

VCS® / VCSI™ LCA Features

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. Verification Planner MS Doc Annotation	
Introduction	18
Use Model	19
Doc Annotator Usage	19
Supported Platforms	21
Compatibility with the Verification Planner Spreadsheet Annotation Flow	21
hvp annotate Command Arguments	22
Capturing a Verification Plan in Doc XML Format	25
Built-In Styles	25
Table Keyword	28
Other Contents	33
Debugging the Doc Plan	34
[Style Table] Keyword Indicator	34
Unique Error Code in Error Messages	36
How to Get Scores Back-Annotated in the Doc	37
Plan/Feature Score Summary	38
Measure Score Table	39
Navigating Down to a Subplan	40

2. XML Doc Plan Generator	
Introduction	41
Use Model	43
Process Flow.....	44
Command Arguments	45
Converting HVP Language to Doc XML	46
HVP Plan	46
HVP Include Files	47
HVP Metric Definitions.....	48
HVP Attribute/Annotation Definition.....	49
HVP Feature/Subplan	50
HVP Measure Statement	52
HVP Assignment and Description	53
Contents Summary	54
3. Using the SVAPP Utility	
Overview	57
Use Model	58
Example using SystemVerilog Testbench	60
Limitations	63
4. IEEE Verilog Standard 1364-2005 Encryption	
The Protection Header File	69
Other Options for IEEE Std 1364-2005 Encryption Mode	72
How Protection Envelopes Work	74
The VCS and VCS MX Public Encryption Key	75

Creating Interoperable Digital Envelopes Using VCS and VCS MX - Example	76
Discontinued -ipkey Option	80
5. Sequential Distance Coverage for Assertions	
Overview	82
Sequential Distance Analysis	83
Rules for Specifying Sequential Distance	86
NTL_COV_ASSERT01	86
NTL_COV_ASSERT02	88
NTL_COV_ASSERT03	89
NTL_COV_ASSERT04	90
NTL_COV_ASSERT05	90
NTL_COV_ASSERT06	91
Rule Output	92
NTL_COV_ASSERT01	92
NTL_COV_ASSERT02	92
NTL_COV_ASSERT03	93
NTL_COV_ASSERT04	93
NTL_COV_ASSERT05	93
NTL_COV_ASSERT06	94
Type of Assertions	94
6. Testbench Separate Compilation	
Overview	96
Use Model	99
Specifying Logical Libraries	100

Analyzing the Code	101
Generating Shared Object Library	106
Generating Shell File	107
Generating simv.	108
Linking Partitions Dynamically at Runtime.	108
 Usage Notes	109
DesignWare VIP	109
Parameterized Programs.	111
Parallel Compilation.	114
Random Stability	114
 Testbench Separate Compile Flow Notes	114
 Testbench-DUT XMR Support.	117
XMR Signal Access	118
XMR Method Access	121
Limitations of XMR Support	124
 NTB OpenVera/SystemVerilog Interoperability	125
Analyzing the Code	125
Generating Shared Object Library	127
Generating Shell File	127
Generating simv.	127
Linking Partitions Dynamically at Runtime.	128
Testbench Separate Compile Flow Notes	128
 Pure NTB OpenVera Flow.	128
 7. Constraints Features	
Using String Indexed Associative Arrays in Constraints	137
Using the array.exists() Function in Constraints	140

8. Coverage Features

Hierarchical Cross Coverage	144
Constituent Covergroup Instances.....	145
Sampling Hierarchical Covergroups	147
Coverage Report-Time Exclusion	149
Checking the Version of an Exclusion File.....	150
Backward Compatibility	154
Saving Exclusions	155
Relative Basic Block ID Change for Line Exclusion.....	155
Impact on Metric Checksums.....	158
Echo Procedural Sampling Enhancement.....	159
Sampling a Non-random Cover Point Variable Assigned with a Random Variable of Another Class	160
Passing a Randomized Object Through a Function/Task Argument	
162	
Stand-alone Covergroups	165
Echo Features	166
Reviewing Exclusions Using Adaptive Exclusion Flow	168
Adaptive Exclusion Review Flow.....	169
Loading an elfile into DVE	170
Understanding the Review Markers in DVE GUI.....	174
Exclusion Reviews in the Navigation Pane	176
Understanding Exclusion Signatures.....	182
Signature in Elf file for Branch Metric.....	187
Exclusion Reviews in the Detail Pane	189
Unmappable Exclusions	192
Coverage Analysis of Unreachable Verilog Code	195
Identifying Conditions in Unreachable Blocks of Verilog Code	196

Specifying Module-Specific Constant Signals in Constant Configuration File.....	198
Module-Specific Constant Limitations	200
Support for with Clause in Cover Groups	202
Cover Point Syntax Enhancement.....	202
Cover Cross Syntax Enhancement	204
Limitations	207
9. Debug Features	
Debug Features.....	210
Enhancement in vpd2vcd Utility.....	210
Reducing Disk Space for Post-process only Debug	210
Reducing Compile Time for Post-process only Debug	212
Debug-DVE Features	213
Using Socket-based Communication in UCLI and DVE	213
Viewing the Full Design Hierarchy in a Partially Dumped VPD	222
Integrating DVE with Protocol Analyzer.....	227
10. Echo	
Diagnostic Mode	232
Echo Targeted Status in the URG Report	232
Runtime Configurable Switches	233
Switches to Split Holes with Multiple Bins	233
Not Annotating Echo Target Mark in covdb	234
Target Priority Change.....	234
Echo all Switches Reference.....	235
Compile-Time Options	236
Echo Option	236

Coverage Model Autogeneration Options	237
Echo Runtime Options	238
URG Options for Bias File Generation	239
Coding Guidelines	240
Constraints and Coverage Model on the Same Variables	240
No Procedural Overwriting of Values Generated by the Solver	241
Coverage Should Be Sampled Between Consecutive Calls to randomize	241
Use Open Constraints	241
Avoid Explicit or Implicit Partitioning in Constraints	242
Avoid In-line Constraints and the Use of “constraint_mode” and “rand_mode”	242
Automatic Generation of a Coverage Model from Constraints	243
Coverage Groups	243
Contribution to Coverage Scoring	251
Coverage Model Inference for In-line Constraints	251
Use Model	251
Understanding Bias Files and Echo	252
Motivation	252
What Is a “Test”?	253
Using Echo Bias File to Target Coverage Holes	254
Automatic Generation of Echo Bias Files	255
Repeatability of Test Results for Parallel Regression Runs	256
Usage Scenarios	257
Running a Single Test with Randomized Configurations	257
Running a Single Test with Randomized Transactions	258
Using a Bias File for a Parallel Regression	258
Autogenerating a Coverage Model	259

Methodology and Flow Issues.....	260
Scenario: All Tests have the same Constraints and Coverage Space (Recommended)	261
Scenario: Tests are Grouped into Categories with each Category having Specific Test Constraints	261
Scenario: Coverage Database Being Loaded in the Beginning of a Test Run	262
Support of Crosses with Non-random Coverage Points	263
Usage Model	263
Procedural Sampling.....	266
Targeting a Non-Random Cover Point Assigned with Random Variables	266
Coverage Model and Randomization in Sibling Classes.....	271
Limitations	273
11. Multicore ALP FSDB Dumping	
Limitations	278
12. VCS Multicore Technology	
Design Level Parallelism (Part 1)	
VCS Multicore Technology Options.....	281
Use Model for Design Level Profiling and Simulation	283
Use Model for Assertion Simulation.....	286
Use Model for Toggle and Functional Coverage	286
Use Model for VPD Dumping.....	287
Running VCS Multicore Simulation	287
Design Level Simulation	287
Assertion Simulation	288
Toggle Coverage	288

Functional Coverage	290
VPD File.....	292
Profiling a Simulation.....	293
Profiling a Serial Simulation.....	293
Specifying Partitions	294
Profiling a VCS Multicore Simulation.....	295
13. VCS Multicore Technology	
Design Level Parallelism (Part 2)	
Running VCS Multicore Examples.....	328
Multicore DLP Autopartitioning	335
Supported Platforms	336
Current Limitations	336
14. New SystemVerilog Features	
Extern Task and Function Calls through Virtual Interfaces	339
Enhancements to the -xlrn uniq_prior_final Compile-Time Option	342
15. Assertion Features	
SystemVerilog Assertions	350
SVA Extensions	350
Limitations	350
Using SystemVerilog Constructs Inside vunits	352
SystemVerilog Constructs Inside Vunits Limitations	353
Using Fail-only Assertion Evaluation Mode.....	354
Fail-only Assertion Limitations	354
Signature-based Control for Deferred Assertions and RT Checks	355

Signature-based Control	356
Limitations	362
16. SystemC Standard Library Classes	
Overview	366
Purpose	366
Brief Introduction to VMM-SC Base Class Library	367
Use Model	391
Using VMM-SC in SystemC Environment	392
Defining Transaction Data Objects Using vmm_data	393
Issuing Message Reports Using VMM Log Macros	394
Options and Configuration Service Using vmm_opts	396
Class Factory Service Using vmm_class_factory Macro	398
Defining Transactors Using vmm_xactor	399
Communication Between Transactors	402
Defining Verification Environment Using vmm_group	402
Extending Phase Methods Having Forever Loops	404
Coordinating Simulation Using vmm_timeline	405
Defining Test Cases Using vmm_test	406
Controlling Tests and Root Components Using vmm_simulation	406
Instantiating VMM-SC Environment in sc_main	407
Using VMM-SC With VMM-SV Interoperability	408
Use Model When vmm_timeline is a Top Level	411
Use Model When vmm_simulation is a Top Level	413
17. SystemC Features	
Simulating Virtual Platform Models	415
Using Platform Architect and Virtualizer MCO	416

What's New in Virtualizer and Platform Architect MCO	417
Debug Flow for Using SystemC with Virtualizer and Platform Architect MCO	424
SystemC Examples	427
SystemC Restrictions.	429
Generating Profile Reports for SystemC Designs	430
Time Profiling	430
Memory Profiling	432
Profiler Example	434
Profile Report Limitations.	439
Compiling SystemC Designs in Partition Compile Flow	439
Using SystemC with Partition Compile	440
SystemC Partition Compile Example.	440
Partitioning your Design.	442
SystemC Partition Compile Limitations	443
Integrating SystemC with Innovator	443
Introduction to Integrating SystemC with Innovator.	444
Single SystemC Compile Flow Enabling RTL Swap-in	445
Platform Analyzer-Controlled Debug with Limited DVE Functionality 446	
18. Using Verification Planner HVP Interactive Editor	
Working with HVP Files.	450
Creating and Editing an HVP File	450
Creating User-Defined Metrics.	453
Creating Feature Nodes	454
Adding User-Defined Attributes	456
Overriding an Attribute.	457
Adding Annotations	457

Goal Overrides	458
Creating a subplan	459
Incompleteness Checks	461
19. Fast Compilation	
20. The Unified Simulation Profiler	
The Use Model	466
Omitting Profiling at Runtime	467
Post Simulation Profile Information	468
Running the profrpt Profile Report Generator	469
Specifying Views	471
The Snapshot Mechanism	474
Specifying Timeline Reports	475
Recording and Viewing Memory Stack Traces	476
The Output Directories and Files	476
HTML Profiler Reports	477
Hypertext Links to the Source Files	505
Single Text Format Report	509
Stack Trace Report Example	509
SystemC Views	512
Constraint Profiling Integrated in the Unified Profiler	520
Changes to the Use Model for Constraint Profiling	520
The Time Constraint Solver View	522
The Memory Constraint Solver View	530
PLI/DPI/DirectC Enhancement in the Unified Profiler	534
Limitations	540

21. Partition Compile	
Partition Compile Use Model	544
Partition Compile Example	546
Cell or Instance Based Partitions	549
Package Based Partitions	550
Limitations of the +optconfigfile Configuration File	551
The Partition Compile Three Step Flow.	551
Specifying Partitions in a V2K or SystemVerilog Configuration	553
Instance or Cell Based Partitions in a V2K/SV Configuration .	555
Specifying a Location for the Partition Data Generation	556
Parallel Compilation of Tests	557
Parallel Compilation of Partitions	559
Cross-Module References (XMRs)	560
Partition Compile Flow for SystemC-on-Top Designs	561
Partitioning Your Design	561
Compiling the SystemC Partition Compile Example	562
Specifying Partitions in a VHDL Configuration File	563
MVSIM Native Mode in Partition Compile	564
Scenarios Causing a Recompilation of a Partition	564
Achieving the Best Turnaround Time (TAT) with Partition Compile	565
Partition Compile Limitations	566
MVSIM Native Mode Limitations	567

[22. Profiling the SystemC Portion of a Design](#)

[23. Multi-Driver Support for Wreal](#)

Introduction	574
Features Supported in this VCS Release	575
Use Model	579
Limitations	579

[24. MVSIM Native Mode Features](#)

Support for Isolation of SVD Constructs	581
Limitations	582
Partition Compile Support	582
Limitations	582
Isolating Interface Modports	583

[25. SAIF Support for SystemVerilog Data Types](#)

Use Model	586
Usage Examples	586
Supported SystemVerilog Data Types	587
SAIF File Format for Unpacked Struct	588

[Index](#)

1

Verification Planner MS Doc Annotation

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

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

Note:

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

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

This section contains the following topics:

- “Introduction” on page 18
- “Use Model” on page 19
- “hvp annotate Command Arguments” on page 22
- “Capturing a Verification Plan in Doc XML Format” on page 25
- “Debugging the Doc Plan” on page 34
- “How to Get Scores Back-Annotated in the Doc” on page 37

Introduction

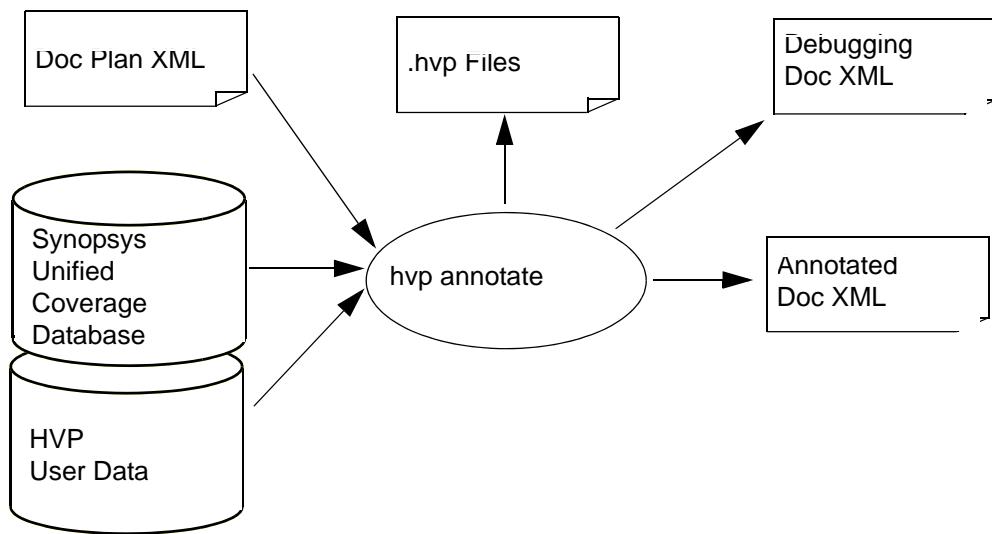
The Verification Planner Doc annotator is used:

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

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

Use Model

Figure 1-1 Word Doc annotator use model



This section contains the following topics:

- “[Doc Annotator Usage](#)” on page 19
- “[Supported Platforms](#)” on page 21
- “[Compatibility with the Verification Planner Spreadsheet Annotation Flow](#)” on page 21

Doc Annotator Usage

MS-Word has many auto-correction options that might cause problems in your Doc verification plan. Before you start editing a Doc verification plan, it is recommended that you turn off auto-correction

options as described below. The following instructions are for MS Word 2003. For other versions of MS Word, turn off the equivalent options.

1. Click the "Office" button at the left-top corner, then-click "Word Options" in the popup menu.
2. Select "Proofing" in the left menu and then click "AutoCorrect Options..." in the right pane.
3. Select the "AutoCorrect" tab, and uncheck each of the "Capitalize ..." options.
4. Select the "AutoFormat" tab, and uncheck "Straight quotes" with "smart quotes".
5. Select the "AutoFormat As You Type" tab, and uncheck "Straight quotes" with "smart quotes".
6. Turn off any other unnecessary options.

Process Flow

1. Capture your verification plan in Doc XML format. This is typically done by starting with a Synopsys-provided template file and marking key paragraphs in the created doc with Synopsys-defined styles, and copying and pasting Synopsys-defined tables.
2. Prepare Synopsys coverage database and HVP user data files.
3. Run **hvp annotate** with the **XML plan**, **covdb**, **userdata** and other desired switches.

4. Fix any errors you see. You can find more error information in `filename.dbg.xml`. Even if you do not see any errors, open the `filename.dbg.xml` file and check to make sure that Verification Planner processed your verification plan hierarchy and contents as you intended. This debug check process only needs to be done on newly created or recently edited plans. Once the plan becomes stable, this debug check step can be skipped when annotating with new coverage databases.
5. If there are no errors, open `filename.ann.xml` to see the annotated scores.

Supported Platforms

RHEL32
RHEL64
suse32
suse64
sparc64
sparcOS5

Compatibility with the Verification Planner Spreadsheet Annotation Flow

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

The `hvp annotate` command is completely backward compatible with the existing Verification Planner Spreadsheet annotator.

hvp annotate Command Arguments

Syntax

```
hvp annotate -plan planfile
[-h]
[-mod hvpfiles]
[-plan_out annfile]
[-feature "hierarchies"|-featurefile txtfile]
[-dir covdbpath|-f txtfile]
[-userdata vedata|-userdatafile txtfile]
[-userdata_out outvedata]
[-metric_prefix prefix]
[-group ratio|merge_across_scopes]
[-show_ratio]
[-show_incomplete]
[-v]
[-q]
```

Options

-plan planfile

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

Example: **-plan myplan.xml**

-h

Show this help message and exit.

-mod hvpfiles

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

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

-plan_out annfile

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

-feature "hierarchy [hierarchy...]"

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

Example:

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

-featurefile txtfile

A text file that contains list of hierarchical filters.

-dir covdbpath

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

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

-f txtfile

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

-userdata vedata

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

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

- userdatafile *txtfile***
Specify a text file that contains a list of user database file paths.
- userdata_out *outvedata***
Dump an annotated score of all measures into the specified *outvedata* user data file.
- metric_prefix *prefix***
The given prefix is used to change a metric name in the output user database file used by -userdata_out.
- group ratio | merge_across_scopes**
 - group ratio:** Aggregate the group metric score as a ratio type (covered/coverable) instead of as a percent.
 - group merge_across_scopes:** Merge the group score across scopes.
- show_ratio**
Display ratio type scores in a ratio form instead of a percent.
- show_incomplete**
Indicate incomplete scores with [inc].
- v**
Verbose mode: Show progress status messages.
- q**
Quiet mode: Turn off all warning messages.

Capturing a Verification Plan in Doc XML Format

This section contains the following topics:

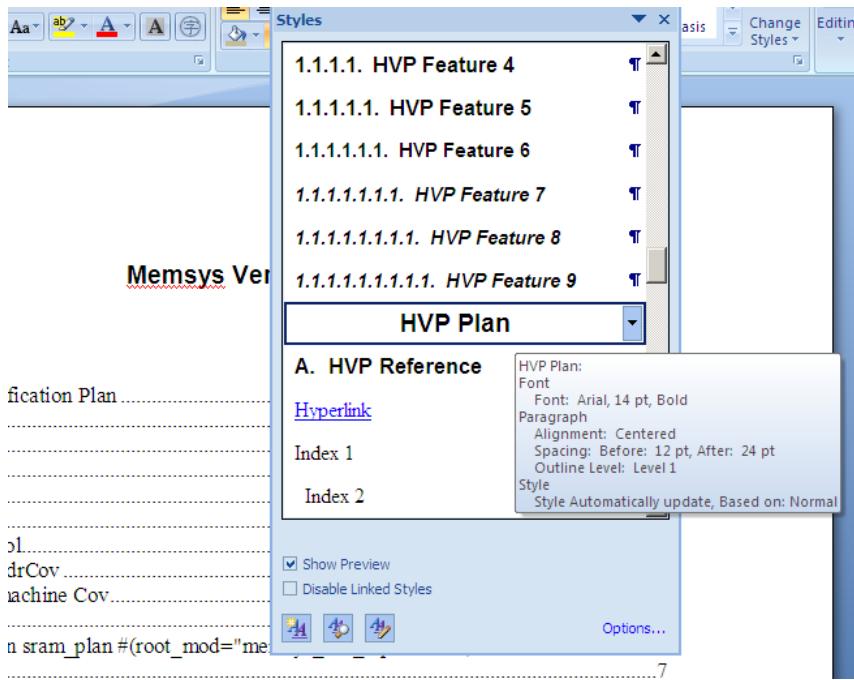
- “Built-In Styles” on page 25
 - “Table Keyword” on page 28
 - “Other Contents” on page 33
-

Built-In Styles

In the Doc XML plan template, there are predefined styles that are prefixed with “HVP”. Verification Planner extracts the contents and their styles to establish the HVP hierarchy. A template document containing predefined styles is provided with the Verification Planner Doc annotator. Make a copy of that template and then use your copy as a basis to create a new plan.

HVP Plan Style

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



Memsys Verification Plan

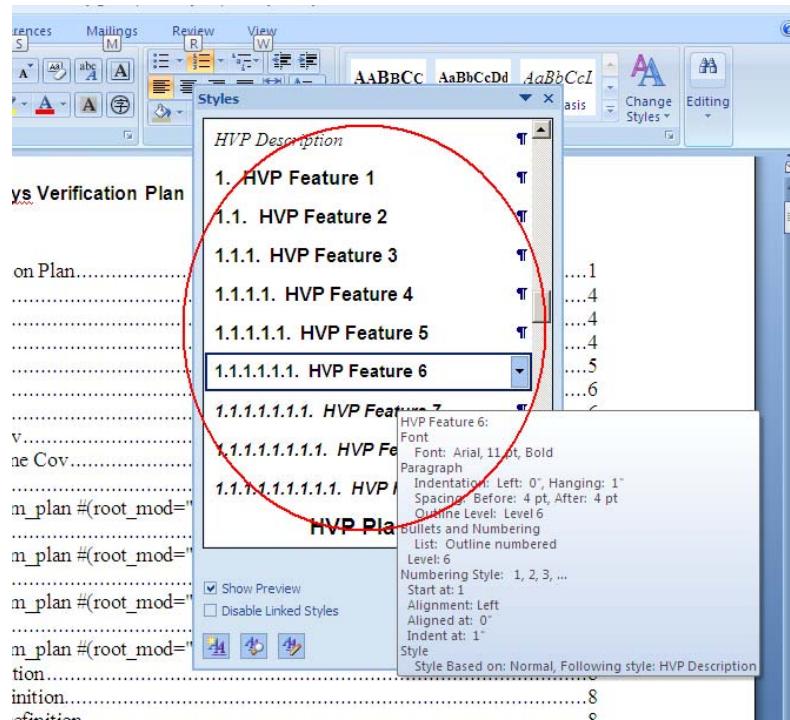
1. Memsys Verification Plan	1
2. 1. CPU_cov	4
3. 1.1. cpu0	4
3. 1.2. cpu1	4
4. 2. Cntrl	5
5. 2.1. code	6
5. 2.2. protocol	6
5. 2.3. busAddrCov	7
5. 2.4. state machine Cov	7
6. 3. memory0	7
7. 3.1. subplan sram_plan #(root_mod=="memsys_test_top.dut.u0")	7
8. 4. memory1	7
9. 4.1. subplan sram_plan #(root_mod=="memsys_test_top.dut.u1")	7
10. 5. memory2	7
11. 5.1. subplan sram_plan #(root_mod=="memsys_test_top.dut.u2")	7
12. 6. memory3	7
13. 6.1. subplan sram_plan #(root_mod=="memsys_test_top.dut.u3")	7

Verification Planner MS Doc Annotation

HVP Feature 1 to HVP Feature 9

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

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



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

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

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

HVP Description

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

Table Keyword

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

- “HVP Assignment” on page 29
- “HVP Measure name” on page 30
- “HVP Metric name” on page 31
- “HVP Attribute name, HVP Annotation name” on page 32
- “HVP Include File” on page 33

HVP Assignment

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

HVP Assignment	
phase	2
owner	Snps

Note:

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

HVP Measure *name*

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

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

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

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

HVP Measure	
source	group: memsys_test_top
source	group: memsys_test_top2
Group	

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

HVP Metric *name*

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

The following are rules for creating an HVP metric table:

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

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

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

Goal is an expression for the coverage goal.

HVP metric bugs	
type	Integer
aggregator	Sum
goal	bugs < 3

HVP Attribute *name*, **HVP Annotation** *name*

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

The following are rules for creating an HVP metric table:

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

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

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

HVP attribute phase	
type	Integer
default	1

HVP annotation spec	
type	String
default	

HVP Include File

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

HVP include file
sram_wordplan.xml
dram_wordplan.xml

Other Contents

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

Debugging the Doc Plan

When an error is detected in a `.doc` verification plan, it is difficult to find where the error occurred. In addition, Verification Planner might interpret your verification plan in a different way than what you intended or expected due to reasons such as typographic errors, incorrect styles, and so on. This is because the paragraph styles are essentially invisible in the normal WYSIWYG view of the document. You might, for example, select the wrong style, not knowing that Verification Planner will not recognize the paragraph properly. When you run `hvp annotate`, a `filename.dbg.xml` is created for each plan file, which provides detailed debugging information.

This section contains the following topics:

- “[Style|Table] Keyword Indicator” on page 34
- “Unique Error Code in Error Messages” on page 36

[Style|Table] Keyword Indicator

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

[HVP Plan]Memsys Verification Plan

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

1. [HVP Feature]CPU_cov

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

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

[HVP Assignment]	
phase	2
owner	Snpa

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

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

[HVP Measure]	
Source	group instance: xxxxxx cpu_cov.my_cpu0::cpu_cov.my_cpu0
Group	

[HVP Measure]	
Source	seabench *
bugs	

3. [HVP Feature]memory0

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

4. [HVP Feature]memory1

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

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

[HVP Attribute]HVP attribute phase	
Type	Integer
Default	1

[HVP Include]HVP include file	
sram_wordplan.xml	

Unique Error Code in Error Messages

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

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

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

How to Get Scores Back-Annotated in the Doc

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

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

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

This section contains the following topics:

- “[Plan/Feature Score Summary](#)” on page 38
- “[Measure Score Table](#)” on page 39
- “[Navigating Down to a Subplan](#)” on page 40

Plan/Feature Score Summary

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

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

For Synopsys coverage,

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

For a test metric,

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

Some examples are shown below.

Memsys Verification Plan

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

1. CPU cov

Group	bugs
100.00%	1

Measure Score Table

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

HVP Measure	
Source	group instance: memsys-test-testbench::cpu::cpu_cov.my_cpu1
Group	100.00%
bugs	N/A

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

HVP Measure	
Source	group instance: memsys-test-testbench::cpu::cpu_cov.my_cpu0
Source	[incomplete]property: non_existing_point
Group	100.00%

Navigating Down to a Subplan

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

3. memory0

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

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

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

2

XML Doc Plan Generator

Introduction

Verification Planner Doc Plan Generator provides a way to convert verification plans in HVP language format to Word Doc XML plan format. The Doc Plans generated from Word Plan generator can be fed into Verification Planner word annotator to annotate scores. This application currently supports MS-Word 2003 and later versions only.

It helps you to generate an annotated report in MS-Word format. In this mode, you run Doc Plan Generator every time you want a new annotated report.

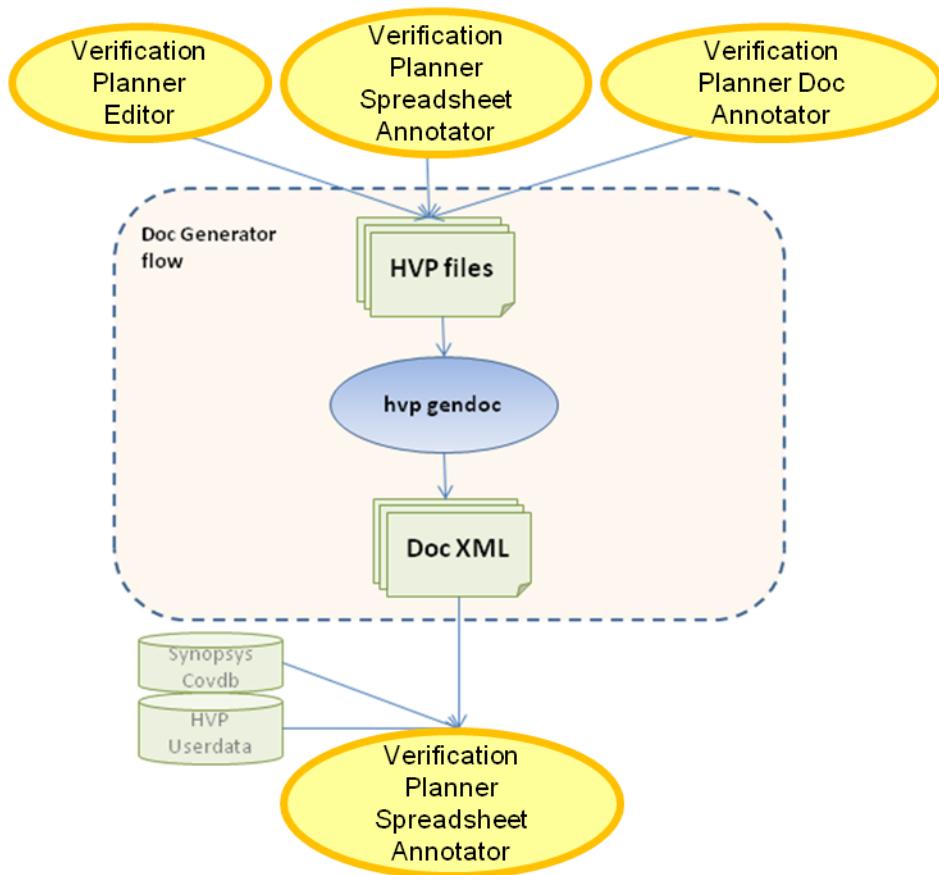
Also, you can make a complete transition from some other medium of maintaining the plans' source (DVE HVP editor or spreadsheet) into MS-Word. In this mode, you run Doc Plan Generator only once and from thereafter you use Doc annotate to generate reports.

The Doc Plan Generator (`hvp gendoc`) simply converts HVP language files to Doc XML format plan document. The output Doc XML is Verification Planner Doc annotation flow compliant format, so you can feed it into the `hvp annotate` command with Synopsys coverage database and user data files to get annotated documents.

The output Doc XML plan contains only a skeleton of feature hierarchy. You might want to add more additional contents such as pictures or descriptions into the documents to make the plan as complete verification report before feeding into the annotation flow.

Use Model

Figure 2-1 Verification Planner Doc Plan Generator Use Model



- `hvp gen -plan <Top Plan HVP file>.`
- One HVP file must contains one plan.
- For example, *MyTopPlan.hvp* will be converted to *MyTopPlan.xml*. A plan can include other plans as subplan, in that case, the HVP files for subplan must be included using ``include` syntax.

- One Doc XML file is generated per each HVP plan file. If multiple HVP files are involved in the given top plan, multiple Doc XML files will be generated accordingly.
- If any included files are not found, `hvp gendoc` skips the missing HVP file with warning. The generated top-plan Doc XML might not work with Doc annotation flow due to the missing file.
- All the Doc XML files are generated in the same directory as the associated HVP files. It means that Doc XML files could be generated in different path if involved HVP files are located in different path. When the `-outdir <path>` option is given, all the output Doc XML files are generated in the given path. In this case, include file path in the Doc XML is reconfigured, so the generated XMLs can work with Verification Planner annotator without additional editing.

Process Flow

1. Capture verification plan in HVP language using Verification Planner GUI Editor, Verification Planner Spreadsheet annotator or manual editing in Text Editor. One plan in one HVP file.
2. `hvp gendoc –plan <top plan HVP file>`
3. `[-outdir <output path>]`
4. Open the generated Doc XML files in MS-Word 2003 or later, then edit them if needed.
5. Apply the Doc XML files to Verification Planner Doc annotator.

Note:The Doc Plan Generator flow is not fully interoperable with DVE HVP edit. This means, you cannot generate a doc, add more comments, diagrams and other details to the doc, generate .hvp and then edit that HVP in DVE. If you do so, you will lose your added comments, diagrams, and other details as these are not recorded in the .hvp file.

Command Arguments

Syntax:

```
hvp gendoc -plan TopPlan.hvp [-outdir output_path]
```

options:

-h

show this help message and exit

-plan <TopPlan.hvp>

Top plan HVP file.

Example

```
-plan myplan.hvp  
-outdir <outpath>
```

output path for the Doc XML files generated. Must exist before run.

Converting HVP Language to Doc XML

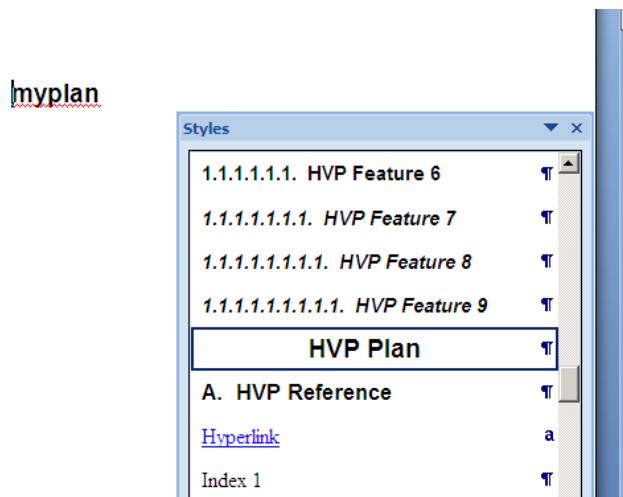
Basically, the Doc XML is using pre-defined styles in Doc Plan template, so generated files will look similar to the Doc XML Plan for Verification Planner doc annotator.

For more details about Doc XML format, see Verification Planner User Guide.

HVP Plan

```
plan myplan;  
....  
....  
endplan
```

At the first page of Doc XML.



The name of plan should be in “HVP Plan” style.

HVP Include Files

```
`include "subplan_sub1.hvp"  
`include "subplan_sub2.hvp"  
plan MyPlan;  
....  
....  
endplan
```

At the last page of Doc XML



A. Include File

HVP Include File
subplan_sub1.xml
subplan_sub2.xml

Note:

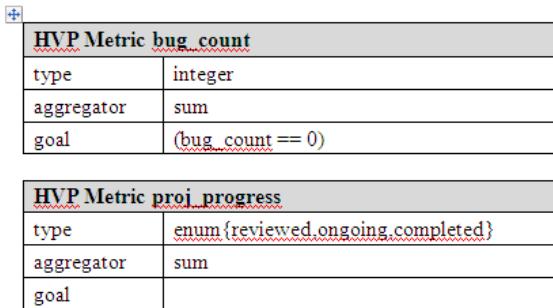
The text in the first cell of table must be “HVP Include File” and style is not specified. The include files are listed from the second row. One in each cell.

HVP Metric Definitions

```
metric integer bug_count;
    aggregator = sum;
    goal = bug_count == 0;
endmetric
metric enum{reviewed, ongoing, completed} proj_progress;
    aggregator = sum;
endmetric
```

At the last part of Doc XML plan, following tables are added.

A. Metric Definition



The screenshot shows a software interface with two tables under the heading "A. Metric Definition".

HVP Metric bug_count

type	integer
aggregator	sum
goal	(bug_count == 0)

HVP Metric proj_progress

type	enum {reviewed,ongoing,completed}
aggregator	sum
goal	

Note:

Each metric definition is defined in a separate table. The first row is composed of single column cell which contains “HVP Metric < name>”. From the second row, each row is composed of two columns. The left column has a keyword, and the right cell has corresponding value string. The style of the text is not specified.

HVP Attribute/Annotation Definition

```
attribute integer phase = 1;
attribute enum{orange, apple, apricot} fruit = orange;
annotation string comment = "";
```

At the last part of Doc XML plan, the following tables are added:

B.Attribute Definition

HVP Attribute phase	
type	integer
default	1

HVP Attribute fruit	
type	enum{orange,apple,apricot}
default	orange

HVP Annotation comment	
type	string
default	

Note:

Each attribute/annotation definition is defined in a separate table. The first row is composed of single column cell which contains HVP Attribute < name> or HVP Annotation <name>. From the second row, each row is composed of two columns. The left column has a keyword, and the right cell has corresponding value string. The style of the text is not specified.

HVP Feature/Subplan

The text of HVP feature/subplan hierarchy uses “HVP Feature1” – “HVP Feature9” depending on the depth of hierarchy. A feature under plan definition is level 1. A new page is always followed by level- 1 features.

```
plan Wishbone;
    feature InputOutput;
        feature ReadIO;
    endfeature
    feature WriteIO;
    endfeature
endfeature
.....
endplan
```



This might look simple, because there is nothing between the features. Normally, between two features, there will be measure tables, attribute/goal assignment tables or some descriptions for features.

If an object of hierarchy is a subplan instead of a feature, the title should be “subplan <name>” as follows.

```
`include "ReadIO.hvp"
`include "WriteIO.hvp"

plan Wishbone;
    feature InputOutput;
        subplan ReadIO;
        subplan WriteIO #(phase=2, owner="Verification Group");
    endfeature
    ....
endplan
```

The plan ReadIO and WriteIO must be defined in included HVP files before they are used as subplan.



Because the subplan ReadIO is at level 2, it uses the “HVP Feature 2” style.

HVP Measure Statement

A measure statement is described in a table. The first row is composed of single column with “HVP Measure <name>” text. The first row is followed by a few two-column rows with “source” at the left cell. The source rows are followed by other two-column rows with metric name at the left cells.

```
feature ReadIO;
    measure Line, Cond mReadComponent
        source = "tree: top.Wishbone.Video.read", "tree:
top.Wishbone.Audio.read";
    endmeasure
    ....
```

1.1. ReadIO	
HVP Measure mReadComponent	
source	tree: top.Wishbone.Video.read
source	tree: top.Wishbone.Audio.read
Line	
Cond	

HVP Assignment and Description

The HVP language allows you to assign attribute/annotation values and goal expression for metric at each level of plan/feature hierarchy. These value assignments are described in a table with “HVP Assignment” at the first cell. The name of attribute/metric and associated values are followed in the next row. One assignment at each row.

One exception is the `description` built-in annotation value. If the `description` value is assigned, it appears at the right next line of feature title in HVP Description style instead of HVP Assignment table.

```
feature ReadIO;
    description = "This feature is testing Read interface of Wishbone
chipset."
    owner = "Verification Group"
    phase = 2
    Line = Line > 95%;
    Group = Group > 90%;
```

1.1. ReadIO

This feature is testing Read interface of Wishbone chipset.

HVP Assignment	
owner	Verification Group
phase	2
Line	Line > 95%
Group	Group > 90%

HVP Description	#
1. HVP Feature 1	#
1.1. HVP Feature 2	#
1.1.1. HVP Feature 3	#
1.1.1.1. HVP Feature 4	#
1.1.1.1.1. HVP Feature 5	#

Contents Summary

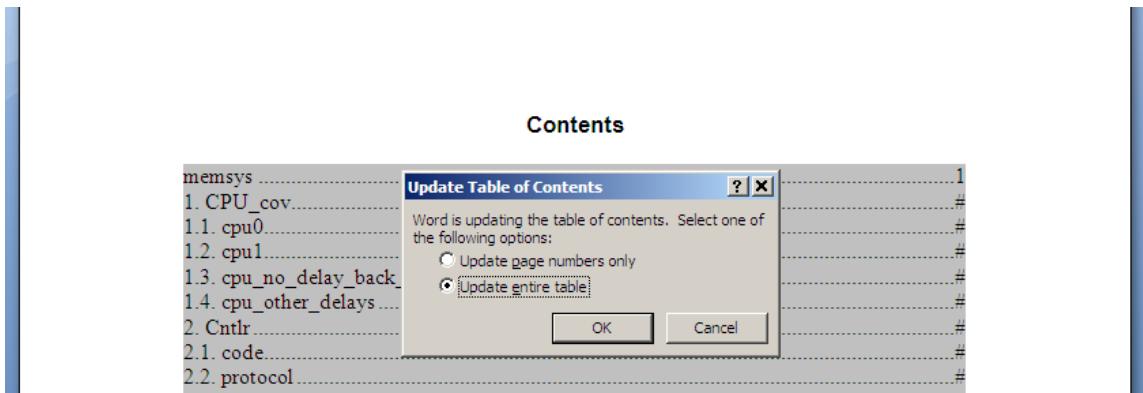
At second page of Doc XML, a summary of hierarchy appears as follows. This summary contains the plan title, features, metric/attribute definition and include file information if they exist.

When the doc XML is generated, the page number and hyperlinks are not resolved completely because they cannot be determined in Verification Planner. As you see in the following figure, the page numbers show up as “#” instead of real number except plan.

Contents

memsys	1
1. CPU_cov.....	#
1.1. cpu0.....	#
1.2. cpu1.....	#
1.3. cpu_no_delay_back_to_back	#
1.4. cpu_other_delays	#
2. Cntrl	#
2.1. code	#
2.2. protocol	#
2.3. busAddrCov	#
2.4. state_machine_Cov	#
3. memory0	#
3.1. subplan sram_plan #(root_mod="memsys_test_top.dut.u0.")	#
4. memory1	#
4.1. subplan sram_plan #(root_mod="memsys_test_top.dut.u1.")	#
5. memory2	#
5.1. subplan sram_plan #(root_mod="memsys_test_top.dut.u2.")	#
6. memory3	#
6.1. subplan sram_plan #(root_mod="memsys_test_top.dut.u3.")	#
A. Metric Definition	#
B. Attribute Definition	#
C. Include File	#

To resolve them, you can simply right-click, then choose “Update Field” at popup-menu, then select “Update Entire Table”.



Then, the right page number will show up as below.

Contents	
memsys ..	1
1. CPU_cov.....	4
1.1. cpu0.....	4
1.2. cpu1.....	4
1.3. cpu_no_delay_back_to_back.....	4
1.4. cpu_other_delays.....	5
2. Cntrl.....	6
2.1. code.....	6
2.2. protocol.....	6
2.3. busAddrCov.....	6
2.4. state_machine_Cov.....	7
3. memory0	8
3.1. subplan sram_plan #(root_mod="memsys_test_top.dut.u0.");	8
4. memory1	9
4.1. subplan sram_plan #(root_mod="memsys_test_top.dut.u1.");	9
5. memory2	10
5.1. subplan sram_plan #(root_mod="memsys_test_top.dut.u2.");	10
6. memory3	11
6.1. subplan sram_plan #(root_mod="memsys_test_top.dut.u3.");	11
A. Metric Definition	12
B. Attribute Definition	12
C. Include File	12

3

Using the SVAPP Utility

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

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

Overview

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

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

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

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

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

Use Model

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

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

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

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

The arguments and options are as follows:

`assertions_filenames`

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

`-vpdin filename`

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

`-vpdout filename`

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

`-vcsflags "compile-time_options"`

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

`-simflags "runtime_options"`

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

`-h`

Displays the required and optional arguments and options.

`-f filename`

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

`-o filename`

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

Example using SystemVerilog Testbench

The following is an example of using SVAPP.

This my.sv file contains a SystemVerilog testbench:

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

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

This my.v file contains a design module:

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

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

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

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

This assert.sva file contains the following SVA code:

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

a : assert property(p);
endmodule

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

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

```

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

endmodule

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

```

Run SVAPP with the following command line:

```
svapp assert.sva
```

SVAPP displays the following SVA messages:

```

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

```

```

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

```

Limitations

SVAPP has the following limitations:

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

```

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

```

    endfunction

endmodule

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

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

```

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

```

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

endmodule

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

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

```

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

```
module assert_mod (input clk, input reset);
```

```

.
.
.

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

a2 : assert property (p2);

endmodule

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

```

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

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

```

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

```

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

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

- SVAPP does not support dynamic variables.
- For VCS MX you cannot bind an SVA module to a VHDL entity. Bind the SVA module to a Verilog module that instantiates the VHDL entity.

4

IEEE Verilog Standard 1364-2005 Encryption

VCS and VCS MX supports encryption of Verilog and SystemVerilog IP code in protected envelopes as defined by the IEEE Standard 1364-2005.

In addition, VCS and VCS MX supports the recommendations from the IEEE P1735 working group for encryption interoperability between different encryption and decryption tools, denoted as “version 1” by P1735.

Note:

VHDL and SystemC encryption is not supported by this feature.

The following option tells VCS or VCS MX to encrypt the specified Verilog or SystemVerilog source files according to the “IEEE Std 1364-2005” standard for encryption envelopes.

