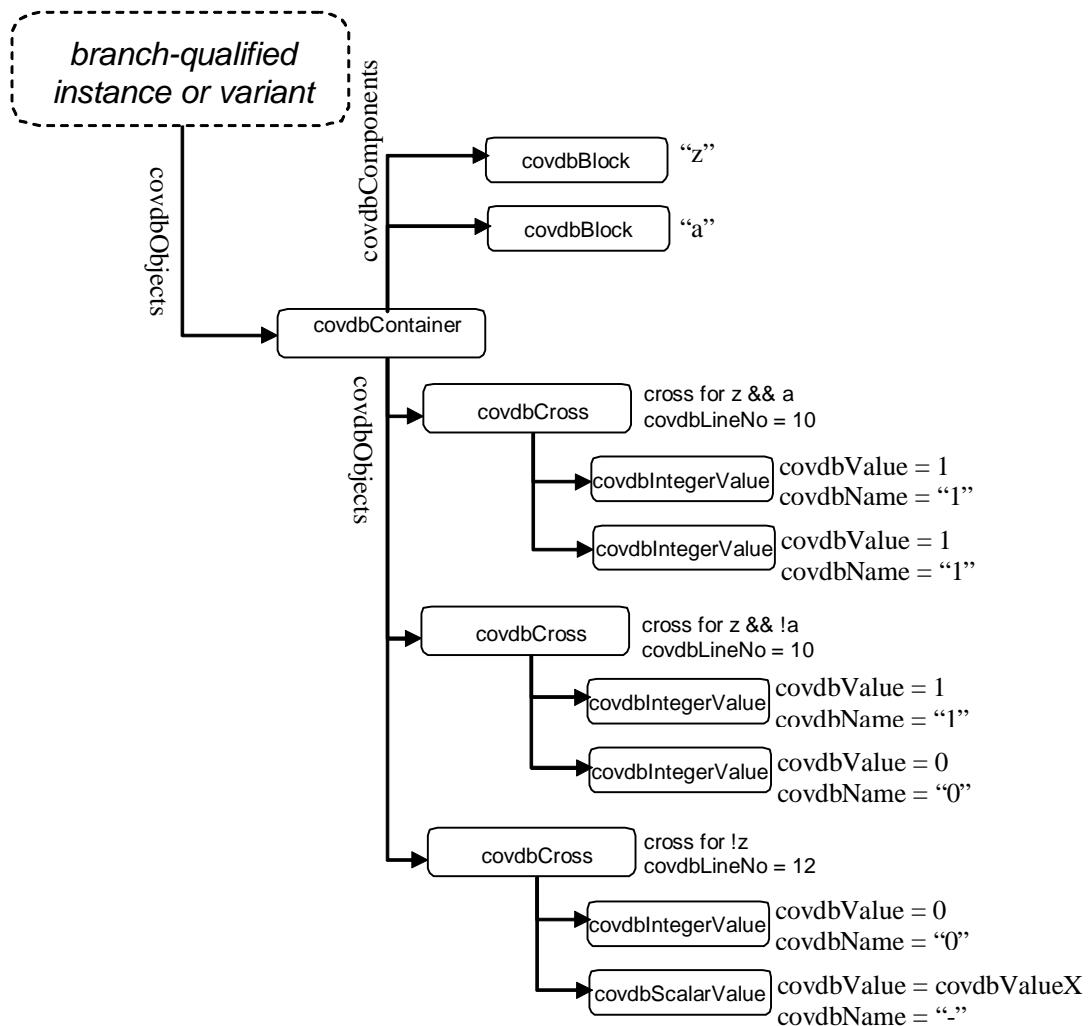
Ternary operators are also monitored by branch coverage. In UCAPI, a ternary conditional is treated the same as an “if/else” condition. For example:

```

9 if (z)
10      x <= a ? b : c;
11 else
12      x <= 1'b0;

```

Turns into this in UCAPI:



Finite State Machine Coverage

Finite State Machine (FSM) coverage models three distinct types of coverable objects.

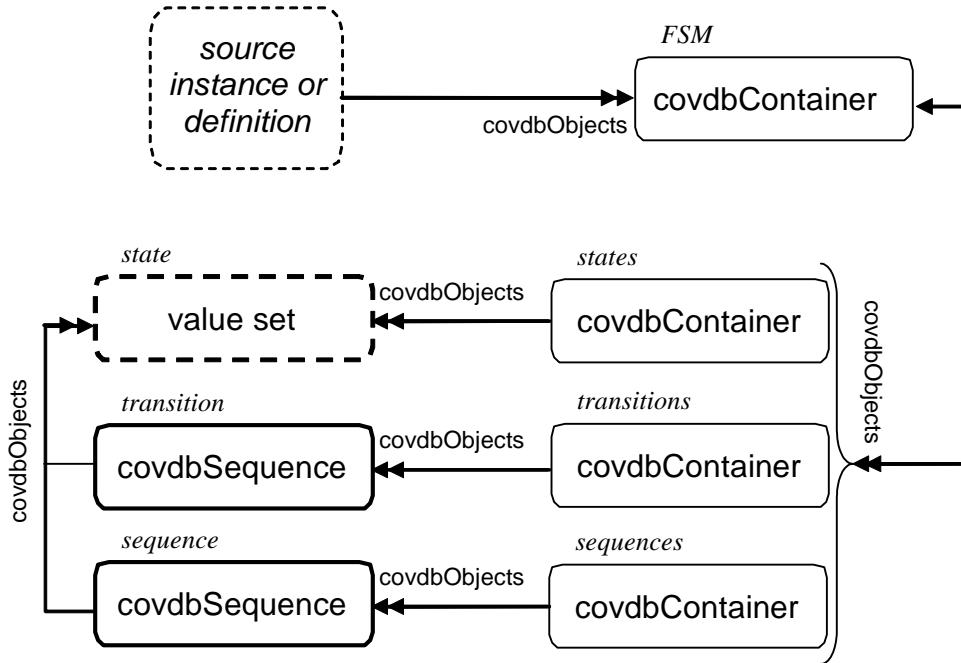
Whether the FSM:

- entered each possible state (state coverage)
- state followed each possible state transition (transition coverage)
- went through all possible sequences of states from the initial state(s) (sequence coverage)

In UCAPI, state coverage is modeled with value sets, and transition and sequence coverage are modeled with `covdbSequence` objects containing value sets. The `covdbCoverable` property of the FSM container is the total number of states, transitions, and sequences for that FSM.

The `covdbFileName` and `covdbLineNo` properties may be read from the FSM handles (of type `covdbContainer`), state value sets, and transition `covdbSequence` handles. No other FSM objects have source information.

State and sequence objects are all considered to be used for information only and do not count for `covdbCoverable` and `covdbCovered` numbers for FSM coverage. Each coverable object for states and sequences will have `covdbExcluded` set in its `covdbCovStatus` property.



For example, consider an FSM encoded as:

```
always @(posedge clk) begin
    case (state)
        `IDLE: if (go) next = `READ else next = `IDLE;
        `READ: if (go) next = `DLY else next = `IDLE;
        `DLY: if (!go) next = `IDLE;
            else if (ws) next = `READ;
            else next = `DONE;
        `DONE: next = `IDLE;
    endcase
end
```

The FSM has four states and seven possible transitions:

States:

IDLE
READ
DLY
DONE

Transitions:

IDLE->READ

READ->IDLE

READ->DLY

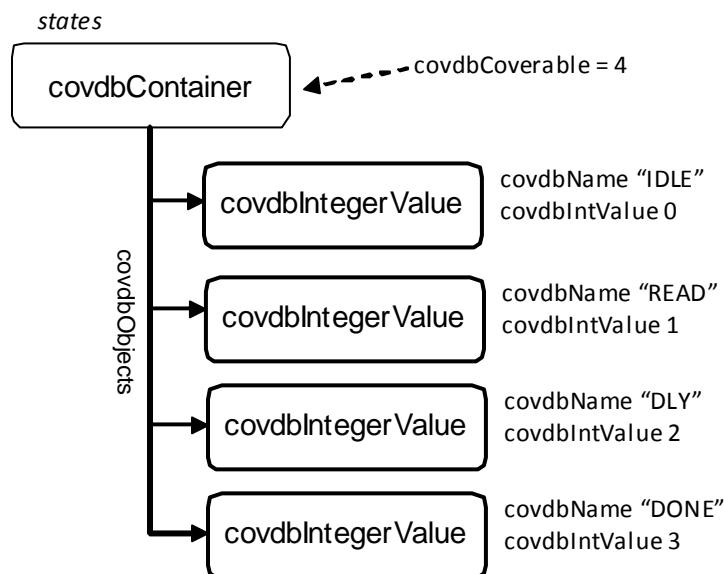
DLY->IDLE

DLY->READ

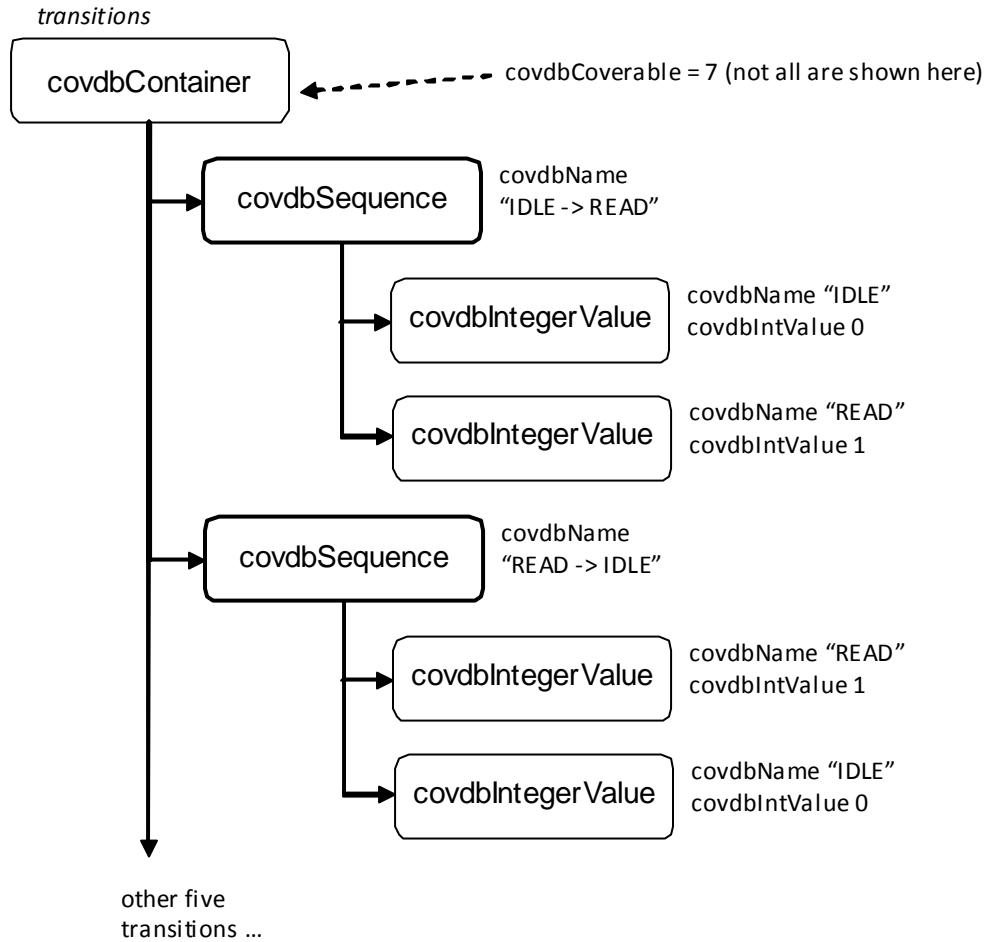
DLY->DONE

DONE->IDLE

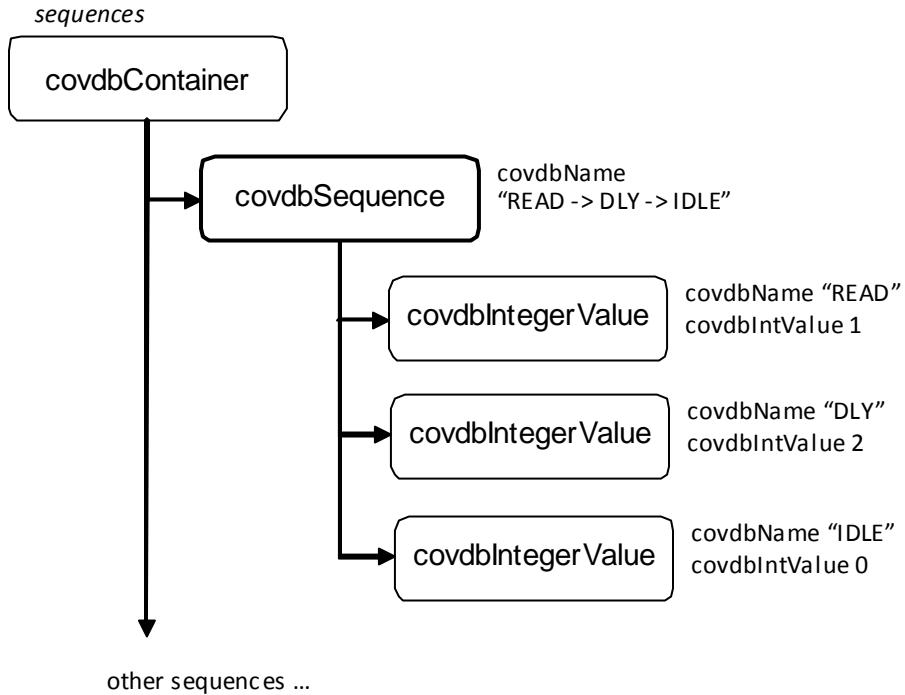
The states in UCAPI will look like:



The transitions are represented as `covdbSequences`, as shown in the following illustration:



The FSM's sequences are covdbSequence objects, but with potentially more than two objects. For example, the sequence READ -> IDLE > IDLE looks like this:



Toggle Coverage

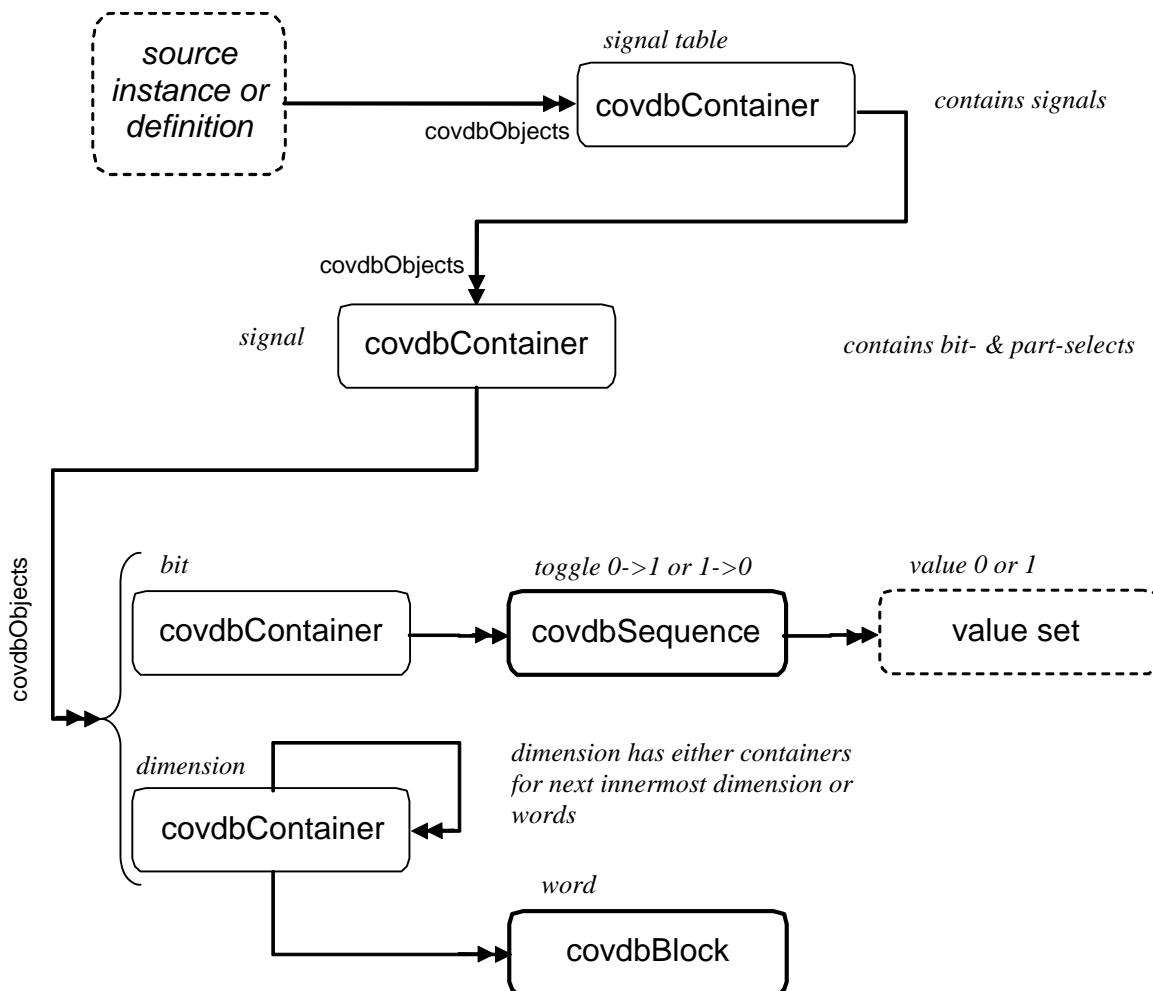
Toggle coverage in UCAPI is organized into tables, each of which contain signals, which are modeled as `covdbContainers`.