```
-ipprotect protection_header_file
```

In this encryption mode, VCS and VCS MX does not compile the Verilog or SystemVerilog source files, but instead encrypts each source file into a separate encrypted Verilog or SystemVerilog file. Each encrypted file is saved under the same filename, but changes its filename extension to .vp. Using the -ipprotect option allows IP providers to specify a *protection_header_file* that contains various protection pragmas.

VCS and VCS MX encrypts:

- the source files on the vcs command line
- the source files specified in `include compiler directives.

Note:

- By default VCS and VCS MX encrypt complete input files. Use the -ipopt=partialprotect option and argument to enable partial protection, with it VCS or VCS MX encrypt only the regions specified by `protect pragma begin-end expressions.
- All `include directives in the encrypted source files are modified by changing the extension of the included filenames from .v to .vp. The modified `include directives are left as unencrypted text. In addition, every file included by a `include directive is also encrypted and saved under the modified filename (changing the extension to .vp). Use the -ipopt=noincludeprotect option and argument with the -ipprotect option to disable processing of `include compiler directives and the source files included by it.

This section on the IEEE Std 1364-2005 encryption mode includes the following:

- “The Protection Header File” on page 69
 - “Other Options for IEEE Std 1364-2005 Encryption Mode” on page 72
 - “How Protection Envelopes Work” on page 74
 - “The VCS and VCS MX Public Encryption Key ” on page 75
 - “Creating Interoperable Digital Envelopes Using VCS and VCS MX - Example” on page 76
 - “Discontinued -ipkey Option” on page 80
-

The Protection Header File

The *protection_header_file* may look like the following:

Example 4-1 Sample IEEE Encryption Header File

```

`pragma protect data_method = "aes128-cbc"
`pragma protect encoding = (enctype = "base64")
`pragma protect key_keyowner="Synopsys"
`pragma protect key_method="rsa"
`pragma protect key_keyname="SNPS-VCS-RSA-1"
`pragma protect key_public_key
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDJJMv7PI1V+DJDaHZuVI
FbAXvr6tEpuM8cAKFuvpIoO6PE3DRqEwaHEJRyIsFnJnavVJ33+Kub54Cr
9JCCh6fnQhtAmKtnAznESOLExCKO1tmjYNCXLJ+QqWFoCuDuI4QS8Ruy1u3
RwABCw7ESQwwIuVsZpOgh0vjrPHzvlc0QIDAQAB

```

The following `pragma protect expressions are required inside the *protection_header_file*:

key_keyowner

Identifies the owner of the key encryption key.

`key_method`

Specifies the key encryption algorithm, the asymmetric method for encrypting or decrypting.

`key_keyname`

Specifies keyowner's key name.

`key_public_key`

Specifies the public key for key encryption

The optional ``pragma protect` expressions that can be included are as follows:

`data_method`

Identifies the data encryption algorithm. Supported methods are as follows:

`aes256-cbc`

`aes192-cbc`

`aes128-cbc`

`des-cbc`

`3des-cbc`

The default `data_method`, if none is specified, is `aes256-cbc`.

`author`

Identifies the author of an envelope.

`author_info`

Specifies additional author information.

`encoding`

Specifies the coding scheme for encrypted data, you can specify either of the following:

base64

uuencode

The default encoding scheme, if none is specified, is base64.

comment

Comment documentation string that is not encrypted.

Note:

These encryption pragmas are only supported inside the *protection_header_file*, which is specified by the -ipprotect option. If they are specified anywhere else (such as in the Verilog or SystemVerilog source files), VCS or VCS MX outputs a warning message and ignores the pragma.

The only 'pragma protect expressions allowed in input Verilog and SystemVerilog files are 'pragma protect begin and 'pragma protect end, when enabled with the -ipopt=partialprotect option and argument to mark the regions to be protected.

Unsupported Protection Pragma Expressions

The `pragma protect expressions that are not currently supported include:

data_keyowner
data_public_key
decrypt_license
reset

data_keyname
data_decrypt_key
runtime_license
viewpoint

Also unsupported are any expressions beginning with digest_.

Other Options for IEEE Std 1364-2005 Encryption Mode

In addition to the `-ipprotect` option, there are other options that you can use in this mode. This section describes them.

`-ipopt=partialprotect`

VCS and VCS MX encrypts complete file by default. Use this option to encrypt only regions marked by the pragmas: ``pragma protect begin` and ``pragma protect end` in the Verilog or SystemVerilog source files.

`-ipopt=noincludeprotect`

VCS and VCS MX in encryption mode encrypt files which are included by the ``include` compiler directive. Use this option to disable the processing of the ``include` compiler directive and files included by it.

`-ipopt=ext=ext`

Use this option to specify the filename extension for encrypted files.

`-ipopt=outdir=dir`

Use this option to specify the target directory for encrypted files.

`+incdir+directory+...`

Specifies the directories that VCS or VCS MX searches for source files specified with the `'include` compiler directive. By default VCS or VCS MX writes encrypted versions of these source files in the directory in which it finds the source files.

The encrypted copies have the same filename and extension of the original except that the `p` character is appended to the filename extension. So for example if it finds a SystemVerilog source file in a Verilog library with the name `dev1.sv`, the encrypted version in that library is `dev1.svp`.

You can specify multiple Verilog libraries with this option by using the plus (+) character as a delimiter, for example:

`+includer+INTRCTR+IOMTR+/DW/SIMENV`

`-f | -F | -file filename`

Specifies a file that contains a list of Verilog or SystemVerilog source files to be encrypted. The `-f`, `-F`, and `-file` options are interchangeable in this encryption mode.

`+define+MACRO=VALUE`

Defines the specified text macro to the specified value.

A text macro so defined at encryption time (when encrypting files instead of compiling files) cannot be overridden at a subsequent compile-time (when including the encrypted files in some later compilation and also entering the `+define` option). VCS and VCS MX ignore the attempted override without displaying any error, warning, or informational message.

`-ipout filename.ext`

This option tells VCS and VCS MX to write the encrypted file for the first Verilog or SystemVerilog source file on the command line with the specified filename and extension. You can enter a pathname for the protected file.

This option only works for the first Verilog or SystemVerilog source file on the `vcs` command line, and does not work for other source files on the command line or files included with the `'include` compiler directive or in Verilog libraries.

How Protection Envelopes Work

As specified in IEEE Std 1364-2005, annex H “Encryption/decryption flow,” section H.3 Digital envelopes:

“The sender encrypts the design using a symmetric key encryption algorithm and then encrypts the symmetric key using the recipient’s public key. The encrypted symmetric key is recorded in a `key_block` in the protected envelope. The recipient is able to recover the symmetric key using the appropriate private key and then decrypts the design with the symmetric key.”

Protection envelopes work as follows:

1. The encrypting tool generates a random key called "session key."
2. The encrypting tool then encrypts the design using this session key.
3. For each potential decrypting tool, information about that tool must be provided using `'pragma protect` expressions in the encryption envelope.

This information includes `key_keyowner`, `key_keyname`, the asymmetric `key_method`, and `key_public_key` for each tool.

4. The encrypting tool then encrypts the session key multiple times, once for each decrypting tool using information provided in the encryption envelope for that tool.

5. The encrypted session key is then recorded in `key_blocks` in the protected envelope.

Multiple `key_blocks` are generated, one for each decrypting tool.

6. The decrypting tool examines the `key_blocks` in the decryption envelope to find one encrypted using a key to which the tool has access.
7. The decrypting tool is able to recover the session key from its `key_block` using the appropriate private key.
8. The decrypting tool then decrypts the design with the session key.

The VCS and VCS MX Public Encryption Key

The VCS and VCS MX base64 encoded RSA public key is:

```
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDjJMv7PI1V+DJDaHZuVI  
FbAXvr6/  
tEpuM8cAKFuvpIoO6PE3DRqEwaHEJRyIsFnJnavVJ33+Kub54Cr/  
9JCCh6fnQhtAmKt/  
nAznESOLExCKO1tmjYNCXLJ+QqWFoCuDuI4QS8Ruy1u3RwABCw7ESQwwIu  
VSZpOghOvjrPHzvlc0QIDAQAB
```

The following `pragma protect expressions will identify this key:

```
`pragma protect key_keyowner="Synopsys"  
`pragma protect key_method="rsa"  
`pragma protect key_keyname="SNPS-VCS-RSA-1"
```

VCS and VCS MX can decrypt and compile source files which are encrypted by VCS and VCS MX or third party tools.

To allow VCS and VCS MX to decrypt encrypted source files, the following snippet must be included while encrypting.

```
`pragma protect key_keyowner="Synopsys"  
`pragma protect key_method="rsa"  
`pragma protect key_keyname="SNPS-VCS-RSA-1"  
`pragma protect key_public_key  
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDjJMv7PI1V+DJDaHZuVI  
FbAXvr6/  
tEpuM8cAKFuvpIoO6PE3DRqEwaHEJRyIsFnJnavVJ33+Kub54Cr/  
9JCCh6fnQhtAmKt/  
nAznESOLExCKO1tmjYNCXLJ+QqWFoCuDuI4QS8Ruy1u3RwABCw7ESQwwIu  
VSZpOghOvjrPHzvlc0QIDAQAB
```

The following example illustrates the protection envelope methodology for using this key in Verilog or SystemVerilog source code.

Creating Interoperable Digital Envelopes Using VCS and VCS MX - Example

VCS and VCS MX allows more than one key_block in a single protected envelope so it can be decrypted by tools from different vendors.

In the following example an IP provider created encrypted source files that can be decrypted by two different EDA tools, VCS and tools from VendorX.

An IP provider retrieves public keys for an EDA tool from its documentation. For VCS or VCS MX it is this section on IEEE Verilog Std 1364-2005 Encryption. For other tools an IP provider might need to contact its vendor.

The *protection_header_file* that this example specifies with the *-ipprotect* option is in [Example 4-2](#).

Example 4-2 Example Protection Header File for Source Encryption with VCS

```
`pragma protect author = "IP Provider"
`pragma protect data_method = "aes128-cbc"
`pragma protect encoding = (enctype = "base64")

`pragma protect key_keyowner="Synopsys"
`pragma protect key_method="rsa"
`pragma protect key_keyname="SNPS-VCS-RSA-1"
`pragma protect key_public_key
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDjJMv7PI1V+DJDaHZuVIFbAXvr
6/tEpuM8cAKFuvpIoO6PE3DRqEwaHEJRyIsFnJnavVJ33+Kub54Cr/9JCh6fnQht
AmKt/nAznESOLExCKO1tmjYNCXLJ+QqWFoCuDuI4QS8Ruy1u3RwABCw7ESQwwIuV
SzpOghOvjrPHzv1c0QIDAQAB

`pragma protect key_keyowner="VendorX"
`pragma protect key_method="rsa"
`pragma protect key_keyname="VENDORX-RSA-1"
`pragma protect key_public_key
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCpMxgDC3D0A3Jcd/tsrH9LyvUK
KiIB5aG+6aGYNm5ngvXgxc8SShMJw0Huye6RSkUtEklJ4AypbBEPGZLZFBYFxZsS
9yLDetHA8GIGsQLRiqKJeRtrQsPb9rYzF9me4JwuCqFe8VW7lBhSfO6gNo5Lk1S4
hB2jvNCz/TLBD3ydZwIDAQAB
```

(1) (2)

(1) key block for VCS
(2) key block for VendorX

Example 4-3 Verilog Source File to be Encrypted

```
// example.v
module secret (a, b);
    input a;
    output b;
```

```

reg b;

initial
begin
    b = 0;
end

always
begin
    #5 b = a;
end
endmodule

```

The following vcs command line generate the encrypted file example.vp which can be decrypted by VCS and tools from VendorX.

```
vcs -ipprotect pragma_header_file example.v
```

Example 4-4 example.vp generated by VCS

```

`pragma protect begin_protected
`pragma protect version=1
`pragma protect encrypt_agent="VCS"
`pragma protect encrypt_agent_info="G-2012.09-A[D]
(ENG) Build Date Feb 18 2012 00:14:12"
`pragma protect author="IP Provider"
`pragma protect key_keyowner="Synopsys"
`pragma protect key_keyname="SNPS-VCS-RSA-1"
`pragma protect key_method="rsa"
`pragma protect encoding = (enctype = "base64",
line_length = 76, bytes = 128 )
`pragma protect key_block
fCY5ZM3A757rFLqRV/
Lk+hy8kPqXMnJ5rmr53Jnyv7u8nCxHUAzVqzmvWhp2pNwbJ+N0
6jmd/
GF3KIexxU1D2nkF+tQEAtBAnHBvxweFAsBa43s1hRJW6TgXGDF
Fktg5qa2b9c1RWl92AggGSqmS+a1btkTZJ7PTjfanUrvtF3g=

```



```

`pragma protect key_keyowner="VendorX"
`pragma protect key_keyname="VENDORX-RSA-N1"
`pragma protect key_method="rsa"
`pragma protect encoding = (enctype = "base64",
line_length = 76, bytes = 128 )
`pragma protect key_block
pNpqXq9REx09UGv+o62OuOYvoyf4mVIDIoaYfyZ6WDOEXZJq3rR
eZu+Jys7JYUhUkHo638PP03pmnEasZjPXi9MqR/
tWCNeva5Ly0bEnkl2mrqxqv0svporedEyFx3swyQ48Kzq76rU7Qs
x1Lz+mN3m97aaD/WusVe/Z0ozXtVo=

```

```

`pragma protect data_method="aes128-cbc"

`pragma protect encoding = (enctype = "base64",
line_length = 76, bytes = 176 )
`pragma protect data_block
+MW2QpXLShFRtT83KhWLbmtcbKLE6jtCrr68RuPfNGys4r5cLDT
NGgytecJ1Br7WF6MXns6NjRxpB7ZMePn/
75UpcyVVUd3hOMVLVvQ+rrWtzVIPWa8td/
wvRA1qhQHVRc3QvW9UJWvOoAj6+6KPEi4TbZwMVFX5g/
J3XN4xASqClubQp+9sR2PJrpwWc3K
RN5dOZaq6Hmr0LVNbraNY4O8JwzNLOrR3gcQSul/86U=

```

- ① Key block for VCS which contains the encrypted session key.
(encrypted using VCS public RSA key)
- ② Key block for VendorX which contains the encrypted session key.
(encrypted using VendorX public RSA key)
- ③ Data block which contains the encrypted IP (encrypted using the session key)

To determine the session key that was used to encrypt the data_block:

- VCS retrieves the session key from first `key_block`
- VendorX uses the second `key_block`

Consequently, both implementations could successfully decrypt the data block which contains the encrypted IP.

Discontinued -ipkey Option

The `-ipkey key` option will be obsolete in future releases.

IP providers should use `-ippprotect` instead. It allows you to specify various protection pragmas (via a protection header file) which are needed while generating interoperably encrypted IPs.

VCS and VCS MX will no longer use the key you pass with the `-ipkey key` option. It will generate a secure key internally.

5

Sequential Distance Coverage for Assertions

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

This feature is supported for designs written in

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

Note:

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

Overview

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

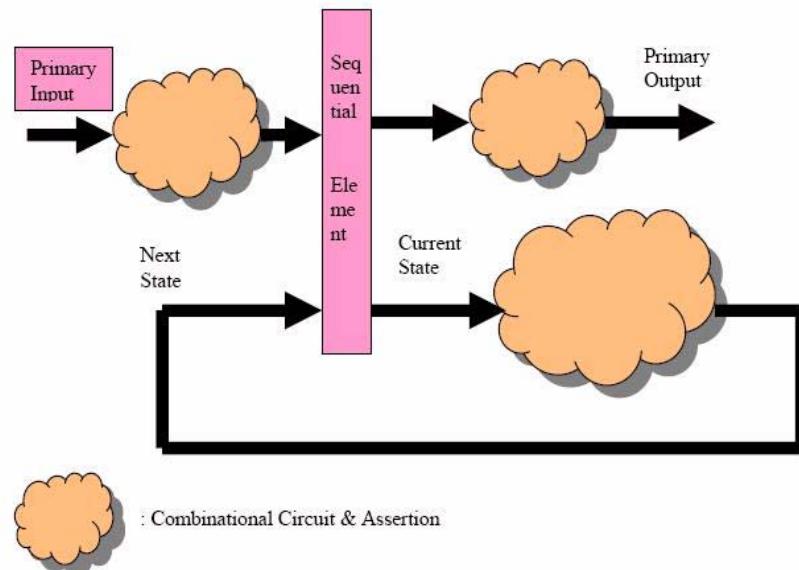
For any complex design, it is not practical to write assertions such that all signals are included in one or more assertions. The number of assertions required for such inclusion of all signals is neither productive nor provides effective use of resources to test a design. Given these consideration,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

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

Sequential Distance Analysis

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

Figure 5-1 Structural view of a digital circuit



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

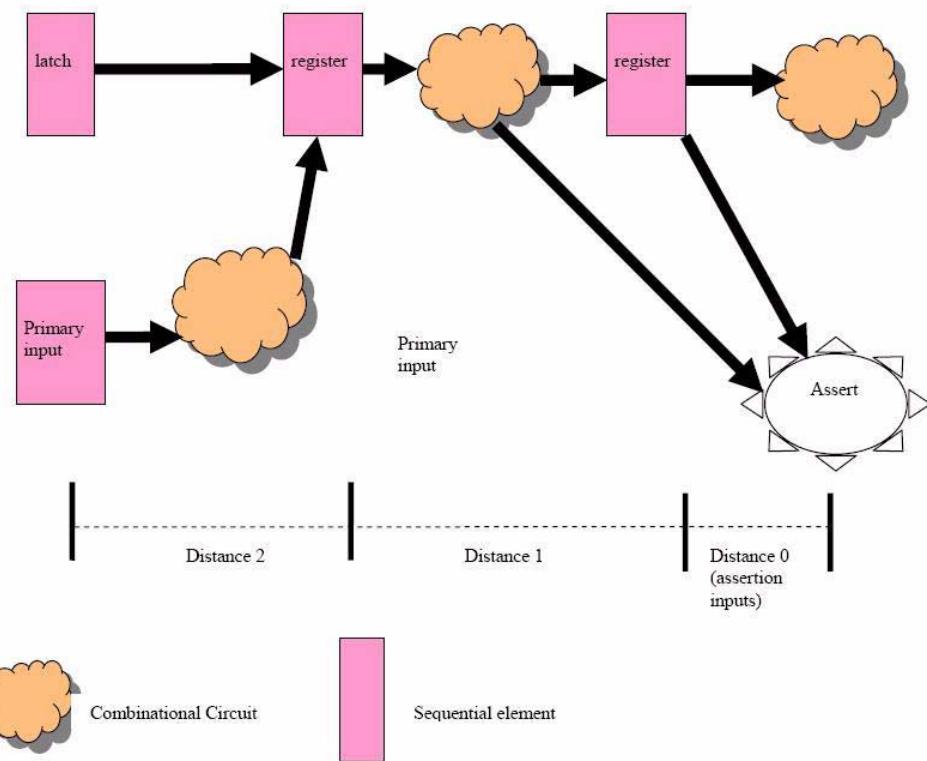
Sequential distance from a sequential element to an assertion is the minimum number of sequential elements in any path from an input of the assertion to that sequential element, excluding the assertion input. The sequential element itself is counted in the distance. For example, if there is no sequential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

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

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

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

Figure 5-2 Sequential Distance



From this analysis, following information is of prime interest:

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

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

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

Rules for Specifying Sequential Distance

The following rules control the specification of sequential distance.

NTL_COV_ASSERT01

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

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

DEPTH

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

Value: Integer

Default: Any depth up to the primary inputs

Usage: Optional

INCLUDE_COVER_ASSERT

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

Value: true or false

Default: false

Usage: Optional

INCLUDE_LATCH

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

Value: true or false

Default: true

Usage: Optional

STATS_ONLY

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

Value: true or false

Default: false

Usage: Optional

SORT_DEPTH

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

Value: true or false

Default: false

Usage: Optional

Output reporting occurs in one of two ways:

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

Examples of the Tcl commands for the configuration file:

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

NTL_COV_ASSERT02

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

ELEMENT

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

Default: None

Usage: Mandatory

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

Examples of the Tcl commands for the configuration file:

```
set_rule -rule NTL_COV_ASSERT02
set_rule_parameter -rule NTL_COV_ASSERT02 -parameter ELEMENT -value
"top.m1.reg1"
```

NTL_COV_ASSERT03

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

DEPTH

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

Value: Integer

Default: Any depth up to the primary inputs

Usage: Optional

MAXASSERTS

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

Value: Integer

Default: None

Usage: Optional

Examples of the Tcl commands for the configuration file:

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

NTL_COV_ASSERT04

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

DEPTH

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

Value: Integer

Default: Any depth up to the primary inputs

Usage: Optional

ASSERTION_NAME

Defines the path name of the assertion.

Value: Hierarchical path to the assertion

Default: None

Usage: Mandatory

Examples of the Tcl commands for the configuration file:

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

NTL_COV_ASSERT05

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

Example of the Tcl command for the configuration file:

```
set_rule -rule NTL_COV_ASSERT05
```

NTL_COV_ASSERT06

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

ASSERTION_NAME

Defines the path name of the assertion.

Value: Hierarchical path to the assertion

Default: None

Usage: Mandatory

DEPTH

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

Value: Integer

Default: Any depth up to the primary inputs

Usage: Optional

SIZE

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

Value: Integer

Default: None

Usage: Mandatory

Examples of the Tcl commands for the configuration file:

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

Rule Output

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

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

NTL_COV_ASSERT01

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

NTL_COV_ASSERT02

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

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

NTL_COV_ASSERT03

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

NTL_COV_ASSERT04

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

NTL_COV_ASSERT05

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

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

NTL_COV_ASSERT06

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

Type of Assertions

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

6

Testbench Separate Compilation

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

- “Overview” on page 96
- “Use Model” on page 99
- “Usage Notes” on page 109
- “Testbench Separate Compile Flow Notes” on page 114
- “Testbench-DUT XMR Support” on page 117
- “NTB OpenVera/SystemVerilog Interoperability” on page 125
- “Pure NTB OpenVera Flow” on page 128

Overview

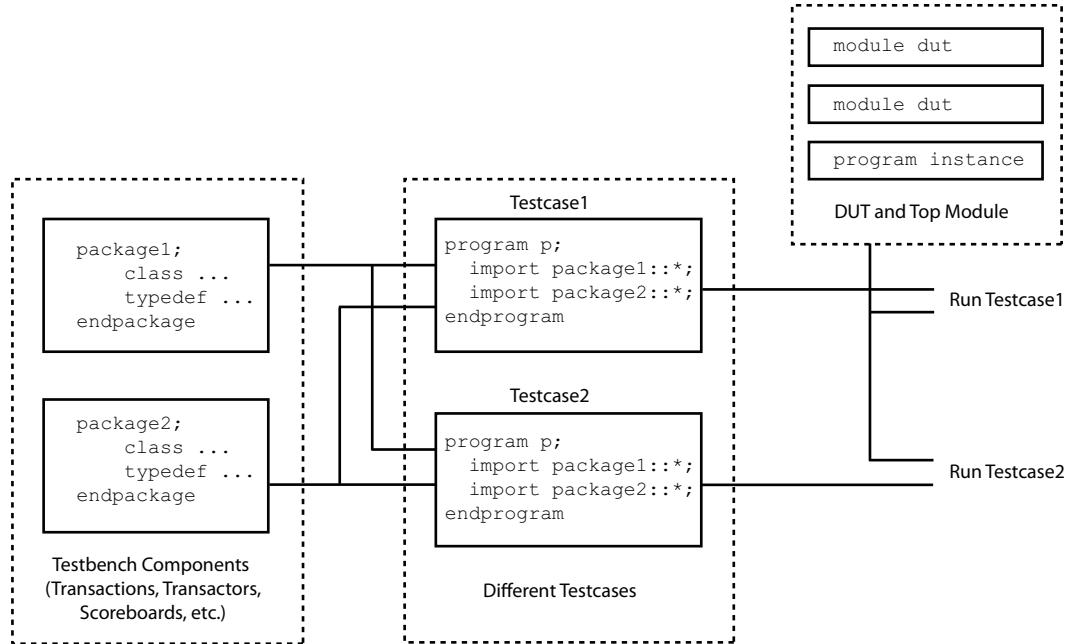
The testbench separate compilation feature accelerates the iterative process of testbench development, and reduces the disk space consumed by a large number of tests.

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

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

For example, consider a typical SystemVerilog verification environment as shown in [Figure 6-1](#). It includes a testbench, DUT, top module to link the DUT with the testbench, and various testcases.

Figure 6-1 Typical SystemVerilog Verification Environment



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

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

```
class ...
typedef ...
endpackage
package2;
class ...
typedef ...
endpackage

Testbench Components
(Transactions, Transactors,
Scoreboards, etc.)
```

```
program p;
    import package1::*;
    import package2::*;
endprogram
```

Different Testcases

```
program p;
    import package1::*;
    import package2::*;
endprogram
module dut
module dut
program instance
    Testcase1
    Testcase2
DUT and Top Module
Run Testcase1
Run Testcase2
```

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

Use Model

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

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

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

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

Note:

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

Specifying Logical Libraries

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

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

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

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

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

Analyzing the Code

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

The command to analyze the code is:

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

Here:

-partition *partition_name*

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

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

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

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

-work *logic_library*

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

-liblist *logic_lib1+logic_lib2+...*

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

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

Examples

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

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

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

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

- Consider the following files:

```
// FILE test1.sv  
package P1;  
...  
endpackage  
  
// FILE test2.sv  
package P1;  
...  
endpackage
```

```

// FILE test3.sv
package P2;
...
endpackage

// FILE test4.sv
program M1;
    import P1::*;
    import P2::*;
...
endprogram

```

The synopsys_sim.setup is:

```

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

```

The packages can be analyzed using the following commands:

```

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

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

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

% vlogan -sverilog test4.sv -work L4 -liblist L2 -liblist
  L1 -liblist L3

```

Package P1 is picked from library L2 and package P2 is picked from library L3.

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

Note:

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

For example, consider the following code:

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

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

The recommended analysis commands for this example are:

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

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

Generating Shared Object Library

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

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

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

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

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

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

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

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

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

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

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

Generating Shell File

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

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

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

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

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

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

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

Generating simv

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

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

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

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

Linking Partitions Dynamically at Runtime

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

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

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

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

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

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

Usage Notes

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

DesignWare VIP

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

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

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

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

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

Virtual ports (the signals' sizes are not defined in Virtual ports) are used in some VIP modules to enable the reuse of VIPs in different environments. If VIPs modules are compiled separately, VCS will not have information on the width of each virtual port signal. By default,

VCS assumes the maximum signal width to be 32. However, the actual signal connected to the port signal can have a larger width (for example, 1024 for a 1024 width AHB system). VCS provides the `-sc_bind_width` option to set the default signal width for Virtual ports.

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

Compiling VIPs with Interface Files

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

For example:

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

```

vcs \
  -sep_cmp -sverilog \
  -ntb_opts rvm \
  -ntb_opts use_sigprop \
  -ntb_opts dw_vip \
  -ntb_vipext .ov \
  SYNOPSYS_VIP_PACKAGE

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

...

```

Compiling VIPs without Interface Files

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

```
-sc_bind_width=actual_signal_size
```

Synopsys recommends you to compile VIP models with interface files.

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

Parameterized Programs

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

For example, consider the following code and script:

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

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

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

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

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

These two simulations incorrectly produce the same result:

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

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

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

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

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

vcs -sep_cmp test1_20_TEST1 -ntb_opts genShellOnly

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

In this script, the program shell file is generated from any one of the program partition and shared by all other partitions. This works well if the parameter is not used to specify the size of program port arguments. In case the parameter is also used to specify the program port size, you must also generate different shell files from different partitions and use the matched shell file for generating simv.

Parallel Compilation

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

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

Random Stability

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

Testbench Separate Compile Flow Notes

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

- Interface use model
 - Pass interfaces only to the partitions which are using it.
 - If the interface is compiled with any of the previous partitions, then do not compile it with the current partition, even if it is using it.
 - Compile (all the interfaces compiled with the TB + interfaces required only by the DUT) in the DUT compilation step.
- Interface method support
 - Tasks, functions, cover groups, initial blocks, property and all other legal constructs within the interface are now supported.
- All testbench classes must be inside a package or program.
- \$root and \$unit calls are not permitted from the testbench.
- Modport is supported only when the program instance and the formal argument for the port definition are identical. Other uses of modport are not permitted. The argument list calling the stub must be identical to what the program block is expecting. For example:

```

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

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

```

- Testcases must be implemented with program block(s) instead of module(s). Testbench separate compile for module is not yet supported.

- Shared packages between the testbench and DUT are not permitted.
- Parameterized programs are supported but the parameters' values cannot be overridden. See “[Parameterized Programs](#)” for information on how to use different parameter values with parameterized programs.
- When you use `-partition` during program compilation, you cannot have multiple program blocks in one file.
- Direction coercion for testbench ports is not permitted. Testbench ports must have directions such that no port direction is coerced.
- Aspect-oriented extensions are only supported in program partitions.
- Compilation options must be the same across all partitions and the DUT. For example, if you use `-debug_all` for one partition, you must use it for all partitions.
- Parameterized interfaces are supported, but the parameters' values are not permitted to be overridden in the testbench. In the DUT, parameterized interface values can be overridden. For example:

```

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

```

- Currently, only mixed-language designs with a Verilog top are supported.
- Function calls in constraints are not permitted.
- You must not use the `-Mdir` option for testbench code generation because all the testbench codes are compiled and saved to logical library directories. You can use the `-Mdir` option for generating `simv`.
- You must use the same set of PLI tab files and C or object files for both the testbench and the DUT.
- DVE interactive debug does not support classes that have later been manipulated with aspect-oriented extensions.
- Waveform dump calls cannot be called from the testbench separate compile flow. This includes `$dumpports` and `$dumpvars`.

Testbench-DUT XMR Support

The testbench separate compile flow has the testbench and DUT compiled in separate partitions. Therefore, the testbench cannot access the DUT XMR data directly. VCS supports this through the config file feature.

Note:

Since XMR access is not allowed in packages. In the SystemVerilog 1800-2009 LRM, in the section named “Package declarations” it specifies:

“Items within packages shall not have hierarchical references to identifiers except those created within the package or made visible by import of another package. A package shall not refer to items defined in the compilation unit scope.”

So XMR access to the DUT signals or tasks is only possible in the program block.

The config file should consist of information of all the XMR signals or XMR tasks/functions accessed. It should specify the scope of the XMR access and the name of the signal, task, or function. Two separate config files are required for XMR signal access and XMR task or function access.

XMR Signal Access

Signals in the DUT scope can be accessed in the testbench side by using a config file. The scope and name of all XMR signals should be entered into the config file. This config file should be passed to the DUT elaboration step using the
+optconfigfile+<config_file_name> option.

The XMR config file requires a strict compilation order. The program elaboration step should be done only after the DUT elaboration, so that the information about the XMR signals can be dumped during the DUT compilation.

Example:

```
// pack.sv
package pack1;
....
endpackage

//prog.sv
```

```

program p1;
import p1::*;
    initial begin
        top.reset = 1'b1;
    end
endprogram

```

```

//top.v
module top;
    reg reset;

    p1 p1_inst();
endmodule

//xmr_sig.cfg
xmr {top} {reset};

```

Following is the flow to be used for XMR signal access:

The package and shell compilation steps remain the same. There is a change only in order in the program block and DUT compilation steps.

Analyzing Program Block Code

You can use the following command to analyze the program block code:

```

vlogan -sep_cmp program_file
[-work logic_library]
[-liblist logic_lib1+logic_lib2+...]
[-timescale scale]
[standard_compile_options]

```

In the above example:

```
% vlogan -sep_cmp -sverilog prog.sv -partition PROG
```

Generating Shell File

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

For example, you can use the following command to generate the shell file for the program block of partition MY_PROG:

```
% vcs -sep_cmp -ntb_opts genShellOnly PROG
```

Generating simv

You can use the following command to compile the DUT code:

```
% vcs -sep_cmp -ntb_vl -sverilog +optconfigfile+xmr_sig.cfg\  
[standard_compile_options]\  
program_shell_file\  
Verilog_filelist
```

For example, you can use the following command to compile the top module with the program shell, and generate simv:

```
% vcs -sep_cmp -sverilog -ntb_vl p1.svshell top.sv  
+optconfigfile+xmr_sig.cfg
```

Generating Shared Object Library for Program Block

You can use the following command to generate shared object library for the specified partition:

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

Example:

```
% vcs -sep_cmp -sverilog PROG
```

XMR Method Access

Tasks or functions in the DUT scope can be accessed in the testbench side by using a config file. The scope, prototype, and the names of all XMR tasks or functions should be entered into the config file, and this config file should be passed to the program analysis, elaboration time, and the DUT elaboration step using the `+dutxmrconfigfile+<config_file_name>` option.

Example

```
// pack.sv
package pack1;
    ...
endpackage

//prog.sv
program p1;
import p1::*;
int a;
initial begin
    top.dut_reset(a);
end
endprogram
```

```

//top.v
module top;

    task dut_reset(int a);
        int reset;
        reset = a;
    endtask

    p1 p1_inst();
endmodule

//xmr_method.cfg
(*xmrpath = "top.dut_reset") task dut_reset(int a);

```

Following are the compilation steps, where the config file has to be passed.

Analyzing Program Block Code

You can use the following command to analyze the program block code:

```

vlogan -sep_cmp program_file
+dutxmrconfigfile+xmr_method.cfg
[-work logic_library]
[-liblist logic_lib1+logic_lib2+...]
[ -timescale scale ]
[standard_compile_options]

```

In the above example:

```
% vlogan -sep_cmp -sverilog prog.sv -partition PROG
+dutxmrconfigfile+xmr_method.cfg
```

Generating Shell File

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

For example, you can use the following command to generate the shell file for the program block of partition MY_PROG:

```
% vcs -sep_cmp -ntb_opts genShellOnly PROG
```

Generating Shared Object Library for Program Block

You can use the following command to generate shared object library for the specified partition:

```
vcs -sep_cmp [logic_library.]partition_name  
+dutxmrconfigfile+xmr_method.cfg  
[standard_library_generation_options]
```

Example:

```
% vcs -sep_cmp -sverilog PROG  
+dutxmrconfigfile+xmr_method.cfg
```

Generating simv

You can use the following command to compile the DUT code:

```
% vcs -sep_cmp -ntb_vl -sverilog  
+dutxmrconfigfile+xmr_method.cfg \  
[standard_compile_options] \  
program_shell_file\  
Verilog_filelist
```

For example, you can use the following command to compile the top module with the program shell, and generate simv:

```
% vcs -sep_cmp -sverilog -ntb_vl p1.svshell top.sv  
+dutxmrconfigfile+xmr_method.cfg
```

Linking Partitions Dynamically at Runtime

The command syntax and examples are described in the “[Linking Partitions Dynamically at Runtime](#)” section.

Limitations of XMR Support

XMR to DUT Signals:

- XMR through VHDL hierarchy is not supported.
 - Though MX design works with separate compile, the XMR path must be a verilog-only hierarchy.
- XMR is not supported for the following types:
 - strings
 - real and shortreal
 - wand, wor, tri, tri0, and trireg
 - structs or unions, and other user-defined types like classes
 - Dynamic arrays and associative arrays
 - enum
 - file handles
 - XMR's to interface clocking blocks
- The following operations are not allowed on XMR signals in the program block:
 - Use of `mytref` (XMR, e, f); where the first port is `ref`.
 - Memory (present in DUT) initialization using the `$readmemb` and `$readmemh` system tasks.

- Limitations of config file:
 - Multiple config files are not supported.
 - You cannot mention two XMR signals in the same scope using “,” operator.

Example: `xmr {top.DUT} {a,b}`

- For XMR to vector or MDA, the whole signal must be specified in the config file. Specifying a part or bit select in the config file is not permitted.

XMR methods:

- Operations not supported:
 - Arithmetic operations on XMR to interface, and DUT signals with their methods.

Example: `S=top.dut.d.num() + top.if1.q.next(i)`

NTB OpenVera/SystemVerilog Interoperability

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

Analyzing the Code

For analyzing OpenVera code, issue the following command:

```
vlogan -sep_cmp -ntb -partition partition_name  
[-work logic_library]
```

```
[-liblist logic_lib1+logic_lib2+...]
[ -timescale scale ]
[standard_compile_options]
Verilog_filelist
```

For analyzing SystemVerilog code, issue the following command:

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

Note:

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

Examples

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

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

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

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

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

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

```
partition p1
```

Generating Shared Object Library

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

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

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

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

Generating Shell File

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

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

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

Generating simv

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

For example, you can use the following command to compile the top module with the program shell, and generate simv:

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

Linking Partitions Dynamically at Runtime

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

Testbench Separate Compile Flow Notes

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

- All OpenVera code must be in one partition.

Pure NTB OpenVera Flow

The testbench separate compilation feature supports pure Open Vera testbench flow.

Since there is no concept of package in Vera, a partition can be a class or set of classes, a program block, or a combination of both. You can have multiple partitions. You must specify the name of every partition using the `-partition <partition_name>` option in all the analysis steps. Every subsequent partition implicitly imports the data from the previously compiled partitions. Thus, files compiled in one partition should not be included in the subsequent partitions.

Note:

VIP data should be compiled into a single partition.

Analyzing OpenVera Code

You can use the following command to analyze the OpenVera code:

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

For example, you can use the following command to analyze the `class_env.vr` vera environment file into the `MY_ENV` partition, and the program block `prog.vr` into the `MY_PROG` partition:

```
% vlogan -sep_cmp -ntb class_env.vr -partition MY_ENV  
% vlogan -sep_cmp -ntb prog.vr -partition MY_PROG
```

Note:

Here, the `MY_PROG` partition implicitly imports all the data from the `MY_ENV` partition. No explicit “import” of that package must be done within the program block.

Generating Shared Object Library

You can use the following command to generate shared object library for a specified partition:

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

For example, you can use the following command to generate shared library for the `MY_ENV` and `MY_PROG` partitions respectively:

```
% vcs -sep_cmp -ntb MY_ENV  
% vcs -sep_cmp -ntb MY_PROG
```

Generating Shell File

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

For example, you can use the following command to generate the shell file for the program block of the MY_PROG partition:

```
% vcs -sep_cmp -ntb_opts genShellOnly MY_PROG
```

Generating simv

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

For example, you can use the following command to compile the top module with the program shell, and generate simv:

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

Linking Partitions Dynamically at Runtime

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

Determining Port Width of Virtual Ports

In the default flow, VCS provides a default width of 32 bits to all virtual port signals because it cannot conclude the exact width till the actual connections are made. The signal, whose width is less than 32 bits, get padded with zeros to make it a 32-bit value. The signal, whose width is greater than 32 bits, generates a compile error when the tool recognizes the correct width.

In the NTB-OV separate compile flow, there are three flows to specify the port width to VCS:

- Use of the `-sc_bindwidth` option in the default flow
- Use of pragmas with the default flow
- Generation of a port config file

Using `-sc_bindwidth` Option in the Default Flow

If you know the maximum port width of all the virtual ports used, then you can specify it to VCS MX using the `sc_bindwidth=<max_bindwidth_used>` option. Therefore, VCS MX considers all the virtual ports to be of length `<max_bindwidth_used>`, and proceeds with the compilation. All ports having width lesser than specified, gets padded with zeros. The above option must be specified with the elaboration step of all the partitions.

To elaborate all testbench partitions:

```
vcs -sep_cmp -sc_bind_width= <max_bindwidth_used>
[logic_library.]partition_name
[standard_library_generation_options]
```

To elaborate DUT partition:

```
vcs -sep_cmp -ntb_vl -sc_bind_width= <max_bindwidth_used>
[standard_library_generation_options] [all dut files]
[shell file]
```

Using pragmas

The virtual port width can also be specified by using pragmas in the virtual port definition. This is the recommended flow, and does not require any additional compile or runtime options. It also provides optimum performance.

Example:

```
port myp {  
    sig1;  
    (* width :: 1 *) sig2;  
};
```

In the above example, the width is specified using a pragma. VCS MX considers the width of `sig2` in port `myp` to be 1.

Using Port Config File

VCS MX can auto-generate a port config file containing the exact width of the virtual port signals by first compiling the design in normal compile flow, and then passing the `-ntb_opts port_config=<cfg_file>` compile-time option to it.

Separate compile can then use this generated port config file to assign the widths to the virtual port signals. This is done by using the `-ntb_opts port_config=<cfg_file>` compile-time option. You can edit the port config file. You can increase the size of any vera port signal to allow all the tests to run.

Generating the config file by compiling the design in the normal compile flow:

```
vcs -ntb -ntb_opts port_config=<cfg_file>
[-work logic_library]
[-liblist logic_lib1+logic_lib2+...]
[ -timescale scale ]
[standard_compile_options]
Verilog_filelist
```

Example:

```
vcs -ntb class_env.vr prog.vr top.sv -ntb_opts
port_config=<cfg_file>
```

Note:

After the above step, a config file `cfg_file` is generated.

Analyzing OpenVera Code

You can use the following command to analyze the OpenVera code:

```
vlogan -sep_cmp -ntb -partition partition_name -ntb_opts
port_config=<cfg_file>
[-work logic_library]
[-liblist logic_lib1+logic_lib2+...]
[ -timescale scale ]
[standard_compile_options]
Verilog_filelist
```

For example, you can use the following command to analyze the `class_env.vr` vera environment file into the `MY_ENV` partition, and the program block `prog.vr` into the `MY_PROG` partition:

```
% vlogan -sep_cmp -ntb class_env.vr -partition MY_ENV -
ntb_opts port_config=<cfg_file>
% vlogan -sep_cmp -ntb prog.vr -partition MY_PROG -ntb_opts
port_config=<cfg_file>
```

Note:

Here, the MY_PROG partition implicitly imports all the data from the MY_ENV partition. No explicit “import” of that package must be done within the program block.

Generating Shared Object Library

You can use the following command to generate shared object library for a specified partition:

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

For example, you can use the following command to generate shared library for the MY_ENV and MY_PROG partitions respectively:

```
% vcs -sep_cmp -ntb MY_ENV
```

```
% vcs -sep_cmp -ntb MY_PROG
```

Generating Shell File

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

For example, you can use the following command to generate the shell file for the program block of partition MY_PROG:

```
% vcs -sep_cmp -ntb_opts genShellOnly MY_PROG
```

Generating simv

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

For example, you can use the following command to compile the top module with the program shell:

```
% vcs -sep_cmp -sverilog -ntb_vl p1.svshell top.sv -ntb_opts  
port_config=<cfg_file>
```

7

Constraints Features

- “Using String Indexed Associative Arrays in Constraints”
- “Using the array.exists() Function in Constraints”

Using String Indexed Associative Arrays in Constraints

To allow randomizing the values of elements in string indexed associative arrays, arrays can be used in constraints. This enables iterating over the string indexes to constrain the values of the associative array elements.

Syntax

```
class class-name;  
    rand integer associative-array-name[string];  
endclass
```

Description

Only string constants or state strings are allowed in the associative array index expressions. This is because the use of random variables would require the introduction of ordering constraints that force those random variables to be solved before the system method, which is not supported.

Random string index associative arrays are not allowed.

The behavior of string indexed arrays is the same as for integer indexed arrays in the case when size is not constrained; that is, in subsequent calls to randomize, the values of the array elements are randomized and the indexes are preserved.

A `foreach` statement over the array indexes is legal, as in the following example:

```
foreach (aa, i) {  
    aa[i] ...      // aa is an associative array with index i  
}
```

When an associative array is printed, its indexes are printed as strings. Debug messages and error messages show the string constants as the array indexes.

The index usage must be limited to referring the index of the array. Using the index as a guard, as in the following, results in an error message:

```
i == "mystring" => a[i].data == 10; // Error
```

Variable string indexes are not allowed.

If none of the array elements are assigned values, the size of the array is considered to be 0.

Constraining the size of the associative array is not allowed, because random string indexes cannot be generated. Each of the following constraint constructs causes an error:

```
constraint c {
    aa.size() == 3;
}

constraint c {
    x == aa.size();
}
```

If no elements have been assigned values, but constraints are written on any of the element's values, an error like the following is issued:

```
Error- [CNST-VOAE] Constraint variable outside array error
```

Example

```
program test ;

class C ;
    rand integer aa[string];
endclass

initial
begin
    integer res;
    C c = new;

    c.aa["s5"] = 10;
    c.aa["ss8"] = 15;
    c.aa["sss10"] = 20;

    foreach(c.aa[i])
        $display("%s %d\n", i, c.aa[i]);
```

```

res = c.randomize();

foreach(c.aa[i])
    $display("%s %d\n", i, c.aa[i]);
end

endprogram

```

Using the array.exists() Function in Constraints

The `assoc-array.exists()` system function can be used in a constraint to check whether a particular element exists in an associative array.

Syntax

`assoc-array.exists(index-expr) -> constraint-expr,`

Description

For every possible element of an associative array, the `assoc-array.exists()` function can be used to set a bit variable that indicates whether the element exists.

The `index-expr` can be of the type of the associative array key, such as integer or string. Only integral types (non-complex classes) are allowed as keys for associative arrays.

Only a constant or state variable can be used as the index expression. Rand variables are not allowed as arguments to `assoc-array.exists()`.

Example

```
class C;
    string s1;
    bit inUse[string]; // State string indexed assoc array.
    rand integer x;
    constraint c_1 {
        inUse.exists("tmp") && inUse["tmp"] -> x == 10;
        inUse.exists(s1) && inUse[s1] -> x == 20;
        foreach (inUse, key) {
            inUse.exists(key) -> guarded-constraints
        }
    }
endclass
```

8

Coverage Features

This chapter contains the following features:

- “Hierarchical Cross Coverage”
- “Coverage Report-Time Exclusion”
- “Echo Procedural Sampling Enhancement”
- “Echo Features”
- “Reviewing Exclusions Using Adaptive Exclusion Flow”
- “Coverage Analysis of Unreachable Verilog Code”
- “Support for `with` Clause in Cover Groups”

Hierarchical Cross Coverage

SystemVerilog supports the covergroup construct to capture the functional coverage model. One of the requirements for a cross is that all the crossed elements must be declared within the same covergroup. This allows seamless and unambiguous sampling, among other benefits.

However, advanced coverage modeling requires the ability to specify crosses of elements in different covergroups. This is called the hierarchical covergroup features. The VCS SystemVerilog implementation has extended the IEEE 1800 standard to enable this feature.

Crosses can be defined between coverpoints that reside in separate covergroup instances. This enables object composition-like behavior for covergroups, in which an instance of a covergroup can make use of the coverage collected by its constituent covergroup instances.

[Example 8-1](#) shows a sample of a hierarchical covergroup, `hier_cg`.

Example 8-1

```
class cl_1;

covergroup cg_0 @(e0);
    cp0 : coverpoint a;
endgroup : cg_0

covergroup cg_1 @(e1);
    cp1 : coverpoint b;
endgroup : cg_1

covergroup hier_cg;
    cr_0 : cross cg_0.cp0, cg_1.cp1;
```

```

// Using implicit class member
// covergroup instances: cg_0, cg_1
endgroup : hier_cg
endclass: cl_1

```

Here, covergroup `hier_cg` crosses coverpoints `cg_0.a` and `cg_1.b`. The `hier_cg` instance is thus a hierarchical covergroup instance, with `cg_0` and `cg_1` as its constituent instances.

Limitations

The covergroups that contain cross of coverpoints from other covergroups, cannot have a declaration of another coverpoint.

For example,

```

covergroup hier_cg;
    cp3 : coverpoint c; // <--- Declaring cp3 is not allowed
    cr_0 : cross cg_0.cp0, cg_1.cp1;
endgroup : hier_cg

```

This section contains the following subsections:

- “[Constituent Covergroup Instances](#)”
- “[Sampling Hierarchical Covergroups](#)”

Constituent Covergroup Instances

A hierarchical covergroup definition can use any covergroup instance visible in its scope to define crosses. Alternatives to using implicit class member instances are shown in the following sections:

- “[Referring to Covergroups using Cross Module Referencing](#)”
- “[Passing Constituent Instances as Arguments](#)”

Referring to Covergroups using Cross Module Referencing

In [Example 8-2](#), the `cg1_inst` and `cg_2_inst` are visible only inside the initial block, named `blk1`. The `hier_cg` hierarchical covergroup refers to them via XMR (Cross Module Referencing):

Example 8-2

```
module hier_cg_ex;
  event cov_e1, cov_e2;
  bit [5:0] a, b;

  covergroup cg_1 @(cov_e1);
    cp0: coverpoint a;
  endgroup: cg_1
  covergroup cg_2 @(cov_e2);
    cp1: coverpoint b;
  endgroup: cg_2

  covergroup hier_cg;
    cr_1 : cross blk1.cg_1_inst.cp0, blk1.cg_2_inst.cp1;
  endgroup: hier_cg

  initial begin: blk1
    cg_1 cg_1_inst = new();
    cg_2 cg_2_inst = new();
    hier_cg h_cg_inst = new();
  end: blk1

endmodule: hier_cg_ex
```

Passing Constituent Instances as Arguments

It is possible to define a covergroup with constituent instances passed as arguments at the time of its instantiation, as shown in [Example 8-3](#).

Example 8-3

```
covergroup cg_0;
    cp0 : coverpoint p;
endgroup: cg_0

covergroup cg_1;
    cp1 : coverpoint q;
endgroup: cg_1

covergroup hier_cg (cg_0 a1, cg_1 b1);
    cr_0 : cross a1.cp0, b1.cp1;
endgroup: hier_cg

initial begin
integer p, q, r;
cg_0 a1 = new;
cg_1 b1 = new;
hier_cg x1 = new;
```

Sampling Hierarchical Covergroups

A hierarchical covergroup infers the sampling event from its constituents and cannot have its own sampling event.

A hierarchical covergroup is considered sampled at a given time stamp if and only if all of its constituent covergroups were sampled at that time stamp. If one or more of the constituent covergroups did not get sampled, the results are not collected for the hierarchical covergroup. The cross hit of a hierarchical cross is composed of the values recorded for the respective coverpoints. An interesting situation occurs when, at a given time stamp, one or more constituent covergroups gets sampled more than once. This is illustrated for [Example 8-4](#) in [Table 8-1](#).

Example 8-4

```
class cl_0;
```

```

covergroup cg_0 @(e0);
    cp0 : coverpoint p;
endgroup : cg_0

covergroup cg_1 @(e1);
    cp1 : coverpoint q;
endgroup : cg_1

covergroup hier_cg;
    cr_0 : cross cg_0.cp0, cg_1.cp1;
endgroup : hier_cg
endclass : cl_0

function new();
cg_0 = new();
cg_1 = new();
hier_cg = new();
endfunction
endclass
...

```

Table 8-1 Hierarchical Covergroup Sampling Results

Simulation Time Step	Events	Values Sampled		
		a1.p	b1.q	x1.cr0
1	e0	6	-	-
2	e1	-	a	-
3	e0, e1	7	b	(7,b)

Multiple Values Sampled in the Same Time Step

If any of the constituent covergroups is sampled multiple times during the same time step, more than one value might be recorded for the constituent coverpoints in a given time step. For such cases, crosses are composed as described below.

Table 8-2 Hierarchical Covergroup Sampling Results

Simulation Time Step	Events	Values Sampled		
		a1.p	b1.q	x1.cr0
1	e1#3	6, 7, 42	-	-
2	e0#1, e1#1	6	a	(6,a)
3	e0#3, e1#3	6, 7, 42	a, b, c	(6,a) (7,b) (42,c)
4	e0#3, e1#1	6, 7, 42	a	(42,a)
5	e0#3, e1#2	6, 7, 42	a, b	(42, b)

If all constituent coverpoints record the same number, say n , of values in the time step, covergroup `hier_cg` records n crosses with the i th cross composed from the i th recorded values of each constituent coverpoint (see the row for Simulation Time Step 2 in [Table 8-2](#)). If the number of values sampled is not the same for all constituent coverpoints, the cross is composed using only the last recorded value of each coverpoint. This is done to ensure that crosses collected are meaningful for the majority of the interesting use cases (see the rows for Simulation Time Steps 4 and 5 in [Table 8-2](#)).

Coverage Report-Time Exclusion

The primary change in coverage report-time exclusion in this release over the previous release of VCS is the enhancement in the granularity of exclusion file version check from the file level to the module or metric-variant level.

To achieve this, the following two changes are introduced in the format of the exclusion file created by the UCAPI API `covdb_save_exclude_file`:

- The module or instance-wise positioning of a design and its metric-variant checksums in the generated file.
- The introduction of the keyword `CHECKSUM`:

The other modifications that are mandated by the version check change are:

- The Line exclusion format is moved from absolute line numbers to basic block IDs. This change is required to shield the exclusion file from unexpected non-content design changes such as the addition of empty and comment new lines, changes in file names, and the location of modules within a file.
- With the shift in the format, the dependency of line exclusion on file names is completely eliminated. Generated exclusion files no longer contain the `FILE:` keyword, because it is not required anymore.
- The absence of checksum information for a module or metric-variant is reported as a checksum mismatch. In other words, manually written exclusion files are loaded only under the `excl_bypass_check` switch; exclusions are not be loaded by default.

Checking the Version of an Exclusion File

VCS checks various design changes brought-in, as compared to the earlier version, with the help of the Version/checksum in the exclusion file. For VCS to check these changes, you must follow the below mentioned flow:

1. Compile the design.
2. Simulate the design.
3. Load the coverage database to DVE.
4. Mark the exclusions, and save them to an exclusion file.
5. Edit the design source.
6. Recompile the edited design.

Note:

This step is important to generate the new checksum.

7. Simulate the edited design.
8. Reload the exclusions.

In the current release, an exclusion file is not discarded when one or more checksums mismatch (exclusion on objects where a checksum match is seen are applied normally, while those having their checksums mismatched are ignored).

Checksums are introduced for modules as well. Previously, only metric-variant checksums were involved in version checks. The checksum of a module considers the source code of the module and ignores all white space, including new lines and comments. Therefore, any addition or deletion of text to the module (barring white space and comments) impacts the module checksum. Essentially, version checks for excluded modules or instances follows a two-level approach.

For code or assertion coverage metrics, first, the module checksum is matched. This is followed by variant checksum matching (if applicable). A mismatch in any of the two checksum checks results in version failure for that excluded scope. Similarly, for covergroups,

matching is done for covergroup definition and covergroup variant checksums in that order (covergroup definition and variant checksums are analogous to module and metric-variant checksums, respectively).

To enable module or metric-variant level granularity in version checks, checksums are dumped along with exclusions, unlike the previous release, where all checksums of the excluded modules were accumulated and saved at the beginning of the file. Specifically, the keyword `CHECKSUM:` is introduced, replacing the existing `// MOD_CHKSUM:` and `// CHECKSUM:` keywords, where in each `MODULE:/INSTANCE:/covergroup` specification is accompanied with a checksum line that is prefixed with the `CHECKSUM:` keyword. The format of this line is as follows:

```
CHECKSUM: <mod_or_cgdef_checksum> <variant_checksum>
```

where, `<mod_or_cgdef_checksum>` can be either the module checksum that is described in the previous paragraph for `MODULE:` and `INSTANCE:` exclusions or the covergroup definition checksum for covergroup exclusions.

For scope exclusions (complete exclusion of a module, instance, or a covergroup definition), only the module or covergroup definition checksum is dumped and matched. This means, even if any of the metric variant checksums of that module or covergroup change (or new variants are added), exclusion is applied on the scope if the module or covergroup checksum matches.

[Example 8-5](#) and [Example 8-6](#) show the format and contents of the generated exclusion file.

Example 8-5 Metric (or Partial Scope) Exclusions

```
CHECKSUM: "<mod_chksum> <metric_variant_checksum>"
```

```

MODULE: <module_name>
      <metric_objects>

CHECKSUM: "<mod_chksum> <metric_variant_checksum>
INSTANCE: <instance_name>
          <metric_objects>

CHECKSUM: "<cg_def_chksum> <cg_variant_checksum>
covergroup <cg_variant_or_instance_name>
          <covergroup_objects>

CHECKSUM: "<cg_def_chksum> <cg_variant_checksum>
covergroup <cg_variant_or_instance_name>
// covergroup variant or instance exclusion

```

Example 8-6 Scope Exclusions

```

CHECKSUM: "<mod_chksum>" // Notice that there isn't any
variant checksum here - only <mod_chksum> is matched
MODULE: <module_name>

CHECKSUM: "<mod_chksum>" // Notice that there isn't any
variant checksum here - only <mod_chksum> is matched
MODULE: <module_name>

CHECKSUM: "<cg_def_chksum>" // Notice that there isn't any
covergroup variant checksum here - only <cg_def_chksum> is
matched
covergroup <cg_def_name>

```

The metric variant checksum is not available for assertion coverage. For design qualified assertion variants and instances, only the corresponding module checksums are dumped, and matched, while loading an exclusion file. For test qualified assertion objects, there is no checksum check (exclusion is applied unconditionally).

The absence of a checksum for a scope in the exclusion file is interpreted as a mismatch, unless the file is loaded in backward-compatible mode (see “[Backward Compatibility](#)” on page 154);

exclusions are not applied for that scope. To enable loading of exclusions for such modules (which means to disable exclusion file version checks) you can use the URG or DVE switch `excl_bypass_checks`.

Backward Compatibility

Exclusion files generated with the previous release of VCS also work with this release, thus ensuring backward compatibility. If the tool encounters any older-style checksums from previous VCS releases (that is, checksums starting with `// MOD_CHECKSUM` and `// CHECKSUM`) it interprets the exclusion file as being generated with a previous release. For such files, version checks are performed in a way similar to the procedure followed in the previous release, where any checksum mismatch results in the non-application of the whole file. Exclusions are loaded from the file only after a successful version check.

Note:

There is some change in the way line metric checksum is computed in the current release, so there are backward compatibility issues with line coverage exclusion (see “[Impact on Metric Checksums](#)” on page 158).

It is possible that an exclusion file generated with a previous release is manually edited and more scopes and exclusions are added. Such extra manual additions (to an existing exclusion file generated through a previous release) cannot be detected; these exclusions are loaded without any version check.

Saving Exclusions

Exclusions are always saved in the new format, which means the checksum information of an excluded scope, variant, or instance and the corresponding exclusions are dumped together. This is true when saving is called in append mode as well, even when the file to which exclusions are appended is an exclusion file generated through a previous release. In this case, the file takes a hybrid format. Version checks of such mixed-formatted exclusion files also follow a hybrid approach. First, all previous release style checksums are read and if there is a mismatch, all exclusions that are saved through the previous release (which means, all scopes that do not come with `CHECKSUM: ...`) are not loaded. If the checksums match, then all of these exclusions are loaded. For the rest, which are generated with the current release (which means, scopes that come with `CHECKSUM: . . .`), checksums are matched scope by scope and exclusions are applied wherever checksums match.

Relative Basic Block ID Change for Line Exclusion

In the current release, the existing absolute line number format is moved to the basic block ID format, for line exclusion. The reasons for this shift are:

- Absolute line numbers are prone to unpredictability. Adding or deleting a few statements, or empty lines or comments, from a module invalidates the line numbers of statements of all modules located in the file after the changed module.
- Migrations of modules from one file to another changes statement line numbers. Any previously generated exclusion files that are based on absolute statement line numbers cannot be used on freshly generated databases.

Line exclusion in NEWDB is actually basic block exclusion. This means that any line marked for exclusion ends up excluding the complete basic block. If this exclusion is saved, you needlessly end up dumping all lines of the basic block.

The introduced basic block ID format addresses both the following issues:

- As the name says, the exclusions are dumped at the block level and not at the statement level. This makes the exclusion files compact.
- Since ordered unique IDs are assigned to the basic blocks, the issues associated with using absolute line numbers, explained in the previous paragraph, are overcome.

Essentially, if a module `mod` contains n basic blocks, then a unique ID ranging from 1 to n is assigned for each of the blocks, where the i th basic block gets an ID of i . For instance, if the third block of the module contains three statements at line numbers 20-22 in File `test.v`, then the change in the styles when any of the three statements is excluded is as follows:

Previous format:

```
FILE: test.v
MODULE: mod
    LINE 20
    LINE 21
    LINE 22
```

Current format:

```
MODULE: mod
```

BLOCK 3

Note:

The file name and the FILE: keyword are not required any more. Further, this keyword can be used to detect the format of line exclusion for each module. If there is a file name specified, then the format is identified to be of absolute IDs, while the lack of it is construed as block ID format.

The scope of a FILE: specification is either EOF or the next FILE: specification, depending upon what comes first. If an exclusion scope lies within the scope of a FILE: specification, then that exclusion scope is loaded in absolute line format, except when the exclusion scope comes with a CHECKSUM: specification, in which case, it is interpreted as generated with this release and hence apply the block ID format.

Following are few more examples:

```
1) FILE: test.v
MODULE: mod1 // will be interpreted as absolute line numbers
        format
        LINE 20
        LINE 21
        LINE 22

INSTANCE: top.i1 // will be interpreted as absolute line
           numbers format as well, since it lines
           inside the scope of "FILE: test.v"
           LINE 2

2) INSTANCE: top.i1 // will be interpreted as block id format
           - instance does not lie in the scope
             of any FILE: scope
           LINE 2
           FILE: test.v
MODULE: mod1 // will be interpreted as absolute line numbers
        format
```

```
LINE 20  
LINE 21  
LINE 22
```

```
INSTANCE: top.i2 // will be interpreted as absolute line  
numbers format as well, since it lies  
inside the scope of "FILE: test.v"  
LINE 2
```

3) FILE: test.v

```
MODULE: mod1 // will be interpreted as absolute line  
numbers format
```

```
LINE 20  
LINE 21  
LINE 22
```

```
INSTANCE: top.i1 // will be interpreted as absolute line  
numbers format as well, since it lies  
inside the scope of "FILE: test.v"
```

```
LINE 2
```

```
CHECKSUM: " "
```

```
INSTANCE: top.i2 // will be interpreted as block id format  
- 'CHECKSUM:' line, which indicates that  
this entry was generated with 2012.09  
LINE 2
```

Impact on Metric Checksums

In previous releases, line coverage metric variant checksums consider absolute line numbers of all statements of the variant. This ensured that any change in the line number of any of the statements of the variant changed the checksum and non-application of exclusions. However, for line coverage exclusions in the block ID format, this restriction does not exist anymore, and line numbers are

removed from checksum computation. In this way, any change in the position or content of the module does not impact the checksum of its line metric variants and exclusions can be seamlessly applied. Due to this change, exclusion files generated in the previous release, which have line metric checksums computed through absolute line numbers, cannot be loaded with the current release - version check fails due to change in line checksums. This is the only backward-compatibility issue. The workaround for this issue is to bypass the version check. There is no change in the way variant checksums are computed for other metrics.

Echo Procedural Sampling Enhancement

Echo analyzes procedural context to find targetable cover points. Echo support for procedural sampling includes:

- Sampling a non-random variable assigned with a targetable random variable.
- Randomizing a sibling class of sampled variables.

This section explains the procedural sampling enhancements. For more information about Echo and procedural sampling, see the Echo chapter under the LCA category in the *VCS / VCS MX Online Documentation*.

The procedural context considered by Echo includes three new coding styles:

- Sampling a non-random cover point variable that has been assigned a targetable random variable present as a member of another class.

- Passing a randomized object through function/task arguments.
 - Stand-alone covergroups.
-

Sampling a Non-random Cover Point Variable Assigned with a Random Variable of Another Class

The following examples show how you can sample a non-random cover point variable assigned with a targetable random member variable of another class.

In [Example 8-7](#), the non-rand cover point variable `i1` (`MyClass`) of object `obj1` is assigned by the targetable rand variable `ri1` (`OtherClass`) of object `oo`.

Example 8-7

```
class OtherClass;
    rand int ri1;
endclass
program test;
    class MyClass;
        int i1;
        event cov_event;
        covergroup MyCov @(cov_event);
            cp1 : coverpoint (i1);
        endgroup
        function automatic new;
            MyCov = new;
        endfunction
        task my_sample();
            ->cov_event;
        endtask
    endclass
    initial begin
        MyClass obj1 = new;
        OtherClass oo = new;
        repeat (5)
```

```

begin
    oo.randomize;
    obj1.i1 = oo.ril;
    obj1.my_sample();
end
end
endprogram

```

Echo works across scopes; that is, the covergroup can be in one scope and the coverable object in another scope (see [Example 8-8](#)).

Example 8-8 covergroup in module scope and coverable object in global scope.

```

class packet;
    rand logic [2:0] data;
endclass

module top;
    packet P;

    class Cover;
        logic [2:0] m_data;
        event ev;

        covergroup cov@(ev);
            CP: coverpoint m_data;
        endgroup
    endclass

    Cover Cov;

    initial begin
        Cov=new();
        P=new();
        void'(P.randomize);
        Cov.m_data = P.data;
        ->Cov.ev;
    end

endmodule

```

Limitation

Non-rand variables with multiple constant or non-constant drivers are not supported.

Passing a Randomized Object Through a Function/Task Argument

[Example 8-9](#) shows how you can use randomized object arguments to target non-random variable associated coverpoints.

Example 8-9

```
class OtherClass;
    rand int ril;
endclass
program test;
    class MyClass;
        int i1;
        event cov_event;
        covergroup MyCov @(cov_event);
            cp1 : coverpoint (i1);
        endgroup
        function automatic new;
            MyCov = new;
        endfunction
        task my_sample(OtherClass oo);
            i1 = oo.ril;
            ->cov_event;
        endtask
    endclass
    initial begin
        MyClass obj1 = new;
        OtherClass oo = new;
        repeat(5)
            begin
```

```

        oo.randomize;
        obj1.my_sample(oo);
    end
end
endprogram

```

Multiple Drivers Present in Different Methods (Functions/Tasks)

Echo identifies dead code and omits the drivers present in dead code in the presence of multiple drivers. Echo does not substitute the sampled variable by the driver of type rand.

You need to check the callers progressively until the root caller is reached. However, if the sampling function is virtual or the function calling the sampling function is virtual, you can resolve whether the function is called or not only at runtime. Echo cannot do this analysis at compile-time to figure out if there is dead code.

In cases where the sampling function is non-virtual, you may check using the switch `-ntb_opts echo_single_scope_check` whether the sampling function is called or not rather than checking the entire caller chain. This is helpful where the non-virtual sampling function is called within a virtual function.

In [Example 8-10](#), the task `my_sample2` is dead code since it is not being called. In effect, there is only one driver on variable `i1`.

Example 8-10

```

program test();
    class OtherClass;
        rand int ri1;
    endclass
    class MyClass;
        int i1;

```

```

event cov_event;
covergroup MyCov @(cov_event);
    cp1 : coverpoint (i1) {
        bins s0 = {0};
        bins s1 = {1};
        bins s2 = {2};
        bins s3 = {3};
    }
endgroup
task my_sample1(OtherClass oo);
    i1 = oo.ril + 1;
endtask

task my_sample2(OtherClass oo);
    i1 = oo.ril + 3;
endtask
function automatic new;
    MyCov = new;
endfunction
endclass

initial begin
    MyClass obj = new;
    OtherClass oo = new;

    repeat(5)
    begin
        oo.randomize;
        obj.my_sample1(oo);
        $display("i1=%d", obj.i1);
    end
end
endprogram

```

Stand-alone Covergroups

Stand-alone covergroup work in both the rand case (where the coverpoint variable is rand) and the non-rand case (where the coverpoint variable is non-rand and is assigned through procedural sampling), as shown in [Example 8-11](#) and [Example 8-12](#).

Example 8-11 // rand case

```
class packet;
    rand logic [2:0] data;
endclass

module top;
    packet P;
    event ev;

        covergroup cov@(ev);
            CP: coverpoint P.data;
        endgroup

    cov Cov;

    initial begin
        Cov=new();
        P=new();
        void'(P.randomize);
        ->ev;
    end

endmodule
```

Example 8-12 non rand case

```
program test();
    class base;
        rand integer r1;
    endclass

    class OtherClass extends base;
```

```

endclass

integer i1;
event cov_event;
covergroup MyCov @(cov_event);
    cp1 : coverpoint (i1) {
        bins s0 = {0};
        bins s1 = {1};
        bins s2 = {2};
        bins s3 = {3};
    }
endgroup

task my_sample(OtherClass oo);
    i1 = oo.ril;
    ->cov_event;
endtask

initial begin
    MyCov cg = new;
    OtherClass oo = new;
    repeat(5)
    begin
        oo.randomize;
        my_sample(oo);
        $display("i1=%d", i1);
    end
end
endprogram

```

Echo Features

The following are the two Echo features:

- “[Support for Procedural Sampling](#)” on page 167
- “[Crosses with Non-Random Cover Points](#)” on page 167

Support for Procedural Sampling

Echo analyzes procedural context to find targetable cover points. Echo supports the following two aspects of procedural sampling:

- Sampling a non-random variable that is assigned with a targetable random variable. Echo targets a non-random cover point when the non-random variable is assigned with an expression of random variables.
- Randomizing a sibling class of sampled variables. At runtime, Echo collects targetable holes from the sibling classes. Echo currently supports only one level of inheritance. Note that all the variables involved in the expression of a cover point must be inherited from the targeted base class.

Crosses with Non-Random Cover Points

You can target a cross with non-random cover points using Echo. The cross must contain at least one random cover point.

While collecting Echo targetable cross holes for non-random cover points, VCS ignores cross-bin combinations with bins that are not sampled. However, if the non-random cover point is properly sampled during the first few cycles, it is collected as a targetable hole at the next collection.

If transition bins are involved in cover points that contribute crosses, VCS ignores the cross bins (including transition bins) when collecting the holes.

For more information about Echo, see the Echo chapter under the LCA category in the *VCS / VCS MX Online Documentation*.

Reviewing Exclusions Using Adaptive Exclusion Flow

Adaptive exclusion is a DVE process you can use to review and reuse an exclusion. To facilitate this, the adaptive exclusion use model refers to an original version of an exclusion file E_A from design version A and the use of E_A with subsequent version B of the same design. The reuse of exclusion files is important in the coverage flow because it improves productivity.

The adaptive exclusion review flow works with all metrics, including line, toggle, FSM, branch, assertion, and covergroup.

- Exclusion reviews can take place at a low granularity in the navigation pane.
- Exclusions reviews of high granularity use signatures and take place in the detail panes.

Note:

Adaptive exclusion is supported for the module-based checksum `elfile` and new database. The old database-based `elfile` format is not supported.

This section explains how to use adaptive exclusion in the following subsections:

- “[Adaptive Exclusion Review Flow](#)” on page 169
- “[Loading an elfile into DVE](#)” on page 170
- “[Understanding the Review Markers in DVE GUI](#)” on page 174
- “[Exclusion Reviews in the Navigation Pane](#)” on page 176
- “[Understanding Exclusion Signatures](#)” on page 182

- “Signature in Elf file for Branch Metric” on page 187
 - “Exclusion Reviews in the Detail Pane” on page 189
 - “Unmappable Exclusions” on page 192
-

Adaptive Exclusion Review Flow

The review flow is to compile and simulate design version A, create an exclusion file E_A based on design version A, and modify the design to create design version B. The goal is to make it easy for you to reuse exclude file E_A on design version B. In order to reuse the exclusion file on the modified part of the design, DVE coverage allows you to review the existing exclusions, then decide what exclusions you can reuse, and which cannot be applied on the newer version of the design.

To review an exclude file with checksum mismatches:

1. Load the coverage database generated with design B in interactive mode.
2. Load the file E_A .
 - 2.1 If you select `bypass_check` to load the elf file, then all exclusion with mismatch signatures are directly applied without the checksum comparison.
 - 2.2 If you do not select `bypass_check`, then continue to Step 3.
3. If there is any checksum mismatch, then DVE opens up all exclusions in the modified design scopes.
4. Select a scope or module that contains unreviewed exclusions and go to the Detail Pane for further review.

5. Review the exclusions using the procedure explained in “[Exclusion Reviews in the Detail Pane](#)” on page 189.
6. If there is an unmappable exclusion, you can use the Unmappable dialog box to review all unmappable exclusions.
 - 6.1 In the Unmappable dialog box, you can mark the exclusion as reviewed.
7. Repeat steps 4 to 6 for every metric until all objects in the instance or module are reviewed completely.
8. Recalculate to apply all reviewed results.
9. Save as a new `elfile` (optional).
10. Save the new exclude file.

Note:

Once a reviewed exclusion is recalculated, it cannot be changed back to unreviewed state. To retrieve the unreviewed exclusions, click **Clear Reviews** button  or **Edit > Review > Clear Reviews**, and then reload the `elfile`.

Loading an `elfile` into DVE

Enabling and Disabling Adaptive Exclusion Review Mode

Adaptive exclusion review mode is enabled by default. The following sections describe how to configure Adaptive exclusion review mode.

Disabling Adaptive Exclusion Review Mode

Use the `-excl_resolve off` option or the DVE Application Preferences dialog box to disable Adaptive exclusion review mode. To disable Adaptive exclusion review mode using DVE Application Preferences dialog box:

1. Select **Edit > Preferences**.

The Application Preferences dialog box appears.

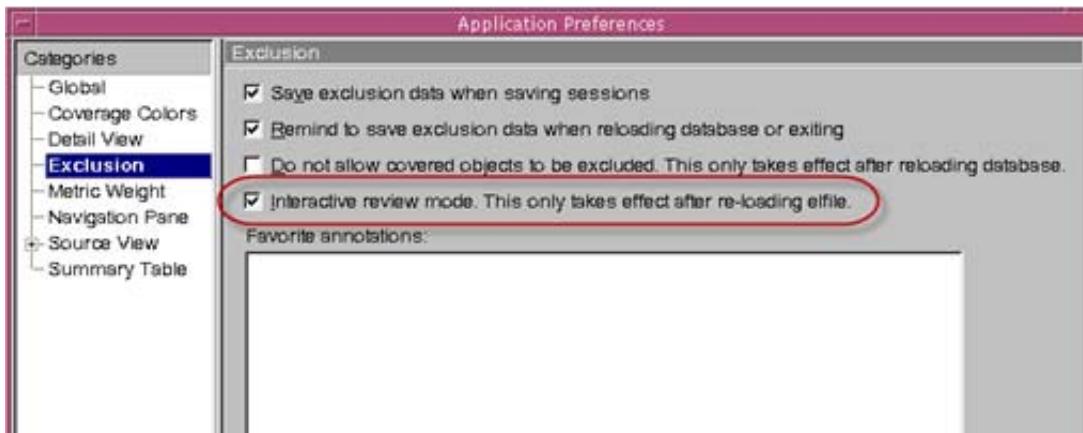
2. In the **Exclusion** category, clear the **Interactive review mode. This only takes effect after re-loading elfile** check box.
3. Click **OK**. This will disable Adaptive exclusion review mode.

DVE makes this mode as default until you again enable Adaptive exclusion review mode.

Enabling Adaptive Exclusion Review Mode

You can enable Adaptive exclusion review mode by using the `-excl_resolve on` option or by selecting the **Interactive review mode. This only takes effect after re-loading elfile** check box in the **Exclusion** category of the DVE Application Preferences dialog box, as shown in [Figure 8-1](#):

Figure 8-1 Enabling Adaptive Exclusion Review Mode

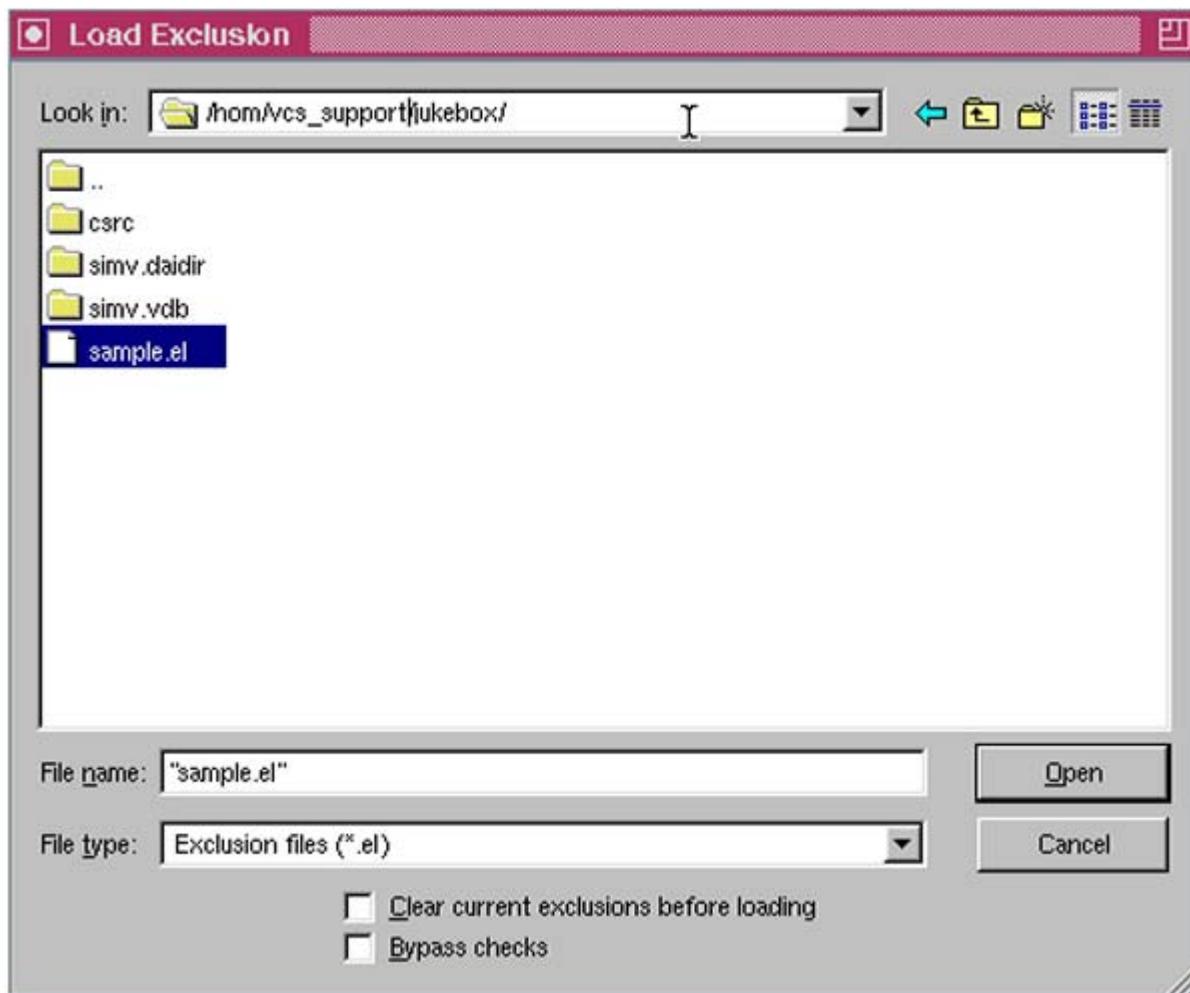


Loading Exclusion Files

If no checksums for a given module or instance are changed, then exclusions are reused for that region without your review. If any checksums for a given module or instance have changed, it is called a changed region.

You must review all exclusions in each changed region, unless you select the **Bypass checks** option. This option appears as a check box in the **Load Exclusion** window in DVE, as shown in [Figure 8-2](#).

Figure 8-2 Load Exclusion Window



The **Bypass checks** option is deselected by default in adaptive exclusion mode.

- If you select the **Bypass checks** option, the checksum comparison is ignored and DVE aggressively excludes the objects.
- If you deselect the **Bypass checks** option, all mismatched exclusions are loaded and opened for review. This allows the use of adaptive exclusion.

After loading an exclude file, if there is any checksum mismatch, DVE prints the summary report of the mismatches in the console window. A sample summary report is shown below:

```
The exclude file "sample.el" is loaded, checksum conflicts  
are found,  
Modules with checksum mismatches: 4  
Instances with checksum mismatches: 5  
Unmappable modules: 2  
Total exclusions need to review: 30  
Total unmappable exclusions: 2
```

Understanding the Review Markers in DVE GUI

After you invoke DVE and load an exclude file with the **Bypass checks** option deselected, markers are used to facilitate the review of the exclusion changes. DVE denotes exclusions with different markers, which are based on their status in review process.

[Table 8-3](#) shows the marker icons for mappable exclusions and explains what they mean.

Table 8-3 Review Markers for Mappable Exclusions in DVE GUI

Icon of Markers	Review Status
?	Its children or subchildren contain an unreviewed exclusion and signature mismatching.
?	The container contains unreviewed exclusions and signature mismatching for child objects.
?	Its children or subchildren contain an unreviewed exclusion and signature matching.
?	The container contains unreviewed exclusions and signature matching for child objects.
?	Unreviewed and signature matching.
?	Unreviewed exclusion and signature mismatching.
!	Unmappable objects or scope.
✓	Review accepted.
✗	The container contains accepted exclusion for child objects.
✗	Reviewed and rejected.
✗ ✓	Accepted and recalculated.

[Table 8-4](#) shows the marker icons for unmappable exclusions and explains what they mean.

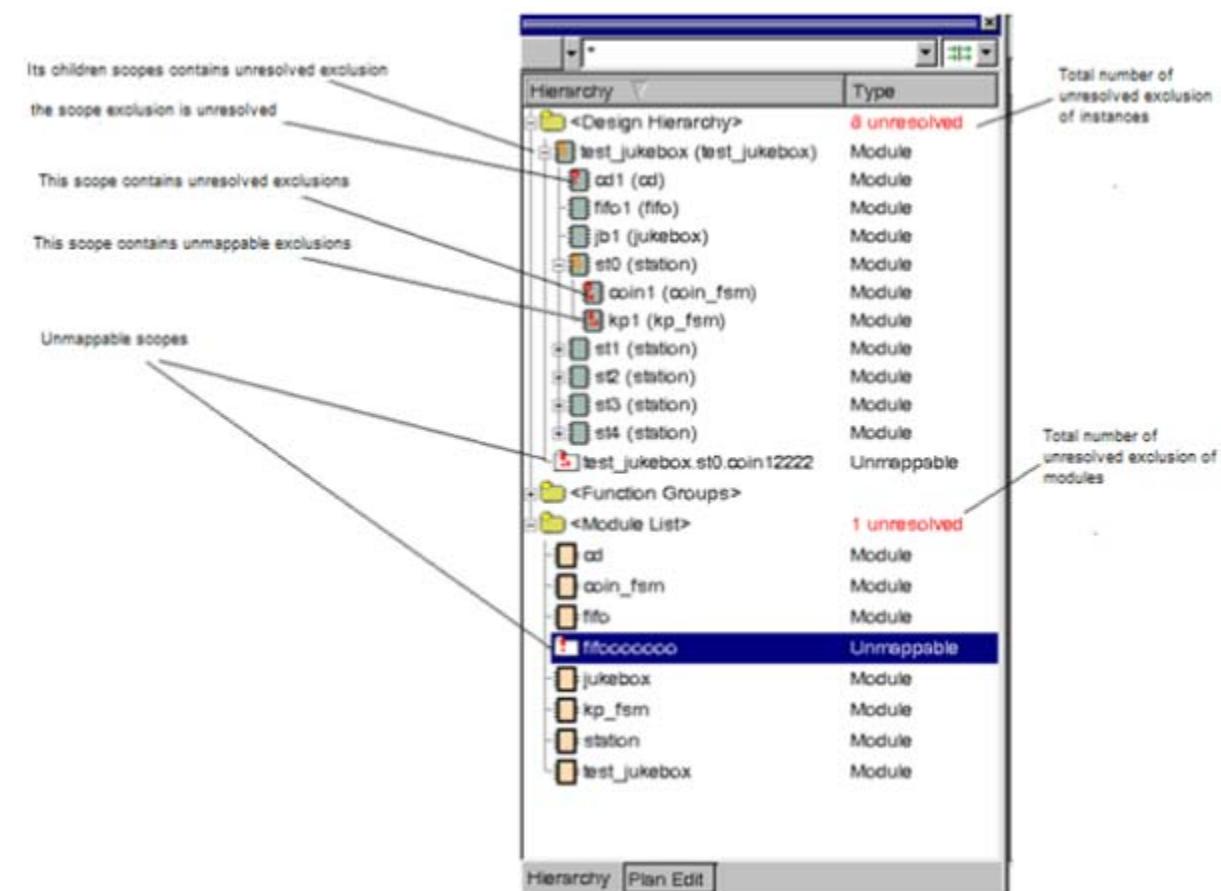
Table 8-4 Review Markers for Unmappable Exclusions in DVE GUI

Icon of Markers	Review Status
!	The unmappable exclusion is in unreviewed status.
!	The unmappable exclusion is reviewed.
	Unmappable scope or module.
!	The unmappable exclusion is reviewed and recalculated.

Exclusion Reviews in the Navigation Pane

Exclusion reviews at a high level of granularity (for example, module or instance) are performed in the Navigation pane. [Figure 8-3](#) shows the Navigation pane with review markers.

Figure 8-3 Navigation Pane with Review Markers

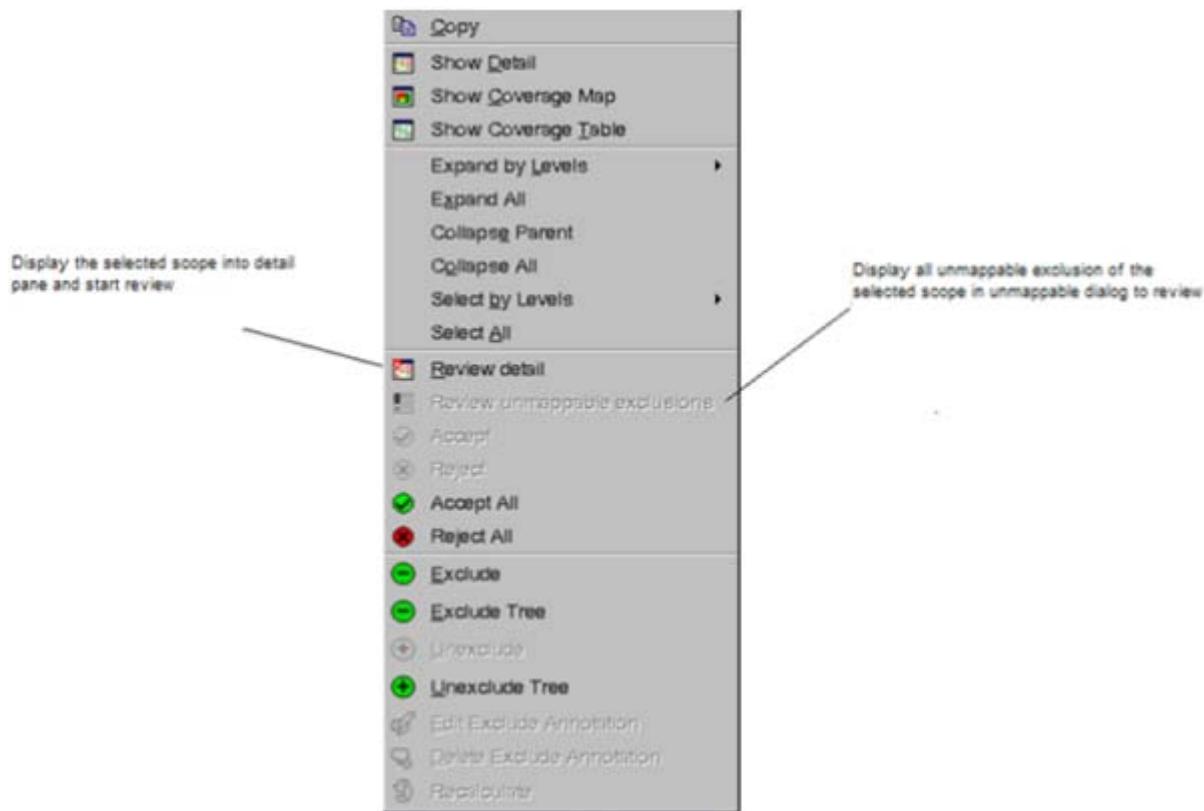


The **?** marker (shown in [Table 8-3](#)) is used to indicate that a given instance or module contains exclusions which you have not yet reviewed. DVE counts and displays the total number of unreviewed exclusions at **<Design Hierarchy>**, **<Function Groups>**, and **<Module List>** headings in the Hierarchy pane, as shown in [Figure 8-3](#), so you know if there are any unreviewed exclusions remaining. If an exclusion is reviewed, then the total number should decrease by 1. Once the total number becomes 0, it turns green.

Exclusions with unmappable module names are grouped together, sorted by their module names and shown in the **<Module List>** with the icon.

The Navigation pane has a context-sensitive menu (right-click the review marker), as shown in [Figure 8-4](#).

Figure 8-4 Context-Sensitive Menu in the Navigation Pane



In [Figure 8-4](#), note the following menu selections:

- Review detail — Starts to review with the selected regions (instances or modules) in detail panes. This is enabled only when at least one unreviewed region is selected.
- Review unmappable exclusions — Displays all unmappable exclusions of the selected scope or module in the Unmappable dialog box for further review. This is dimmed if not applicable.

- Accept All — Provides two options, **All** and **Signatures matching**, to accept all exclusions or only those with signatures matching, respectively (see [Figure 8-5](#)).

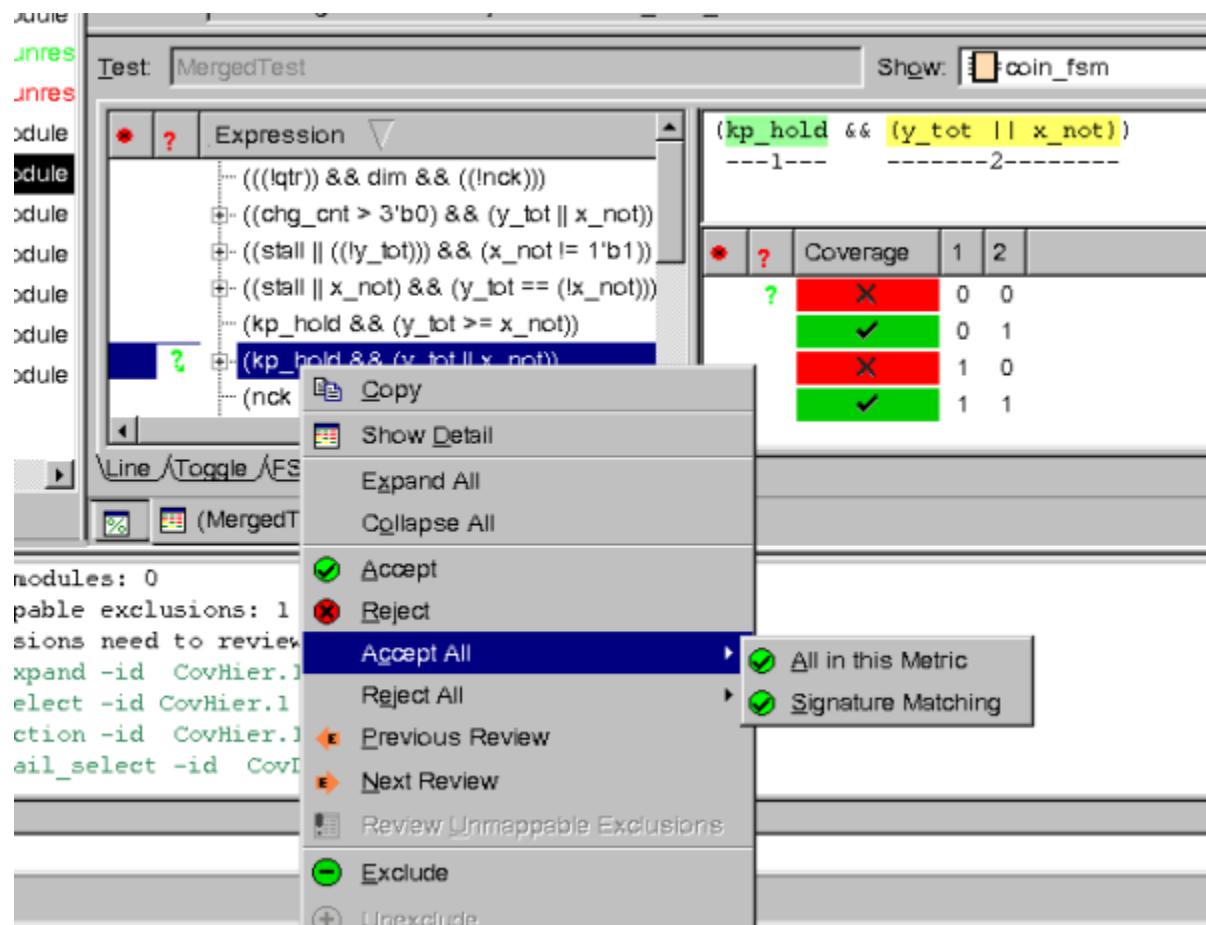
Figure 8-5 Options in Accept All Menu



- All in this metric — If you right-click a list item or source window in the Detail pane, then in this context, the **Accept All** menu displays the **All in this metric** option. You can choose this option to accept or reject all unreviewed items in the current metric.

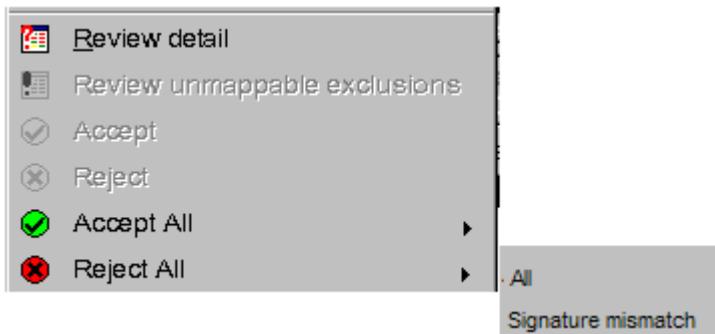
[Figure 8-6](#) shows the **All in this metric** option on the Detail pane, which shows coverage items of each metric.

Figure 8-6 All in this Metric Option in the Accept All Menu



- Reject All — Provides two options, **All** and **Signatures mismatch**, to reject all exclusions or only those with signatures that do not match, respectively (see [Figure 8-7](#)).

Figure 8-7 Options in Reject All Menu



The ToolTip in the Navigation pane displays unreviewed exclusions for each metric. For example:

```
Mappable : a/b/c  
Fsm: d  
Condition: e  
...  
Unmappable: f/g
```

where,

- $a/b/c$ means there are totally c mappable items, a accepted, and b rejected.
- d and e are total mappable numbers in each metric.
- f/g means there are totally g unmappable, and f reviewed (you can only review unmappable; cannot accept or reject).

Understanding Exclusion Signatures

For high-granularity exclusion reviews, you need to review signatures to decide to keep or discard the exclusion.

A signature describes an excluded object in the exclusion file. Such an object can be an entire expression or a vector of an expression.

Signature in Elf file for Condition Metric

Consider the following code, which contains two conditional expressions:

```
102:     if (stall || (x_not && y_tot))  
103:         n_state = five;  
104:     else if (qtr && !dim && !nck)
```

Both of these expressions contain excluded vectors. Also, the exclusion on line 102 is annotated with This exclusion is for testing.

The elf file contains the following information:

```
INSTANCE: test_jukebox.st0.coin1  
Condition 6 "(stall || (x_not && y_tot)) 1"  
ANNOTATION: " This exclusion is for testing "  
Condition 8 "(qtr && ((!dim)) && ((!nck))) 1"
```

Note:

The signature that mentions Condition 6 is from line 102 and the signature that mentions Condition 8 is from line 104. Both exclusions are in the instance test_jukebox.st0.coin1.

If you change one of the above expressions, a review of all the exclusions in the instance is triggered. For example, if you change `stall` to `kp_hold` in the expression on line 102, as shown below:

```
102: if (kp_hold || (x_not && y_tot))
```

You then recompile, resimulate, and load the new coverage database into DVE along with the exclusion file. As the checksum is different for `test_jukebox.st0.coin1`, a review of the signatures for this instance is required.

Review information appears when you mouse-over an expression, and it also appears in the console window. The information in the console window contains:

- The line number of the exclusion in the current design, and if the two signatures match or mismatch.
- The signature of the exclusion in the current design (if the two signatures match, then it is not shown).
- The signature of the exclusion from the exclude file.
- The annotation for the exclusion from the exclude file, if any.

The console displays the following two expressions.