The tables are groupings of signals by category, such as “signal” and “port”. The `covdbName` property of the table container is the category of all signals in that table (e.g., “ports”).

Each signal’s container has a list of sub-containers in it, representing bits of that signal. Each sub-container has two `covdbSequences` in it, one representing the transition from 0 to 1, and one representing the transition from 1 to 0.

Signals and MDA dimension container objects for toggle coverage have the annotations “`lsb`” and “`msb`” set on them. These return a string given the least-significant bit and most-significant bit indices, respectively, as strings.

The `covdbCoverable` of a given signal for toggle coverage is twice the number of distinct bits for that signal – that is, the `covdbCoverable` is the number of `covdbSequences` under the container for the signal. The `covdbWidth` property is the total number of bits.



For example, consider a module “M” with these signals:

```
reg x[3:0];  
input y;  
wire y;
```

Now assume the toggle coverage data is as shown in the following table:

Table 6-2

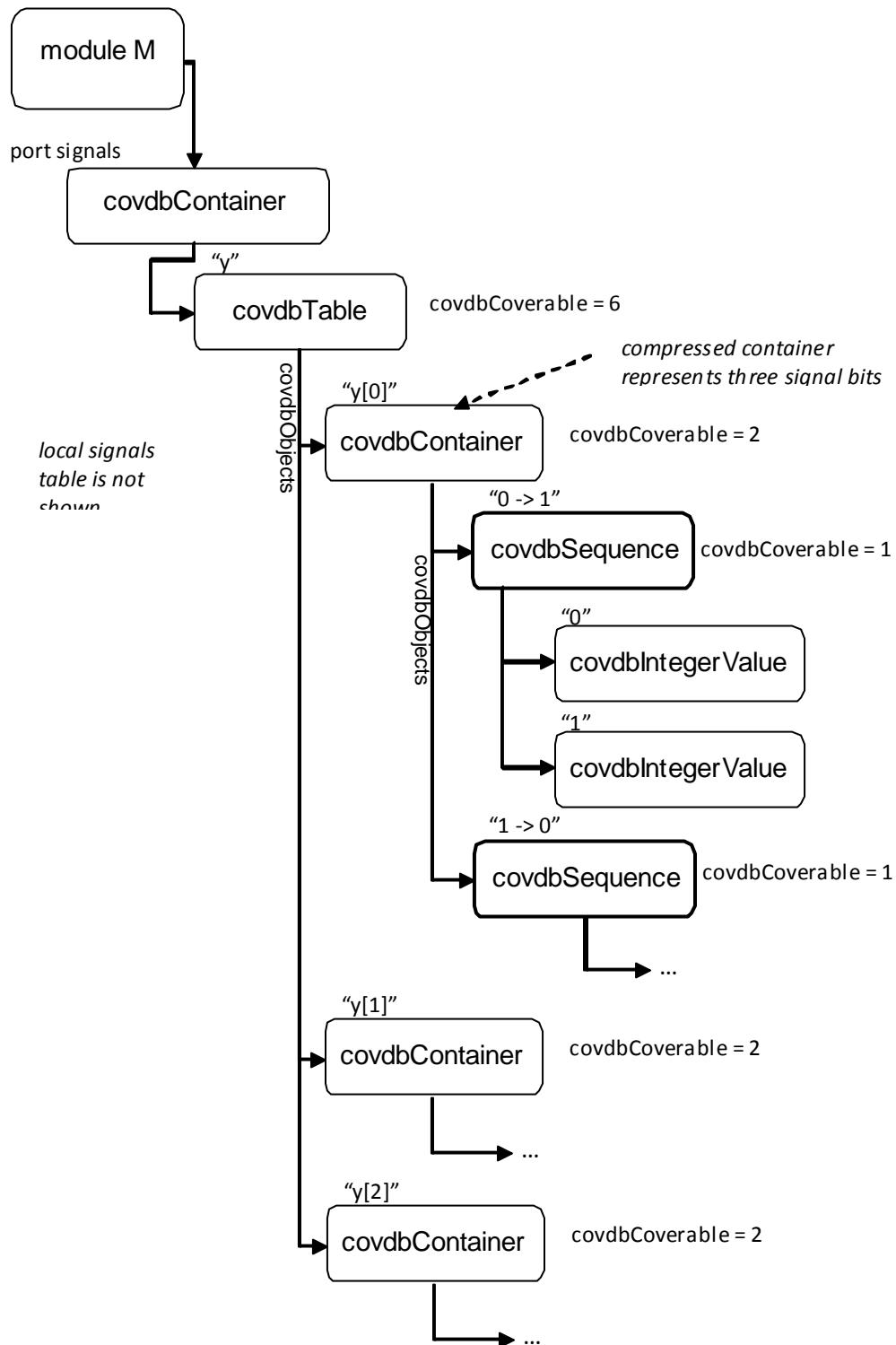
Signals	0->1	1->0
x[3]	Y	N
x[2:1]	Y	Y
x[0]	N	Y
y	Y	N

The toggle coverage structure would look like the following diagram.
(In this figure, only some of the covdbSequence and value set children are shown.)

Now consider a case with a multiple-dimension array signal:

```
reg [2:0] mda[3:0][1:0];
```

Then the data would look like the following (only the handles containing mda [1] [0] [2] are shown):



Assertion Coverage

Assertion coverage includes coverage information about assertions, cover properties and sequences. The `covdbName` of the assertion coverage metric handle is **Assert**.

Assertion coverage objects may appear directly in source definitions and instances (inlined assertions) or in special source regions called **units**, which are test-qualified source regions (see “[Test-qualified Source Regions](#)” on page 40). Inlined assertions are metric-qualified but not test-qualified, and they may be read without loading any test. Assertions in units (and the list of units itself) are not available except with respect to a given test handle.

In the current version, inlined assertions are accessible through the interface as described, but will only appear in the design as tests are loaded.

There are three types of assertion coverage objects: assertions, cover properties, and cover sequences. The handles for these different assertion coverage objects are organized into `covdbContainers` in each source definition and instance.

Assertions are modeled in UCAPI as `covdbContainers`. The `covdbBlock` objects in these containers represent the different type of information collected. For example, there is a `covdbBlock` representing success, a `covdbBlock` for failures, and another for attempts. The `covdbName` of each `covdbBlock` indicates what it represents.

All but one of the `covdbBlock` objects will have the `covdbStatusExcluded` flag set, indicating they should be ignored for coverage. For example, for assertions there will be a block for

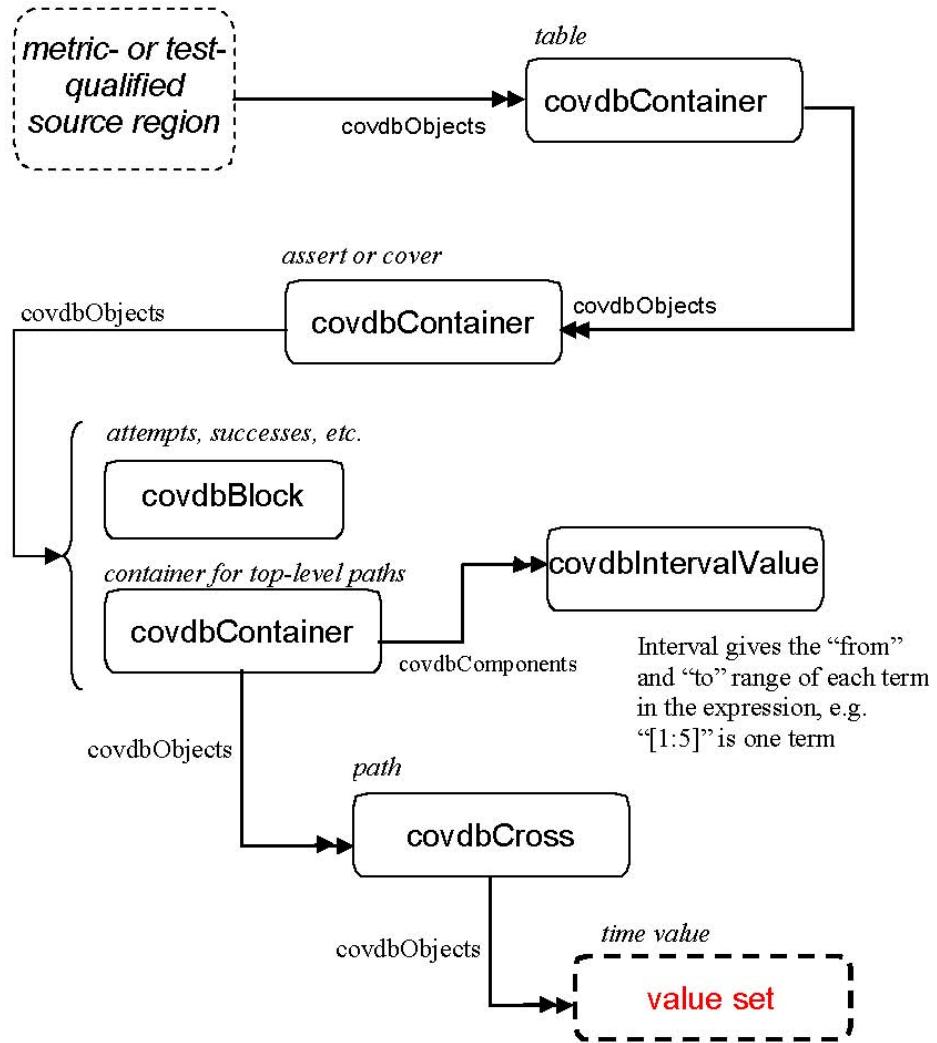
attempts. The number of attempts is not used to compute the coverage score, so this block is marked excluded. The assertion is covered if its non-excluded covdbBlock object is covered.