```
Current exclusion under review is at Line 102, signatures mismatch:
```

```
Database signature: if (kp_hold || (x_not && y_tot))
```

```
Elfile signature:      if (stall || (x_not && y_tot))
```

```
Elfile annotation:    This exclusion is for testing.
```

```
Current exclusion under review is at Line 104, signatures match:
```

```
The signature from the exclude file is: (qtr && !dim && !nck)
```

The expression for the second exclusion is not changed, as its review is triggered by the change of the `cksum` of the instance.

An exclusion review of line 104 may seem unnecessary in this example, but consider that the `cksum` change occurred because the signals `gtr`, `dim`, or `nck` were given different assignments in the surrounding code. In this case, a review is important.

Signature in Elf file for Line Metric

VCS does not perform line coverage. Instead, it performs basic block coverage. A basic block is a chunk of statements that always execute together. For example:

```
if (w) begin
    a <= 1'b0;      (1)
    b <= w & z;    (2)
end
```

In this example, statements (1) and (2) make up a single basic block. When any statement in a basic block executes, they all do (by definition). Line coverage only monitors which basic blocks are covered during simulation.

The signature for an excluded statement for line coverage is:

```
<the text of the starting line of the block in
source code>
```

Multiple basic blocks can appear on a single line of text. In this case, the signature of an excluded statement still includes the entire source text of the line.

Adaptive Exclusion Basic Block Limitations

The following limitations apply to basic blocks in the adaptive exclusion flow:

Incomplete consideration of source

Because only the first line is considered in signatures, any changes to subsequent lines of the basic block go undetected. In this case, you can see false signature matches.

Over consideration of source

When multiple basic blocks exist on the same line, all of them contain the same signature. Even in this case, you can see false signature matches.

For example:

```
#1 a = 0; #1 b =1;
```

This line completely contains the following basic blocks:

```
b1: a = 0; #1 (say block id = 3)
b2: b = 1; #1 (say block id = 4)
```

Both b1 and b2 contain the same signature, since the line number is same.

If this module is changed and one additional basic block gets into the code before this line, then the block IDs of these two blocks are shifted down by 1 (4 for b1 and 5 for b2). So id = 4 points to different basic blocks in the initial and modified designs, while retaining the same signature.

Signature in Elfile for Toggle Metric

The signature for excluding a signal for toggle coverage is:

<type> <signal_name> [msb:lsb] for vector

<type> <signal_name> for scalar

The toggle direction is not included in the signature.

Toggle exclusions are mapped to objects in the design by signal names. If no signal exists in the given region with the same name, then the exclusion goes into the unmappable list.

Signature in Elfile for FSM Metric

FSM coverage contains three types of exclusions:

- Whole FSM
- State
- Transition

The signature for an FSM state exclusion is:

- Name of the FSM current state signal
- Value of the state
- Name of the state

The signature for an FSM transition exclusion is:

- Name of the FSM current state signal
- Values and names of both states

FSM exclusions are mapped using the current state signal name and the names of the states involved. If an exclusion exists for a state signal that no longer exists, or a state whose name no longer exists, the exclusion is categorized as unmappable. For example, for transition `fiftn->idle`, it should be `8->1`, where `8` is the value of `fiftn` and `1` is `idle`.

Signature in Elfile for Branch Metric

The signature information for a whole branch block is:

<the top condition expression>

The signature information associated with a branch is:

<the top condition expression> <branch vector>

Consider the following example:

```
if (a&&b)
    if(d)
        //Branch 1
        else if(e&&b)
            //Branch 2
            else
                //Branch 3
        //Branch of MISSING_ELSE
```

Branch block signature: a&&b

The signatures for these branches are:

Branch 1: a&&b 11

Branch 2: a &&b 101

Branch 3: a&&b 100

Branch of MISSING_ELSE: a&&b MISSING_ELSE

For case switching, <as the separators for case> is used, as shown in the following example:

```
if (b)
case(sw)
53: ... Signature: b 1'53'
```

Note:

With this signature scheme, all changes to the top expressions are detected and lead to signature mismatches. The disadvantage is that changes in the structure or inner expressions of the branch are not detected.

Signature in Elfile for Assertion Metric

The signature for an assertion exclusion is the name and type (assertion, cover property, cover sequence) of the assertion.

For unnamed assertions, VCS assigns names using a counter. If one unnamed assertion is deleted, then it effectively changes the names of all following unnamed assertions. So, for example, if an assertion unnamed\$\$_3 is excluded in design version A, and the module changes, then the signature is unnamed\$\$_3. Because it is difficult to know whether or not to accept the exclusion, it is recommended to name the assertions.

Note:

Changes in assertion content are not detected, because they are not part of the signature.

Signature in Elf file for Covergroup Metric

Covergroup bins also contain unique names that are used for coverage exclusion. The signature of a covergroup bin is its name. User-defined bins contain user-defined names. Automatically created bins contain names generated according to the LRM rules.

Exclusion of entire covergroups and crosses is done by name only for those with user-defined labels (ignoring the lists of bins, and so on).

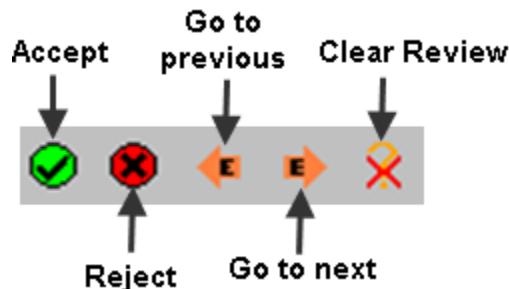
Exclusion Reviews in the Detail Pane

This section explains how to use the Detail pane in DVE to perform high-granularity exclusion reviews.

Icons in the Review Toolbar

You select an instance or module to populate the Detail pane by clicking the **Review detail** icon in the context-sensitive menu described in the previous section. You can then click the icons in the Review toolbar (see [Figure 8-8](#)) to walk through exclusions and review them one by one.

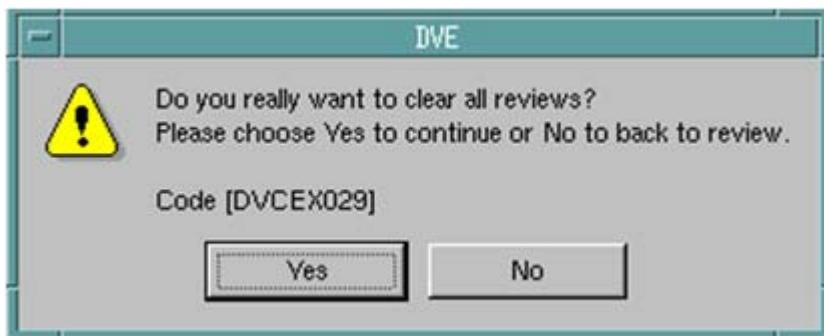
Figure 8-8 Review Toolbar



Following are brief descriptions of the icons:

- **Accept** — When you click this icon, it marks the current exclusion under review as accepted. DVE then automatically moves forward to the next unreviewed exclusion.
- **Reject** — When you click this icon, it marks the current exclusion under review as rejected. DVE then automatically moves forward to the next unreviewed exclusion.
- **Go to previous** — Use to move to the previous unreviewed exclusion.
- **Go to next** — Use to move to the next unreviewed exclusion.
- **Clear Review** — Use to undo your previous reviews. This is useful if you want to start the review from scratch. Clicking this icon displays the message box shown in [Figure 8-9](#).

Figure 8-9 Confirmation Window for Clear Review Option



Click **Yes** to clear all the review marks (accept or reject mark along with any condition open for review).

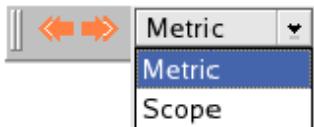
Note:

After clearing all reviews, you need to reload the exclusion to continue the review process.

Metric and Scope Menus in Review Toolbar

To support the exclusion review flow for all metrics, the **Metric** and **Scope** menus are added in the Review toolbar as shown in [Figure 8-10](#).

Figure 8-10 Metric and Scope Menus in Review Toolbar



If you select **Metric**, then **Go to next** takes you to the next metric of the current instance or module that contains unreviewed exclusions. If there is no next metric to review, then the **Go to Next** button is disabled.

If you select **Scope**, then **Go to next** takes you to the next instance or module. It goes through instances first, then modules. If there is no next scope or instance available to review, then **Go to next** is disabled. When a new instance or module is populated in the Detail pane, the Summary report is printed in the console and the Detail pane switches to the first available metric for review.

- In code coverage, the sequence of **Go to next** metric is: Line -> Toggle -> Condition -> FSM -> Branch -> Assert.
- In function coverage, if you select **Scope** as the criterion, then two buttons are disabled, but **Metric** is allowed. The sequence is: Cover-Group -> Assertion.

When you decide to accept or reject an exclusion, you need to compare the signature from the database to the signature from the elfile to see if they match.

Any and Conflict Menus in Review Toolbar

A combo-box with **Any** and **Conflict** menus is added into the Review toolbar, as shown in [Figure 8-11](#).

Figure 8-11 Any and Conflict Menus in Review Toolbar



If you select **Any**, then **Go to next** goes through any unresolved exclusion. If you select **Conflict**, then **Go to next** goes through an unresolved exclusion with signature mismatching only.

Saving and Restoring a Review Session

DVE supports save and restore. So when you save a session file, all reviewed exclusions get dumped into the session Tcl file. If you restore a session Tcl file, then all reviewed exclusions are retrieved from the session file.

Unmappable Exclusions

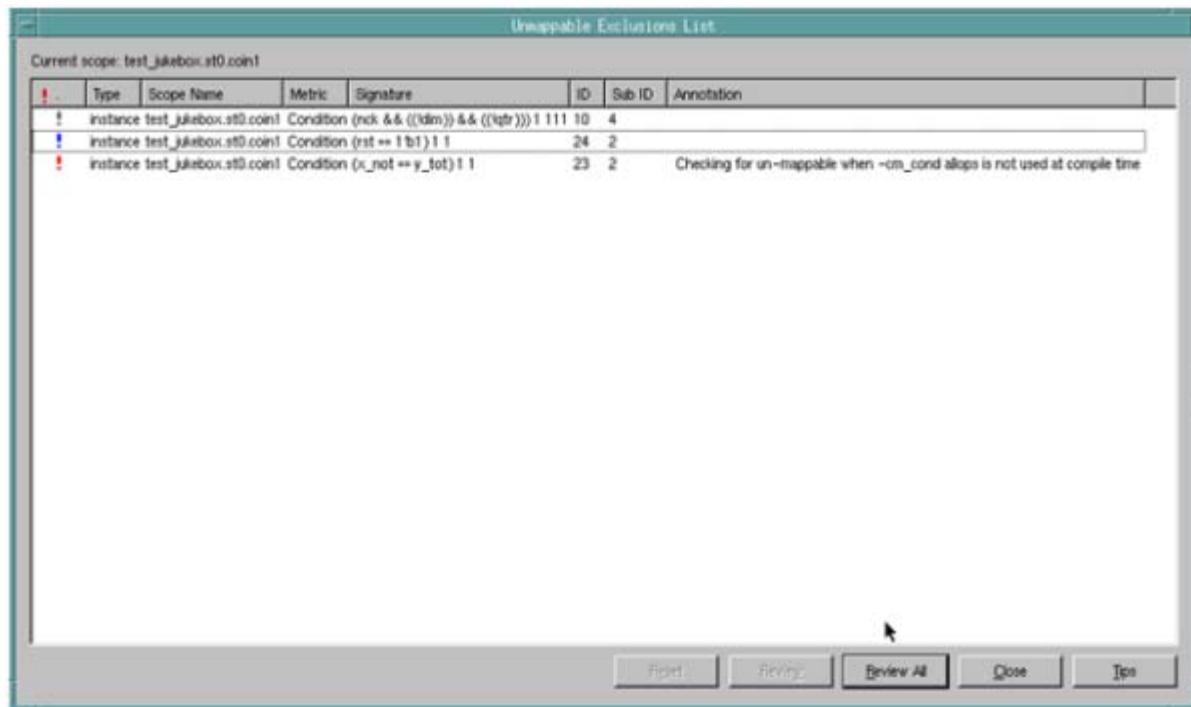
Sometimes, you may have a module, instance, or expression that has changed a lot. In such cases, the excluded objects in the elfile cannot be mapped to the changed design. To display unmappable exclusions, you can use the **Unmappable exclusions** dialog box.

To invoke the Unmappable exclusions dialog box:

- From **Edit > Review**, select **Show unmappable exclusions**. The Unmappable Exclusions List dialog box appears, displaying all unmappable exclusions.
- (or)
- Select a scope or module. Then from the context sensitive menu, select **Review unmappable exclusion**. The dialog box displays unmappable exclusions of the selected scope or module.

[Figure 8-12](#) shows the Unmappable Exclusion List dialog box displaying a few unmapped exclusions.

Figure 8-12 Unmappable Exclusion List Dialog Box



The two modes of unmappable exclusion are:

- Not reviewed !
- Reviewed !

After recalculation, it is displayed as !.

The dialog box is synchronized with the current detail window and only displays one metric at a time. When the metric in the detail window changes this dialog box is updated with the new metric.

The five buttons in the Unmappable Exclusion List dialog box are:

- **Reset** — Marks all selected exclusions as Not reviewed.
- **Review** — Marks all selected exclusions as Reviewed.
- **Review All** — Marks all exclusions in the dialog box as Reviewed.
- **Close** — Closes the dialog box.
- **Tips** — Provides review tips.

Coverage Analysis of Unreachable Verilog Code

In previous versions, VCS analyzed the operands of a condition to determine which vector combinations were unreachable. It could not determine whether a condition was present within an unreachable block of code or not. Consider the code shown in [Example 8-13](#).

Example 8-13 Verilog Block

```
if (x)
begin
    ....
end
else
begin
    if (a || b)
    ....
end
```

In [Example 8-13](#), if `x` is computed to have constant value 1 at compile-time, the entire `else` block becomes unreachable, which makes the condition `a || b` unreachable as well. In previous versions, VCS did not consider control flows like this, and reported all the vectors of this condition as uncovered.

Also in previous versions of VCS, you could only specify instance-specific constant signals in the constant configuration file. VCS now allows you to specify module-specific constant signals to mark a signal as constant for all instances of a module.

VCS now supports the following enhancements for coverage analysis of unreachable Verilog code:

- Identifies conditions within unreachable blocks of Verilog code and removes them from coverage monitoring. For more information, see “[Identifying Conditions in Unreachable Blocks of Verilog Code](#)” on page 196.
- Allows you to specify module-specific constant signals in the constant configuration file (-cm_constfile option). For more information, see “[Specifying Module-Specific Constant Signals in Constant Configuration File](#)” on page 198.

Identifying Conditions in Unreachable Blocks of Verilog Code

VCS now detects unreachable conditions by analyzing procedural blocks, evaluating the conditions controlling those blocks, and determining which blocks are actually unreachable. Consider the code shown in [Example 8-14](#).

Example 8-14 Unreachable Verilog Block #1

```
22 case (my_case_var)
23 2'b10 : a = b & c
24 2'b11 : a = b | c
25 default : a = b ^ c;
26 endcase
```

The constant configuration file contains the following line:

```
Top.M1.my_case_var 2'b10
```

In [Example 8-14](#), the conditions at lines 24 and 25 are unreachable for instance `top.M1` regardless of the values of `b` and `c`.

Next, consider the code shown in [Example 8-15](#).

Example 8-15 Unreachable Verilog Block #2

```
27  if (x || y)
28    begin
29      $display(" in IF");
30    end
31  else begin
32    if ( a || b ) begin
33      $display(" In nested IF");
34    end
35  end
```

The constant configuration file contains the following line:

```
Top.M1.x 1'b1
```

In [Example 8-15](#), the condition at line 40 is unreachable because $x || y$ evaluates to constant 1 regardless of the value of y .

VCS now detects these type of unreachable conditions under any level of nesting of control statements, including

- case block within an if block
- if block present within a case block and so on.

VCS does this analysis for all selection statements, including case, casex, casez, if-else, priority-if, unique-if, priority-case, unique-case, except when comparisons with x or z values are involved. For more information, see “[Module-Specific Constant Limitations](#)” on page 200.

All conditions found to be unreachable because of block-level unreachability are marked unreachable in coverage report.

VCS also supports marking a condition in ternary condition as unreachable if that condition is never evaluated, as ternary condition evaluates to be constant. For example, consider the following code:

```
assign r= a? b&&c : d || e ;
```

Here, if a is a constant and has value 0, all vectors of condition b&&c are marked unreachable in URG.

Specifying Module-Specific Constant Signals in Constant Configuration File

You can use the VCS `-cm_constfile` coverage option to specify compile-time constants which VCS cannot compute automatically. In previous releases, such constants could only be specified using full instance names. VCS now allows you to specify module-specific constant signals within a constant configuration file (`-cm_constfile` option). Following is the syntax:

```
<Full-Signal-Name> := <Module-Name>. <Signal-Name>
```

where,

```
<Module-Name> := [<library-Name>.] Name-Identifier
```

For example, to make signal `x` a constant 0 for module `M` and the module is inside library `LIB`, specify the following in the `-cm_constfile`:

```
\LIB.M .x 1'b0
```

For register constants, use `-cm_constfile` to specify them explicitly. You must also use the `-cm_noconst` option for the constant analysis to happen.

For a particular signal, if both module-specific and instance-specific constant signals are specified in the constant configuration file, the constant value (module-specific or instance-specific) specified first is given preference for the particular instance.

For example, if the content of a constant configuration file is:

```
\LIB.M .x 1'b0  
Top.M1.x 1'b1
```

Here, `M1` is an instance of module `M` instantiated in another module `Top`, and `x` is a signal in module `M`. Even though you explicitly assign value `1'b1` to signal `x` for instance `M1`, because the module-specific constant is specified first in the constant configuration file, it is given preference, and signal `x` gets a constant value of `1'b0`.

However, if the content of a constant configuration file is:

```
Top.M1.x 1'b1  
\LIB.M .x 1'b0
```

Then signal `x` gets value `1'b1` for instance `M1`, because the instance-specific constant is specified first in the constant configuration file.

Module-Specific Constant Limitations

Note the following limitations when working with module-specific constants:

- VHDL is not supported.
- No analysis is done to determine if registers are constant in a design. For example, if you have a register declared as:

```
reg [1:0] my_case_var;
```

And you initialize it at the beginning of an initial block as follows:

```
initial  
begin  
my_case_var = 2'b10;
```

And the always block where `my_case_var` is driven is:

```
always @ (posedge clk)  
begin  
if (a)  
    my_case_var = 2'b10;  
else  
    my_case_var = 2'b11;  
end
```

And the constant configuration file contains:

```
Top.M1.a 1'b1
```

Then, the register `my_case_var` always gets constant value `2'b10`. But you cannot determine such constants for registers.

- VCS does not perform constant analysis within any kind of task or functions.

- Constant X or Z values are not considered for constant analysis. Any comparison which involves X or Z values is not determined as constant. For example, for the following code:

```
wire [2:0] w;
assign w = 2'b0x;
initial
begin
    case (w)
        2'b00 : $display("00");
        2'b0x : $display("0x");
        default : $display("default");
    endcase
end
```

The value of w actually matches with case-item 2'b0x. However, VCS is unable to determine this kind of constant because an X value comparison is involved.

Again, for the following example:

```
wire [2:0] w;
assign w = 2'b01;
initial
begin
    casex (w)
        2'b00 : $display("00");
        2'b0x : $display("0x");
        default : $display("default");
    endcase
end
```

Here, case item 2'b0x is always reachable, whereas default and 2'b00 are unreachable.

- Control flow constructs such as while or for are not supported for control flow-based reachability for condition coverage.

Support for with Clause in Cover Groups

VCS now extends the syntax of cover point and cross bins to support with clause, as described in the following sections. The with clauses help you to define cover points clearly, which otherwise needed a lot of writing in cover group specification.

You can use this enhancement to precisely specify the functional coverage space for your SystemVerilog based verification environments. This enhancement is applicable only for regular bins (transition bins will not be supported).

Cover Point Syntax Enhancement

The syntax for specifying ranges for cover point bins is extended to include with clause as follows:

```
bins_or_options ::=  
  
[ 'wildcard' ] bins_keyword bin_identifier [ '[' [  
expression ] ']' ] = '{' open_range_list '}' [  
'with' '(' with_expression ')' ] [ 'iff' '('  
expression ')' ]  
  
| [ 'wildcard' ] bins_keyword bin_identifier [  
'[' [ expression ] ']' ] = cover_point_identifier  
'with' '(' with_expression ')' [ 'iff' '('  
expression ')' ]
```

Where,

```
bins_keyword ::= bins | ignore_bins | illegal_bins
```

```
with_expression ::= expression
```

The `with` clause further tightens the range of values specified through `open_range_list` and subsumed by a cover point bin. Essentially, each value in the `open_range_list` must satisfy the condition represented by the `with_expression` in order to be considered a part of the bin. The `with_expression` can be an arbitrary expression (including function calls) with the cover point identifier used as one of the operands in the expression. Instead of the cover point identifier, the keyword `item` should be used.

Additionally, the cover point name can be used to substitute the whole range of values taken by the cover point variable. The evaluation of the bin ranges and the `with_expression` is done at the time of cover group instantiation. All variables, barring the cover point name and the `item` keyword are contained in the `with_expression`.

They are treated as run time constants and are evaluated at the time of cover group instantiation; the treatment is similar to the one used for guard expressions.

Examples

```
bit [7:0] x;
a: coverpoint x
{
  bins mod3 [] = { [0:255] } with (item % 3);
}
```

This bin definition selects all values from 0 to 255 that are not divisible by 3.

Since the range list above includes all values of the cover point variable, it can actually be substituted simply by the name of the cover point, as shown below:

```
bins mod3 [] = a with (item % 3);
```

An example for function calls used in “with” expression:

```
bins mod3 [] = { [0:255] } with (myFunc(item));
```

If none of the values in the `open_range_list` satisfy the `with_expression`, the bin is deemed to be excluded from the cover point and is dropped. This is similar to specifying an invalid (or a null) `open_range_list`. VCS currently generates a runtime warning to indicate that such a bin is dropped and extend the warning for bins determined to have a null solution set satisfying `with_expression`:

Warning- [FCPSBU] Invalid values in bin

For unconstrained array bins, VCS puts a threshold of 50000 on the number of distinct values that can be specified in `open_range_list`, and issues a runtime error when this threshold is breached. This logic will be extended for bins under `with_expression`; when the number of values of `open_range_list` satisfying `with_expression` exceeds the set threshold, the following error is issued and the bin is dropped:

Error- [FCABEL] Array bin limit exceeded

Cover Cross Syntax Enhancement

The syntax for specifying ranges for cover point bins will be extended to include `with` clause as follows:

The syntax for specifying the cover point bin combinations subsumed by a cross bin will be enhanced as follows:

Additions to Syntax

```
select_expression ::=  
...  
| select_expression 'with' '(' with_expression ')' '  
| cross_identifier  
  
with_expression ::= expression
```

The `with` clause for the cross further restricts the set of cover point bin combinations (or ‘tuples’) that are specified by the select expression `select_expression`. The subsumed combinations must satisfy `with_expression`. Instead of `select_expression` (that is, expression with `bins`), you can also specify the cross identifier itself (that is, the name of the cross), to indicate that the `with` clause needs to be applied to all the bin combinations for the cross. As mentioned earlier, while discussing cover point bins, `with_expression` can be an arbitrary expression (including function calls), and involves the cover points that compose the cross. The expression should have at least one cover point (specified by the name of the cover point) as an operand.

Examples

```
logic [0:7] a, b;  
  
covergroup cg(bit [0:7] mask);  
    a1:coverpoint a  
    {  
        bins foo[] = { [0:127] };  
        bins bar = { [128:255] };  
    }  
    b1:coverpoint b  
    {  
        bins two[] = { [0:255] } with (item % 2);  
    }
```

```

        bins three[] = {[0:255]} with (item % 3);
    }
X: cross a1,b1
{
    bins cherry = (binsof(b1) intersect {[0:50]} &&
binsof(a1.foo) intersect {[0:50]}) with (a1==b1);
    bins plum = (binsof(b1.two) with (b1 > 12)) ||
(binsof(a1.foo) with (a1 & b1));
    bins apple = X with (a1+b1 < 100);
}
endgroup

```

The cross bin `cherry` demonstrates using the `with` clause on a complex `select_expression`. First, those bin tuples consisting of a bin from `b` containing a value between 0 and 50 are selected; then, the `&&` operator selects from those bin tuples ones with a bin from `a.foo` containing a value between 0 and 50. The `with` clause then selects from those only the bin tuples containing at least one value tuple where `a==b`.

The cross bin `plum` demonstrates a `select_expression` composed of 'with' expressions. The first `with` expression selects those bin tuples containing bins in the `b.two` bin array whose values are greater than 12. The `||` operator then adds the bin tuples selected by the second 'with' expression (those containing a bin from `a.foo` and for which the bitwise AND of the `a` value, `b` value, for some values `a` and `b` in the bins of the bin tuple).

Finally, bin `apple` illustrates a bin specification written without the `bins of` or the `intersect` constructs. Similar to the cover point bins, the name of the cross can be used to represent the whole cross space. This bin subsumes all bin combinations where the sum of the values of the two constituent cover points is below 100.

Limitations

- This feature does not support the solver for constraints involving function calls.
- The `-covg_dump_range` feature, meant for displaying the values subsumed by bins in coverage reports, will not be supported for regular bins involving large bin ranges and having 'with' expressions on them.
- A run time error is issued for bins which uses with clause and has 32767 matching items.

```
Error- [FCWCBREL] State value ranges limit exceeded /slowfs/vgpv3/ndutta/with_clause/ range_error.v, 19 top, "cov1"
```

The number of state value ranges evaluated by the 'with' clause on bin 'b1' of coverpoint 'c1' in covergroup 'cov1' exceeds the limit '32767'.

Covergroup instance: 'cg1'

Design hierarchy: 'top'

For Example:

```
integer i1;  
.....  
c1: coverpoint i1 {  
    bins b1 = {[$:$]} with( item%2 == 1);  
}  
.....
```


9

Debug Features

This chapter contains the following features:

- “Enhancement in vpd2vcd Utility” on page 210
- “Reducing Disk Space for Post-process only Debug” on page 210
- “Reducing Compile Time for Post-process only Debug” on page 212
- “Using Socket-based Communication in UCLI and DVE” on page 213
- “Viewing the Full Design Hierarchy in a Partially Dumped VPD” on page 222
- “Integrating DVE with Protocol Analyzer”

Debug Features

Enhancement in vpd2vcd Utility

The vpd2vcd utility converts a VPD file generated using the system task \$vcdblplus or UCLI dump commands to a VCD or EVCD file.

A new command-line option `-expand_dumpport_scope` has been added in the vpd2vcd utility.

Usage

```
% vpd2vcd -f optionfile <vpdfile> <vcdfile> \
-expand_dumpport_scope
```

where,

`-expand_dumpport_scope`

Generates a VCD file with \$scope information in a hierarchical format, versus the default flattened format. Printing in a hierarchical format with this option matches the \$scope print format generated through an EVCD dumped from UCLI.

Reducing Disk Space for Post-process only Debug

If you want to perform only post-process debug, then you can use the `-debug_perf=splitdebugdir` compile-time option to significantly reduce the `simv.daidir` disk space. When this option is used, VCS creates the `debug_dump` directory in the `simv.daidir` directory at compile-time.

DVE uses the `debug_dump` directory to run in post-process mode, so even if the rest of the contents of `simv.daidir` are deleted, it will continue to work seamlessly. You can use the `trim_daidir` script file to delete the files that are not required for post-process debug flow.

Note:

The `-debug_perf=splitdebugdir` option will not enable dumping by itself, it will only create the `debug_dump` directory inside `simv.daidir`.

Use Model

Perform the following steps:

1. Use the `-debug_perf=splitdebugdir` compile-time option to create the `debug_dump` directory.

Example:

```
% vcs -sverilog -debug_all <other_vcs_options> -  
debugperf=splitdebugdir file_name.sv
```

2. Run the simulation once and dump the VPD file.

```
% simv <simv_options>
```

3. Use the `trim_daidir` script file, as shown below, to delete the files that are not required for post-process debug flow.

Example: `trim_daidir simv.daidir`

4. Perform the following command to invoke DVE.

```
% dve -vpd <vpdfile>
```

Limitations

- No files would be deleted for pure VHDL designs.

Reducing Compile Time for Post-process only Debug

If you want to perform only post-process debug with an existing VPD file and recreate `simv.daidir`, then you can use the `-static_dbgen_only` compile-time option and significantly reduce the compilation time.

Use Model

Perform the following steps:

1. Use the following syntax to compile your design file.

```
% vcs -debug_pp <other_vcs_options> file_name.v
```

2. Run the simulation once to generate the VPD file.

```
% simv <simv_options>
```

3. Use the `-static_dbgen_only` compile-time option, as shown below, to regenerate `simv.daidir` that is required to do post-process debug.

```
% vcs -static_dbgen_only -debug_pp  
<other_vcs_options> file_name.v
```

4. Perform the following command to invoke DVE.

```
% dve -vpd <vpdfile> (you can use all the DVE features and  
do post-process debug).
```

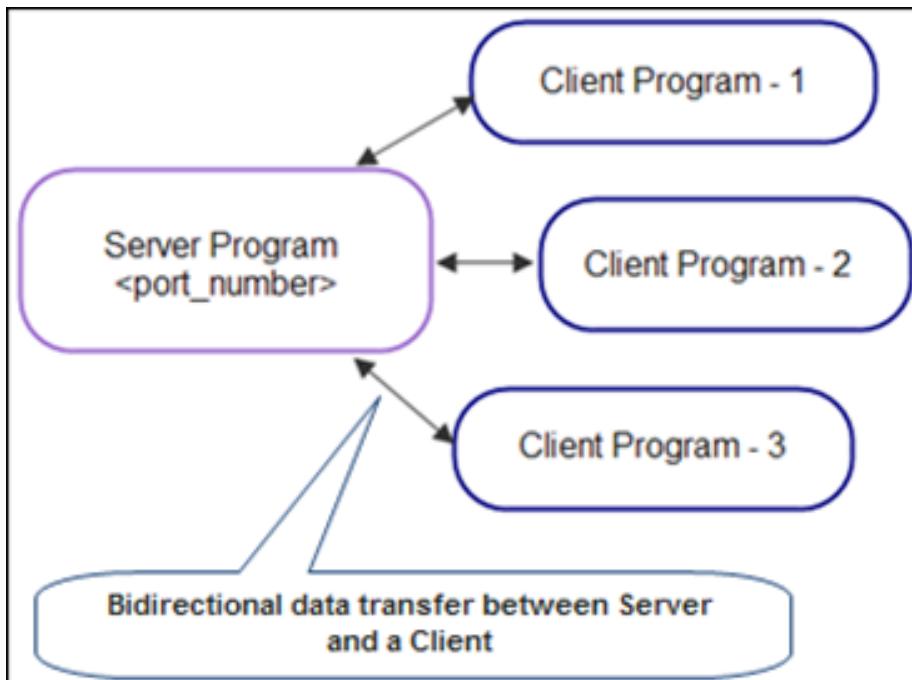
Debug-DVE Features

Using Socket-based Communication in UCLI and DVE

The Tcl fileevent command is used to establish socket-based communication between a client program and a server program. A socket is a software endpoint that establishes bidirectional communication between a server program and one or more client programs.

The socket associates the server program with a specific hardware port on the machine where it runs, so that any client program anywhere in the network with a socket associated with that same port can communicate with the server program. [Figure 9-1](#) illustrates client-server communication.

Figure 9-1 Socket-based Communication



DVE and UCLI now uses the Tcl fileevent feature (socket-based Tcl communication) between server program and a client program to execute UCLI commands using the `ucliCore::tempPause` and `ucliCore::tempPauseResume` UCLI commands. This feature is supported only in UCLI and Interactive DVE.

This feature is implemented in UCLI and DVE as follows:

Implementing Socket-based Tcl Communication in UCLI

In UCLI, if you specify inputs in the socket, then the simulator is temporarily paused using the `ucliCore::tempPause` command, to run the specified UCLI command. When the UCLI command is executed, then the simulation is resumed using the `ucliCore::tempPauseResume` UCLI command.

This feature allows you to establish an interface between:

- UCLI (server) <-> UCLI (client)
- DVE (server) <-> DVE (client)
- UCLI (server) <-> DVE (client)

The following example describes socket-based Tcl communication between UCLI (server) and UCLI (client):

UCLI (server):

```
./simv -ucli2Proc -ucli -do server.do
cat server.do
fileevent $ServerPort readable [list proc1]
proc proc1{} {


```

UCLI (client):

```
./simv -ucli2Proc -ucli -do client.do
set socket_name [socket $env(HOST) $SeverPort]
fconfigure $socket_name -buffering line
puts $socket_name xxxxx
```

Note:

In this scenario, you can specify a value through socket to invoke UCLI command call.

Implementing Socket-based Tcl Communication in DVE

In DVE, the flow is same as in UCLI. The simulator is paused to run the specified UCLI command. At the same time, you can stop the loop in the `do` script to debug interactively. In DVE, it also runs in two processes.

This feature allows you to establish an interface between:

- DVE (server) <-> Terminal (client)

This feature allows multiple users to access the same DVE GUI from different terminals. The following example illustrates socket-based communication between DVE (server) and Terminal (client):

DVE (server):

```
% ./simv -gui -do server.do
```

Terminal (client):

```
% telnet machine_name <port_number>
```

Using Tcl Fileevent

This section describes socket-based communication between DVE (server) and a terminal (client). For example, consider the following Tcl file `fileevent.tcl`. This file consists of essential code to implement this feature.

```
# -*- tcl -*-
# This callback is called when data is received from the
terminal
proc GetData {chan} {
    flush $chan
```

```

if {[eof $chan]} {
    set data [gets $chan]
    if {$data == "exit"} {
        close $chan
    } else {
        if {[regexp ^gui $data]} {
            puts "DVE Gui Command : $data"
            ucliCore::tempPause
            if {[catch {eval $data} fid]} { //////
                You
                must capture the error or warning message from the UCLI/DVE
                command; otherwise the socket communication will be lost.

                puts "Command Error : $fid"
            } else {
                puts [eval $data]
            }
            ucliCore::tempPauseResume
        } else {
            puts "UCLI Command : $data"
            ucliCore::tempPause
            if {[catch {eval $data} fid]} {
                puts "Command Error : $fid"
            } else {
                puts [eval $data]
            }
            ucliCore::tempPauseResume
        }
    }
} else {
    puts "channel $chan closed"
    close $chan
}
}

# This callback is called when a connection is initiated
from DVE
proc Accept {channel clientaddr clientport} {
    puts "Connection from $clientaddr registered"
    fileevent $channel readable [list GetData $channel]
    puts $channel "welcome\n"
    flush $channel
}

```

```

# Start a server, so that you can communicate with DVE

set listenSocket 0
# seed the random number generator
expr srand([clock clicks])
while {$listenSocket == 0} {
    # pick a random port
    set port [expr int((rand()*(pow(2,16) - 1024))+1024)]
    puts "trying port $port"
    catch {set listenSocket [socket -server Accept $port] }
}

set host [info hostname]
puts "now listening to port $port host $host"

```

Consider the following test case `test.v`:

```

module test;
    reg clk=0;
    reg [2:0] cmd;
    reg [31:0] addr;

    initial
    begin
        forever
            begin
                #5 clk=~clk;
            end
    end

    initial
    begin
        #20;
        addr=31'h006A0018;
        cmd=1;
        #9;
        addr=0;
        cmd=0;
        $finish;
    end

```

```
endmodule
```

To use the above code to perform socket-based Tcl communication between DVE (server) and terminal (client):

1. Compile the above example code

```
% vcs -nc -debug_all -sverilog test.v
```

2. Open the DVE GUI

```
% ./simv -gui &
```

3. In the DVE GUI, perform the following command:

```
Dve%source fileevent.tcl
```

The following sample message is generated in the DVE console window:

```
trying port 40343
```

```
now listening to port 40343 host vgamddual99
```

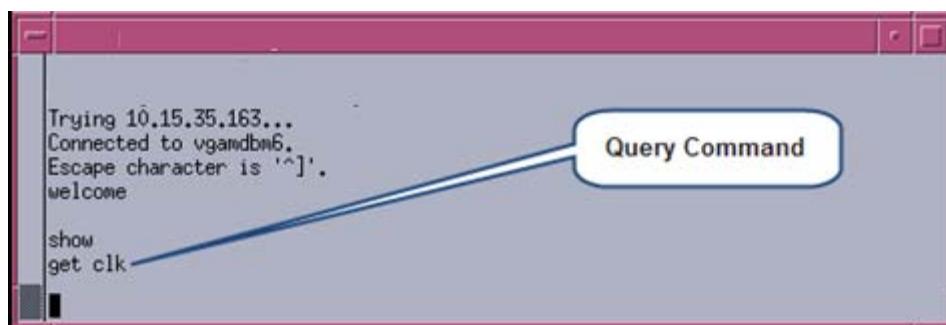
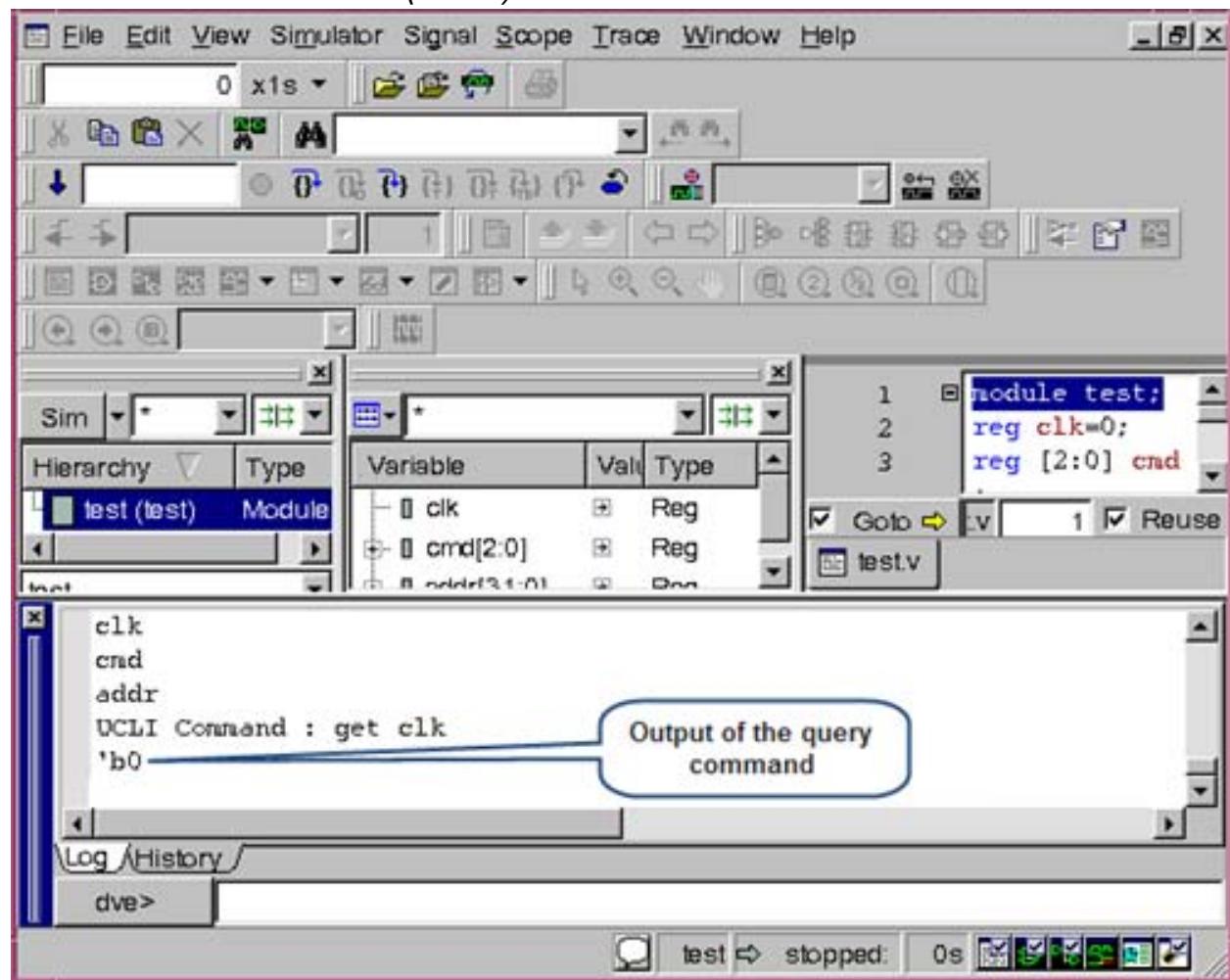
Where, `vgamddual99` is the machine name (`machine_name`) and `40343` is the server port number (`<port_number>`).

4. Open a new terminal and login to another machine (for example, `new_machine`).
5. Perform the following command:

```
% telnet machine_name <port_number>
```

You can now execute the UCLI/DVE commands using the above example. For instance, if you type the `get` command in a new terminal, you can view the output of this command in DVE. [Figure 9-2](#) illustrates this example.

Figure 9-2 Socket-based Tcl Communication between DVE (server) and Terminal (client)



Limitations

The following are the limitations of the Tcl fileevent support in UCLI and DVE:

- Debugging of Specman, SystemC, and Cbug objects using fileevent socket channel is not supported.
- This feature is only supported with the `-ucli2Proc` flag.
- You must capture the error or warning message of the UCLI/DVE command. If you simulate without capturing the error or warning message of the UCLI/DVE command, then the socket communication is lost.
- This feature is not supported in DVE post-processing mode.
- The following UCLI commands are not supported:
`run`, `restart`, `finish`, `pause`, and `resume`

Viewing the Full Design Hierarchy in a Partially Dumped VPD

You can view the complete design hierarchy in DVE post-process mode even if you have dumped partial hierarchy in the VPD file.

Example

tokens.v

```
module d_ff(d, clk, reset, q);  
  
    input    clk;  
    input    reset;  
    input    d;  
    output   q;
```

```

    reg      q;

    always @(posedge reset or negedge clk) if (reset) begin
        q = 1'b0;
    end
    else begin
        q = d;
    end
endmodule

module t_ff(clk, reset, q);
    output   q;
    input    clk;
    input    reset;

    wire     d;

    d_ff df1(d, clk, reset, q);

    not (d, q);
endmodule

module ripple_counter(clk, reset, q);
    output [3:0]   q;
    input          clk;
    input          reset;

    t_ff tff0(clk, reset, q[0]);
    t_ff tff1(q[0], reset, q[1]);
    t_ff tff2(q[1], reset, q[2]);
    t_ff tff3(q[2], reset, q[3]);
endmodule

module stimulus;
    reg      clk;
    reg      reset;
    wire [3:0]   q;

```

```

ripple_counter c1(clk, reset, q);

initial
    $monitor($time, "count q = %b reset= %b clock = %b
count %d", q,
            reset, clk, q);
initial begin
    clk = 1'b1;
    forever #(10) clk = (~clk);
end
initial begin
    reset = 1'b1;
    #(10) reset = 1'b0;
    #(350) reset = 1'b1;
    #(50) reset = 1'b0;
    #(10) reset = 1'b1;
    #(100) reset = 1'b0;
end
initial begin
    #(1000) $finish;
end
endmodule

```

Steps to compile the example

```

% vcs -sverilog -debug_all tokens.v

% simv -ucli -i partial.ucli

% dve -script partial.tcl

```

The files partial.tcl and partial.ucli are available in your \$VCS_HOME directory.

To view the complete hierarchy in a partially dumped VPD

1. Compile the example and open DVE.
2. Select **File > Open Database**.
3. Select the VPD file and click **Open**.

A partial design hierarchy appears.

The screenshot shows the ModelSim interface with three main panes:

- Left Pane (Hierarchy):** Shows a tree structure with "stimulus..." as the current module.
- Middle Pane (Variables):** A table showing internal variables:

Variable	Type
clk	Reg
reset	Reg
q[3:0]	Wire
- Right Pane (Code Editor):** The Verilog code for the module:

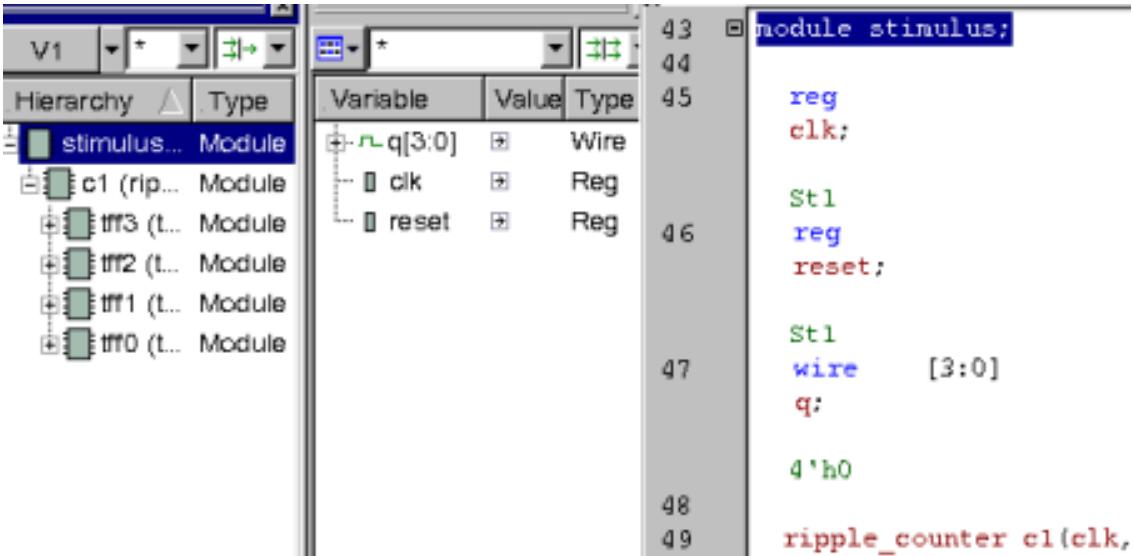
```
43 module stimulus;
44
45     reg clk;
46     reg reset;
47     wire [3:0] q;
48
49     ripple_counter cl(clk, reset);
50
51     initial
52         $monitor($time, "count q = %b clock = %b count %d",
53             set, clock, count);
54
55 endmodule
```

- #### 4. Select **Edit > Preferences**.

The Application Preferences dialog box appears.

5. In the Design Debug category, select the **Use design debug library for design hierarchy for post process** check box and click **OK**.
 6. Select **File > Reload databases**.

The full hierarchy is now visible in the Hierarchy pane.



The screenshot shows a software interface for digital design simulation. On the left, the 'Hierarchy' pane displays a tree structure of modules: 'stimulus ... Module' at the top, followed by 'c1 (ripple_counter)', 'tff3 (tff3)', 'tff2 (tff2)', 'tff1 (tff1)', and 'tff0 (tff0)'. To the right of the hierarchy is the 'Source' pane, which contains the Verilog code for the 'stimulus' module. The code defines a wire 'q[3:0]', a reg 'clk', and a reg 'reset'. It also includes a 'Stl' block for a ripple counter, defining a wire 'q' of type [3:0] initialized to 4'h0, and instantiating a 'ripple_counter' module with inputs 'clk' and 'reset'.

```
module stimulus;
    reg q[3:0];
    reg clk;
    reg reset;
    Stl
    wire [3:0] q;
    4'h0
    ripple_counter cl(clk,
```

Limitations

- Separate compile flow is not supported.
- OVA is not supported.
- SystemC-top design is not supported.

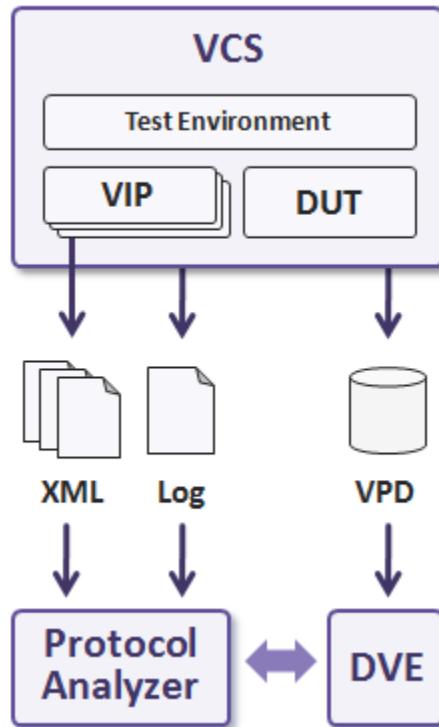
Integrating DVE with Protocol Analyzer

VCS enables you to invoke Protocol Analyzer from DVE, and connect it automatically to DVE. It also allows you to directly load VIP configured session files from Protocol Analyzer to DVE.

This integration allows you to use the existing DVE VIP session files from Protocol Analyzer, and provides VIP based debug configuration setups (session files) which are useful in DVE.

The Protocol Analyzer is a protocol-oriented analysis environment which can be used with DVE to debug designs using Verification IPs (VIPs). [Figure 9-3](#) describes the link between Protocol Analyzer and DVE.

Figure 9-3 Integrating Protocol Analyzer with DVE



The following points describe in detail the link between Protocol Analyzer and DVE:

- Protocol Analyzer can be started and connected to DVE. This link helps both tools to be synchronized (simulation time, and so on) during the debug session.
- Protocol Analyzer provides a set of .tcl files for Synopsys VIPs that create a specific setup for DVE. This setup creates groups of signals and visualization radix to visualize signals which are related together in the DVE waveform viewer.

Use Model

Invoking Protocol Analyzer from DVE

You can use one of the following methods to invoke Protocol Analyzer from DVE:

Using DESIGNWARE_HOME Environment Variable

Set the DESIGNWARE_HOME environment variable to point to VIP directory. When started from DVE, Protocol Analyzer is automatically connected to DVE.

When this environment variable is set, DVE automatically creates a VIP menu. This menu contains the entry to start Protocol Analyzer. In addition, DVE scans the DESIGNWARE_HOME directory and adds a menu entry for each session file corresponding to a VIP which is instantiated in one of the currently opened designs.

The session file menu entries follow the below hierarchy:

<VIP Version> -> <VIP Module> -> session file

There can be more than one session file for a given VIP.

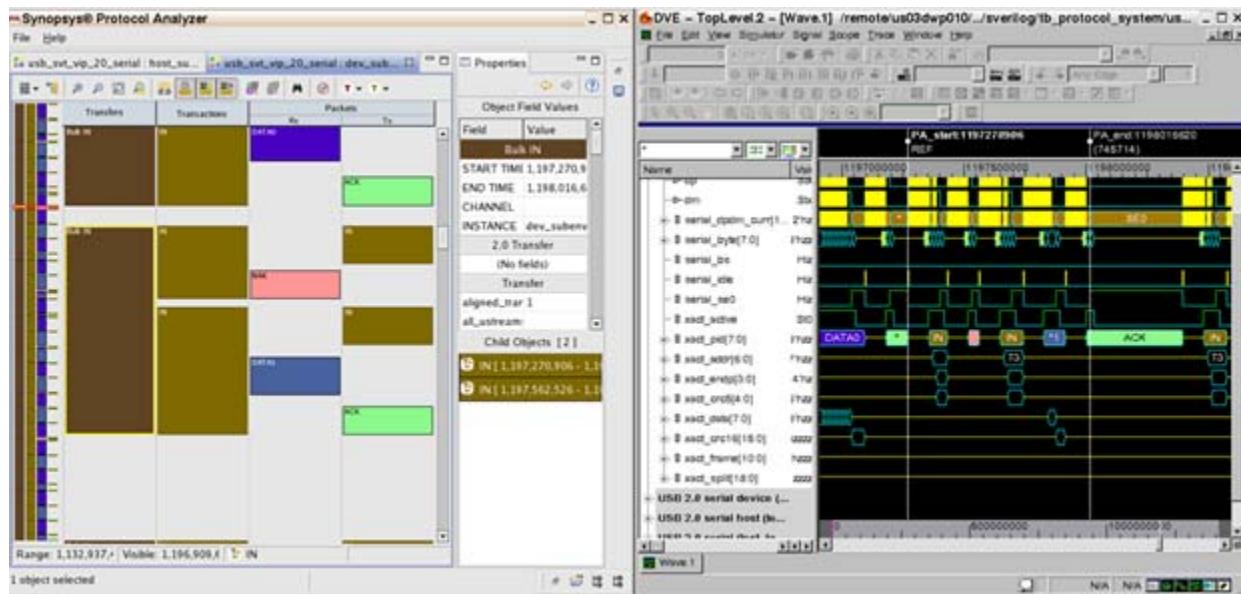
Using -dve Option

Specify the following option in Protocol Analyzer:

```
pa -dve [host:]port
```

This option allows DVE to start Protocol Analyzer.

Figure 9-4 Invoking Protocol Analyzer from DVE



10

Echo

This document describes all Echo-specific options.

This chapter also includes coding guidelines that allow you to get the most out of the Echo technology. Some common user flows are also provided.

Echo is a technology that automatically analyzes the constraint space and generates stimuli that significantly reduce the manual effort required to verify a large number of functional scenarios. In addition, it allows for the auto-generation of a stimulus coverage model, and the use of bias files to guide the convergence process of either the auto-generated or user-developed stimulus coverage.

Diagnostic Mode

To enable diagnostic mode at compile-time, you need to provide the `-ntb_opts echo_diagnostics` switch with `-ntb_opts echo`.

In the compile-time diagnostic messages, targetable cover points and crosses, number of targetable bins and untargetable objects are displayed with reasons. At the end of compilation, Echo status summary is provided. The number of targetable objects may not be exact, as the result is based on the compile-time analysis. The actual status and the number is referred in runtime diagnostic messages.

To enable diagnostic mode at runtime, you need to provide the `-echo_enable_diagnostics` switch with `-echo_enable`.

In the runtime diagnostic messages, holes that are targeted per each randomization cycle and the result are displayed. At the end, it provides total number of randomization calls, if holes remain untargeted after the simulation.

Echo Targeted Status in the URG Report

Every coverage bin (cover point and cross) that is targeted by Echo now has an additional annotation called **Echo Targeted**. This annotation is displayed for both covered and uncovered objects and is shown in a separate column. For covered bins, it means that the bins are targeted by Echo and got hit (and covered). For uncovered bins, it means that Echo did target the bin but the bin was not hit. In URG Report group information page, **Echo Targeted** annotation is visible in the **Status** column.

Table 0-1. Covered Bins

Name	Count At Least	
auto [0] 3	1	
auto [1] 1	1	Echo Targeted
auto [2] 3	1	
auto [5] 3	1	Echo Targeted
auto [6] 1	1	Echo Targeted
auto [8] 1	1	
auto [11] 2	1	Echo Targeted
auto [13] 3	1	Echo Targeted
auto [14] 2	1	Echo Targeted
auto [15] 1	1	

Runtime Configurable Switches

Following are the runtime configurable switches:

Switches to Split Holes with Multiple Bins

Echo targets one hole per one randomize call. If more than one bins are in a hole, Echo splits the hole into many holes, so each hole can contain only one bin. In this way, one randomize call can target a single bin.

You may disable this split hole behavior with the switch
`-echo_enable=no_split_hole`.

Following switches allows to split holes with multiple bins.

- `-echo_enable=no_split_hole` — Specifies not to split any holes.
 - `-echo_enable=split_sample_hole` — Split holes with multiple sample bins. Not splitting holes with cross bins.
 - `-echo_enable=split_cross_hole` — Split holes with multiple cross bins. Not splitting holes with sample bins.
-

Not Annotating Echo Target Mark in covdb

Echo targeted flags are automatically annotated at targeted bin objects in the coverage database when `-echo_enable` is on. They are eventually displayed in the URG Report.

If you do not want this annotation in the coverage database for some reason, you can use the switch `-echo_disable_in_db`.

Target Priority Change

Echo targets sample bins prior to cross bins, by default. You can change the priority by using the

`-echo_enable=priority_cross` switch, so that the cross bins are targeted first.

It makes sense to use this switch in many cases, because it is guaranteed that all samples are covered if all the cross are covered, but not vice versa.

Echo all Switches Reference

The following tables describes all the compile-time and runtime switches available in Echo:

Table 10-1 Compile-time switches

Switch Name	Description
-ntb_opts echo	enable Echo mode
-ntb_opts echo_diagnostics	enable diagnostic mode
-ntb_opts echo_cov_gen	enable coverage model auto-generation mode
-ntb_opts echo_cov_gen_dir <path>	set path for coverage model files

Table 10-2 Runtime Switches

Switch Name	Description
-echo_enable	enable Echo mode
-echo_enable=on_start	target coverage holes from the very first call to randomize
-echo_bias_file=<file_name>	Specifies the bias file
-echo_enable=no_split_hole	not splitting holes when collecting targetable holes
-echo_enable=split_sample_hole	splitting sample holes only
-echo_enable=split_cross_hole	splitting cross holes only
-echo_disable_in_db	hide "Echo Targeted" flag in the URG Report
-echo_enable=priority_cross	targeting cross holes in advance of the sample holes

Following are the URG switches for Echo Bias File generation:

Table 10-3 URG Switches

Switch Name	Description
-echo gen_bias <N>	generate <N> number of bias files
-echo gen_bias_dir <path>	generate bias files in the given path
-echo max_hole_size <N>	overriding maximum number of bins in a hole

Compile-Time Options

This section describes the options you use to enable the Echo technology when using the VCS compiler. Note that using these options does not guarantee that Echo will occur. For Echo to occur, the constraints and coverage model need to meet certain requirements.

You also need to specify the options described in this chapter (as appropriate) when you execute simulation (that is, when you run simv).

Echo Option

`-ntb_opts echo`

Ensures that the required Echo-related processing is done for all relevant coverage points.

Example for OpenVera testbench:

```
vcs -ntb -ntb_opts echo foo.vr
```

Example for SystemVerilog testbench:

```
vcs -sverilog -ntb_opts echo foo.sv
```

Coverage Model Autogeneration Options

`-ntb_opts echo_cov_gen`

Generates a coverage model for a stimulus object—that is, for a class that contains random variables and/or constraints.

See “[Automatic Generation of a Coverage Model from Constraints](#)” on page 243 for details on how a coverage model is inferred from constraints.

When you specify this option, Echo writes automatically inferred cover groups (if any) to the directory `$(SIMV_DIR)/echo_cov_gen`. You can specify a different directory with the `-ntb_opts echo_cov_gen_dir` option.

Note that if the `echo_cov_gen` directory already exists, any files in that directory will be overwritten when you rerun Echo with the `-ntb_opts echo_cov_gen` option.

The names of the files created (and the names of the coverage groups themselves) depend on the names of the classes and constraints blocks in the source file.

Example for OpenVera testbench:

```
vcs -ntb -ntb_opts echo_cov_gen foo.vr
```

Example for SystemVerilog testbench:

```
vcs -sverilog -ntb_opts echo_cov_gen foo.sv  
-ntb_opts echo_cov_gen_dir <directory_name>  
Overrides the default directory (./echo_cov_gen) into which  
Echo writes automatically inferred covergroups.
```

Example:

```
vcs -ntb -ntb_opts echo_cov_gen -ntb_opts  
echo_cov_gen_dir ./my_dir foo.vr
```

Echo Runtime Options

To enable the Echo functionality, specify the following command-line options when executing the `simv` generated by VCS.

`-echo_enable`

Enables Echo at runtime.

The `-echo_enable` option is required for running Echo. The following arguments are optional.

`-echo_enable=on_start`

Tells Echo to target coverage holes from the very first call to `randomize`. By default, the targeting of holes does not start until Echo determines that merely running pure random stimulus (based on constraints) is no longer resulting in increasing coverage.

You can use this option when few randomize calls occurring in the test so each randomize call can be coverage-aware. This option is also enabled automatically when you use bias file options.

`-echo_bias_file=<file_name>`

Specifies the bias file to be used for the test run. The bias file specifies the coverage holes that the test should target in the current run. Different runs of the test can use different bias files as input. Although it is possible to create a bias file manually, we recommend that you generate the bias file using the URG utility. See “[URG Options for Bias File Generation](#)” on page 239 for more details.

For the bias file to have any effect, you must enable other options (such as `-echo_enable`). When you specify the `-echo_bias_file` option, the `-echo_enable=on_start` option is automatically enabled.

URG Options for Bias File Generation

This section describes the URG options related to Echo. These options generate bias files. You use the bias files to bias Echo to target specific coverage holes. The URG options described in this section are in addition to other required URG options, such as `-dir` and `-format`.

`-echo gen_bias <non-zero integer>`

Instructs URG to partition all the coverage holes in the input database into the specified number of bias files.

Echo writes the bias files to the `./echobiasConfigs` directory. The bias files are text files that represent coverage holes. Avoid editing the bias files by hand. However, if you choose to edit the bias files, do not deviate from the bias file format. You can use each bias file generated with the `echo_gen_bias` option as an input to a test run.

See “[Understanding Bias Files and Echo](#)” on page 252 for more details.

```
-echo gen_bias_dir ./<my_dir>
```

Overrides the default directory setting that determines where the bias files are written. By default, the bias files are written into the `./echobiasConfigs` directory. Echo creates the directory if the directory does not exist.

Coding Guidelines

The following sections provide coding guidelines for using Echo.

Constraints and Coverage Model on the Same Variables

Echo works when the coverage model is on the stimulus variables. In other words, the cover points (and cover point expressions) should be the `rands` variables that represent the stimulus. Constraints and the coverage model should be in the same class. Further, the randomize and coverage sampling should happen on the same instance of the generator class.

No Procedural Overwriting of Values Generated by the Solver

Echo works by generating constraints for all the coverage holes and by enabling these “coverage constraints” during consecutive calls to `randomize`. If there is procedural code (in `post_randomize` for example) which overwrites the values generated by the solver, then it is possible that coverage bins might not be hit, even though the solver is generating the right values for the random variables.

Coverage Should Be Sampled Between Consecutive Calls to `randomize`

The sampling event for the coverage model should be triggered between consecutive calls to `randomize`, to ensure that all generated values are sampled. An alternative is to use the `@randomize sample` event. Using this sample event ensures that the cover points are sampled as soon as randomization is completed.

Use Open Constraints

In an ideal scenario, you should specify only the legal (or environment) constraints when running Echo. Do not use test constraints. Test constraints further constrain the legal environment to focus the solver on a specific part of the legal space. With Echo enabled, the solver is automatically focussed on the portion of the legal space that is covered by the coverage model.

If you want to focus the solver in Echo mode, then load a preexisting coverage database before starting the stimulus generation. Echo will not target anything that is covered in the loaded database.

Avoid Explicit or Implicit Partitioning in Constraints

Echo might not work efficiently when there are explicit or implicit partitions in the constraint problems. Explicit partitioning is enabled by using “solve before” constructs. Implicit partitioning is enabled when there are function calls in constraints or when object allocation is enabled.

If partitioning does happen and Echo is enabled, then partitioning might lead to some loss of efficiency for Echo: some coverage bins might be targeted but not covered.

Avoid In-line Constraints and the Use of “constraint_mode” and “rand_mode”

Echo is designed to work on the static structure of the constraints and coverage. If the structure of the constraints is changed using in-line constraints or turning constraints or randomness off and on using “constraint_mode” and “rand_mode”, then the performance of Echo might be compromised.

Automatic Generation of a Coverage Model from Constraints

The goal of Echo is to generate stimuli that efficiently cover the testbench constraint space. In order to achieve this, Echo extracts a functional coverage model from the constraint expressions, and automatically covers it. The intuition is that a better sampling of the stimulus space will exercise the design behaviors more exhaustively, and therefore verify a large number of functional scenarios more efficiently.

The plan is to provide a first-class “contract” to the user which specifies how the coverage model is inferred and how goals are named.

Coverage Groups

A cover group will be generated for each class by default.

A cover group will be generated for each class to contain the coverage model derived from variable declarations in the class. The cover group for the class that will contain the coverage model for variables will be called `covg_<class_name>`.

The members of the cover group (cover points, crosses) will be annotated with comment attributes in a manner that will allow tracking back to the original constraint expressions. The following sections detail the types of cover points and crosses that will be inferred.

The cover groups will contain items as described below.

Cover Points

Variable Coverage

Each `rand` variable will have a cover point associated with it, except for variables that have an unguarded set membership constraint at the top-level, in any of the constraint blocks of the class. For example:

```
enum PktType = {Type1, Type2};  
rand bit[7:0] x;  
rand PktType p;  
PktType prevp;  
constraint c1 {  
    x in {0:1, 5};  
}
```

Here, a cover point will be inferred for `p`, but not for `x` or `prevp`. Note that a set membership cover point will be inferred for `x`.

The cover point expression will be just the variable. The following rules apply to the creation of bins:

- Autobinning for `enum` type variables.
- Equal volume autobinning, with a maximum of 64 bins, for variables with precision 8 bits or less.

For example:

```
rand bit[7:0] x;  
rand PktType p;  
rand bit[31:0] y;
```

In the above example, autobinned cover points will be created for `x` and `p`, with the bins for `x` having the ranges 0–3, 4–7, …, 252–255, and the bins for `p` having the values `Type1` and `Type2`.

The built-in function `rand_mode` can be used to turn off the randomness of a `rand` variable. But the collection of coverage for such variables will continue irrespective of the mode change.

The cover points for variables will be called `covp_<cover_point_id>`. `<cover_point_id>` is a unique integer identifier for every cover point. The autobins for the cover points will be named by the usual naming convention for autobins. The comment for the cover point will indicate details about the variable.

Branch Coverage

Each conditional block will have an associated Boolean cover point. The cover point expression will be the conjunction of all the enclosing conditional guards. For example:

```
if (p) {  
if (q) {  
    x == y;  
}  
}
```

In the above example, there will be a coverpoint with the expression `p&&q`.

The comment for the cover point will have the expression string.

Condition Coverage (Sensitized)

Each sensitizing guard condition will have a Boolean cover bin associated with it. For example:

```
if (p || (q && r)) {  
...  
}
```

In the above example, there will be a cover point with expressions p || ($q \&& r$) with one bin corresponding to the case when expression p is ON (or true) and one bin corresponding to the case when expression $q \&& r$ is true. The idea is to make sure that the conditional expression becomes true at least once for every implicant in the expression being true.

The cover points will be guarded by the conjunction of all the enclosing conditional guards.

Note:

The subexpressions of the guard expression involving operators other than `&&`, `||`, `!` are considered as atomic, and the sensitizing conditions are defined over these atomic subexpressions.

When the condition expression is a set expression, the condition coverage model will be generated like the set membership coverage model. For example:

```
if (p in {0:4, 7}) {  
    ...  
}
```

Here the cover point expression will be p , and the bins 0, 1–3, 4, 7 will be created.

The comment for the cover point will have the expression string.

Set Membership Coverage

Echo will infer a cover point for constraints coded using the `in` and `dist` operations. Equivalent ways of writing the same constraints using other operators might not result in an inferred coverage model. For example:

```
x in {0:5};
```

results in an inferred cover point, whereas

```
x >= 0 && x <= 5;
```

does not result in an inferred cover point. Also, use of set membership in complex constraints does not result in an inferred cover point.

For membership in sets with constant members:

- A single cover point will be created. The sample expression will be the same as the LHS
- value in the set membership constraint.
- A bin will be created for the low value of each range.
- A bin will be created for the high value of each range, for non-singleton ranges.
- A bin will be created for the range excluding the low and high values of each range, if the resulting range is non-empty.

For example:

```
x in {0:5, 7};
```

In the above example, a cover point with expression `x` and four bins, 0, 1–4, 5, 7 will be created.

The functional coverage language does not allow dynamic variables (those that are not parameters to the coverage groups) to describe bin ranges. Hence, for membership in sets with nonconstant members:

- A cover point with three Boolean cover bins will be created for each of the ranges as follows:
 - A bin with the expression (range_low_expression).
 - A bin with the expression (range_low_expression + 1 : range_high_expression - 1).
 - A bin with the expression (range_high_expression).

For example:

```
x in {a:5, b};
```

In the above example, a cover point will be created for variable x with the following bins:

- One bin with value a;
- One bin with value range (a+1:4);
- One bin with value 5;
- One bin with value b;

Note:

If the set membership range expressions use random variables, then no coverage bin will be inferred for that range. For example, if `a` in the above example were specified as `rand`, then the first two bins would not be inferred.

Note:

All the cover points above will be guarded by the conjunction of all the enclosing conditional guards.

Crosses

Combinations of control variables in the class, will be used to generate crosses.

We will define control variables as variables that are of enumerated types or bit-vectors that are 8 bits wide or less, for which a variable coverage cover point is generated.

Cross on Variables Within a Class

If a class contains control variables `x`, `y`, then a cross `x, y` will be generated. The cross will be autobinned. The cover points and bins for `x`, `y` will be as described in the section on variable coverage.

The set of variables that participate in one cross is determined as follows:

Control variables will be added to a cross until the total number of expected bins (product of underlying cover point bins) of the cross does not exceed 64000.

If more control variables still remain after previous step, then a new cross will be created.

The crosses will be called `covc_<cross_id>`. `<cross_id>` is a unique integer identifier for the cross. The comment for the cross will indicate details about the crossed variables (class name, file, line, etc.).

Cross of Condition and Constraint Expressions

The sections on condition coverage and set membership described two kinds of cover points. In the case of set membership constraints inside conditionals, we will generate a cross of the cover points for coverage of the condition (sensitizing). This cross will be autobinned. For example:

```
if (p || (q && r)) {  
    x in {0:5, 7};  
}
```

In the above example, cover points for `x` will be generated from the set membership constraint. Cover points for the condition expression will be generated. A cross for the two cover points will be generated.

Sampling Event

A special built-in sample event called `@(randomize)` will be used to determine the sampling for the generated coverage groups.

This event can be used in user defined cover groups that are embedded in classes. The event is triggered when an instance of this class is randomized, either directly or through containment in an instance of another class being randomized. When the event is triggered, coverage is tracked for the instance and/or the cover group definition (that is, cumulative).

The @ (randomize) event is triggered on all `rand` objects in the context being randomized, after the solver is done, and values have been populated in the objects. If objects are being allocated by `randomize`, then the allocation of the objects is done before the values are populated back, hence before @ (randomize) is triggered.

Contribution to Coverage Scoring

The automatically generated coverage model will contribute to the coverage score by default with a weight of 1.0.

Coverage Model Inference for In-line Constraints

Since in-line constraints (`randomize with { .. }`) is recommended for use in the specification of test constraints, we will not infer a coverage model from such constraints. In fact, we expect the use of test constraints to be minimized after coverage driven stimulus generation is implemented.

Use Model

Autogeneration of coverage model is done as part of the VCS compile step. The switch to enable coverage model generation is `-ntb_opts echo_cov_gen`. All the coverage models will be written out as text files in the `echo_cov_gen` directory. The `echo_cov_gen` directory will be present wherever the `simv.daidir` and `simv` files are generated by VCS compiler. Both System Verilog and OpenVera testbench formats are supported. The language in which the coverage model will be generated is the same as the syntax of the testbench.

Example command line for an OpenVera testbench:

```
vcs -ntb -ntb_opts echo_cov_gen foo.vr
```

Example command line for a SystemVerilog testbench.

```
vcs -sverilog -ntb_opts echo_cov_gen foo.sv
```

It is expected that the user will first create a coverage model using the `auto_gen` option. Then the user will include the coverage model into their testbench (using ``include` or `#include` directives) and then make sure the coverage model is instantiated in the new task for the enclosing class (that is, the class where the constraints and random variables are specified). The user is encouraged to view the autogenerated coverage model and make changes if required.

The VCS compiler does not compile the design when `echo_cov_gen` is specified (that is, no `simv` is created).

Understanding Bias Files and Echo

Motivation

Echo is a technology that automatically analyzes the testbench constraint space with the goal of generating the most efficient stimuli to explore a wide range of functional scenarios. In order to achieve this, Echo automatically generates stimulus functional coverage, that is, covergroups for the various random variables in the testbench. Echo then seeks to automatically target these covergroups by directing the constraint solver to generate stimulus such that the coverpoints in these covergroups will be hit.

In addition, if you have developed your own stimulus cover model, then Echo can identify the holes in the stimulus cover model and similarly direct the constraint solver to generate stimulus such that these cover holes will be hit. Echo works at a test level, that is, Echo tries to hit coverage holes within a test.

Verification tests are mostly run on a regression farm, where multiple copies of the tests are being executed in parallel on different machines in the farm, with each simulation being assigned a different random seed. It is desirable that with Echo enabled, each test be able to target different portions of the coverage space in order to meet the coverage goals in the shortest possible clock time.

What Is a “Test”?

A test for the purpose of this document is an executable that can be simulated (for example, `simv` generated by VCS). The source files (testbench files, design files etc.) are compiled by VCS compiler and linked with the simulation engine to create the executable. The executable can then be invoked multiple times with different runtime options, such as different random seeds or configuration files and the simulation results can be observed. Note that the executable is created once, that is, the source files (constraint, coverage models, transactors, modules etc.) do not change with every execution. The different options passed with every execution lead to different behavior of the simulation.

From the Echo point of view, a “test” should consist of a testbench with an “open” set of constraints: the constraints should represent the entire legal stimulus space. Note that in many existing methodologies, the constraints in a “test” consist of the legal constraints as well as a set of test constraints that limit the solution

space of the legal stimulus. This is typically done to focus the test towards some verification targets such as coverage or specific features.

The bias file approach described below works best with tests using an open set of constraints and a full coverage model. Thus any invocation of the simulation executable can target any of the existing valid coverage holes.

Using Echo Bias File to Target Coverage Holes

The user can influence the coverage holes being targeted by Echo in a test run is by using a Echo bias file. This is a file that contains a list of coverage holes which the Echo engine will target when the test starts running. The Echo bias file format is a text file representing coverage holes. This file can be modified by the user if needed.

Runtime Option to Specify a Bias File

The Echo bias file will be passed in as a runtime argument for the test run, using

`-echo_bias_file=<path_to_echo_bias_file>`. When the test starts executing, the Echo engine will read the bias file and populate its in-memory coverage hole database with the data from the Echo bias file. Then it will systematically start targeting the holes. After all the holes are targeted, if more randomize calls occur, then they will not be reactive, that is, no coverage hole will be targeted.

- The holes in the Echo bias file are all at a coverage group definition level. They will apply to all instances of the coverage group in the test, for which reactivity can be applied and can be enabled.
- Only the coverage holes present in the bias file will be targeted.

- The bias file will be read and processed by the Echo runtime engine. The test does not need to be recompiled if the bias file is changed or a new bias file is to be used for the test.
- The bias file has no impact on the coverage engine and on coverage reporting. It will only influence Echo. If the data in the bias file is for a coverage group which cannot be targeted by Echo, then all the data for that coverage group is ignored by the Echo engine.
- If the Echo bias file as pointed to by the `-echo_bias_file` option cannot be read, then an error will be issued the file will be ignored. The test will run as though no bias file has been specified.
- There are no restrictions on the name of the Echo bias file.
- The bias file will provide a guideline as to what coverage bins are targeted by Echo, but Echo can choose to override these guidelines in certain situations. For example, Echo will not target coverage bins that have been assigned a zero weight by the user or coverage bins belonging to disabled by the user using the `collect` attribute or by using `coverage_control` system task.

Automatic Generation of Echo Bias Files

URG automatically generates Echo bias files for a particular test. URG can be invoked on a coverage database file and will generate N bias files, where N is supplied by the user. The utility can be invoked as follows:

```
$VCS_HOME/bin/urg -dir <coverage_db_dir> -echo gen_bias
<number_of_bias_files_required>
```

The utility will enumerate all the coverage bins in the input database files and will create N random partitions of the holes where N is supplied by the users using the `num_bias_files` argument. It is an error if N is greater than the number of coverage bins in the coverage database. The bias files will be written out in the `$(PWD)/echoBiasConfigs` directory as `config0, config1, config (N-1)`. Other URG options (such as `-format`) can also be used simultaneously to process the coverage data. Since the Echo bias file represents coverage bins at a coverage group definition level, it will be assumed that there is only one shape for every coverage definition in the input coverage database. If multiple coverage definitions are detected, then an error will be issued and no bias files will be generated.

The bias files generated by URG can then be used as input to tests being run in parallel.

Repeatability of Test Results for Parallel Regression Runs

An essential requirement for the approach outlined above is that it should be possible to reproduce the results of a test run in a parallel regression environment in a stand alone manner. More specifically, if a test fails in the parallel regression run, it should fail in exactly the same manner when run on its own. Echo ensures that given the same inputs and the same command line arguments, the sequence of values generated by the solver will be exactly the same. Here the inputs are the bias file and the test source code, and the arguments are the random seed and the `-echo_bias_file` argument. *Note that this means that the regression system needs to preserve the bias file used by a test in addition to other test source code and scripts.*

Usage Scenarios

This section discusses Echo usage in various common verification scenarios. In all the scenarios in this section, it is assumed that the guidelines mentioned in the previous section are being followed (for example, coverage and constraints are on the same variables, and so on).

You can load a preexisting coverage database in every scenario to screen out already-covered coverage bins. By doing so, the current run can focus only on uncovered bins. The coverage database must be loaded after all the testbench components have been instantiated, so that the coverage data from the database can be loaded in the proper (in-memory) runtime database.

Running a Single Test with Randomized Configurations

This scenario generally applies to multimedia devices, in which the device has many interfaces and many modes and the device can be configured to select some interfaces and some modes.

The object of verification is to exercise all the interesting configurations of the device. In the testbench, configurations are selected, and then data (possibly random) is passed through the device.

Configurations for a device are typically chosen (through `randomize`) a few times per test run. In the extreme case, only one configuration might be selected (for example, `config.randomize` is performed only once).

In this scenario it is important to preload the coverage database to ensure that already-selected configurations are not chosen.

Running a Single Test with Randomized Transactions

This scenario applies to transaction-based verification environments. This scenario is required for packet-processing devices or instruction-based processor verification.

The device is set in some mode, and then multiple transactions are generated and passed through the device. You use constraints to generate the transaction objects. In this scenario, you can use the default arguments for Echo.

If you assume that many transactions will be generated (randomized), then it is appropriate for the convergence heuristics to take over and apply reactive calls as needed.

```
status = transObj.randomize();
```

or

```
repeat (1000) {
    transObj = new;
    status = transObj.randomize();
}
```

Using a Bias File for a Parallel Regression

In this scenario, you use the bias file generation utility to bias inputs for different parallel test runs.

1. Run the first batch of tests using the following commands:

```
$SIM -echo_enable +ntb_random_seed = ${SEED}
```

Assume that each batch contains 100 tests.

2. Wait for first batch run to complete.
3. Merge the coverage results for all 100 tests in the first batch and store the results in merged.vdb.
4. Perform bias file generation for the next batch of tests using the following command:

```
$VCS_HOME/bin/urg -dir merged.vdb -echo gen_bias 100
```

This command distributes the remaining coverage holes into 100 bias files.

5. Run the second batch of 100 tests with each test using a different bias file as an input:

```
$SIM -echo_enable -echo_bias_file=<bias_filename>
```

6. Merge results from the second batch of runs.
7. If coverage has not reached your coverage goal, then repeat steps 3 through 5.

Autogenerating a Coverage Model

Echo can generate a coverage model from the constraints specification. However, you must instantiate the coverage model that you intend to use in the simulation.

1. Autogenerate the coverage model:

```
vcs -ntb -ntb_opts echo_cov_gen foo.vr
```

Echo generates the model in the file
./echo_cov_gen/MyClass.vr.

2. Examine the generated model to determine if the model meets your expectations.
3. Add the declaration for the generated cover group as part of the class member declarations.

For example, if the name of the covergroup is MyCov and it is of the class MyClass, then you must add to the other member declarations for the class MyClass:

```
coverage_group MyCov;
```

4. The autogenerated coverage group must be instantiated in the new task of the enclosing class. Perform the instantiation with the following statement:

```
MyCov = new;
```

5. Make the cover group instantiation the last statement of the new task. If the new task does not exist, then you must add the task to the cover group instantiation.
6. Because the name of the autogenerated cover group is deterministic, the declaration and instantiation can be done once during testbench development phase.

Methodology and Flow Issues

This section addresses methodology and regression execution issues.

Scenario: All Tests have the same Constraints and Coverage Space (Recommended)

When all the tests being executed in parallel have exactly the same coverage space and the same legal stimulus space (that is, the constraints are exactly the same for all the tests), then the Echo bias file approach can be used to achieve high efficiency test runs. Each test should have its own bias file. The input database for the URG utility can be obtained by running any test and using the output coverage database. The number of test runs should be used as the value for the `echo_gen_bias` argument for URG. Once the bias files are generated, they can be used again and again for repeated regression runs. It is desirable that the number of times the stimulus objects are randomized in a test be at least equal to the average number of coverage bins in the bias files. After all the tests are done running, the coverage database from all the tests can be merged to get the final coverage number. It is possible but not efficient to use both the random seed and the bias file for every test.

Scenario: Tests are Grouped into Categories with each Category having Specific Test Constraints

In this scenario, tests have test specific test constraints which further constrain the legal stimulus space and thus also reduce the hittable coverage bins. (Coverage bins whose value ranges lie outside the space of the test constraints cannot be hit). A bias file can be created for every category of tests. The bias file should be such that all the coverage bins in the file are hittable given the test constraints for that category. Each test within this category can use the same bias file but with a different random seed. The URG bias generation utility can be used to generate a bias file template. (Note that the URG does not look at test constraints when generating the bias files). The test

writer can then modify this template to include the hittable coverage bins. As in the previous scenario, assuming test constraints remain constant across regression runs, then the bias files have to be generated once and can be used for multiple regression runs. Even if all the bins in a bias file are not hittable, all it causes is less efficient utilization of resources. Echo may use some cycles in the test run to try to target unhittable bins, but remaining cycles will still target hittable coverage bins. In this scenario too, the coverage data from all the tests can be collected and merged. Then a sequential run can be used to target any remaining coverage holes. For this final sequential run, the merged coverage database can be used as an input along with the fully open constraints.

Scenario: Coverage Database Being Loaded in the Beginning of a Test Run

If a coverage database is being loaded as part of the configuration for a test, then there is no need to use a Echo bias file. Echo will only target coverage bins that are not covered in the loaded database, so it is already biased in some sense. If the same database is loaded in multiple tests, then a random seed can be used so that each test targets different holes. This is useful in a batch mode type of regression environment. Here all tests have the same constraints and coverage space, but one batch of tests is run at one time. After the batch is complete, the coverage databases are merged and the merged database is used as input for the next batch of tests. *Note that for repeatability purposes, the input database associated with a test needs to be preserved in case the test needs to be run in a stand alone mode.* If a Echo bias file is used, then it will override the bias from the loaded coverage database, that is, holes in the bias file will

be targeted even if they are marked as covered in the loaded coverage database. Hence it does not make sense to use both coverage database load and Echo bias file approaches for a test.

Support of Crosses with Non-random Coverage Points

You can now target a cross with non-random cover points using Echo. The cross must contain at least one random cover point.

While collecting Echo targetable cross holes for non-random cover points, cross bin combinations with bins that are not sampled, are ignored. However, if the non-random cover point is properly sampled during the first few cycles, it is collected as targetable hole at the next collection.

If transition bins are involved in cover points that contribute crosses, the cross bins including transition bins are ignored when collecting the holes.

Usage Model

Following example of three cover points explains how Echo targets crosses with non-random cover points:

```
rand integer i1, i2;
integer i3;
...
cp1 : coverpoint i1 {
    bin s0 = {0};
    bin s1 = {1};
}
cp2 : coverpoint i2 {
    bin s2 = {2};
```

```

        bin s3 = {3};
    }
cp3 : coverpoint i3 {
    bin s4 = {4};
    bin s5 = {5};
}
cc : cross cp1, cp2, cp3;

```

There are 8 possible combinations of cross bins in the cross 'cc', but Echo does not collect all the 8 cross bin combinations for targeting.

If 'cp3' is sampled with bin 's4', when Echo collects holes in cross 'cc', it only collects holes from all possible cross bins with 'cp3.s4'.

cp1	cp2	cp3	
s0	s2	s4	Targetable
s0	s3	s4	Targetable
s1	s2	s4	Targetable
s1	s3	s4	Targetable
s0	s2	s5	Untargetable
s0	s3	s5	Untargetable
s1	s2	s5	Untargetable
s1	s3	s5	Untargetable

If the value of 'cp3' changes in the middle of simulation, for instance in the collected holes, there are still two cross bins (s1-s2-s4 and s1-s3-s4) to be targeted at the next two randomizations. However, if cp3 is sampled with 's5', targeting 's1-s2-s4' and 's1-s3-s4' will actually hit 's1-s2-s5' and 's1-s3-s5', respectively.

cp1	cp2	cp3	
s0	s2	s4	Targeted & hit
s0	s3	s4	Targeted & hit
s1	s2	s4	Targeted, but not hit
s1	s3	s4	Targeted, but not hit
s0	s2	s5	
s0	s3	s5	
s1	s2	s5	hit without being targeted
s1	s3	s5	hit without being targeted

After running out of holes to target, at the next round of hole collection, 's0-s2-s5' and 's0-s3-s5' are collected as hole, because cp3 is sampled with 's5'.

As you see, 's1-s2-s4' and 's1-s3-s4' will never have a chance to be hit, unless 'cp3' is sampled with 's4' back again. Hence, Echo gives a notification message as follows to let you know the non-random bin changes.

Note- [CCT_NRBIN_CHG] Non-random sample bin changed.

The sampled bin of the non random coverpoint 'cp3' in covergroup 'MyClass :: MyCov', which is part of cross, has changed from 's4' to 's5'.

At the very first hit of collecting the target holes during simulation, if the non-random cover point is not sampled, none of the cross bins is collected as the target holes. However if the non-random cover point 'cp3' is properly sampled during the first few cycles, the cross bins are collected as the target hole at the next collection.

Considering simulation with bias files, cross bins in bias file is targeted regardless of any of the above conditions. However, the targeted cross bin may cause some other cross bin to be hit depending on the actual value of non-random sample as described in the previous example.

Procedural Sampling

Echo analyzes procedural context to find targetable cover points. The support for procedural sampling by Echo contains the following two aspects:

- sampling a non-random variable that is assigned with targetable random variable.
 - randomizing a sibling class of sampled variables.
-

Targeting a Non-Random Cover Point Assigned with Random Variables

Echo targets a non-random cover point when the non-random variable is assigned with an expression of random variables.

Case 1 : Non-random variable is assigned with an expression of random variables.

Consider the following example, a non-random variable i3 is procedurally assigned 'i1+i2', where i1 and i2 are both random variables. In such case, Echo is able to target i1 due to the procedural relationship.

```
program test() ;
```

```

class MyClass;
    rand integer i1, i2;
    integer i1;
    event cov_event;

covergroup MyCov @(cov_event);
    cp1 : coverpoint (i1) {
        bins s0 = {0};
        bins s1 = {1};
        bins s2 = {2};
        bins s3 = {3};
    }
endgroup

function automatic new;
begin
    MyCov = new;
end
endfunction

task my_sample();
begin
    i1 = i1+i2;
    ->cov_event;
end
endtask
endclass

initial begin

MyClass obj1 = new;
repeat(4)
begin
    obj1.randomize();
    obj1.my_sample();
    ->obj1.cov_event;
end
end

```

```
endprogram
```

Case 2 : Similar to Case 1, except expression, contains variables inherited via class extension.