The covdbCovCount may be read from each of these covdbBlock objects to get the number of times each type of thing occurred. Non-excluded covdbBlocks are covered if their covdbCovCount is greater than or equal to the assertion's covdbCovCountGoal.

SystemVerilog assertions and cover directives will always be in the assert-qualified module definitions and instances in the covdbObjects list. OpenVera Assertions are contained in test-qualified source regions, as discussed in [“Test-qualified Source Regions” on page 40](#).

The covdbFileName and covdbLineNo properties may be read from handles of type covdbContainer for assertion coverage. There is no source information for covdbBlock handles in assertion coverage.

The following diagram shows the model for assertion coverage:



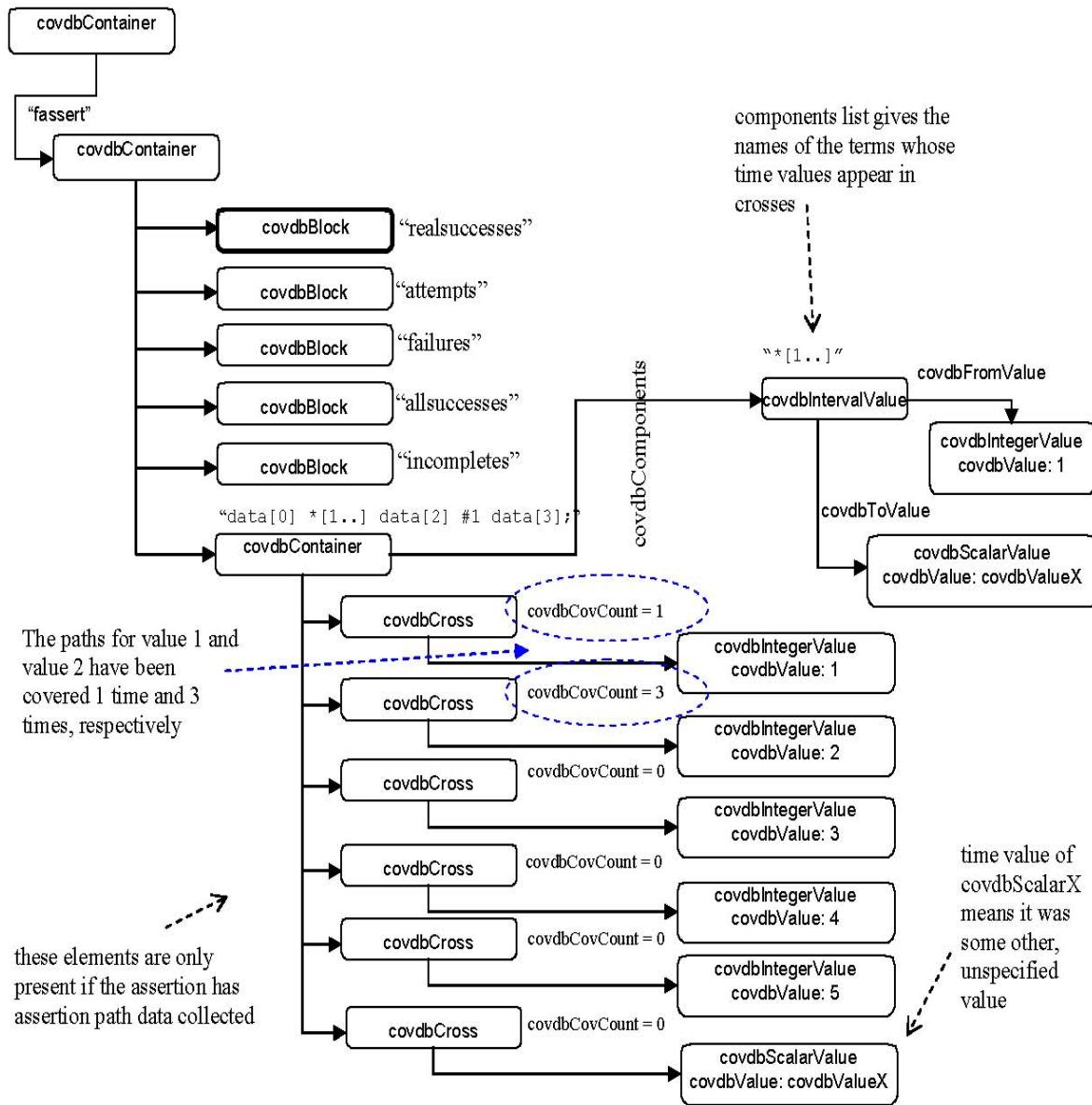
For example, consider the following assertion:

```

clock posedge clk {
    event fevent : data[0] * [1..] data[2] #1 data[3];
}
assert fassert : check(feevent);
  
```

The following diagram shows the UCAPI structure for this assertion with basic coverage. Note that there are five covdbBlocks under the assertion, but that only the `realsuccesses` block counts for coverage, since the other blocks are all marked excluded. These other blocks (e.g., `attempts`, `incompletes`) are in the database, but are never considered for coverage.

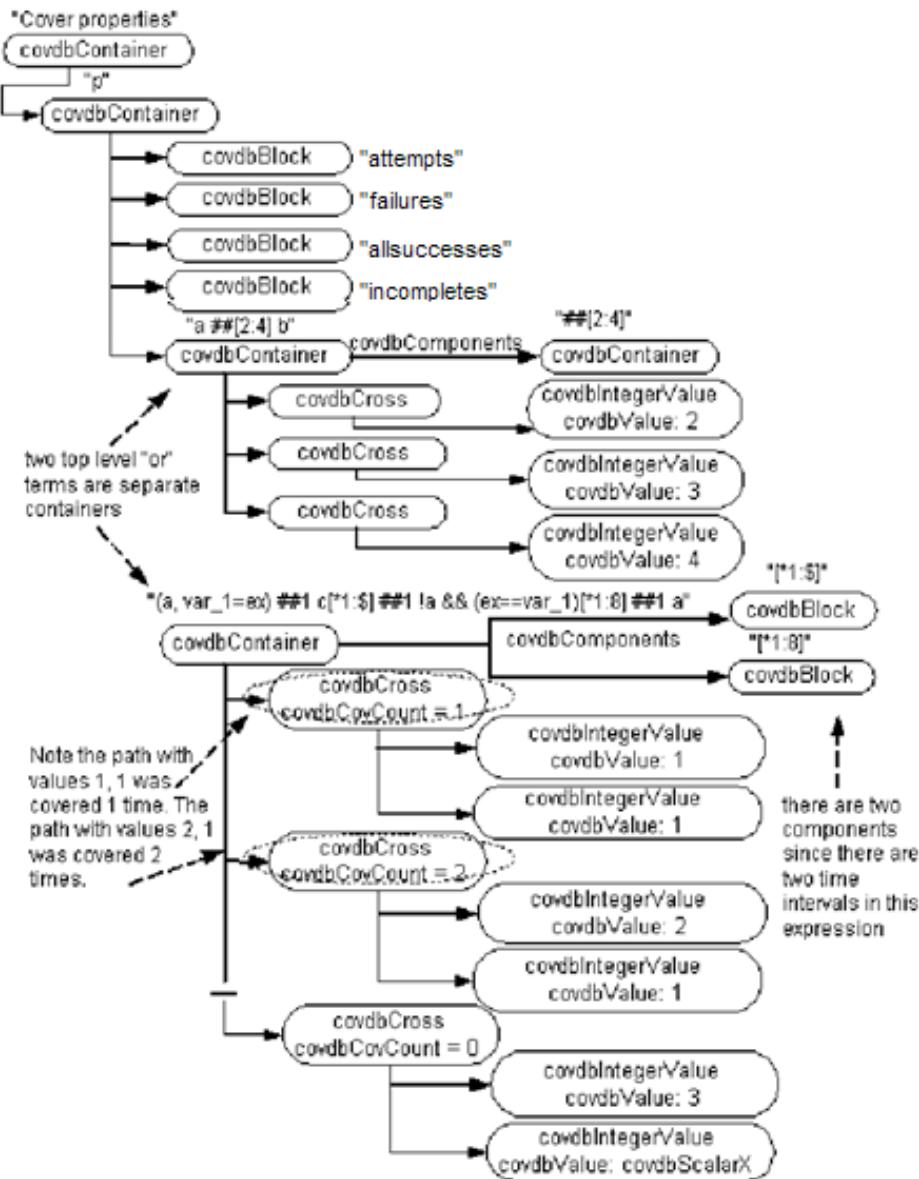
In this example that only the paths where the first range (# [1 ..]) had value 1 or 2 was the assertion covered. It was covered one time for the value 1 and three times for the value 2. You can see this as the covdbCovCount of the individual covdbCross bin handles.