```
program test();

class Base;
    rand integer i1, i2;
endclass

class Derived extends Base;
    integer i3;
    event cov_event;

    covergroup MyCov @(cov_event);
        cp1 : coverpoint (i3);
    endgroup

    function automatic new;
    begin
        MyCov = new;
    end
    endfunction

    task my_sample();
    begin
        i3 = i1+i2;
        ->cov_event;
    end
    endtask
endclass

initial begin

    Derived obj1 = new;
    repeat(4)
    begin
        obj1.randomize();
        obj1.my_sample();
        ->obj1.cov_event;
    end
end
```

```
end  
  
endprogram
```

Case 3 : Similar to Case 1, random variables are within scope, but not part of the class inheritance model.

The procedural assignment can be resolved when a VMM style random variable is used. (Random variables in another class object.)

```
program test();  
  
class OtherClass;  
    rand integer ril;  
endclass  
  
class MyClass;  
    OtherClass objo;  
    integer il;  
    event cov_event;  
  
    covergroup MyCov @(cov_event);  
        cp1 : coverpoint (il) {  
            bins s0 = {0};  
            bins s1 = {1};  
            bins s2 = {2};  
            bins s3 = {3};  
        }  
    endgroup  
  
    function automatic new;  
    begin  
        objo = new;  
        MyCov = new;  
    end  
    endfunction  
  
    task my_sample();
```

```

begin
    i1 = objo.ril;
    ->cov_event;
end
endtask
endclass

initial begin

MyClass obj1 = new;
repeat(4)
begin
    obj1.objo.randomize();
    obj1.my_sample();
end
end

endprogram

```

Case 4 : Echo resolves and targets when the ternary expression is assigned to the sampled variable.

```

program test();

class MyClass;
    rand integer i1,i2;
    integer i3;
    bit cond;
    event cov_event;

covergroup MyCov @(cov_event);

cp1 : coverpoint (i3) {
    bins s0 = {0};
    bins s1 = {1};
    bins s2 = {2};
    bins s3 = {3};
}

endgroup

```

```

function automatic new;
begin
    MyCov = new;
end
endfunction

task my_sample();
begin
    i3 = (cond ? i1 : i2);
    ->cov_event;
end
endtask
endclass

initial begin

    MyClass obj1 = new;
    integer cnt;
    cnt = 0;
    repeat (4)
begin
    obj1.cond = (cnt++)%2;
    obj1.randomize();
    obj1.my_sample();
    ->obj1.cov_event;
end
end

endprogram

```

Coverage Model and Randomization in Sibling Classes

Assume that the two sibling classes (class D1, D2) are derived from the same base class (class B) that contains some random variables. The class D1 has a covergroup that samples the random variables defined in the base class. In the simulation, the randomization takes place in the class D2 which has no coverage model (see the following example).

At runtime, Echo collects targetable holes from the sibling classes to target. Echo currently supports only one level of inheritance. Note that all the variables involved in the expression of coverpoint must be inherited from the base class to be targeted.

```
class B;
    rand integer i1;
    event cov_event;
endclass

class D1 extends B;
    covergroup MyGrp@(cov_event);
        cp1 : coverpoint (i1);
    endgroup
endclass

class D2 extends B;
endclass

initial begin
    D1 objd1 = new;
    D2 objd2 = new;
    ->objd1.cov_event;
    objd2.randomize();
end
```

An extended scenario is as follows.

The classes D1 and D2 are derived from the base class B, and the actual covergroup is defined in another class E. A coverpoint in E is sampling d2.i1, but randomization happens on D1 object.

```
class B;
    rand integer i1;
    event cov_event;
endclass

class D1 extends B;
endclass
```

```

class D2 extends B;
endclass

class E;
    D2 d2;
    covergroup MyGrp@(cov_event);
        cp1 : coverpoint (d2.i1);
    endgroup
endclass

initial begin
    D1 objd1 = new;
    D2 objd2 = new;
    E obje = new;
    obj1.randomize();
    ->obje.cov_event;
end

```

Limitations

- If multiple assignments are found, the procedural context at compile-time cannot be resolved as shown in the following example,

```

class MyClass;
    rand integer i1, i2;
    integer i3;
    event cov_event;

    covergroup MyCov @(cov_event);

        cp1 : coverpoint (i1) {
            bins s0 = {0};
            bins s1 = {1};
            bins s2 = {2};
            bins s3 = {3};
        }

    endgroup

```

```

        function automatic new;
        begin
            MyCov = new;
        end
        endfunction

        task my_sample1();
        begin
            i3 = i1;
            ->cov_event;
        end
        endtask
        task my_sample2();
        begin
            i3 = i2;
            ->cov_event;
        end
        endtask

    endclass

```

- The RHS expression of an assignment must be a class member variable, so it can be scoped. Echo cannot target procedural assignments written with respect to global variables, local function variables, or variables assigned by parameters as shown in the following example,

```

rand integer r1;

class MyClass;
    integer i1;
    event cov_event;

    covergroup MyCov @(cov_event);
        cp1 : coverpoint (i1) {
            bins s0 = {0};
            bins s1 = {1};
            bins s2 = {2};
            bins s3 = {3};

```

```
}

endgroup

function automatic new;
begin
    MyCov = new;
end
endfunction

task my_sample();
begin
    i1 = r1;
    ->cov_event;
end
endtask
endclass
```

11

Multicore ALP FSDB Dumping

FSDB is a simulation history file format for Novas (SpringSoft) tools.

You enable the parallel writing of this simulation history file with the `-parallel+mtfsdb` compile-time or runtime option. The `mt` in `mtfsdb` stands for multi-threaded.

To enable the same executable for both serial and parallel FSDB dumping, enter the runtime option instead of the compile-time option.

You can use the same system tasks, beginning with `$fsdb`, in your Verilog source code.

You can use window-based dumping, start and resume dumping, and change the dumping hierarchy during the simulation.

Limitations

- You need to request a special library from Novas (SpringSoft).
- Parallel FSDB dumping will not work with Multicore DLP (an LCA feature).

12

VCS Multicore Technology Design Level Parallelism (Part 1)

VCS Multicore Technology takes advantage of the computing power of multiple processors in one machine to improve simulation turnaround time. For the current release, Design Level Parallelism is an LCA feature. For the sake of clarity, this document describes the whole VCS Multicore Technology feature, including both design level parallelism (DLP) and the GA Application Level Parallelism (ALP).

Candidates for DLP should be a long running simulation. Short running simulations of an hour or less may not show much value to user even if DLP can show some gain. It's the long running testcase typically several hours/days where you will see most value if DLP can demonstrate any gain.

To evaluate your design's suitability for DLP:

1. Run the unified profiler (with the `-simprofile` compile-time and runtime options and the `profprt` report writing utility).
2. Read the *CPU Time Instance View* to determine if it is a good candidate for parallel simulation and identify partitions for use. (See “Profiling a Serial Simulation” and “Multicore DLP Autopartitioning”.)

Note:

The unified profiler is now an LCA feature, see [The Unified Simulation Profiler](#).

- If the *CPU Time Instance View* shows that there are sub-hierarchies that use most CPU time (instead of having few percents scattered across the design), then these sub-hierarchies are good candidates for DLP partitions.
- If there is uniform distribution of CPU time or one instance takes most of the CPU time, then the design is not suitable for DLP.
- If there are too many instances and all of them take about 5% or less of the total CPU time, then the design is not suitable for DLP either. The distribution should be at least 10% or higher per instance to indicate a good DLP opportunity.
- The cumulative simulation activity in the partitions chosen should be at least 50% or higher (for example you can have two partitions taking 25% each or five partitions taking 10% each).
- There should be at least 2 partitions to try out DLP apart from master partition.
- For multicore designs, the cores are typically the partitions.

- Ensure from serial profile output that there is minimal Testbench overhead (either in the form PLI or SVTB or any other test-bench language). If most of the time spent shows up in program block (test bench), then there is not too much opportunity for DLP.
-

VCS Multicore Technology Options

You use the VCS `-parallel` option to invoke parallel compilation. The syntax is:

```
vcs filename(s).v -parallel [+multicore_option(s)]
[-parallel+show_features] [-o multicore_executable_name]
[vcs-options]
```

These options and properties are as follows:

`-parallel`

When used without VCS Multicore options, `-parallel` enables all VCS Multicore Technology options, except for design level parallelism. When used with VCS Multicore options, `-parallel` enables only those options specified.

This option is available at compile-time only.

`+design=FILENAME`

Enables design level parallelism and specifies the name of the partition configuration file. Note: this option is available at compile-time only. (Also see “[Multicore DLP Autopartitioning](#)”).

`+fc [=NCONS]`

This compile-time option, enables Multicore Functional Coverage, and with `NCONS` specifies the number of PFC consumers. `NCONS` can be changed at run time.

```
vcs -parallel+fc ...
```

```
vcs -parallel+fc=3 ...  
  
+profile  
    Enables Multicore design and application level profiling.  
  
+profile_value  
    Enables value-based design level profiling.  
  
+sva [=NCONS]  
    This compile-time option enables multicore SVA, and with NCONS specifies the number of multicore SVA consumers. NCONS can be changed at run time.  
  
+tg1 [=NCONS]  
    Enables multicore Toggle Coverage, and specifies the number of multicore toggle coverage consumers. To enable the use of the same executable for both serial and parallel runs, use this option at runtime.  
    NCONS specifies the number of multicore SVA consumers. For ALP, NCONS can be changed at run time.  
  
+vpd [=NCONS]  
    Enables multicore VCD+ Dumping, and specifies the number of multicore VCD+ consumers. To enable the use of the same executable for both serial and parallel runs, use this option at runtime.  
    NCONS specifies the number of multicore SVA consumers. For ALP, NCONS can be changed at run time.  
  
[-o multicore_executable_name]  
Using the VCS -o option to specify the simulation executable binary filename allows work on multiple simultaneous VCS Multicore compiles and runs. VCS Multicore-specific data is stored in a directory executable_name.pdaidir. The default path name is simv.pdaidir.
```

Note:

If [NCONS] is not specified, the default is 1 client.

`-parallel+show_features`

Displays enabled VCS Multicore features. Note that you must enter the `-parallel` option with `+show_features`

Examples:

`-parallel+vpd` is equal to `-parallel+vpd=1`
`-parallel+tgl` is equal to `-parallel+tgl=1`

VCS Multicore option examples:

```
vcs -parallel+design=part.cfg ....  
vcs -parallel+fc .... -o psimv  
vcs -parallel+vpd+fc -parallel+tgl -o par_simv ....  
vcs -parallel+design=part.cfg+sva ....
```

Use Model for Design Level Profiling and Simulation

Design level parallelism (DLP) performance gains depend on a number of factors, including

- The degree of parallelism in the design
- Design partition activity (Also see “[Multicore DLP Autopartitioning](#)”.)
- Location of clock logic in the master partition or replicated in each partitions. Avoid clock dependency between partitions in which one partition generate the clock and feeds into the other partition.

Design Suitability

Characteristics of design types suitable for DLP include:

- Parallel scan DFT designs
- Large BIST designs with parallel cores
- Multicore designs
- Large blocks instantiated multiple times
- Processor type of designs
- Design pipelines making the blocks run in parallel (e.g., network processors)

DLP Use Model

Follow these steps to determine design qualification criteria for VCS Multicore DLP.

1. Ensure the design (and tests) runs well with the latest VCS version.
The selected testcase for DLP should be a long running simulation. Short running simulations of an hour or less may not show much value to user even if DLP can show some gain. It's the long running testcase typically several hours/days where you will see most value if DLP can demonstrate any gain.
2. Run the serial profiler and generate vcs.prof data. The VCS profiler tells you the subhierarchies in your design that use the most CPU time. See "[Profiling a Serial Simulation](#)" .
3. Analyze the instance based profiling to determine if it is a good candidate for parallel simulation and identify partitions for use.

If the Instance view shows that there are sub hierarchies that use most CPU time (instead of having few percents scattered across the design), then these sub hierarchies are good candidates for DLP partitions. For Example:

Instance	%Totaltime
top.A1	35.82
top.A2	33.31
top.A3	30.87

Here top.A1, top.A2 and top.A3 are good candidates for DLP partitions. (Also see “[Multicore DLP Autopartitioning](#)” .)

If there are no uniform distribution or one instance takes most of the %Totaltime, the design is not suitable for DLP.

If there are too many instances and all of them take about 5% or less %Totaltime it is not suitable for DLP either. The distribution should be at least about 10% or higher per instance to indicate a good DLP opportunity.

4. Cumulative simulation activity in the partitions chosen should be at least 50% or higher (e.g. you can have 2 partitions taking 25% each or 5 partitions taking 10% each)

There should be at least 2 partitions to try out DLP apart from master partition.

For multicore designs, the cores are typically the partitions.

Ensure from serial profile output that there is minimal Testbench overhead (either in the form PLI or SVTB or any other test-bench language). If most of the time spent shows up in program block (test bench), it means not too much opportunity for DLP.

5. Specify the partitions in the VCS Multicore configuration file. See “[Specifying Partitions](#)” . (Also see “[Multicore DLP Autopartitioning](#)” .)

6. Run VCS Multicore compilation, specifying the VCS Multicore configuration file. For more information, see “[Design Level Simulation](#)” .

```
vcs -parallel+design=partition.cfg  
simv
```

If you wish to increase the performance gains from VCS Multicore, additional steps follow:

7. Run VCS Multicore simulation again and this time collect data for the VCS Multicore profiler. See “[Profiling a VCS Multicore Simulation](#)” .

Use Model for Assertion Simulation

1. Run VCS Multicore compilation specifying the `-parallel+sva` option:
2. Run VCS Multicore simulation.

Use Model for Toggle and Functional Coverage

1. Run VCS Multicore compilation specifying the VCS Multicore `tgl` option and coverage metric options for toggle coverage, and/or the VCS Multicore `fc` option for functional coverage. You can optionally specify the number of consumers for each.
2. Run the simulation to generate coverage results.
3. Generate coverage result reports.

Use Model for VPD Dumping

1. Run VCS Multicore compilation specifying the `-parallel+vpd` option.
 2. Run the simulation to generate the VPD file.
-

Running VCS Multicore Simulation

VCS Multicore Technology takes advantage of the computing power of multiple processors to improve simulation turnaround time

You can generate results for one of all the following VCS Multicore Technology options in a simulation:

- Design simulation
 - Assertion simulation
 - Toggle coverage
 - Functional coverage
 - VPD file generation
-

Design Level Simulation

Once you have profiled your design and created a partition configuration file, you can simulate your design with design level parallelism only or in combination with some of the ALP options.

1. Compile using the VCS Multicore `-parallel` option and other VCS Multicore and VCS options.

```
vcs filename(s).v -parallel+design=partition_filename.cfg  
[multicore_options] vcs_options
```

2. Run the simulation with VCS and VCS Multicore run-time options.

```
simv
```

(Also see “[Multicore DLP Autopartitioning](#)”.)

Assertion Simulation

You can process only assertion level results or assertion level results along with other VCS Multicore options with this compile-time option.

1. Compile using the VCS Multicore `-parallel+sva` option, the assertion compilation option or options, and other VCS Multicore and VCS options.

```
vcs filename(s).v -parallel+[sva [=NCONS] ]  
[ multicore_options vcs_options
```

2. Run the simulation with VCS and VCS Multicore run-time options.

```
simv
```

Toggle Coverage

Generate results for only toggle coverage or toggle coverage along with other results by compiling the design with VCS Multicore options that include the `+tg1` option and VCS coverage metrics options. You can use the `+count` option to report total executed transactions. After generating coverage results, you can examine them using the Unified Report Generator.

Note:

To enable the use of the same executable for both serial and parallel runs, use this option at runtime.

tgl [+count]

Report total executed transactions.

1. Compile using the VCS Multicore `-parallel` option, coverage option or options, and other VCS Multicore and VCS options.

```
vcs filename(s).v -parallel+tgl [=NCONS] -cm tgl  
[multicore_options] [vcs_options]
```

2. Run the simulation to generate coverage results.

```
simv -cm tgl [vcs_options]
```

3. Generate coverage result reports:

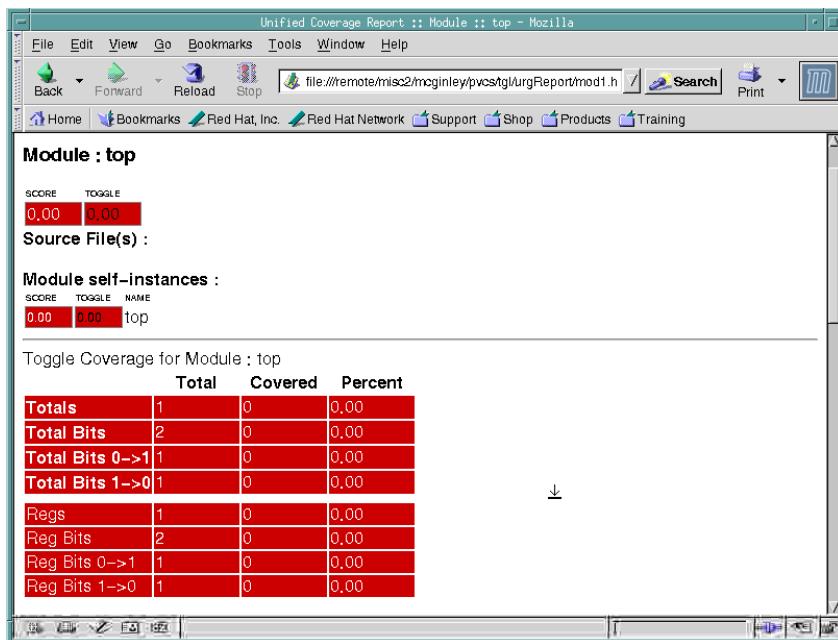
```
urg -dir coverage_directory.cm urg_options
```

Example

In this example, toggle coverage results only are generated and the URG report is produced in the default HTML format.

```
% vcs -cm_tgl mda -q -cm_dir pragmaTest1.vdb -cm tgl \  
-sverilog -parallel+tgl=2 pragmaTest1.v  
% simv -cm tgl  
% urg -dir pragmaTest1.vdb
```

Results can then be examined in your default browser.



Functional Coverage

Generate results for only functional coverage or functional coverage along with other results by compiling the design with VCS Multicore options that include the `+fc` option and VCS coverage metrics options. Note that this is a compile-time option. After generating coverage results, you can examine them using the Unified Report Generator.

1. Compile using the VCS Multicore `-parallel` option, coverage option or options, and other VCS Multicore and VCS options.

```
vcs filename(s).v -sverilog -parallel+fc [=NCONS]  
[parallel_vcs_options] [vcs_options]
```

2. Run the simulation to generate coverage results.

```
simv
```

3. Generate coverage result reports:

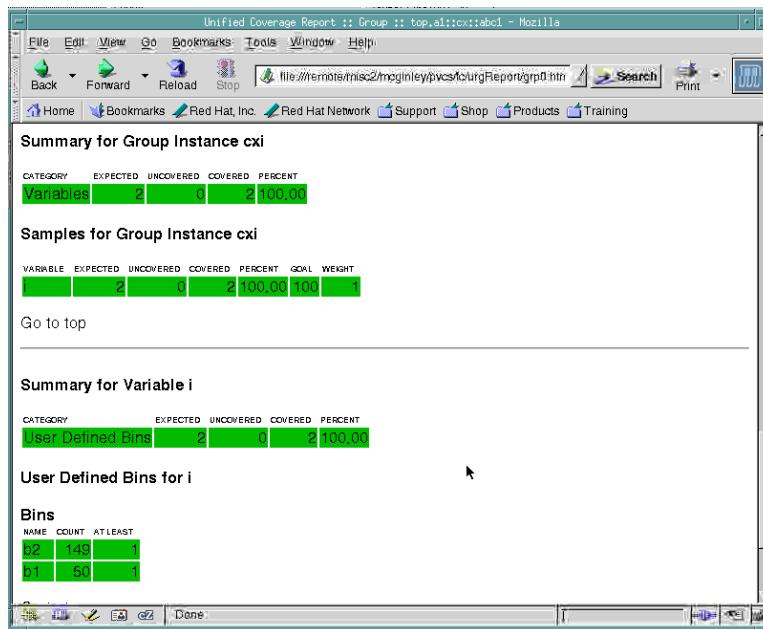
```
urg -dir coverage_directory.vdb urg_options
```

Example

In this example, functional coverage results only are generated and the URG report is produced in the default HTML format.

```
% vcs iemIntf.v -sverilog -parallel+fc=2  
% simv -covg_cont_on_error  
% $urg -dir simv.vdb  
% cat urgReport/gr*  
%
```

Results can then be examined in your default browser.



VPD File

You can enable VCS Multicore VPD+ Dumping and specify the number of VCS Multicore VPD+ consumers using the VCS Multicore `vpd` option. To enable the use of the same executable for both serial and parallel runs, use this option at runtime.

Note:

When used with multiple consumers, VPD file size blow up might be an issue. Use `-parallel+vpd_buffer=<N>`, where N=256, 512 etc.

1. Compile using the VCS Multicore `-parallel` option with the `vpd [=NCONS]` option, and other VCS Multicore and VCS options.

```
vcs filename(s).v -debug_pp -parallel+vpd [=NCONS]  
[multicore_options] [vcs_options]
```

2. Run the simulation.

```
simv
```

You can post-process the results with the generated +VPD database.

Example

In this example, a VPD+ file with three specified consumers is generated.

```
% vcs -debug_pp -parallel+vpd=3 design.v  
% simv
```

Profiling a Simulation

You can profile your VCS Multicore simulation to ensure efficient use of resources. This section details the profiling process. It contains the following sections:

- “[Profiling a Serial Simulation](#)”
- “[Specifying Partitions](#)”
- “[Profiling a VCS Multicore Simulation](#)”
- “[Running VCS Multicore Examples](#)”

Profiling a Serial Simulation

The unified profiler is a serial profiler that can tell you the following:

- if your design and testbench is a good candidate for VCS Multicore simulation
- identify partitions to use.

You enable, monitor, and report unified profile information with the `-simprofile` compile-time and `-simprofile` time runtime option and the `profrpt` report writing utility.

The `profrpt` utility, among other views, writes the *CPU Time Instance View* that tells you the CPU time used by the individual instances. This view contains, for an instance the exclusive time — the CPU time used by the instance alone, and the inclusive time — the CPU time used by the instance itself and all the instances that are hierarchically under the instance—in other words the sub-hierarchy.

The sub-hierarchies that use the most CPU time are good candidates for VCS Multicore partitions. In [Figure 12-1](#) you would use separate partitions for the subhierarchies.

Note:

The unified profiler is now an LCA feature, see [The Unified Simulation Profiler](#).

Specifying Partitions

The `-design` option requires a partition configuration file. (Also see [“Multicore DLP Autopartitioning”](#).) You specify the partitions in the VCS Multicore configuration file using the following syntax:

```
partition {hierarchical_name(module_identifier) ,...} ;
partition {hierarchical_name(module_identifier) ,...} ;
partition {hierarchical_name(module_identifier) ,...} ;
.
.
.
```

The syntax is as follows:

partition

The keyword that specifies that what follows are the contents of one partition.

hierarchical_name

The hierarchical name of a Verilog module instance that is the top-level instance of the subhierarchy that you want to simulate as a partition.

module_identifier

The name of the module definition that corresponds to that top-level instance.

All parts of the design not covered in any of the specified partitions together form an implicitly defined master partition. The user-specified partitions are called slave partitions.

Example 12-1 VCS Multicore Configuration File

In this example, there is a separate line for each partition. Each hierarchical name for a top-level module instance must be followed by its module name in parentheses. You can specify more than one subhierarchy in a partition. That is, you can group multiple subhierarchies together to form a single partition.

```
partition {top.A1_inst(A1)};  
partition {top.A2_inst(A2)};  
partition {top.A3_inst(A3)};
```

Note: You can use the Verilog comment syntax to comment your partition file. For more information on creating configuration files, see the *VCS User Guide*.

Example 12-2 A Free Format Configuration File

You can also enter a free format of the specifications in a configuration file entering new lines, blanks, and comments.

```
partition { top.bus1 (MyBus) ,  
            top.cpu (CPU) } ; // two instances per partition  
partition {  
            top.bus2 (MyBus)  
} /* one instance per partition */
```

Profiling a VCS Multicore Simulation

Run the simulation by entering a command line with the name of the master executable (by default named `simv`) and runtime options. The syntax for this command line is as follows:

```
simv -parallel+profile [VCS_runtime_options]
```

VCS Multicore Profiling Options

If you entered the `-parallel+profile` option on the `simv` command line, VCS Multicore Technology will write the data files that the VCS Multicore profiler needs to report on the VCS Multicore simulation.

You start the profiler with the `pvcsProfiler` command. Its syntax is as follows:

```
pvcsProfiler [profileDumpDir=simvName.pdaidir]
[doTimeProfiling=1|0] [doToggleProfiling=0|1]
[graphImHeight=integer] [graphImWidth=integer]
[graphTnHeight=integer] [graphTnWidth=integer]
[lowerSimTimeBound=float] [runVersion=string]
[toggleCutOff=float] [upperSimTimeBound=float]
[-help]
```

These arguments and properties are as follows:

`profileDumpDir=simvName.pdaidir`

Use `profileDumpDir` to specify the directory containing the dump files that the profiler reads. The directory is the name you specified for the simulation executable binary file with the extension `.pdaidir`. The default path name is `simv.pdaidir`.

The profiler writes HTML files that display profile information about the master and slave simulations. By default the profiler creates the `ppResults_0` directory and writes these files in this directory.

`doTimeProfiling=1|0`

Calculate the time accumulation and produce the graphs. The default argument is 1. If the argument is 0, the profiler does not make this calculation or produce the graphs.

`doToggleProfiling=1|0`

Calculate the partition port toggle counts and produce the high count listing. The default argument is 1. If the argument is 0, the profiler does not make this calculation.

`graphImHeight=integer`

Height of the full size graphs in pixels. The default height is 1000.

`graphImWidth=integer`

Width of the full size graphs in pixels. The default width is 1000.

`graphTnHeight=integer`

Height of the thumbnails size graphs in pixels. The default height is 250.

`graphTnWidth=integer`

Width of the thumbnails size graphs in pixels. The default width is 250.

`lowerSimTimeBound=float`

A floating point number for the simulation time when profiling starts. The default argument is 0.0.

`runVersion=string`

By default the profiler creates the `ppResults_0` directory in the current directory and writes its output HTML files in this directory. If you want a different name for this directory, enter this property. With this property the profiler creates the `ppResultsstring` directory.

`toggleCutOff=float`

Specifies that the port toggle count cutoff for a partition is equal to this percentage of highest toggle count. The default argument is 1.0.

`upperSimTimeBound=float`

A floating point number for the simulation time when profiling stops. The default argument is 1e+100.

`-help`

Displays the valid properties and their definitions.

Examining the Profiler Results

The main results file that the profiler writes is named `results.html`. By default it writes it in a directory named `ppResults_0` in the current directory. The following is an example of a `results.html` file (as viewed in a regular web browser). Note that if `ppResults_0` already exists, it creates a directory `ppResults_#` where `#` is the next available number.

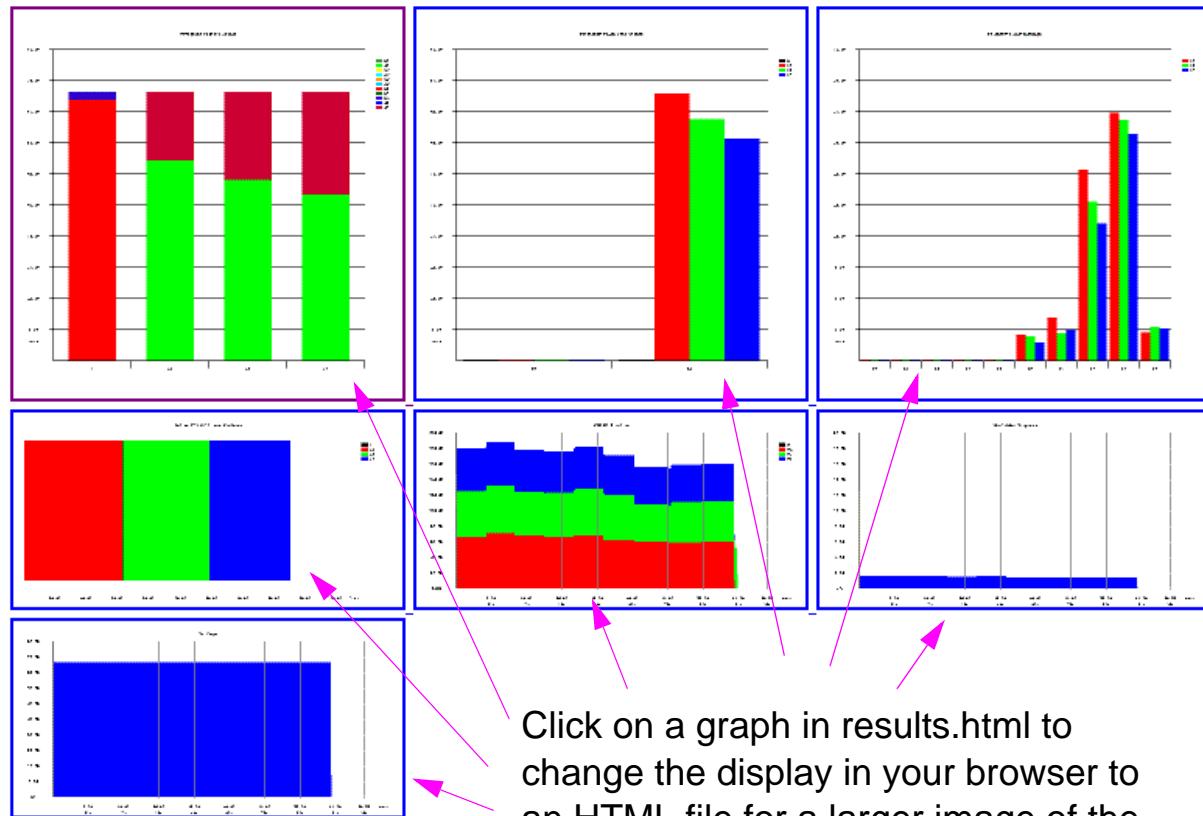
Profiler Results Showing Good Parallelism

The following figures show the output from the profiler that indicated good parallelism and the design simulated with these results can benefit from Multicore DLP.

Figure 12-1 `results.html` File

Summary Profiling Results

Created Fri Oct 2 08:58:13 2009



Click on a graph in `results.html` to change the display in your browser to an HTML file for a larger image of the graph.

Click here to see how to read the graphs

[Notes on how to read the graphs](#)

The `results.html` file contains links to HTML files for the seven graphs. The most important graphs are as follows:

- The Processor Segment Totals (`graph1.html`) see [Figure 12-2](#)
- The Processor Delta Time Totals (`graph2.html`) see [Figure 12-5](#)
- The S1 Balance Distribution (`graph3.html`) see [Figure 12-8](#)
- The Active VCS Multicore Features (`graph4.html`) see [Figure 12-9](#)

The graphs of lesser importance are as follows:

- The CPU Utilization (`graph5.html`)
- Simulation Progress (`graph6.html`)
- The VM Usage (`graph9.html`)
- There is not `graph7.html` or `graph8.html` in the current implementation.

Each image in `results.html` is a hypertext link to an HTML file with a larger image of the graph.

The `results.html` file also contains the [Notes on how to read graphs](#) link to the `howtoInterpretGraphs.html` file that explains how to interpret the graphs.

Figure 12-2 The Processor Segment Totals Graph (graph1.html)

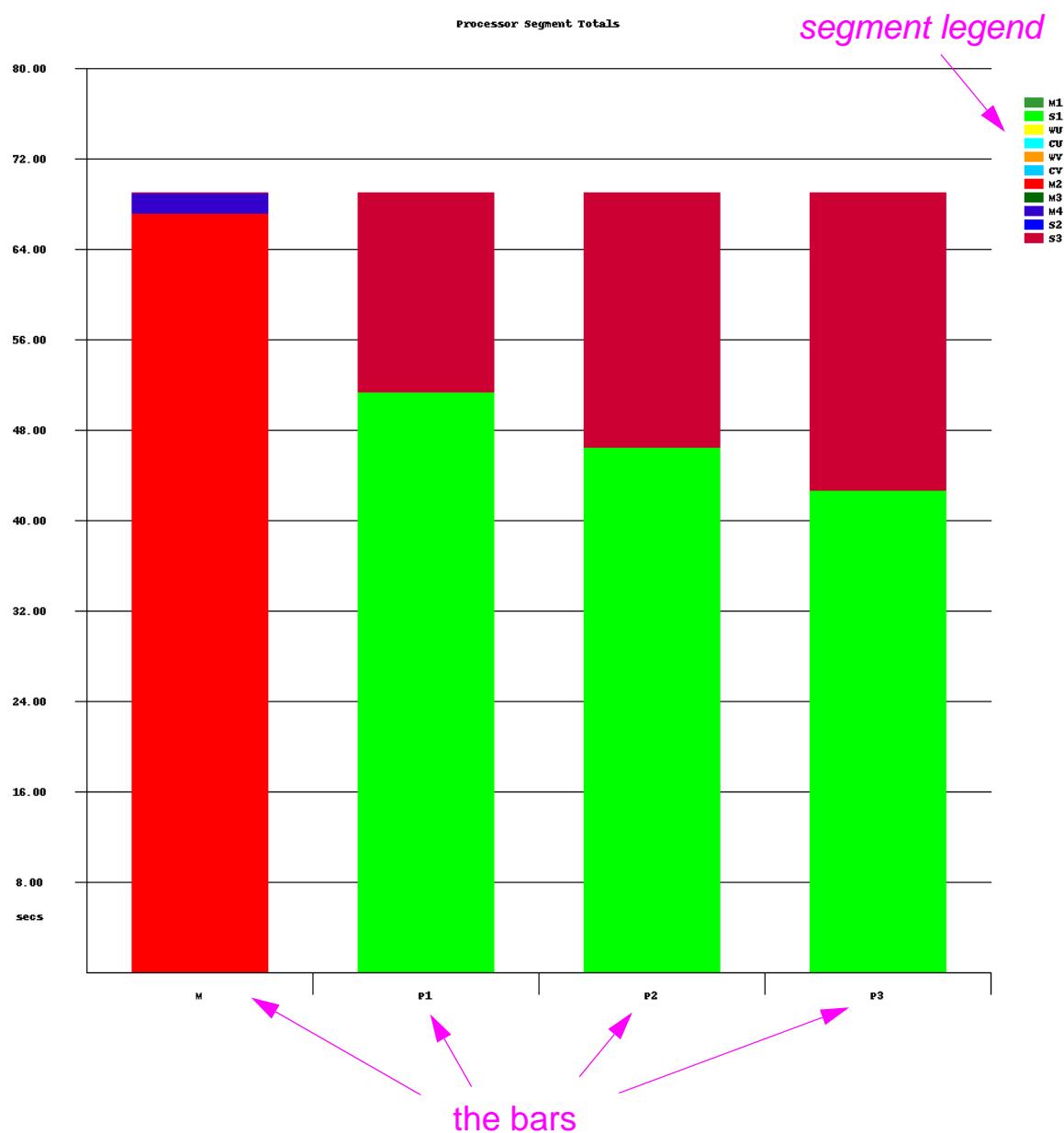


Figure 12-3 The Bars in the Processor Segment Totals Graph

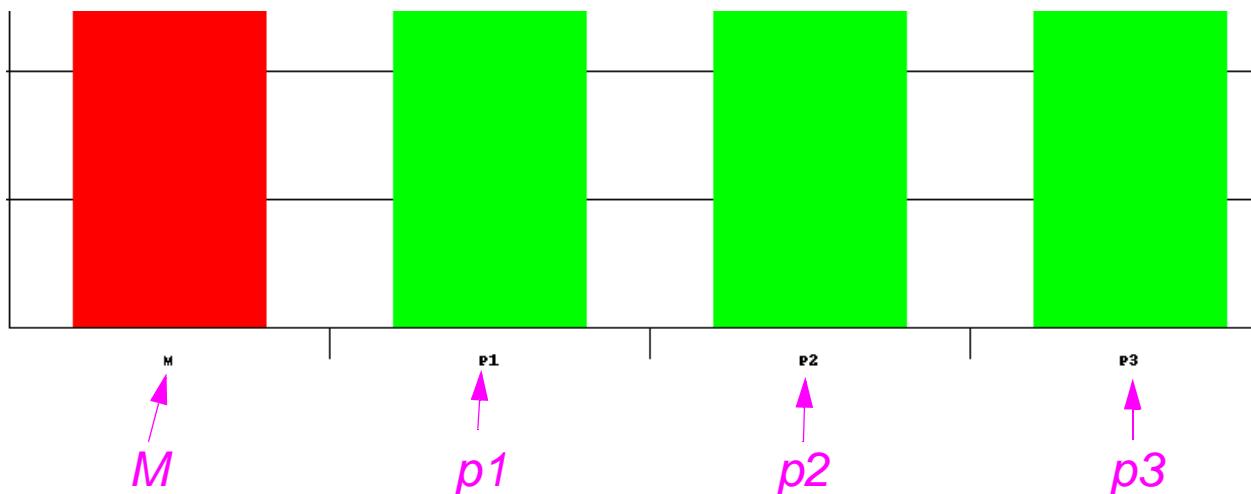


Figure 12-3 shows you the names of the bars in the Processor Segment Totals Graph:

M: shows the bar for the master partition

p1: shows the bar for a slave partition, in this case the first core specified in the partition file.

p2: shows the bar for a slave partition, in this case the second core specified in the partition file.

p3: shows the bar for a slave partition, in this case the third core specified in the partition file.

The partition file in this example is as follows:

```
partition {top.A1 (A1)};  
partition {top.A2 (A2)};  
partition {top.A3 (A3)};
```

Figure 12-4 The Segment Legend



[Figure 12-4](#) shows the legend for the Processor Segment Totals Graph. It shows you what each segment in the bars represents. (Complete information on these segments is in the `howtoInterpretGraphs.html` file.)

M1: The M1 segment is only in the left-most bar, the one for the master partition. It represents the time spent by the master partition on its own events. There is no M1 segment in the bar for the master partition in [Figure 12-2](#).

M2: The M2 segment is also only for the master partition, it shows the time the master partition waits for synchronizing events in the slave partitions. (The slave partitions are the cores you specified in your partition file.) In [Figure 12-2](#) most of the time for the master partition is in this segment.

M3: The M3 segment is also only for the master partition, it shows the time spent propagating values received during segment the M2 segment. You should seek to minimize the amount of time in this segment. There is no M3 segment in the bar for the master partition in [Figure 12-2](#).

M4: The M4 segment is also only for the master partition, it represents the time spent propagating values to the slave partitions, the cores. You should seek to minimize the amount of time in this segment. A small amount of time in [Figure 12-2](#) is spent in this segment.

S1: The S1 segments show the time used by each slave partition, or core, to execute its own events. These events happen in parallel. Most of the time in the slave partition bars in [Figure 12-2](#) is in this segment.

S2: The S2 segments show the time used by the slave partitions to send signal values and simulation times to the master partition. You should seek to minimize the amount of time in this segment.

[Figure 12-2](#) shows no time spent in this segment.

S3: The S3 segments show the time used by the slave partitions waiting for the master partition to send updated signal values. You should seek to minimize the amount of time in this segment. As shown in [Figure 12-2](#), most of the time in the slave partitions is in segment S1 but a significant amount of time for each slave partition is in segment S3.

WU: The WU segments accumulate the time spent waiting to enter the critical section to execute user code. These times should be as small as possible. [Figure 12-2](#) shows no time spent in this segment.

Important:

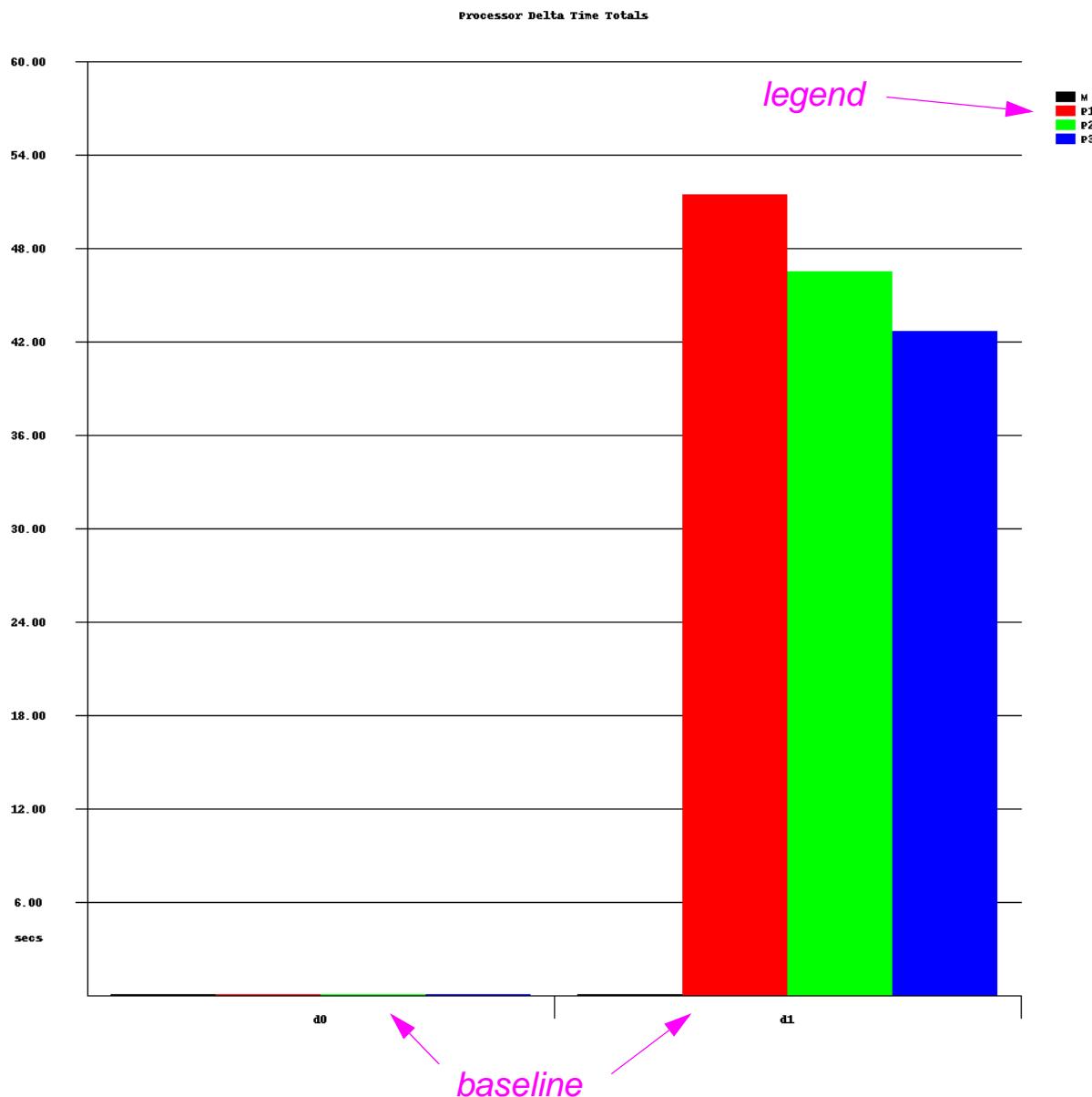
The critical section is a phase in the simulation when only one thread or partition is executing. The partition could be accessing a shared resource, something in another partition, or user PLI code.

CU: The CU segments accumulate the time spent inside the critical section while executing user code. These times should be as small as possible. [Figure 12-2](#) shows no time spent in this segment.

WV: The WV segments accumulate the time spent waiting for entering the critical section to execute VCS code. These times should be as small as possible. [Figure 12-2](#) shows no time spent in this segment.

CV: The CV segments accumulate the time spent inside the critical section while executing VCS code. These times should be as small as possible. [Figure 12-2](#) shows no time spent in this segment.

Figure 12-5 The Processor Delta Time Totals Graph (graph2.html)



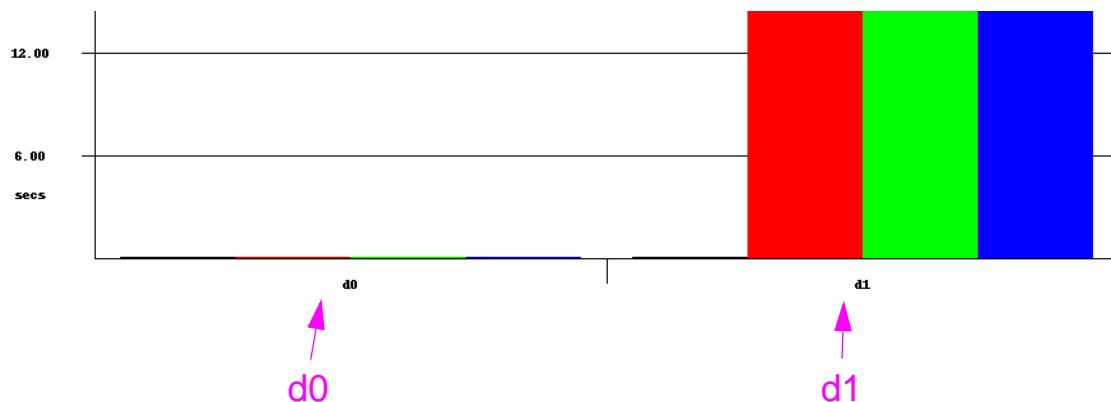
Simulation time is a series of simulation time slots and a time slot can contain one or more deltas. A delta, in this context, is the time VCS used to pass values from one partition to another.

Figure 12-6 The Processor Delta Time Total Graph Legend



In each delta in the graph a black bar is for the master partition, red for slave partition P1, green for P2, and green for P3.

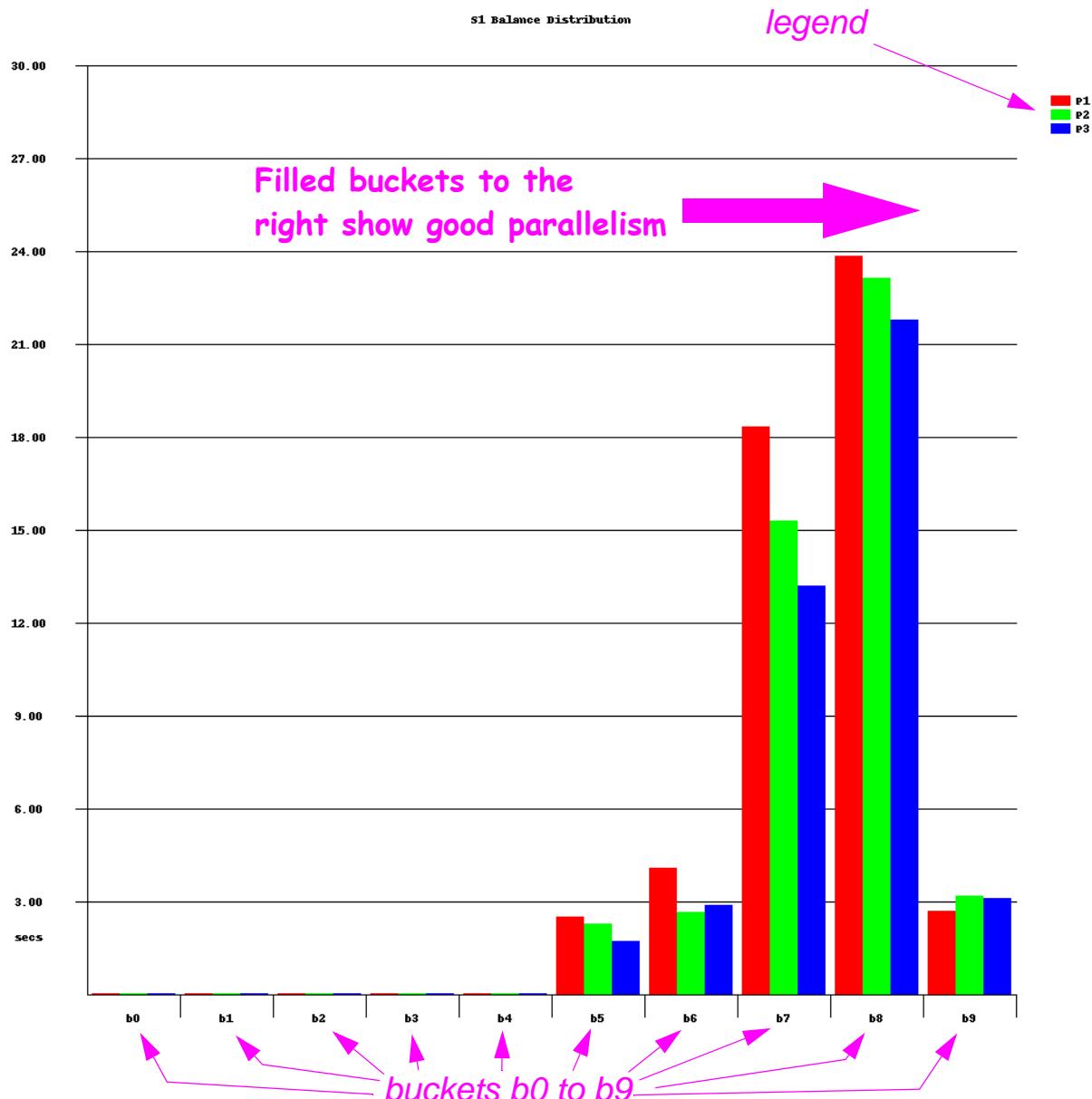
Figure 12-7 The Baseline of the Processor Delta Time Total Graph



In this example the simulation had two deltas, d_0 and d_1 , with almost all of the delta time used by the slave partitions (cores) in the second delta, d_1 . In delta d_1 the core for P1 uses the most delta time and the core for P3 uses the least, however the bars are roughly the same length indicating good parallelism in the delta.

A d3 delta, or subsequent delta, would show signal propagation from one slave partition to another, which could be a cause of an undesirable imbalance of the slave partition parallelism.

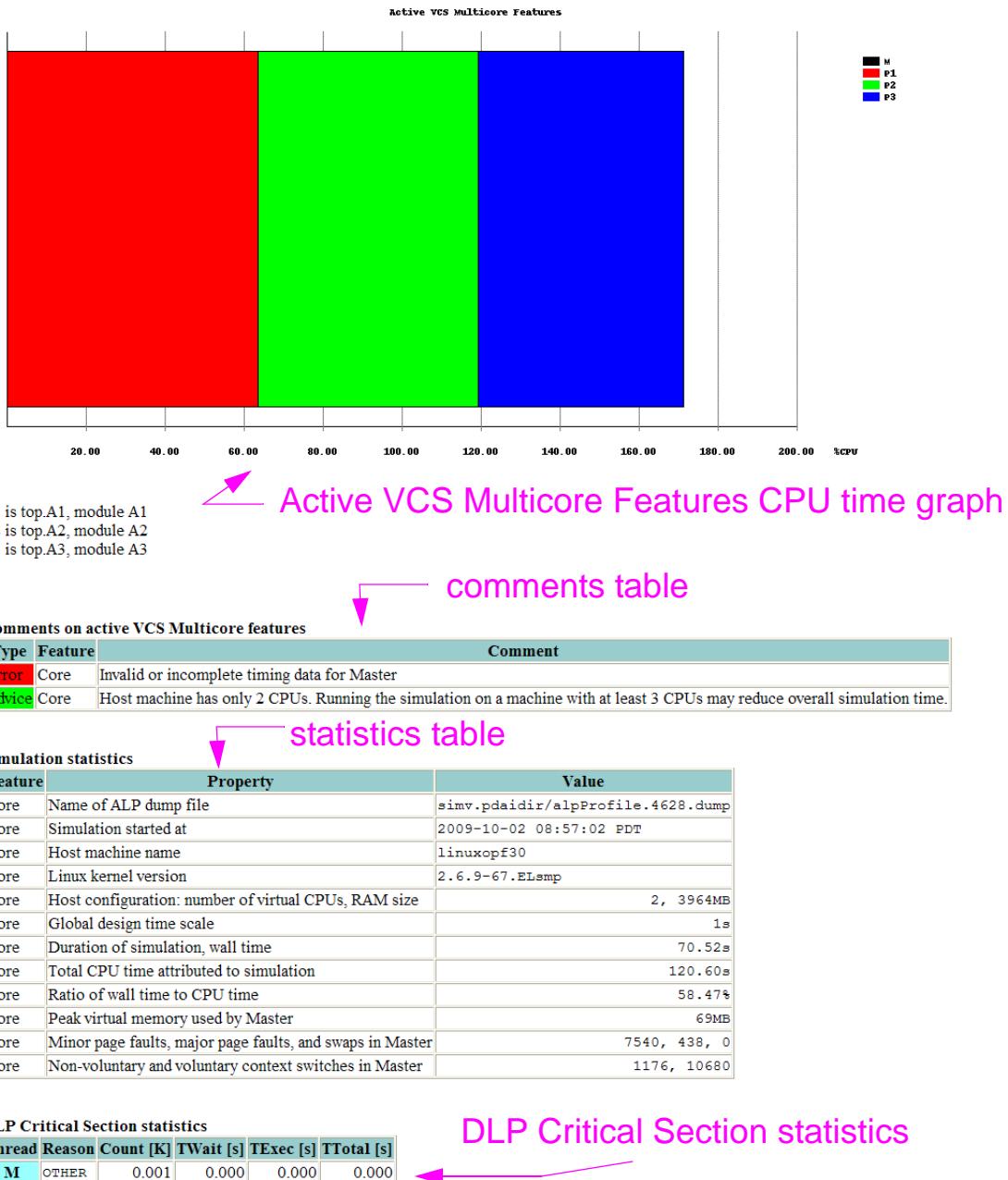
Figure 12-8 S1 Balance Distribution Graph (graph3.html)



As you remember from the Processor Segment Totals graph, S1 is the time used by slave partitions (cores) on their own events.

The S1 Balance Distribution Graph shows accumulations of S1 simulation times for the slave partitions (cores) divided into buckets. The buckets contain bars of simulation times for each slave partitions. Buckets on the right have more parallelism and those on the left have less parallelism. You want the accumulations of simulation times for each partition to be in the buckets on the right, like they are in this example.

Figure 12-9 Active VCS Multicore Features (graph4.html)



Appended to the Active VCS Multicore Features graph are tables for comments, statistics, and DLP Critical Section statistics.

This graph shows you how much of the total CPU time was used by each partition, in this example there is no CPU time usage by the master partition. Also notice that the widths of each slave partition (core) is approximately equal.

The comments table advises you of error conditions and also provides advice. In this example:

- There is an error condition, “Invalid or incomplete timing data for Master” which means that there was an error in collecting data for the master partition.
- There is also the advice to use a machine with more CPUs. This is the most important advice that is displayed in this table.

Other important possible advice is that your machine is thrashing.

All features in the statistics table in this release are core features.

For users, the important statistics are the following:

- Duration of simulation wall time
- Total CPU time attributed to simulation

In this example the wall time is almost half the CPU time, showing the advantage of using Multicore DLP.

The other statistics are more important for internal use.

Figure 12-10 DLP Critical Section Statistics

DLP Critical Section statistics					
Thread	Reason	Count [K]	TWait [s]	TExec [s]	TTotal [s]
M	OTHER	0.001	0.000	0.000	0.000

The **critical section** is a phase in the simulation when only one thread or partition is executing. The partition could be accessing a shared resource, something in another partition, or user PLI code.

In this example the table has information on thread M, in other words, the master partition. The reason OTHER is for processing user PLI code, but this is of no concern because it required no significant wait or execution time.

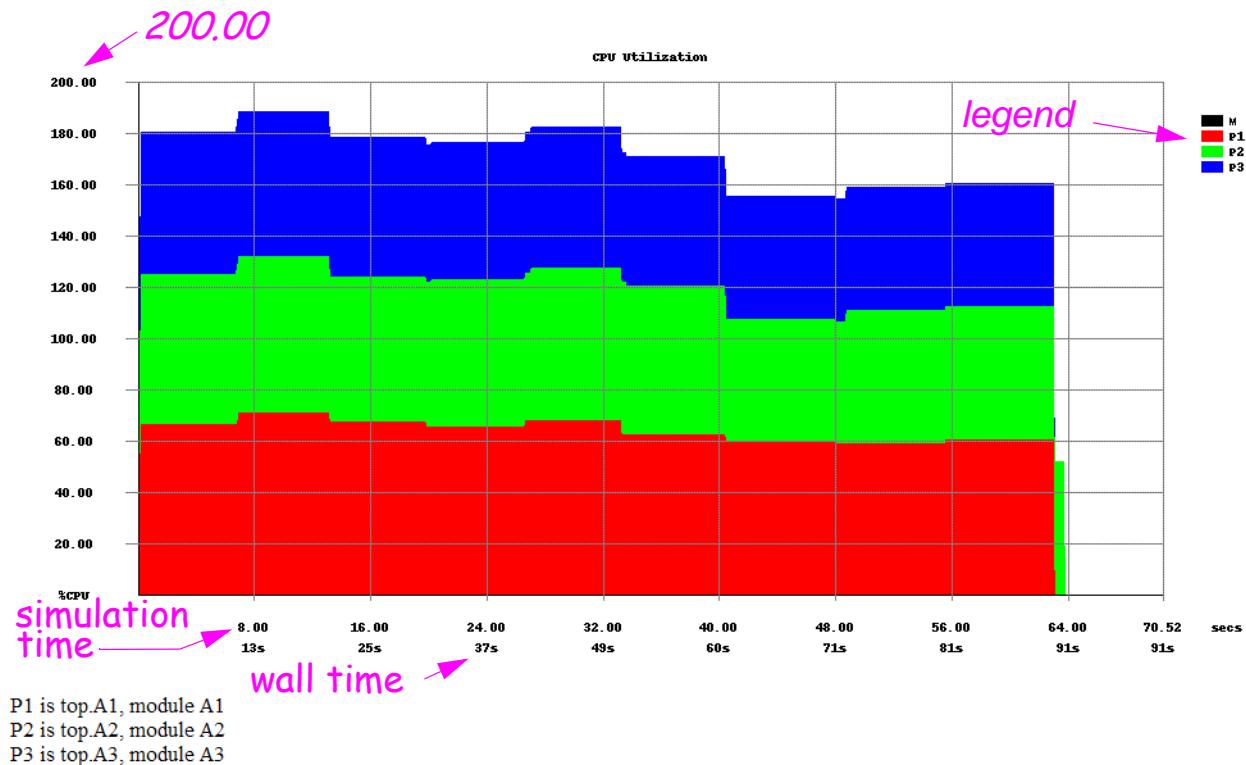
A high execution rate impedes the performance of all threads or partitions. If you see a high rate change your design to use less PLI code.

13

VCS Multicore Technology Design Level Parallelism (Part 2)

[Figure 13-1](#), the CPU Utilization graph shows you how much CPU time is used by each slave partition (core) during the simulation. This graph is less important than the earlier graphs and it is mainly for internal use.

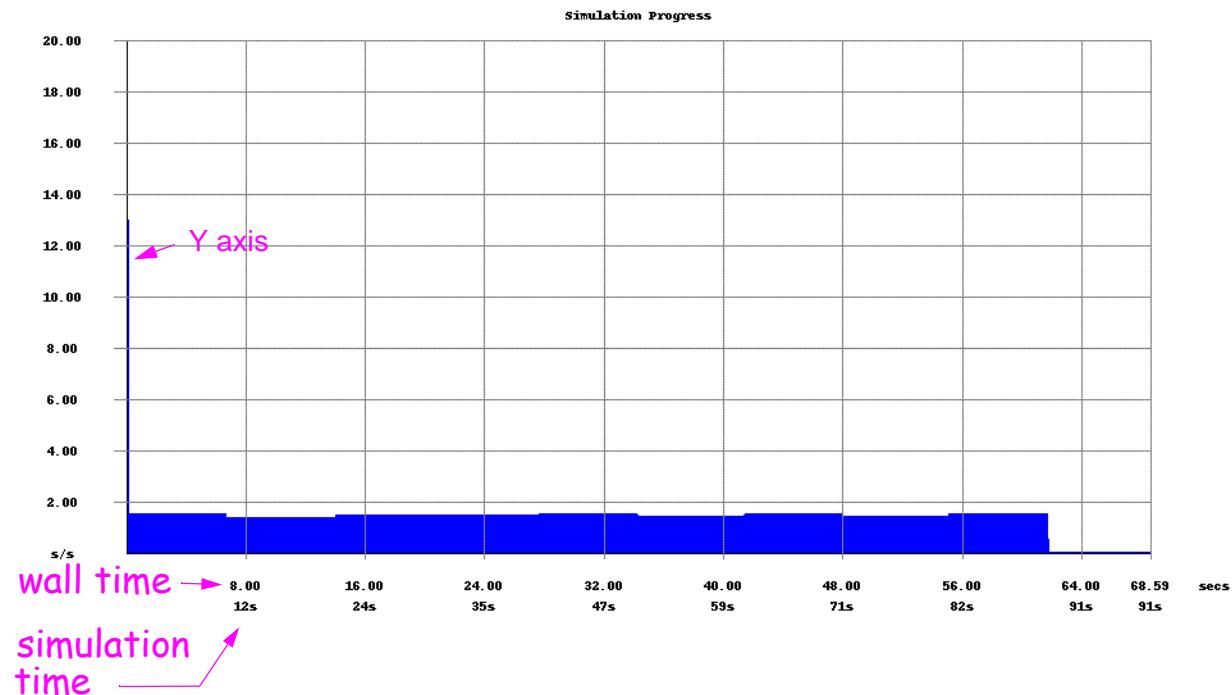
Figure 13-1 CPU Utilization Graph (graph5.html)



The legend shows you the colors for each of the partitions. As shown in this figure, partitions P1 and P3 end simulation time shortly before wall time (top line for the X axis) 64 seconds, followed shortly by partition P2.

Looking at the Y axis, each slave partition is using approximately 60% of the CPU time, indicating that there is around a 40% wait time for each slave partition.

Figure 13-2 Simulation Progress (graph6.html)

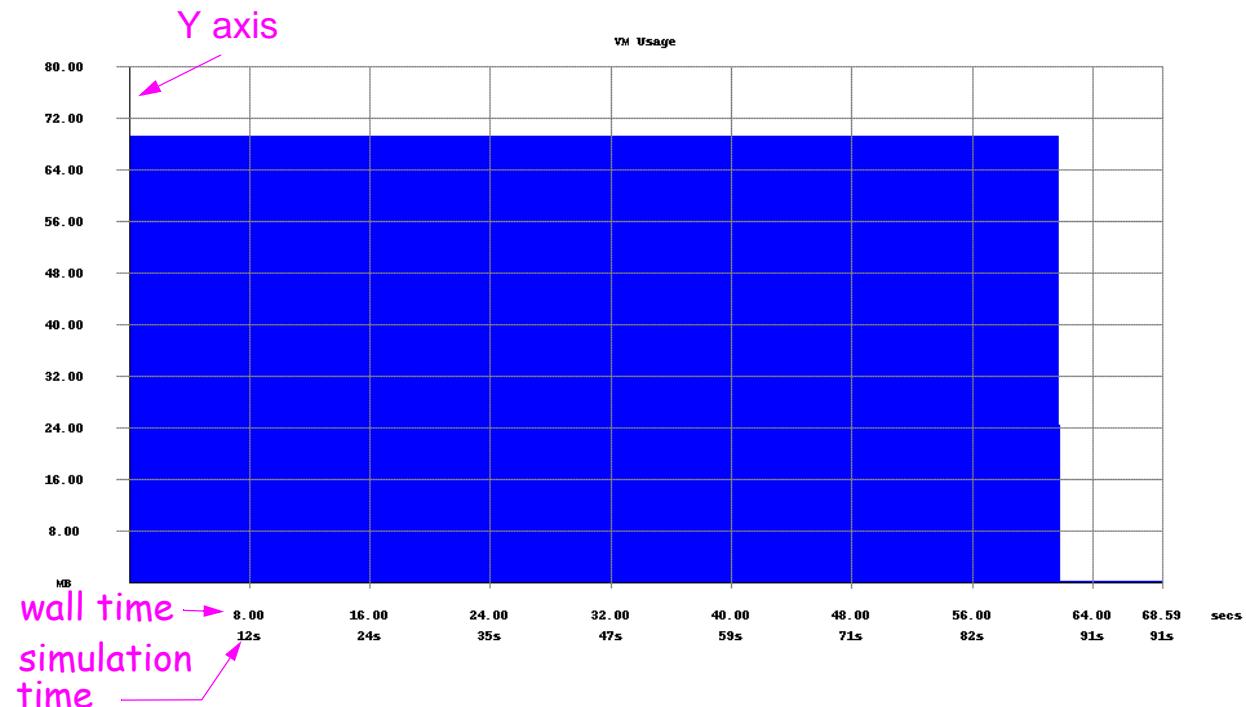


The Y axis of this graph shows you the wall time divided by the simulation time.

The X axis shows you the wall time and the simulation time in two separate lines.

In this example the simulation progress is fairly uniform but it would show a different if there were different stages such a long initialization period.

Figure 13-3 The VM Usage Graph (graph9.html)

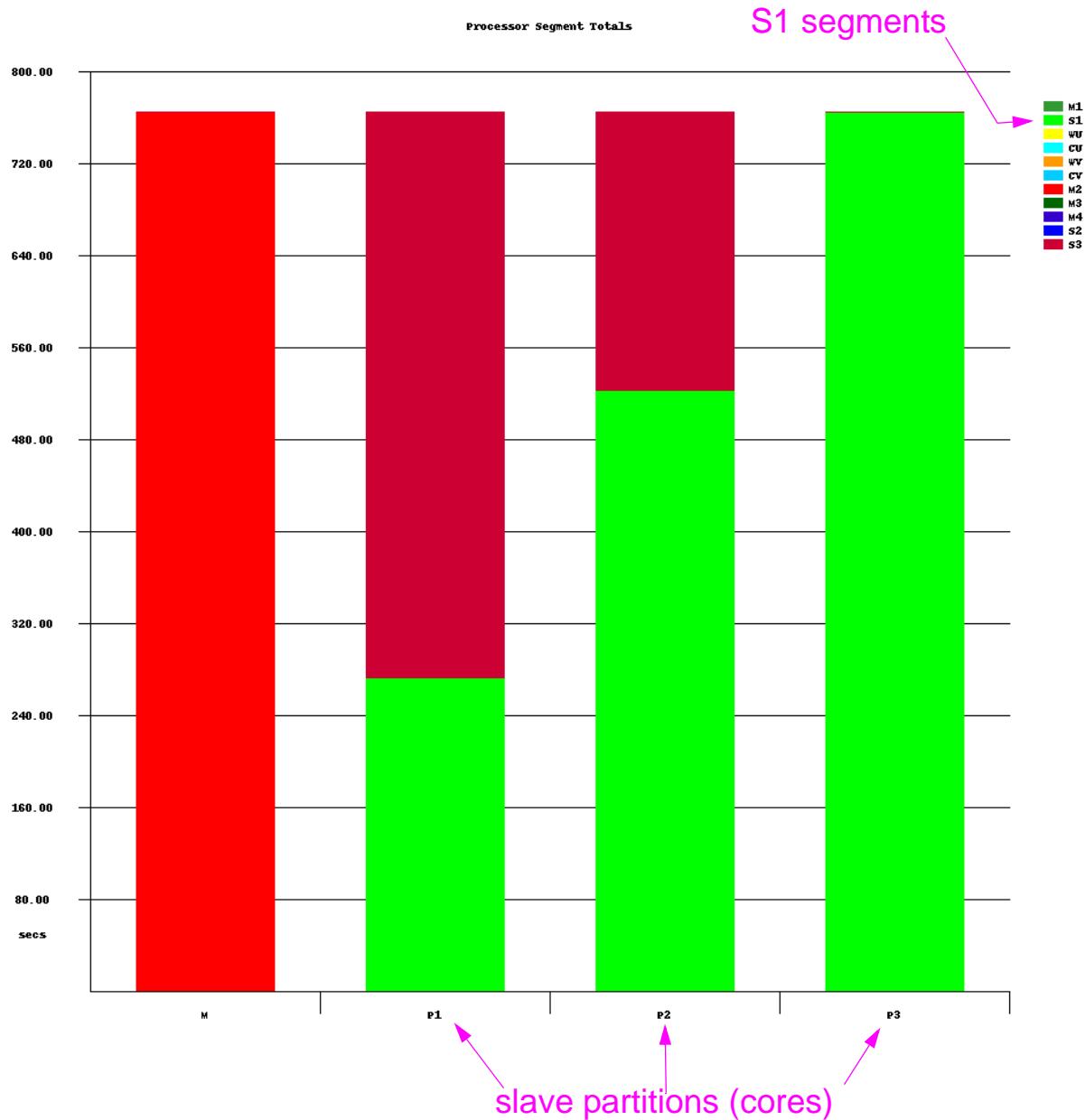


The VM Usage graph shows you the virtual memory usage over time.

Profiler Results Showing Less Parallelism

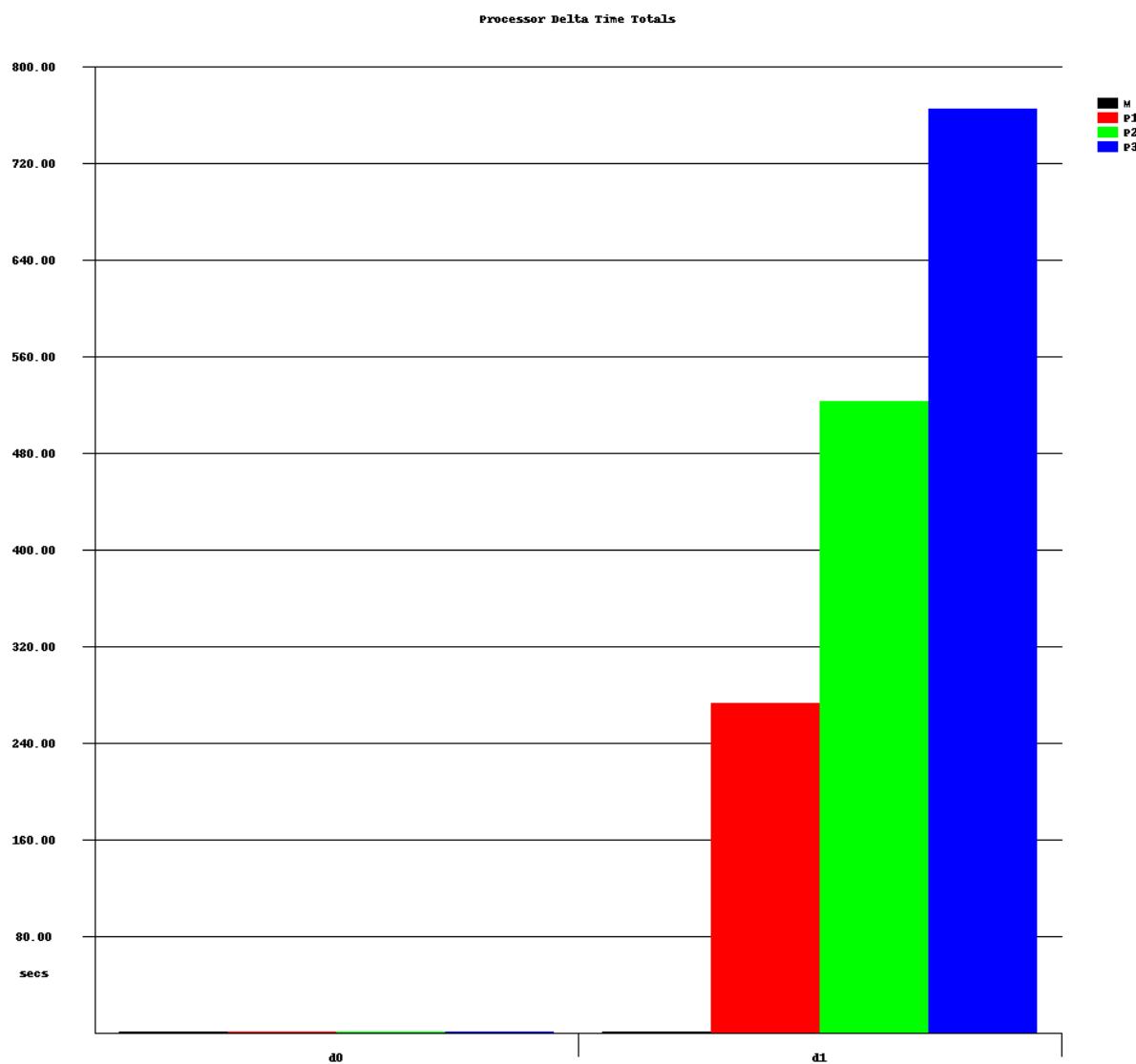
The following figures show the output from the profiler that indicate less parallelism and the design simulated with these results can benefit less from Multicore DLP.

Figure 13-4 The Processor Segment Totals Graph (graph1.html) with Less Parallelism



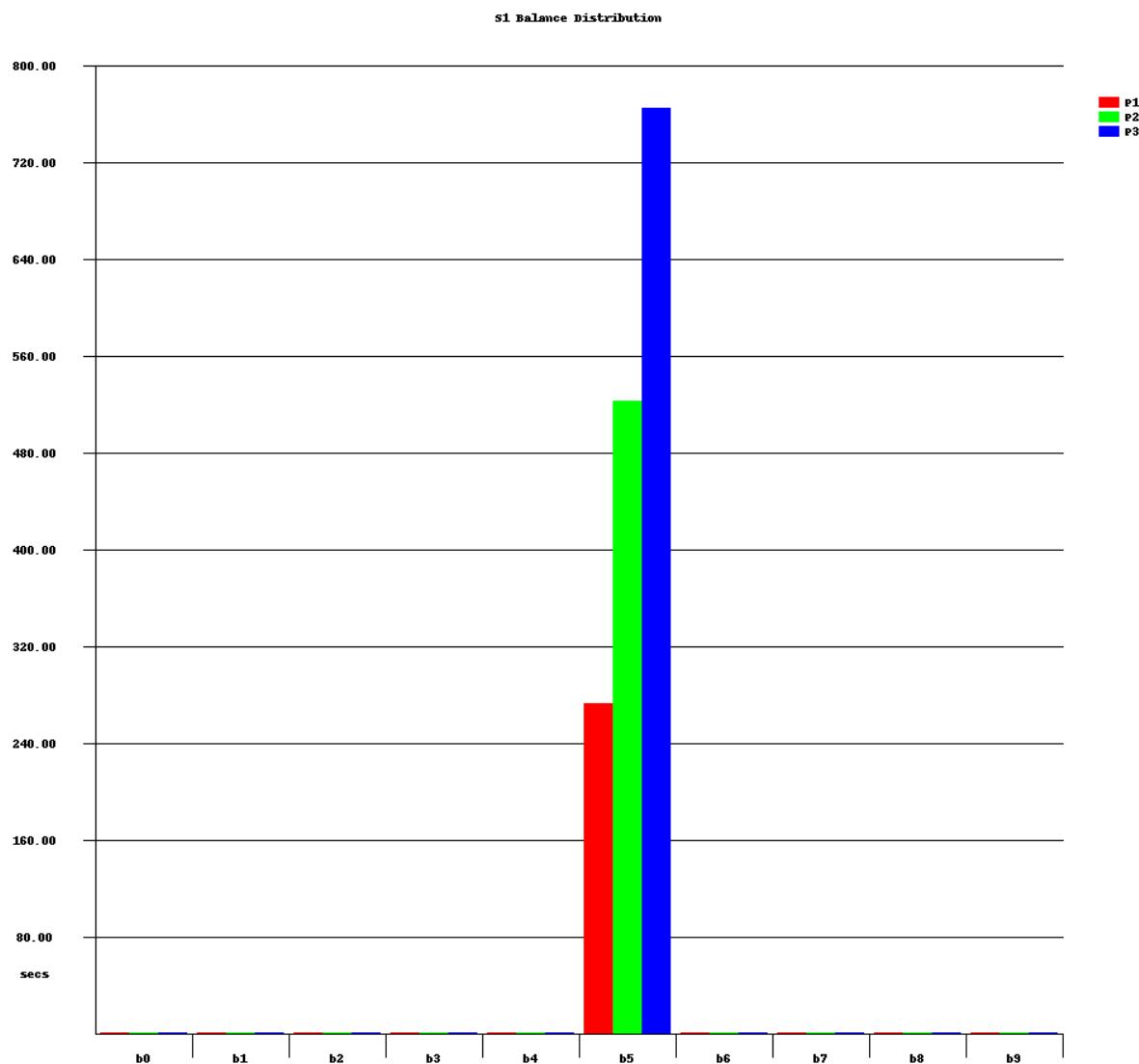
In [Figure 13-4](#) The S1 segments are of significantly different heights, showing less parallelism. Compare this graph with a corresponding one for good parallelism in [Figure 12-7](#).

Figure 13-5 The Processor Delta Time Totals Graph (graph2.html) with Less Parallelism



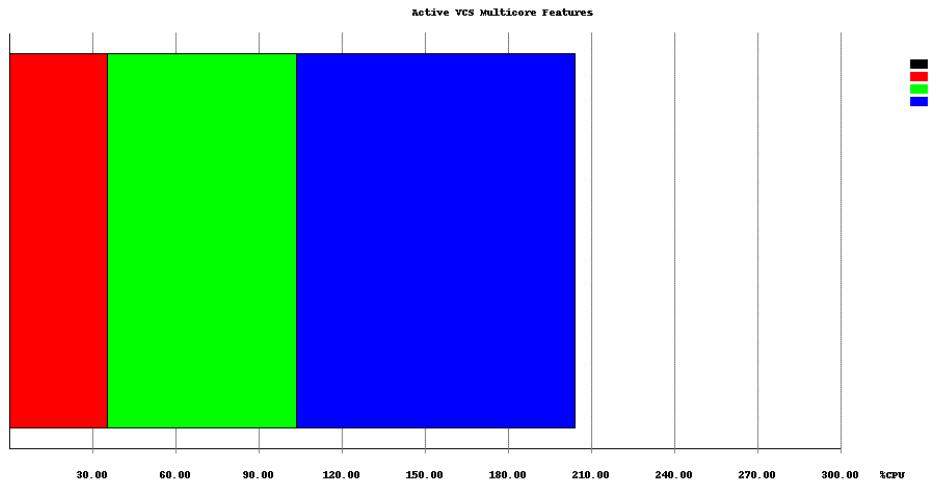
In Figure 13-5 the bars for the different slave partitions (cores) are of significantly different heights in delta d1. Compare this graph with a corresponding one for good parallelism in Figure 12-7.

Figure 13-6 S1 Balance Distribution Graph (*graph3.html*) with Less Parallelism



In [Figure 13-6](#) only bucket b5 contains significant activity, there is not much activity in the buckets on the right such as buckets b6 to b9. Compare this graph with a corresponding one for good parallelism in [Figure 12-7](#).

Figure 13-7 Active VCS Multicore Features (graph4.html) with Less Parallelism



P1 is top.A1, module A1

P2 is top.A2, module A2

P3 is top.A3, module A3

Simulation statistics

Feature	Property	Value
Core	Name of ALP dump file	simv.pdaidir/alpProfile.805.dump
Core	Simulation started at	2009-11-10 16:01:36 PST
Core	Host machine name	vgvnsvrl
Core	Host configuration: number of virtual CPUs, RAM size	16, 8184MB
Core	Global design time scale	1s
Core	Duration of simulation, wall time	780.35s
Core	Total CPU time attributed to simulation	1590.53s
Core	Ratio of wall time to CPU time	49.06%
Core	Peak virtual memory used by Master	47MB
Core	Minor page faults, major page faults, and swaps in Master	0, 118, 0
Core	Non-voluntary and voluntary context switches in Master	1756, 24006

DLP Critical Section statistics

Thread	Reason	Count [K]	TWait [s]	TExec [s]	TTotal [s]
M	OTHER	0.001	0.000	0.000	0.000

In [Figure 13-6](#) The Active VCS Multicore Features graph shows slave partitions (cores) of significantly different widths. Compare this graph with a corresponding one for good parallelism in [Figure 12-7](#).

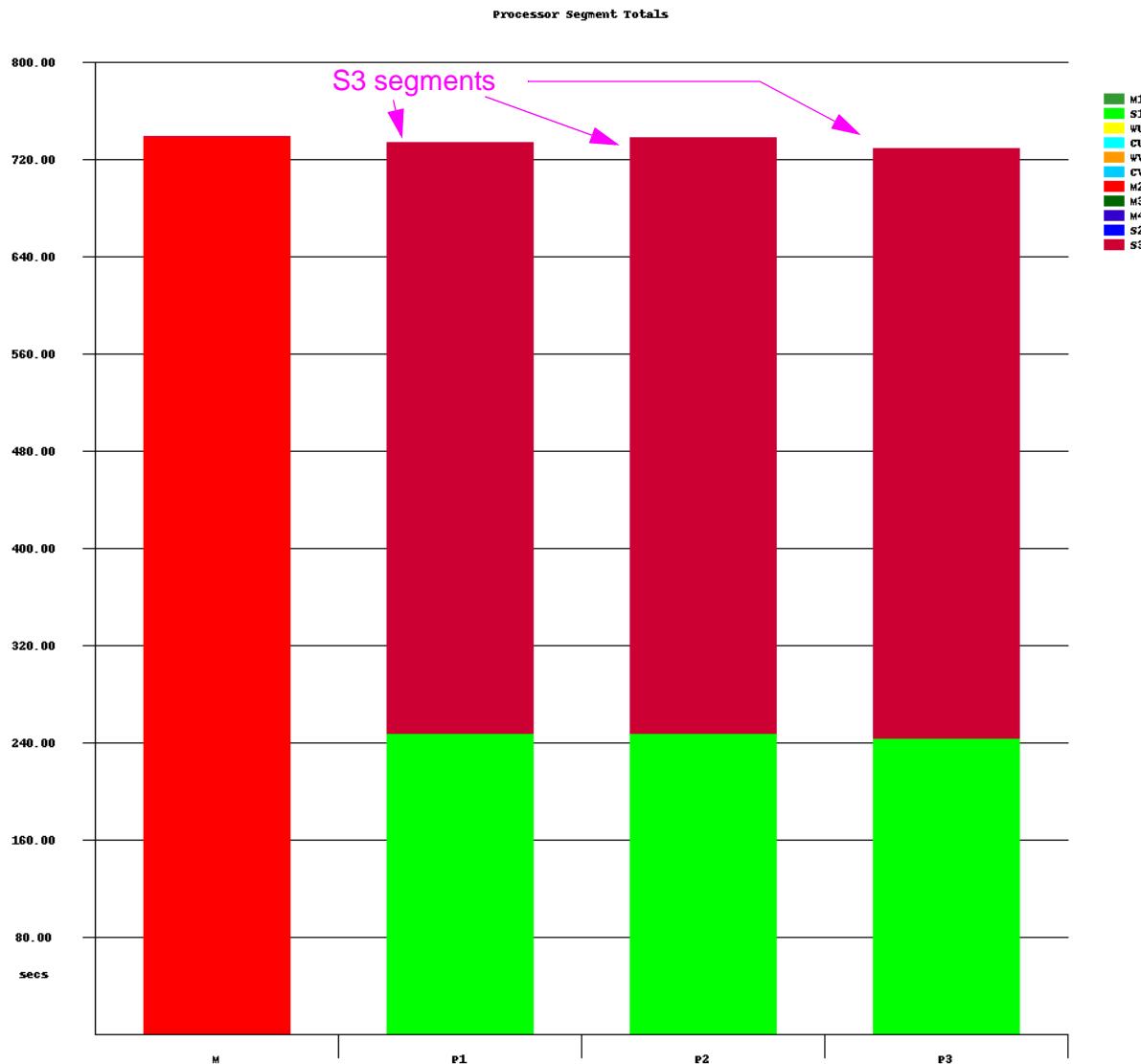
No comments table means that Multicore DLP does not have advice for this design.

Example of a Worst Case Scenario

The next four graphs are from a design in which the partitions get their clocks together. Workloads are balanced, but the partitions do now execute in parallel. This example is slower than a serial run.

[Figure 13-8](#) Shows that the partitions are doing the same amount of work, but they are spending most of their time waiting for events in other partitions.

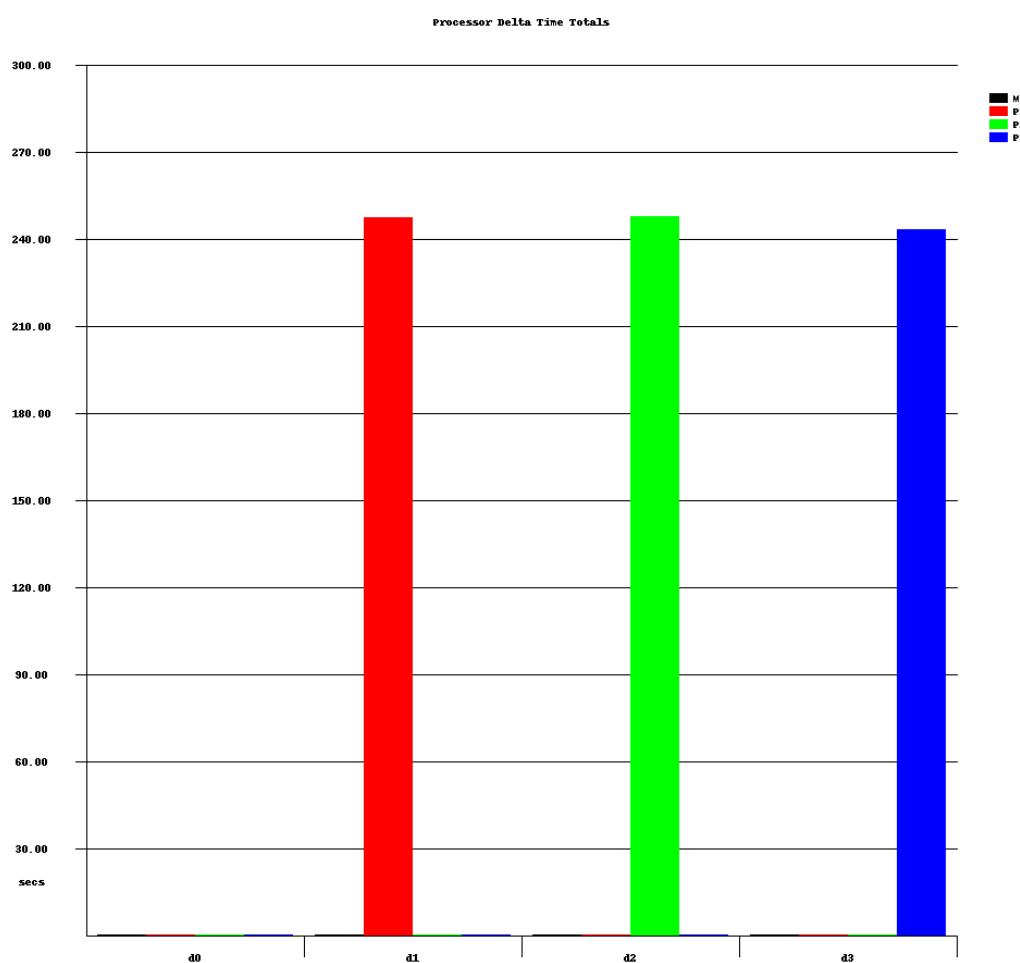
Figure 13-8 Processor Segment Totals Graph for a Worst Case Scenario



S3 segments, as stated earlier, show the time used by the slave partitions waiting for the master partition to send updated signal values. Compare this graph with the Processor Segment Totals graph for a design with good parallelism in [Figure 12-7](#).

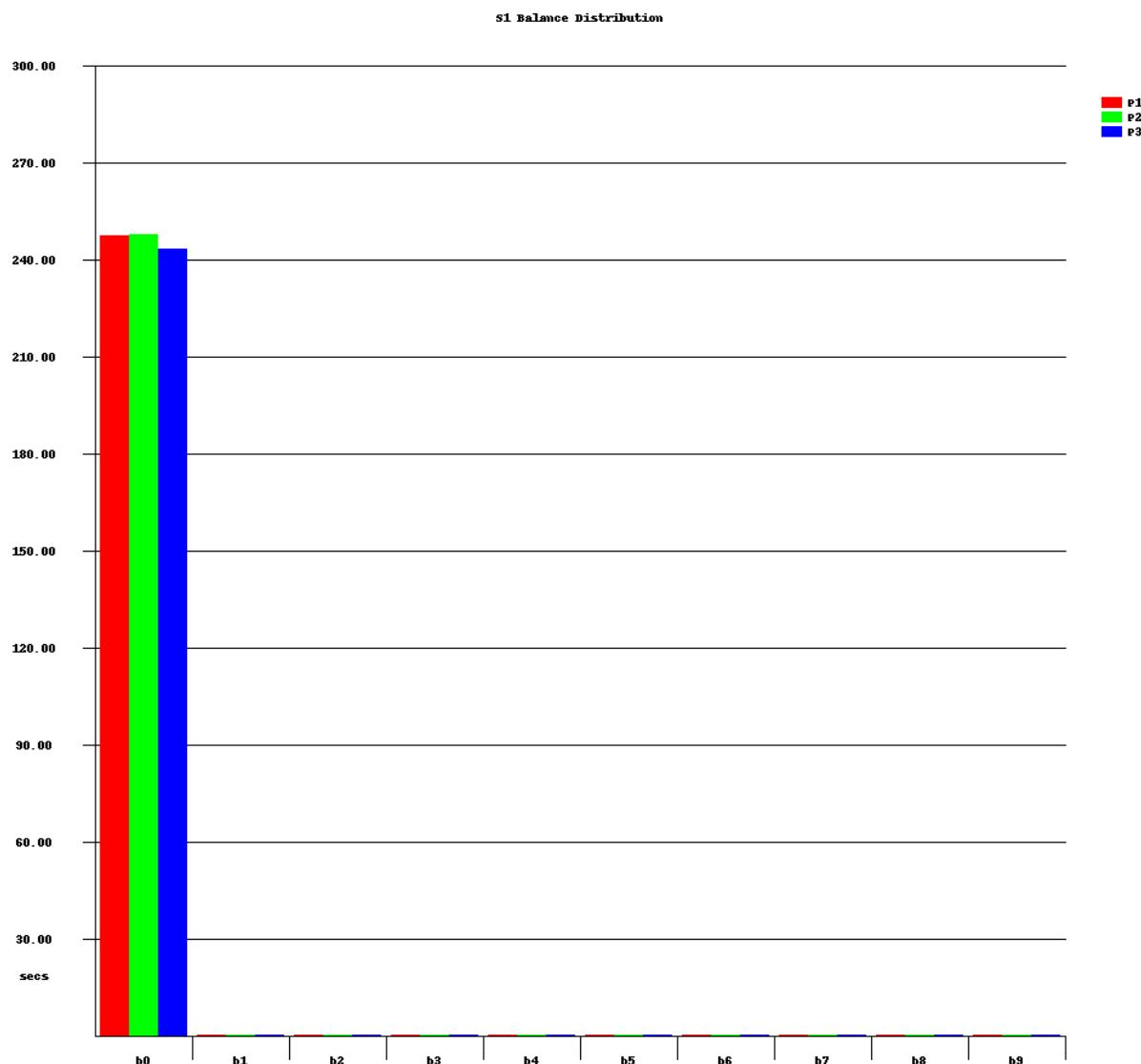
[Figure 13-9](#) shows that all the work for each partition is done in a different delta.

Figure 13-9 Processor Delta Time Totals Graph for a Worst Case Scenario



[Figure 13-10](#) shows the bars completely to the left, indicating that there is no parallelism in the design.

Figure 13-10 S1 Balance Distribution for a Worst Case Scenario



Running VCS Multicore Examples

Included in your installation is a design example and suite of scripts that allow you to perform serial / parallel performance comparisons on your system and profile the results. The tests include:

- A serial run
- A VCS Multicore run
- Profiler runs
 - A balanced workload VCS Multicore run
 - An unbalanced workload VCS Multicore run
 - A run with a delay in a partition causing one partition not to run in parallel with the other two partitions
 - A run in which a clock is passed from partition to partition causing non-parallel execution

Please contact VCS Support if you are looking for these examples.

Serial Run Time

The serial run example allows you to view wallclock time for comparison with the VCS Multicore run.

1. Run the script `run_serial.csh` .
2. Examine `DATE.serial` for the wallclock time taken for serial simulation as shown below.