Now consider another example, a cover property with multiple time intervals in it:

```
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
```

Since this property has a top-level `or`, it will have two containers for paths under the assertion container. Also, since the second term has two time intervals in it, the crosses contains two values each.



You can read the category and severity failures for a given assertion handle using the `covdb_get_annotation` function:

```

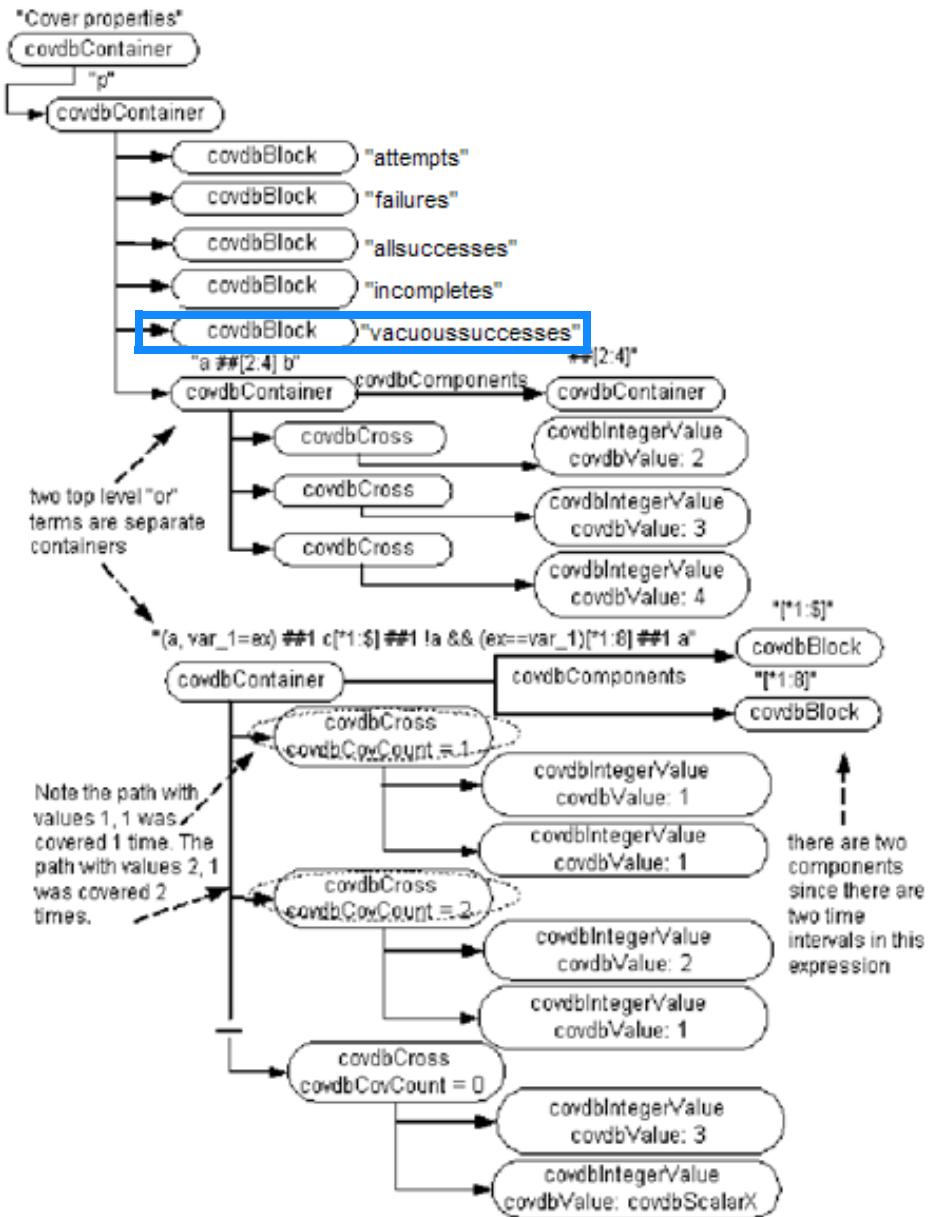
char *category = covdb_get_annotation(assertionHdl,
"FCOV_ASSERT_CATEGORY"); char *severity =
covdb_get_annotation(assertionHdl,

```

```
"FCOV_ASSERT_SEVERITY") ;
```

For cover properties, there are four blocks under the property, namely, "attempts", "failures", "allsuccesses" and "incompletes". Additionally, if vacuous passes are monitored at runtime using the

switch -assert vacuous, a fifth block, named "vacuoussuccesses" will be added to the container (as shown in the blue rectangle in the figure) as follows:



Testbench Coverage

Testbench coverage monitors user-specified expressions during simulation. Testbench coverage data may be monitored through coverage constructs in Vera, Native Testbench (NTB), or System Verilog. The `covdbHandle` of the testbench coverage metric handle is `Testbench`.

Testbench coverage monitors three different types of coverable objects: states (value sets), transitions (modeled as `covdbSequence` objects), and crosses (`covdbCross` objects). For testbench coverage, the containers of these coverable objects are called bin tables.

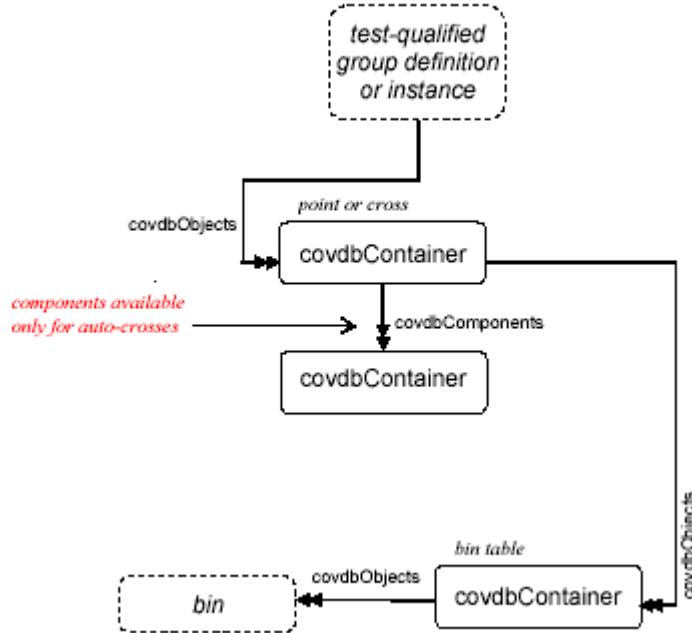
As for assertion coverage, many of the coverable objects in testbench coverage may be marked `covdbStatusExcluded`. Such objects are not used when computing coverage information such as the `covdbCovered` or `covdbCoverable` properties.

The `covdbFileName` and `covdbLineNo` properties may be read from group, group instance, point and cross declaration handles for testbench coverage. No other handle types for testbench coverage have source information.

When testbench covergroups are defined within HDL modules (for example, inside a SystemVerilog module), they are accessed as the `covdbObjects` list from the testbench-qualified HDL module handle.

The following figure shows the structure of testbench coverage within a test-qualified group or instance definition. The top-level list is containers representing point or cross declarations within the group. Each point or cross declaration has one or more bin tables, and each bin table has its list of bins.

For cross declarations, there may be a `covdbComponents` list, which will contain the list of point declarations that are being crossed.



The bin handle can be different depending on the type of bin table. The types of bin tables are user-defined bin table, auto bin table, user-defined cross table, and auto cross table. The type of the table is given by its `covdbName` property.

Currently there is no representation in UCAPI of the containing program for covergroups declared in programs.

Setting Hit Count on Bins

You can set hit count on user bins, compressed and uncompressed auto bins for coverpoint and cross.

Usage

`covdb_set(binHandle, coverGroupHandle, testHandle,
covdbCovCount, hitCount)`

`covdb_get(binHandle, coverGroupHandle, testHandle,
covdbCovCount)`

Hit count can be set on bin handles at definition or instance level.

It's allowed at definition level only when `per_instance` is off. Because, definition level hit count should be equal to the sum of hit counts of the bin in all instances. If it's set at definition level, you can't distribute it among instances. So, you get an error when you try to set the hit count on definition level bin handles keeping `per_instance` ON.

You can set hit count on instance level bin handles. The increase or decrease in hit count is populated to the corresponding bin at the definition level.

Set covdbStatusCovered

To mark a bin as covered, increment the hit count of that bin to the coverpoint's least value.

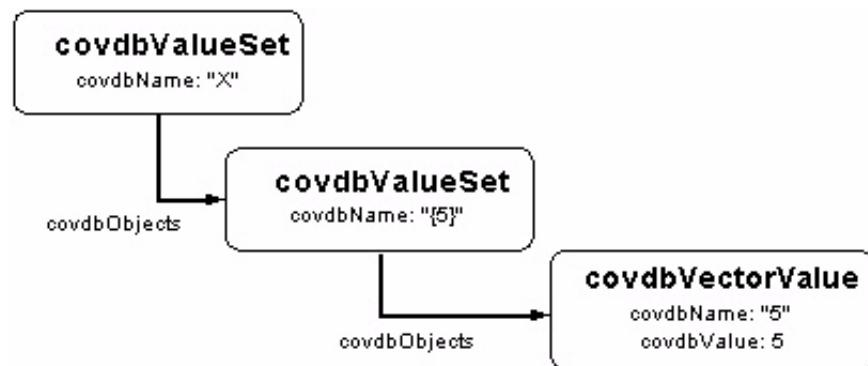
Usage

`covdb_set(binHandle, coverGroupHandle, testHandle,
covdbCovStatus,covdbStatusCovered)`

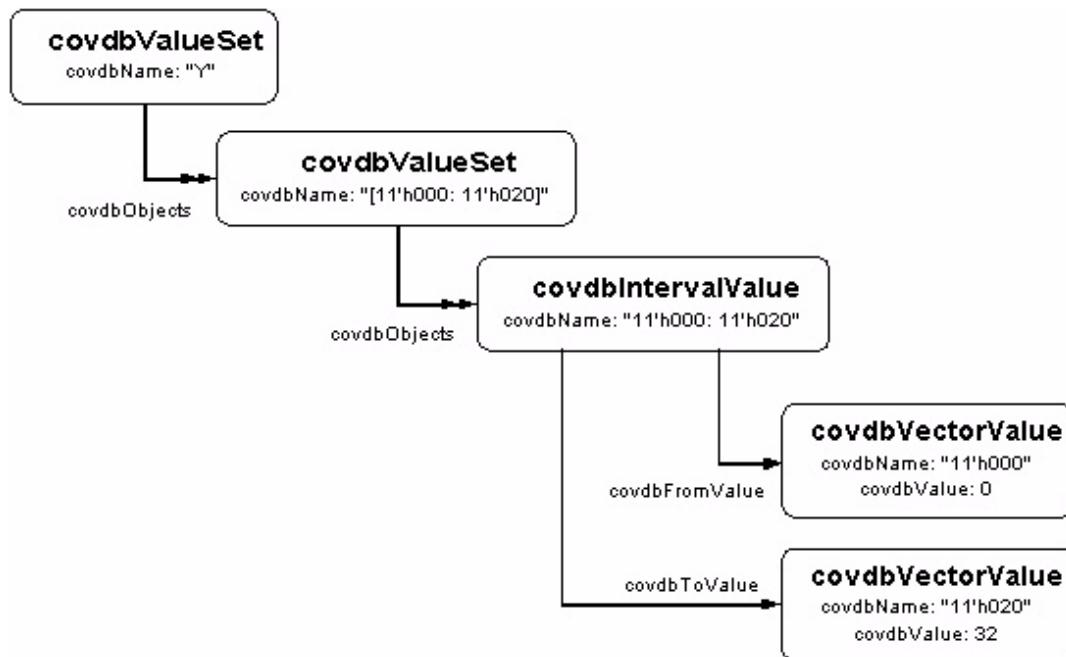
`covdb_get(binHandle, coverGroupHandle, testHandle,
covdbCovStatus)`

Contents of the Bin Tables

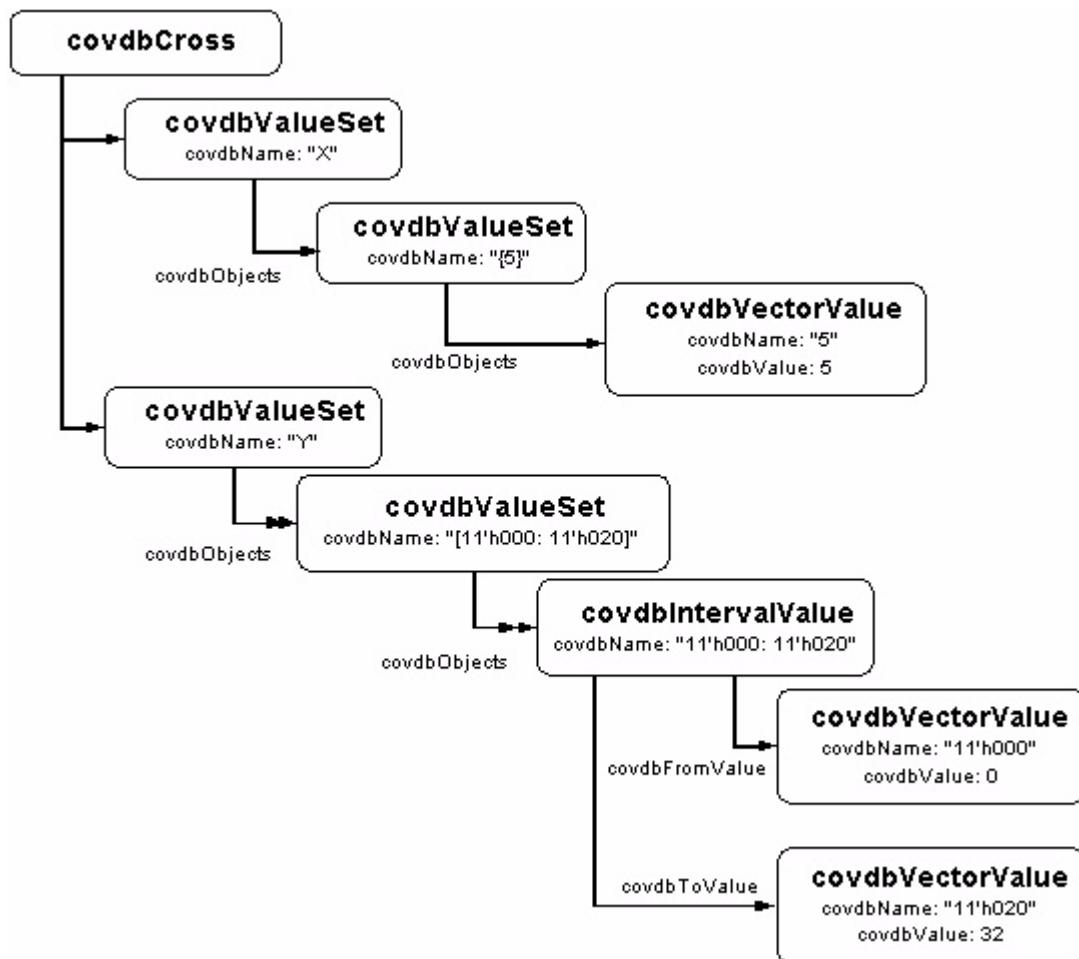
For user-defined and auto bin tables, the bins will all be value sets.
For example, a bin x with the value 5 would be:



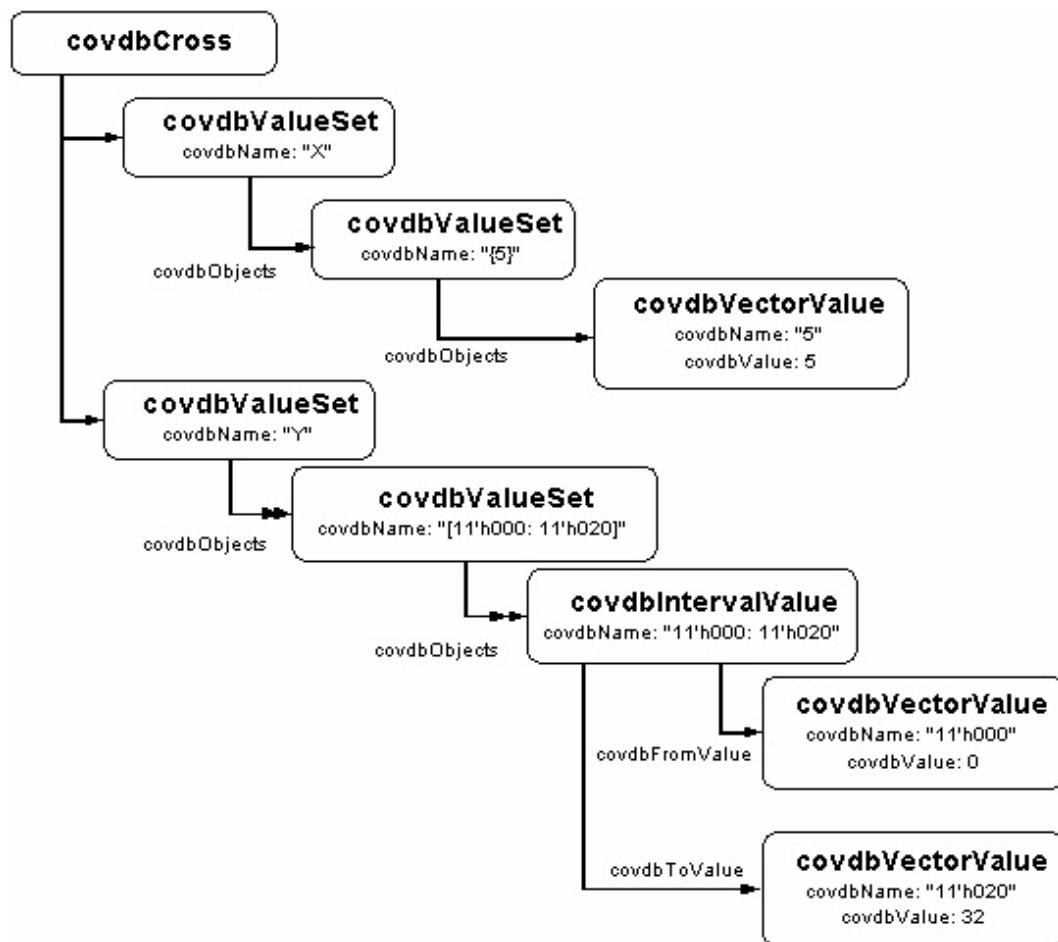
And a bin $\text{Bin Y} = [0:32]$ will be represented by the following:



Crosses contain handles to the bins being crossed, so a cross of
 $x = 5$ and y in $[11'h000 : 11'h020]$ would be:



User-defined cross bins are defined by the (what would otherwise be) auto-cross bins they contain. The user-defined cross bins (in the user-defined cross bins table), therefore have an extra level of structure:



The `covdbName` string property of a point container gives the name of the expression being sampled. For example, a coverpoint on variable "myvar" would have `covdbName = "myvar"`. A cross container has the relation `covdbComponents`, which is an iterator over the coverpoint containers that were crossed to create the cross

itself. Using these two methods (names for points and covdbComponents for crosses), applications can determine the meaning of the values given in the bins.

For example, if a coverpoint's name is "myvar", then each of its value set bins will refer to a value of "myvar" that was observed. If a cross's components are points x and y , then the value set bins in each cross bin are bins from coverpoints x and y , respectively.

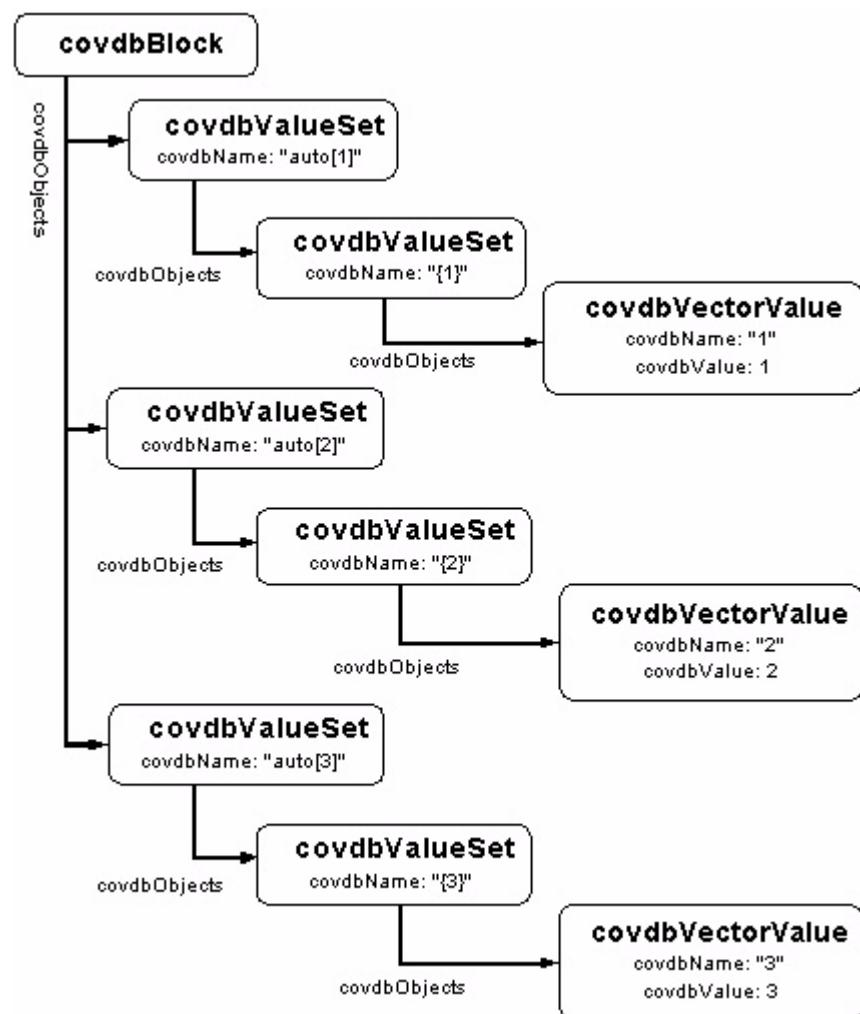
Compressed Bins

Each bin handle will have the string annotation "COVERPOINT_BIN" set to 1 or 0. If it is set to 1, it means the bin is a regular bin. If it is set to 0, it means the bin is a compressed bin. Compressed bins have covdbCoverable greater than 1 (the value is the number of bins that were compressed).

The property 'covdbObjects' gives an iterator over individual coverpoint bins of compressed coverpoint bin handle. In case of compressed cross bins, there are two kinds of iterators:

- iterating over individual uncompressed auto-cross bins. The property 'covdbObjects' returns this iterator.
- iterating over constituent coverpoint bins. The property 'covdbComponents' returns this iterator.

For example, the following bin, `auto[1:3]`, is a compressed bin made up of three bins, one each for the values 1, 2 and 3. The compressed bin is a `covdbBlock`, and its `covdbObjects` list is the list of bins that make it up:



Limitations

Currently, only automatically-generated cross bins are represented as `covdbCross` objects. User-defined cross bins are represented as `covdbBlock` objects. All transitions are currently represented as `covdbBlock` objects.

The `covdbComponents` list is currently only available for automatically generated crosses, not for user-defined crosses.

Loading a Design

Note that when using VCS, designs are loaded with the `covdb_load()` function. The following function, `loadDesign`, accepts a string argument representing the path to the design and then loads the design.

```
covdbHandle loadDesign(char *design)
{
    covdbHandle designHandle = covdb_load(covdbDesign, NULL,
                                           design);
    if (!designHandle)
        printf("Error: could not load design %s\n", design);
    else
        printf("Info: loaded design %s\n", design);
    return designHandle;
}
// example of calling the function:
covdbHandle designHandle = loadDesign("simv"); //
```

Note:

A handle returned by `covdb_load` is persistent until unloaded by `covdb_unload`.

Available Tests

Loading a design does not load the test(s) associated with a design. Given a valid design handle, the available tests can be accessed.

Calling `covdb_iterate()` with `covdbAvailableTests` returns an iterator to a list of the testnames associated with the design. These tests are not automatically loaded; they must be loaded with one of the API functions. The following C function shows the available tests associated with a design handle.

```
void showAvailableTests(covdbHandle designHandle)
{
    covdbHandle availableTestName availableTestNameIterator;
    availableTestNameIterator = covdb_iterate(designHandle,
                                              covdbAvailableTests);
    printf("Design: %s has the following tests:\n",
           covdb_get_str(designHandle, covdbName));
    while(availableTestName =
          covdb_scan(availableTestNameIterator))
        printf("\ttest: %s\n", covdb_get_str(availableTestName,
                                             covdbName));
    covdb_release_handle(availableTestNameIterator);
}
```

Loading Tests

Tests are not automatically loaded with a design. Once a design handle has been obtained, the handle is used to access the list of test names and then load one or more of the tests. A single test is loaded with `covdb_load` while two or more tests can be loaded with successive calls to `covdb_loadmerge`.

Loading a single test requires that a valid design handle and test name string be available:

```

/* load a single test from a design */
covdbHandle testHandle;
testHandle = covdb_load(covdbTest, designHandle, testName);
if(!testHandle) {
    printf("UCAPI Error: could not load test %s\n", testName);
    exit(1);
}

```

Loading and merging two or more tests first requires the loading of a single test with `covdb_load` followed by multiple calls to `covdb_loadmerge` to load and merge the additional tests' data. The following example shows a function used to load all the available tests related to a design. It is assumed that all of the test data is contained in one location with the design data.

```

covdbHandle loadTests(covdbHandle designHandle)
{
    covdbHandle availableTestName; availableTestNameIterator;
    covdbHandle mergedTest;

    // get an iterator to the list of availableTests
    availableTestNameIterator = covdb_iterate(designHandle,
                                              covdbAvailableTests);

    // Using the iterator get the first available test
    availableTestName =
    covdb_scan(availableTestNameIterator);

    // load the first test
    mergedTest = covdb_load(covdbTest, designHandle, ,
                           availableTestName);
    if(mergedTest)
        printf("Successfully loaded test %0s\n",
               covdb_get_str(availableTestName, covdbName));
    else
    {
        printf("Unable to load test\n");
        return NULL;
    }

    // using loadmerge load and merge the remaining tests

```

```

while((availableTestName =
       covdb_scan(availableTestNameIterator)))
{
    mergedTest = covdb_loadmerge(covdbTest, mergedTest,
                                 availableTestName);
    if(!mergedTest)
        printf("Could not load test %s\n",
               covdb_get_str(availableTestName, covdbName));
    else
        printf("Successfully loaded and merged test %s\n",
               covdb_get_str(availableTestName, covdbName));
}
covdb_release_handle(availableTestNameIterator);
return mergedTest;
}

```

The UCAPI method `covdb_release_handle()` function is called to release the iterator `availableTestNameIterator`. This is done because a handle returned from a `covdb_iterate()` call is persistent. Releasing the handle frees the associated memory and prevents a memory leak.

Coverage Correlation : Determining Which Test Covered an Object

This section describes how to use UCAPI to find which test covered a given coverable object. When you merge two or more tests in UCAPI, the information about which test covered which object is retained. You can recover it by using the `covdbTests` relation from a coverable object handle.

Note:

This feature is limited to testbench and assertion coverage metrics only.

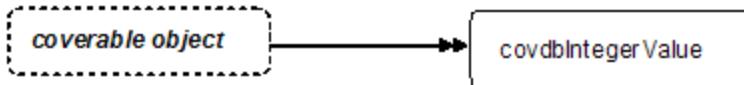
For example:

```

design = covdb_load(covdbDesign, NULL, "mysimv");
test = covdb_load(covdbTest, design, "mysimv/test1");
test = covdb_loadmerge(covdbTest, test, "mysimv/test2");
...
tstsIter = covdb_qualified_object_iterate(binHdl, groupHdl,
                                         testHdl, covdbTests);
while((testCt = covdb_scan(tstsIter))) {
    printf("Bin %s was covered by test %s %d times\n",
           covdb_get_str(binHdl, covdbName),
           covdb_get_str(testCt, covdbName),
           covdb_get(testCt, grpHdl, testHdl, covdbValue));
}
covdb_release_handle(tstsIter);

```

`covdb_qualified_object_iterate(objHdl, regHdl, mergedTestHdl, covdbTests)`



The handle returned by `covdb_scan` for `tstsIter` is a `covdbIntegerValue` handle. Its `covdbName` is the name of the test that was loaded/loadmerged. Its `covdbValue` is the hit count for `binHdl` for that test. For example, if a bin “b1” had the following hits in these different tests:

test	hits
mysimv/test15	5
mysimv/test20	2
mysimv/test32	3
mysimv/test40	1

Then the above code would print the following for b1:

```

Bin b1 was covered by test mysimv/test1 5 times
Bin b1 was covered by test mysimv/test3 2 times

```

The tests that did not cover the bin are not given in the iterator. All the metrics and modes do not track the hit counts. Some will only record that a given object was covered or not covered in a given test. In this case, each test in the list will have a count value of 1 or it will not be in the list.

Metrics or modes that do not support this relation will return a NULL iterator handle when `covdb_qualified_iterate` is called. However, no error will be flagged. Once a merged test handle is saved using `covdb_save`, the information about which of the tests that were loaded or loadmerged covered which object, is discarded by default. That is, if you then reload the saved test, no data about the individual tests that were merged to create it will have been retained. You can change this behavior by using `covdb_configure` to set `covdbKeepTestInfo` to true before you load the initial design as follows:

```
covdb_configure(design, covdbKeepTestInfo,  
                "true");
```

You can use the coverage API `covdbMaxTestsPerCoverable` to set a maximum number of tests for each coverable object. The default value of this API is "3". You can also set `covdbMaxTestCoverable` to override the default value. If set to non-zero, the value that you set for this API becomes the maximum number of tests preserved for each object when multiple tests are loadmerged. For more information about this API, see the chapter *Unified Coverage API Functions* in the *Coverage Technology Reference Manual*.

Index

Symbols

-cm_line contassign [1-10](#)

&&

in condition coverage [1-16](#)

A

annotated source code [4-23, 4-35](#)

assertion argument to the -cm option [2-6](#)

assertion coverage

specifying [2-6](#)

B

blocks

code blocks in line coverage [1-7-??](#)

in line coverage [1-5, 1-11](#)

branch argument to the -cm option [2-6](#)

Branch Coverage [4-35](#)

branch coverage

specifying [2-6](#)

C

case statement

in line coverage [1-10](#)

Chart Linkage [3-104](#)

-cm_line contassign [1-10](#)

-cm_tgl fullintf [3-14](#)

-cm_tgl portsonly [3-15](#)

Code coverage, metrics reported [3-2](#)

compiling for coverage [2-2](#)

cond argument to the -cm option [2-6](#)

condition coverage

defined [1-14, 1-14-??](#)

specifying [2-6](#)

Condition Coverage Detail Window [4-27](#)

conditional expression

in condition coverage [1-14](#)

conditional operator

in condition coverage [1-14](#)

continuous assignment statements

in condition coverage [1-16](#)

coverage database

open [4-2](#)

Coverage Detail Window [4-18](#)

coverage information, summary [3-87](#)

Coverage Map Window [4-17](#)

coverage report, FSM [3-25](#)

Coverage Table Window [4-15](#)

coverage type representation [3-80](#)

coverage, toggle report [3-12](#)

coverage, viewing results [4-13](#)

coveritem [5-79](#)

Customizing URG Charts [3-97](#)

D

dashboard.html [3-81](#)

DVE [5-54](#)

starting [4-2](#)

E

elfile [5-54](#)

environment variable

URG_FAKE_TIME [3-99](#)

environment variables

URG_NO_SESSION_XML [3-99](#)

exclusion, commands [5-51](#)

F

filter coverage display [4-51](#)

forever statement

in line coverage [1-10](#)

fsm argument to the -cm option [2-6](#)

FSM coverage [3-25](#)

criteria for an FSM [1-27](#)

specifying [2-6](#)

FSM transition mode [4-31](#)

G

Generating Trend Charts [3-94](#)

-grade index [5-22](#)

groups

creating coverage [4-21](#)

groups.html [3-86](#)

H

Hierarchical Linkage [3-113](#)

hierarchy.html [3-83](#)

I

if statement

in line coverage [1-5, 1-9](#)

if-else statement

in line coverage [1-9](#)

index-based grading [5-20](#)

indexlimit [5-21](#)

invokingDVE [4-2](#)

L

line argument to the -cm option [2-6](#)

line coverage

defined [1-4](#)

difference from statement coverage [1-8](#)

line coverage, displaying [4-23](#)

load coverage session [4-10](#)

load exclusion state [5-65](#)

Loading and Saving Sessions [4-10](#)

M

map window, coverage [4-17](#)

mapping coverage for subhierarchies also used in another design ??-[5-41](#)

Metric-wide Breakdown Linkage [3-108](#)

modN.html [3-87](#)

module

displaying coverage results by [4-16](#)

monitoring for coverage [2-3](#)

N

Navigating Trend Reports [3-102](#)

navigation menu [3-80](#)

net and register coverage results [4-24](#)

O

objects tables [3-7](#)

Older Session URG Links [3-116](#)

Organization of Trend Charts [3-105](#)

P

pages, report generated [3-2](#)

procedural assignment statement

in condition coverage [1-14](#)

procedural assignment statements

in condition coverage 1-16

R

recalculate the coverage 5-58

report pages 3-2

reported code coverage metrics 3-2

reports

generating 2-4

Running a Quick Start Example 4-2

S

save exclusion state 5-63

sequences view

 FSM 4-32

session.xml file

 suppressing the generation of 3-99

–show brief 3-67

simulating while monitoring for coverage 2-3

source code

 displaying 4-23, 4-35

–srcmap 5-47

start coverage mode 4-2

starting DVE 4-2

statement coverage

 difference from line coverage 1-8

statistics table, formatting 3-6

summary coverage information 3-87

SVA coverage 4-36

T

task definition

 in line coverage 1-9

TCL scripts 4-50

tgl argument to the -cm option 2-6

The 3-7

timestamp, manipulating 3-99

toggle coverage 3-12

 specifying 2-6

Top Level Chart 3-105

-trend option 3-93

Trend Plot 3-95

type, coverage representation 3-80

U

URG

 annotating exclusion 5-97

urg –metric 3-67

URG_FAKE_TIME environment variable 3-99

URG_NO_SESSION_XML environment variable 3-99

user-defined coverage groups 4-21

V

Viewing Coverage Results 4-13

W

wait statement

 procedural delay is interpreted as 1-8

while statement

 in line coverage 1-5, 1-9