```
Tue Oct 10 09:59:22 PDT 2006
```

```
Tue Oct 10 10:00:45 PDT 2006
```

VCS Multicore Run Time

The balanced VCS Multicore run example allows you to view wallclock time for comparison with the serial run.

1. Run the script `run_balanced.csh` .
2. Examine `DATE.parallel` for the wallclock time taken for VCS Multicore simulation as shown below.

```
Tue Oct 10 10:13:11 PDT 2006  
Tue Oct 10 10:13:42 PDT 2006
```

Profiler Runs

Serial Run to Identify Partitions

This is example of a serial run used to identify partitions.

1. The script for the serial profiler run is `run_serial_prof.csh` .
2. Examine the INSTANCE VIEW section in `vcs.prof` to see that instances `top.A1_inst`, `top.A2_inst` and `top.A3_inst` are candidates for partitions, as shown in [Figure 13-11](#).

Figure 13-11 Instance View from a Serial Run

=====			
INSTANCE VIEW			
Instance		%Totaltime	
top	(1)	100	
top.A1_inst	(2)	31	
top.A2_inst	(3)	29	
top.A3_inst	(4)	30	

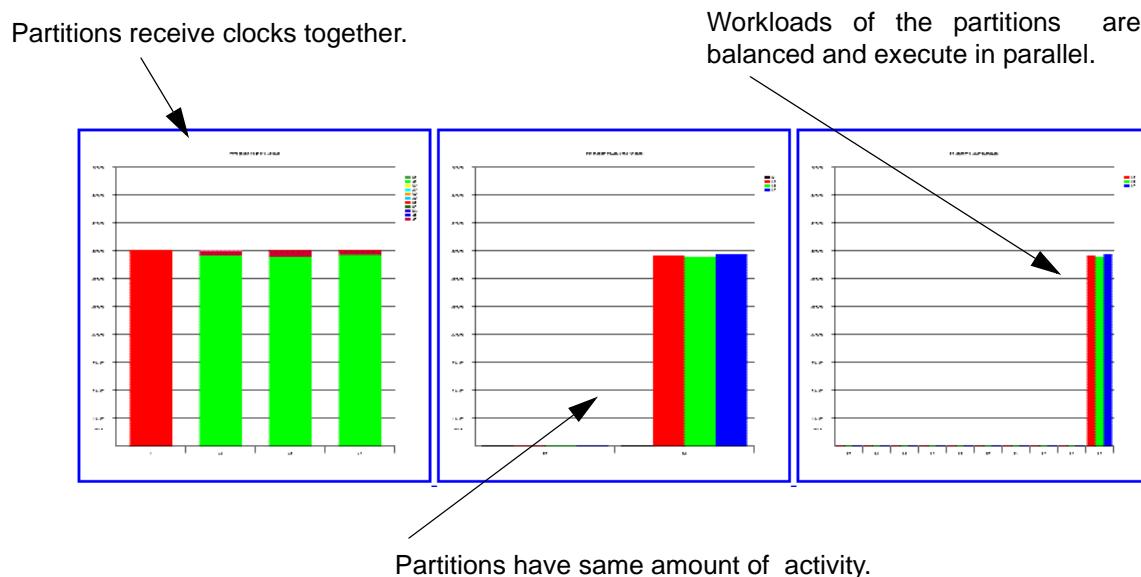
VCS Multicore Runs

+define+BALANCED

This example is the best-case scenario.

1. Run the script `run_balanced_prof.csh` .
2. Examine the output in the `ppResults_balanced` directory by opening the file `results.html` as shown in [Figure 13-12](#).

Figure 13-12 Balanced VCS Multicore Run



To view details, double-click on a graph in your browser to view the results in a full-screen.

In this example:

- On the left, the Processor Segment Totals graph shows that each partition is doing the same amount of work and the partitions receive their clocks together.

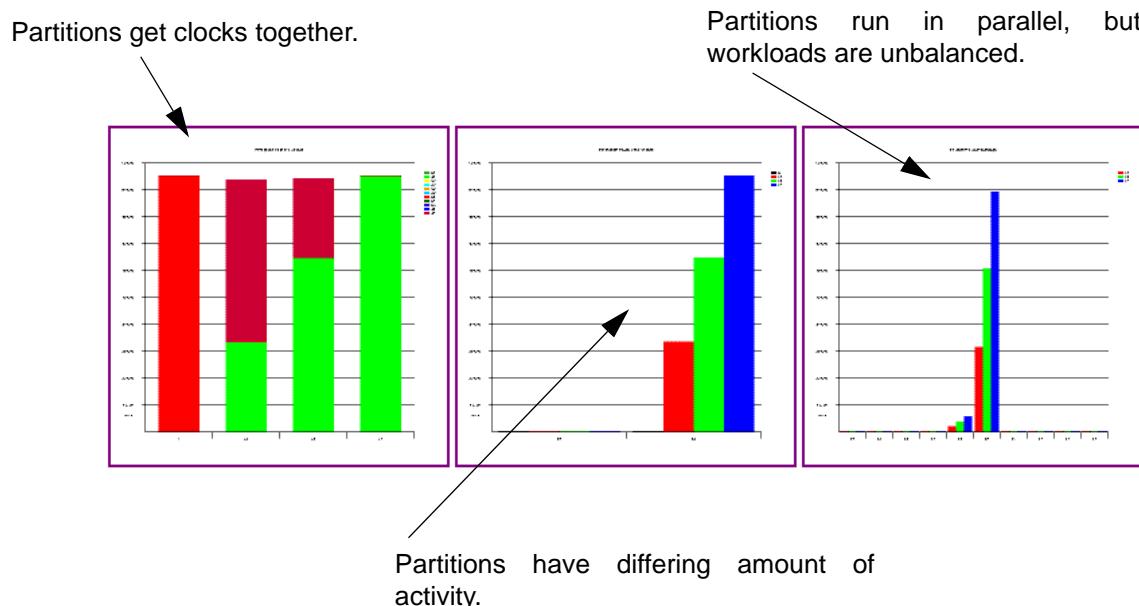
- In the center, the Processor Delta Time Totals Graph shows two deltas, δ_0 , in which the design generated the clock signal, and δ_1 , where each partition did its work in the same delta and has the same amount of activity.
- On the right, the Balance distribution graph shows that workloads of the partitions are balanced and execute in parallel.

+define+UNBALANCED

In this VCS Multicore run unbalanced workloads hinder performance.

1. Run the script `run_unbalanced_prof.csh`.
2. Examine the output is in the `ppResults_unbalanced` directory by opening the file `results.html` as shown in [Figure 13-13](#).

Figure 13-13 Unbalanced VCS Multicore Run



In this example:

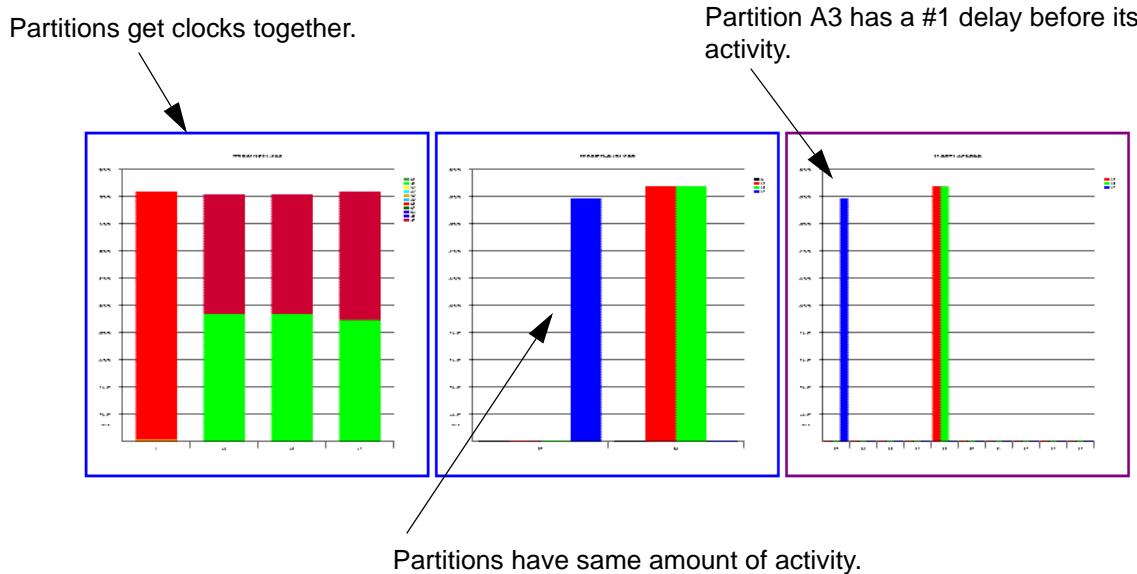
- The left chart, the Processor Segment Totals graph shows that the partitions are not doing the same amount of work. Some partitions are doing a significant amount of waiting for events in other partitions.
- The center graph, the Processor Delta Time Totals Graph, shows uneven processor delta times, indicating uneven work distribution among the partitions.
- On the right, the Balance distribution graph shows that partitions execute in parallel, but the workloads are unbalanced.

+define+BALANCED+DELAY

This is an example of a delay in a partition causing one partition not to run in parallel with the other two partitions.

1. Run the script `run_delay_prof.csh`.
2. Examine the output in the `ppResults_delay` directory by opening the file `results.html`. [Figure 13-14](#) shows Partition A3 with a #1 delay before its activity. Hence workloads are balanced, but partition A3 does not execute in parallel with A1 and A2.

Figure 13-14 A Delay in a Partition



In this example:

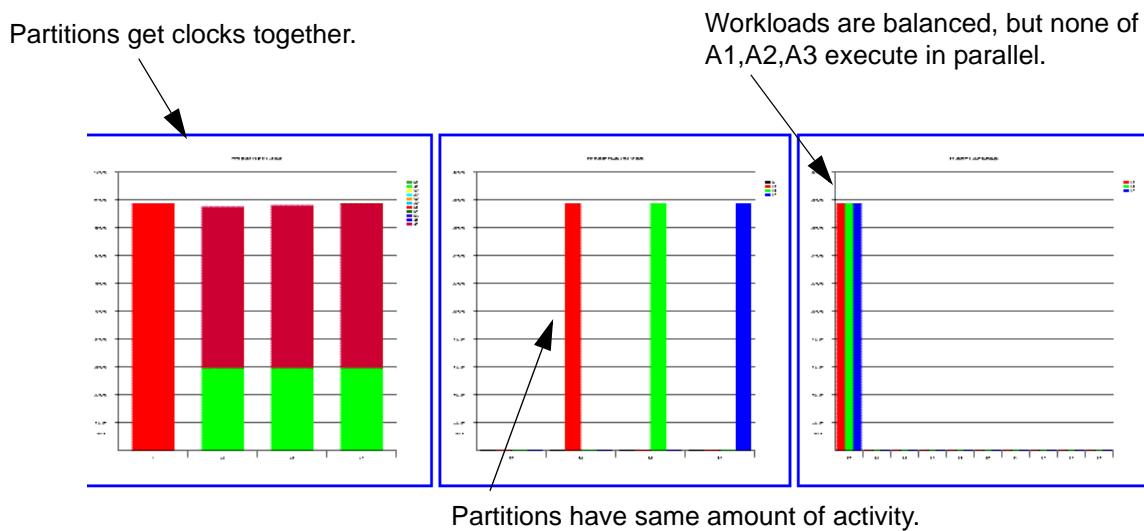
- The left chart, the Processor Segment Totals graph shows that each partition is doing the same amount of work but are spends significant amounts of time waiting for events in other partitions.
- The center graph, the Processor Delta Time Totals Graph, shows the partitions have the same amount of activity, but they execute across two deltas
- On the right, the Balance distribution graph shows that workloads of the partitions are balanced, but partition A3 does not execute in parallel with A1 and A2.

+define+BALANCED+CLOCK

This is an example of a clock being passed from partition to partition causing non-parallel execution. This is the worst-case scenario, and will be slower than the serial run.

1. Run the script `run_clock_prof.csh`.
2. Examine the output in the `ppResults_clock` directory by opening the file `results.html`. [Figure 13-15](#) shows the example in which Partition A2 gets its clock from A1, and A3 gets its clock from A2.

Figure 13-15 A Clock Passed from Partition to Partition



In this example:

- The left chart, the Processor Segment Totals graph shows that each partition is doing the same amount of work but are spending most of the time waiting for events in other partitions.
- The center graph, the Processor Delta Time Totals Graph, shows the partitions have the same amount of activity, but the work for each partition is done in a different delta.
- On the right, the Balance distribution graph shows that workloads of the partitions are balanced, but none of the three partitions execute in parallel.

Multicore DLP Autopartitioning

If your design is suitable for multicore simulation, VCS can automatically generate a Multicore DLP partition file when you enter the `-parallel+autopart=N` compile-time option, where N is the number of cores (or Multicore DLP slave partitions) in your design. The value of N should be at least 2.

The partition file VCS writes is named `autopart.cfg` and VCS writes the file in the directory in which you entered the `vcs` command.

The purpose of autopartitioning is to quickly show you the advantage of using Multicore DLP without your first writing a Multicore DLP partition file for your design. Autopartitioning is not intended as an improvement on a manually written partition file.

The specified number of cores and slave partitions, N , is the maximum number of slave partitions autopartitioning writes in the `autopart.cfg` file. Autopartitioning might write fewer slave partitions in this file in some cases, and a slave partition might have multiple instances in the file.

If autopartitioning finds that it cannot write the `autopart.cfg` file, it writes a file named `autopartFail.txt` explaining why it can't write the `autopart.cfg` file.

Each time you enter the `-parallel+autopart=N` compile-time option, autopartitioning writes a new `autopart.cfg` file. If an `autopart.cfg` file already exists in the current directory, autopartitioning overwrites it, so for example, if you are experimenting with different values of N , save the previous `autopart.cfg` file with a different name or location.

The `autopart.cfg` file can also be entered in the regular Multicore DLP flow, for example:

```
vcs -parallel+design=autopart.cfg -lca other options and arguments
```

SystemVerilog dynamic data types are not supported in the slave partitions. autopartitioning attempts to avoid these data types in its generated slave partitions.

Limitations

SystemVerilog constructs such as classes, covergroups, associative and dynamic arrays, strings and events, throughout the design will probably prevent autopartitioning from generating a valid `autopart.cfg` file because these constructs are not supported by Multicore DLP.

Also the limitations on Multicore DLP remain with DLP autopartitioning.

Supported Platforms

- RHEL32 and RHEL64 32/64bit RH4.0: Multi-core multi-processor machine.
- Solaris 32 bit: Multi-core Multi-processor machine.

Current Limitations

The limitations of Multicore ALP or DLP are as follows:

- SystemVerilog constructs, such as, but are not limited to, classes, covergroups, associative and dynamic arrays, strings and events have the following limitations in Multicore DLP:
 - Their use throughout the design can prevent autopartitioning from generating a valid autopart.cfg file.
 - Their use is prohibited in cores that are slave partitions.

SystemVerilog covergroups, however, are supported by Multicore DLP but only if you also run Multicore ALP with a core (or a consumer) for Multicore functional coverage (PFC) for these covergroups.

The SystemVerilog data types `logic` and `bit` are supported throughout the design with Multicore ALP and DLP.

- Multicore ALP Parallel SAIF is not supported with VCS Multicore DLP.
- The unified profiler (`-simprofile`), an LCA feature, is not supported in Multicore ALP or DLP.
- SystemC on top is not supported for VCS Multicore DLP.
- The UCLI dump command with `-add` and `-ports` is not supported for Multicore ALP parallel VPD file dumping.
- Multicore ALP parallel VPD file dumping does not support the `$vcdplusmsglog` or `$vcdplustblog` system tasks.
- The dynamic race detection tool (`-race` and `-racecd`) is not supported with Multicore DLP.
- Saving and Restarting the simulation with checkpoint files (`$save` system task) is not supported with Multicore DLP.

- For SystemC cosimulation the following are not supported in Multicore DLP:
 - partial build with shared libraries
 - partial build with object files
 - SystemC and C/C++ cosimulation
- VMC is not supported with Multicore DLP.
- Analog mixed-signal simulation is not supported with Multicore DLP.
- For VCS MX, only Verilog or SystemVerilog code can be in cores that are slave partitions.

14

New SystemVerilog Features

The new SystemVerilog features are as follows:

- “Extern Task and Function Calls through Virtual Interfaces”
- “Enhancements to the -xlrn uniq_prior_final Compile-Time Option” on page 342

Extern Task and Function Calls through Virtual Interfaces

You can define tasks and functions in an interface with one or more of the modules connected by the interface. You declare them as export in a modport or as extern in the interface. When they are called through virtual interfaces, the actual task or function that VCS executes depends on the interface instance of the virtual interface.

Example of exporting tasks in modports

```
interface simple_bus ; // Define the interface
modport slave (export task Read) ;
endinterface: simple_bus

module memMod (simple_bus sb_intf);
task sb_intf.Read; // Read method
...
endtask
endmodule

module top;
simple_bus sb_intf(); // Instantiate the interface
memMod mem(sb_intf.slave); // exports the Read tasks
endmodule
```

Example of extern tasks in interfaces

```
interface intf;
extern task T1();
extern task T2();
endinterface

module top;
intf i1();
intf i2();

virtual intf vi;
M1 m1(i1);
M2 m2(i1);
M3 m3(i2);
M4 m4(i2);

initial begin
    vi = i1;
    vi.T1(); // Task i1.T1 in M1
    vi.T2(); // Task i1.T2 in M2
    vi = i2;
    vi.T1(); // Task i2.T1 in M3
    vi.T2(); // Task i2.T2 in M4
end
endmodule
```

```

module M1(intf i1);
task i1.T1;
...
endtask
endmodule

module M2(intf i1);
task i1.T2;
...
endtask
endmodule

module M3(intf i2);
task i2.T1;
...
endtask
endmodule

module M4(intf i2);
task i2.T2;
...
endtask
endmodule

```

The definition of extern subroutines within an interface shall observe the following rules:

- Each interface instance may have different implementations of its **extern** subroutines
 - The same **extern** subroutine of different interface instances can be defined in different modules
 - Different **extern** subroutines of the same interface instance can be defined in different modules
- Every interface instance must have one and only definition of its **extern** subroutines

- If an interface instance containing an **extern** subroutine, one of the modules connected must define that subroutine
- Any **extern** subroutine of an interface instance cannot be defined in more than one module
- The module implementing any **extern** subroutine can be instantiated only once
- These rules apply for exported subroutines in modports as well.

Limitations

- Extern task and function calls through virtual interface are not supported in separate compile flow
- Extern task and function calls through virtual interface are not supported in constraints
- The interface containing an extern task or function can only be passed as port to module and program scopes. It cannot be instantiated inside module defining the extern task/function of interface and doing so will give an error.

Enhancements to the -xIrm uniq_prior_final Compile-Time Option

The `-xIrm uniq_prior_final` compile-time option and keyword argument tell you when:

- a case statement is modified by the `unique`, `unique0`, or `priority` keywords and there is a case item expression that is not unique
- a conditional `if` statement is modified by the `unique`, `unique0`, or `priority` keywords and both of the following:

- the conditional expression is not met
- there is no `else` statement for the `if` statement

When these conditions happen VCS or VCS MX issues a violation report.

The violation reports, starting with the F-2011.12 release and now the G-2012.09 release, contain more information, specifically:

- The hierarchical name of the module instance that contains the `case` or `if` statement as described above, or contains a call to a user-defined task or function that contains these statements.
- The value of the evaluation expression which must be true for a `for` loop statement to continue to iterate.

Also in these releases a violation report occurs when both of the following occur:

- there is a call to a function that contains the `case` or `if` statements as described above
- the function called is a possible assignment expression for the conditional `? :` operator or in the RHS expression in a continuous assignment.

The following code examples illustrate these enhancements.

Example 14-1 unique case Statement in Multiple Module Instances

```
function foo (input a);
    unique case (a)
        1'b1: ; ← case item expressions
        1'b1: ; ← are not unique
    endcase
endfunction

module T (input wire a);
    always_comb
        foo(a); ← module T calls the function
    endmodule

module Top;
    reg a;
    T t1(a);
    T t2(a); ← multiple instances of module T

    initial
    begin
        a = 1'b1;
    end

    endmodule
```

The violation reports in versions F-2011.12 and G-2012.09 refers to the hierarchical names of the instances:

```
RT Warning: More than one conditions match in 'unique case' statement.
          "doc_ex1.sv", line 2, for \$unit ::foo.
          Line      3 &      4 are overlapping at time      0s.
#0 in foo      at doc_ex1.sv:2
#1 in Top.t2      at doc_ex1.sv:10

RT Warning: More than one conditions match in 'unique case' statement.
          "doc_ex1.sv", line 2, for \$unit ::foo.
          Line      3 &      4 are overlapping at time      0s.
#0 in foo      at doc_ex1.sv:2
```

```
#1 in Top.t1      at doc_ex1.sv:10
```

VCS and VCS MX identify both instances and they output two violation reports.

The violation report in version E-2011.03 refers to the module name instead of the module instance hierarchical name:

```
RT Warning: More than one conditions match in 'unique case' statement.
```

```
    "doc_ex1.sv", line 2, for \$unit ::foo.  
    Line      3 &      4 are overlapping at time      0.  
#0 in foo      at doc_ex1.sv:2  
#1 in T  at doc_ex1.sv:10
```

There is only one violation report.

Example 14-2 for Loop Statement Iteratively Calling a Function with a unique case Statement

```
function foo (input a);
    unique case (a)
        1'b1:;
        1'b1;; ← case item expressions
    endcase
endfunction

module T (input wire a);
    int P;
    always_comb
        for (int i=0;i<P;i++)
            foo(a); ← for loop that iteratively
    initial begin                                calls the function
        P = 2;
    end
endmodule

module Top;
    reg a;
    T t1(a);
    initial begin
        a = 1'b1;
    end
endmodule
```

In the for loop the value of i must be less than p. There are violation reports because the unique case statement does not have unique case item expressions.

In versions F-2011.12 and G-2012.09 the violation reports include the value of i in the iteration.

```
RT Warning: More than one conditions match in 'unique case'
statement.
    "loop_expr.sv", line 2, for \$unit ::foo.
    Line      3 &      4 are overlapping at time      0s.
#0 in foo          at loop_expr.sv:2
```

```

#1 in Top.t1      at loop_expr.sv:12
#2 in loop with i= 0      at loop_expr.sv:11

RT Warning: More than one conditions match in 'unique case'
statement.
    "loop_expr.sv", line 2, for \$unit ::foo.
    Line      3 &      4 are overlapping at time      0s.
#0 in foo      at loop_expr.sv:2
#1 in Top.t1      at loop_expr.sv:12
#2 in loop with i= 1      at loop_expr.sv:11

```

In version E-2011.03 the value of the expression that must be true for continued iteration of the `for` loop, `i` in this example, is not in the violation report:

```

RT Warning: More than one conditions match in 'unique case'
statement.
    "loop_expr.sv", line 2, for \$unit ::foo.
    Line      3 &      4 are overlapping at time      0.
#0 in foo      at loop_expr.sv:2
#1 in T  at loop_expr.sv:12

```

Example 14-3 unique case Statement in a Function Call in a Possible Assignment Expression with the Conditional Operator

```
function foo (input a);
    unique case (a)
        1'b1: ;
        1'b1: ;
    endcase
endfunction

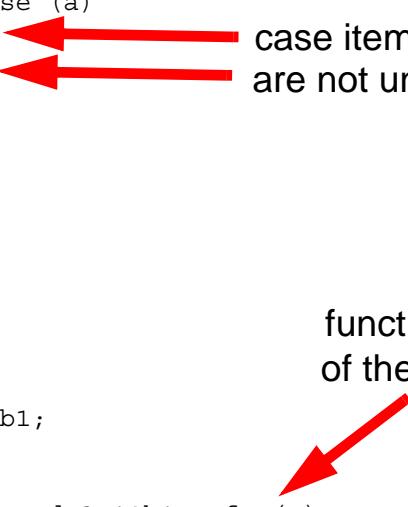
module Top;
    reg a, b;
    reg cond;

initial
begin
    a = 1'b1;
    cond = 1'b1;
end

assign b = cond ? 1'b1 : foo(a);

endmodule
```

function call is an assignment expression
of the conditional operator



In versions F-2011.12 and G-2012.09 there is a violation report:

```
RT Warning: More than one conditions match in 'unique case'
statement.
    "cond_op.sv", line 2, for \$unit ::foo.
    Line      3 &      4 are overlapping at time      0s.
#0 in foo      at cond_op.sv:2
#1 in Top      at cond_op.sv:18
```

In version E-2011.03 there is no violation report.

Limitations

This enhancement was done only for unique/priority case

- unique0 is not yet supported

unique/priority with if is not covered by this enhancement.

15

Assertion Features

This chapter contains the following features:

- “SystemVerilog Assertions” on page 350
- “Using SystemVerilog Constructs Inside vunits” on page 352
- “Using Fail-only Assertion Evaluation Mode” on page 354
- “Signature-based Control for Deferred Assertions and RT Checks” on page 355

SystemVerilog Assertions

SVA Extensions

This section describes VCS extensions to SystemVerilog Assertions that are not standard LRM features.

Functions for Counting Unknown Bits

Use the following system functions to count the number of X's and/or Z's in a vector.

- `$countx (expression)` returns the number of expression bits set to X.
- `$countz (expression)` returns the number of expression bits set to Z.
- `$countunknown (expression)` returns the number of expression bits set to either X or Z.
- `$onedriven` returns true if only one bit of the expression is not Z, and its value is defined (not X).
- `$onedriven0` returns true if at most one bit of the expression is not Z, and if such a bit exists, its value is defined (not X).

Limitations

This section describes the limitations.

Debug Support for New Constructs

Use `-assert dve` at compile/elab to enable debug for assertions. While basic debug support is available with this release, assertion tracing in DVE not supported completely. DVE provides information such as: `start_time`, `end_time` for every attempt and statistics for every assertion/cover. DVE also groups all signals involved in an assertion on tracing an attempt. However the extra "hints" that are provided for SVA constructs are not available for new constructs as of now.

UCLI support for new assertions is not fully qualified.

Note on Cross Features

Some of the features in new assertions have known limitations with cross feature support, such as Debug, Coverage. Please check with Synopsys support if there are unexpected results with cross feature behavior for these new constructs.

Some known issues:

- `-cm property_path` is not available for the new constructs
- New sequence operators when used as sampling event for covergroups may not function well.

Using SystemVerilog Constructs Inside vunits

VCS supports using SystemVerilog (SV) and SystemVerilog Assertions (SVA) inside a Property Specification Language (PSL) verification unit (vunit). This feature:

- Allows SV or SVA code inside a vunit.
- Allows you to more easily bind checkers containing assertions and modeling code in a vunit to a design.

Use the `-assert svvunit` compile-time option, as shown in the following example, to enable this feature. You can specify this option with the `vcs` or `vlogan` command.

```
% vcs -assert svvunit <filename.v> <design_filename.v> \  
<vunit_filename.psl>
```

or

```
% vlogan -assert svvunit <filename.v> <design_filename.v> \  
<vunit_filename.psl>
```

where,

`<vunit_filename.psl>` — PSL vunit that contains SV or SVA code. For example:

```
% vcs -assert svvunit test.v design.v vunit_checker.psl
```

SystemVerilog Constructs Inside Vunits Limitations

- Inheritance of vunits is not supported.
- Vunit binding is supported only for modules.
- In the above use model, you cannot specify PSL constructs in any vunit specified in the same vlogan command. You must separate the PSL vunits from SV vunits and use them in two separate compilations, as shown in the following examples:

```
% vlogan vunit_psl.psl design.v  
% vlogan -assert svvunit vunit_sv.psl test.v
```

where,

- vunit_psl.psl is a vunit that contains PSL code
- vunit_sv.psl is a vunit that contains SV or SVA code.

Using Fail-only Assertion Evaluation Mode

Fail-only is a new assertion evaluation mode by which VCS provides an optional optimization controlled by the `-assert failonly` or `-assert concfail` compile-time options. Using these options improves runtime assertion performance.

Immediate/deferred assertions and concurrent assertions without pass action blocks, local variables, match operators, or multiple clocks tend to benefit from this evaluation mode.

Use the compile-time options shown in [Table 15-1](#) to enable fail-only mode.

Table 15-1 Fail-only Assertion Compilation Options

Option	Description
<code>-assert failonly</code>	Enables fail-only mode for concurrent assertions.
<code>-assert concfail</code>	Enables fail-only mode for both immediate/deferred and concurrent assertions.

Fail-only Assertion Limitations

Following are the limitations of this feature:

- No offending expression reported upon failure
- No success reporting (including vacuous success)
- No VPI callback on success (including vacuous success)
- No attempt start-time reporting

Signature-based Control for Deferred Assertions and RT Checks

VCS now provides a way to control RT and deferred assertions (labelled assertions) using the “signature” method.

From release G-2012.09 onwards, VCS generates a “signature” associated with each error/warning report related to labelled deferred assertion and RT checks.

A signature is a string made up of tokens extracted from the function call stack trace of assertions. The tokens are separated by dots. The signature is not an XMR, but an XMR-like string which is not restricted to legal Verilog syntax.

Each signature uniquely determines a path from where the error/warning occurs, as shown in the following example. This feature helps you to locate the exact place where the assert failure occurs. You can use signature to control assertions and RT checks. The following section describes how to use signature to achieve desired control.

Example

Consider the following example:

Example 15-1 Function call stack trace

```
// Currently there is no legal syntax in Verilog to disable  
the assertion "f.L"  
function f(input a, b);  
    L: assert final(a > b);  
endfunction // function is defined in $unit, used by many  
           callers as  
           // intellectual property
```

```

module top;
    reg a, b, c, d;
    always_comb begin : block1
        f(a, b);
    end
    always_comb begin : block2
        f(c, d);
    end
endmodule

```

The following output contains the signature (highlighted in bold) for function call stack trace:

```

"tel.v", 3: \$unit ::f.L: started at 0s failed at 0s
Offending '(a > b)'
#0 in f.L at tel.v:3
#1 in top.block1 at tel.v:10
[Warning/Error Signature: "top.block1.f.L"]

"tel.v", 3: \$unit ::f.L: started at 0s failed at 0s
Offending '(a > b)'
#0 in f.L at tel.v:3
#1 in top.block2 at tel.v:13
[Warning/Error Signature: "top.block2.f.L"]

```

Signature-based Control

Existing assertion control tasks, including `$assertoff()` and `$asserton()`, accept signature as an argument to perform the control. The following topics illustrate how to use signature to control the assertions in various ways:

- “[Full-signature control \(single control\)](#)”
- “[Partial-signature control \(group control\)](#)”
- “[Static Configuration File](#)”
- “[Runtime control](#)”

Full-signature control (single control)

In [Example 15-2](#), VCS provides all assertion failures and corresponding signatures for them, as shown in “[Output: run.log.ref](#)” :

Example 15-2 Full-signature Control

```
reg a;
function reg foo (reg a);
    for(int i=1;i<2;i++) begin:BL
        for(int j=-1;j<1;j++) begin
            A1 : assert final (a !== 1'b1);
        end
    end
    foo = 1;
endfunction
module Top;
    reg b;
    reg cond;
initial begin
    // here, the signature of each path is presented
    a = 1'b1;
    #0;
end
always_comb begin : AL
    for(int i=0;i<1;i++) begin:BL1
        foo(a);
    end
    for(int i=1;i<2;i++) begin:BL2
        foo(a);
    end
end
endmodule
```

Output: run.log.ref

```
"cts2.v", 5: \$unit ::foo.BL.A1: started at 0s failed at 0s
Offending '(a !== 1'b1)'
#0 in foo.BL.A1 at cts2.v:5
#1 in loop with j=-1 at cts2.v:4
```

```

#2 in loop with i= 1 at cts2.v:3
#3 in Top.AL.BL1 at cts2.v:26
#4 in loop with i= 0 at cts2.v:25
[Warning/Error Signature:
"Top.AL.BL1[0].foo.BL[1].GenLoop_j_0[-1].A1"]

"cts2.v", 5: \$unit ::foo.BL.A1: started at 0s failed at 0s
Offending '(a !== 1'b1)'
#0 in foo.BL.A1 at cts2.v:5
#1 in loop with j= 0 at cts2.v:4
#2 in loop with i= 1 at cts2.v:3
#3 in Top.AL.BL1 at cts2.v:26
#4 in loop with i= 0 at cts2.v:25
[Warning/Error Signature:
"Top.AL.BL1[0].foo.BL[1].GenLoop_j_0[0].A1"]

```

You can pass the complete signature produced by VCS in a previous run to turn ON/OFF a particular assertion failure corresponding to that signature.

For example:

```
$assertoff(0, "Top.AL.BL1[0].foo.BL[1].GenLoop_j_0[-1].A1");
```

Partial-signature control (group control)

Full-signature can only control assertion failure in one particular path at a time. You can use partial signature, if you want to turn ON/OFF a set of assertion failures.

Scope-level control

When a partial signature ending with a scope name (including a block or function and so on) is passed to assert control tasks, assertion failures rooted from that scope are enabled/disabled.

```
$assertoff(0, "Top.AL");
```

All failures in [Example 15-2](#) will be shut off.

Wildcard character control

This feature supports “*” wildcard character in signature string. The “*” wildcard character represents zero or many tokens in the signature.

For example, the string Top . AL . * [0] . A1 will match the signatures (the matched part of “*” is highlighted in bold):

```
Top . AL . BL1 [0] . foo . BL [1] . GenLoop_j_0 [0] . A1  
Top . AL . BL2 [1] . foo . BL [1] . GenLoop_j_0 [0] . A1
```

If you apply the above wildcard character by specifying the following system tasks:

```
$assertoff(0, "Top . AL . * [0] . A1");
```

Instance-level control

You can continue to perform instance level control, using different depth levels. The behavior is compatible with VCS releases prior to G-2012.09, with the below exception:

Control task	Effect
\$assertoff(0, "Top")	All assertions under “Top” will be shut off (including all those from grand children instances).
\$assertoff(1, "Top")	Assertions directly under “Top” will be shut off. Others remain ON.
\$assertoff(2, "Top")	Assertions under “Top”, “Top . L1” will be shut off. Others (in “Top . L1 . L0”) remain ON.
\$assertoff(-1, "Top")	Invalid level, generates error message.

Note:

Even for those assertion failures that do not have signatures generated, due to one of the reasons listed in “[Contexts where no Signature is Generated](#)”, they can still be turned on or off by instance-level control. This is to match legacy semantics.

Static Configuration File

The signature can be used as an argument for static control. You can use the `signature` option with the signature string, as shown in the following example, to define signature in static configuration file.

Example 15-3 Specifying signature in static configuration file

```
//----sig.cfg-----  
+signature M.AL.GenLoop_i_0[0].foo.loopA[0].U1  
-signature Top.AL.*.A1  
//-----  
  
+signature
```

Enables all assertions represented by the signature.

```
-signature
```

Disables all assertions represented by the signature.

Since there is no presence of static file configuration for RT checks, you can use the `-assert hier_uniq=` option, as described in [Table 15-2](#).

The configuration file (`sig.cfg` in [Example 15-3](#)) can be used to control either assertions or RT checks, at compile time or at runtime.

Note:

`-assert hier` and `-assert hier_uniq` runtime options require `-assert enable_hier` at compile time.

Table 15-2 The -assert hier_uniq= option commands

Command	Effect
vcs -assert hier_uniq=sig.cfg	Controls all applicable RT checks at compile-time.
vcs -assert hier=sig.cfg	Controls all applicable assertions at compile-time.
simv -assert hier_uniq=sig.cfg	Controls all applicable RT checks at runtime.
simv -assert hier=sig.cfg	Controls all applicable assertions at runtime.

If you want to control assertions and RT checks uniformly, the same configuration file can be passed to different switches, as shown below:

```
vcs -assert hier_uniq=sig.cfg -assert hier=sig.cfg
```

In this way, all assertions/RT checks whose signatures match those defined in configuration file will be disabled/enabled.

Note:

- All existing syntaxes, such as -assert, -tree, and so on will continue to work as before.
- You can use the -signature option in the existing file along with legacy -assert, -tree.

Runtime control

You can use the `call` command, as shown below, to call the signature-based control at runtime.

```
call {$assertoff(0, "Top.AL.BL1[0].foo.BL[1].GenLoop_j_0 [-1].A1")};
```

All types of control (single or group) presented in sections “[Partial-signature control \(group control\)](#)” and “[Static Configuration File](#)” will work using the call command.

Limitations

- Partition Compile and XPROP are not supported.
- If you pass an invalid signature to assert control tasks, then there will be no warnings or errors reported. Such tasks will be simply ignored. Therefore, you must pass the VCS generated signatures to ensure the intended behavior.

Contexts where no Signature is Generated

The signature is introduced to locate the exact place where the assert failure occurs. Each signature corresponds to a real hardware unit when the design is synthesized. If an assertion is not placed in a synthesizable code, or if it is not identifiable, no signature is generated. As a result, such assertions cannot be controlled using signatures. [Example 15-4](#), [Example 15-5](#), and [Example 15-6](#) show contexts where signatures will not be generated.

Example 15-4 Assert function inside always_comb

```
always_comb begin : AL
    assert final (0); //No Label
end
```

Example 15-5 for-loop statement is inside always_comb

```
always_comb begin // Not Named
    for(int i=0;i<1;i++)
        A1: assert final (0);
end
```

Example 15-6 for-loop statement is inside initial

```
initial begin :INI // Behavioral
    for(int i=0;i<1;i++)
        A1: assert final (0);
end
```

16

SystemC Standard Library Classes

The chapter explains the SystemC Standard Library classes in the following major sections:

- “Overview” on page 366
- “Purpose” on page 366
- “Brief Introduction to VMM-SC Base Class Library” on page 367
- “Use Model” on page 391
- “Using VMM-SC in SystemC Environment” on page 392
- “Using VMM-SC With VMM-SV Interoperability” on page 408

Overview

This chapter describes the availability of the VMM-SystemC library (VMM-SC) with VCS E-2011.03 and later versions. It consists of a set of base classes equivalent to the VMM library in SystemVerilog (SV). It also describes the use model for the VMM-SC based verification methodology and how you can use this methodology with VMM SystemVerilog (VMM-SV).

The VMM-SC implementation is based on VMM D-2010.06 and later versions of VMM-SV. VMM-SC does not implement all the classes and utilities that are provided with VMM-SV. The supported classes and utilities are discussed in detail in this chapter.

Purpose

The purpose of this document is to familiarize you with the VMM-SC class library. The following section provides a brief introduction to VMM-SC base classes, followed by the different use model for VMM-SystemC.

The objective of VMM-SC is to provide a unified reference verification methodology in conjunction with VMM-SV for system-level verification to RTL verification. The system-level environments primarily use the SystemC/C-based models.

You use VMM-SC to develop a verification environment at the system level and also to verify the system on a chip (SoC) at the RTL level with VMM-SV effectively. For details, see the use model section.

Brief Introduction to VMM-SC Base Class Library

The summary of all the available methods linked to their respective detailed documentation page at the beginning of each class specification is provided:

- “[vmm_unit](#)” on page 368
- “[vmm_group](#)” on page 370
- “[vmm_xactor](#)” on page 371
- “[vmm_subenv](#)” on page 373
- “[vmm_timeline](#)” on page 374
- “[vmm_simulation](#)” on page 376
- “[vmm_test](#)” on page 377
- “[vmm_data](#)” on page 378
- “[vmm_object](#)” on page 383
- “[vmm_log](#)” on page 385
- “[vmm_opts](#)” on page 387
- “[vmm_factory](#)” on page 390
- “[vmm_interop](#)” on page 391

vmm_unit

Description

This class is used as the base class that provides predefined simulation phases to structural elements, transactors, and transaction-level models. The purpose of this class is to,

- Support structural composition and connectivity
- Integrate into a simulation timeline

This class serves as a base class for implicitly phased verification components such as, `vmm_xactor`, `vmm_group`, `vmm_subenv`, and `vmm_timeline` and is not meant to be directly extended by you. It provides the base functionality of implicit-phased phase methods.

Member Functions

You can access the following public/protected member functions. For more functions provided by this base class, see the *Doxygen* document.

```
vmm_unit (string name, string inst, vmm_object  
*parent=NULL) // Class constructor, which takes three  
arguments, class name, instance name and parent  
(optional).
```

```
virtual void    build_ph () // Virtual functions to be  
                     implemented by you  
virtual void    configure_ph ()  
virtual void    connect_ph ()
```

```
virtual void    configure_test_ph ()
virtual void    start_of_sim_ph ()
virtual void    disabled_ph ()
virtual void    reset_ph ()
virtual void    training_ph ()
virtual void    config_dut_ph ()
virtual void    start_ph ()
virtual void    start_of_test_ph ()
virtual void    run_ph ()
virtual void    shutdown_ph ()
virtual void    cleanup_ph ()
virtual void    report_ph ()
virtual void    final_ph ()
~vmm_unit () //Destructor
```

vmm_group

Description

This class is used as the base composition class for building structural elements composed of transactors or other groups.

Member Functions

You can access the following public/protected member functions.
For more functions provided by this base class, see the *Doxygen* document.

`vmm_group (sc_module_name name, string inst,
vmm_object *parent=NULL) // constructor which takes three
arguments class name, instance name and parent (optional).`

`~vmm_group ()`

Note:

The first argument to the constructor is not a `string` or a `char*`,
but `sc_module_name`.

vmm_xactor

Description

This class is used as a base class for all transactors, including bus-functional models, monitors, and so on. It provides a standard control mechanism expected in all the transactors.

Member Functions

You can access the following public/protected member functions. For more functions provided by this base class, see the *Doxygen* document.

```
vmm_xactor (sc_module_name name, string inst, int
stream_id=-1, vmm_object *parent=NULL) // Class
constructor which takes four arguments, class name,
instance name, stream_id and parent (optional).
```

```
virtual      start_xactor () // used to start the xactor
void

virtual      stop_xactor () // used to stop the xactor
void

virtual      reset_xactor (reset_e rst_typ=SOFT_RST) // used
void          to reset the xactor

virtual      main () //protectd Method used to start the
void          xactor

virtual      prepend_callback (vmm_xactor_callbacks *cb) //
void          used to prepend to the callbacks vector

virtual      append_callback (vmm_xactor_callbacks *cb) //
void          used to append to the callbacks vector
```

```
virtual      unregister_callback (vmm_xactor_callbacks *cb) /
void         / used to unregister a callback from the vector
virtual      register_callback (vmm_xactor_callbacks *cb,
void        bool prepend=0) // used to register a callback
                      extension
~vmm_xactor () // Destructor
```

Note:

- The first argument to the constructor is not a `string` or a `char*`, but `sc_module_name`.
- In explicit phasing, threads (main) are not killed when the `stop_xactor` is called.

vmm_subenv

Description

This is a base class used to encapsulate a reusable subenvironment.

Member Functions

You can access the following public/protected member functions. For more functions provided by this base class, see the *Doxygen* document.

```
vmm_subenv (sc_module_name name, string inst,  
vmm_object *parent=NULL)
```

virtual void	start ()
virtual void	stop ()
virtual void	reset ()
virtual void	cleanup ()
virtual void	report ()
~vmm_subenv () Destructor	

Note:

The first argument to the constructor is not a `string` or a `char*`, but `sc_module_name`.

vmm_timeline

Description

The `vmm_timeline` user-defined class coordinates simulation through a user-defined timeline with predefined test phases. You can add or remove the phases subsequently as needed.

Member Functions

This class provides the same set of public/protected member functions as mentioned for `vmm_group`, as it is the derivative of `vmm_group`. For more functions provided by this base class, see the *Doxygen* document.

```
vmm_timeline (sc_module_name name, string inst,  
vmm_object *parent=NULL, int is_from_simulation=0)  
// Class constructor which takes three arguments class name,  
instance name and parent (optional).
```

```
virtual int run_phase (const char *nm="$")  
int insert_phase (const char *name, vmm_phase_def  
*ph, const char *before="$")  
int add_phase (const char *name, vmm_phase_def *ph)  
int delete_phase (const char *name)  
char * get_previous_phase_name (const char *name)  
char * get_next_phase_name (const char *name)  
char * get_current_phase_name ()  
virtual void prepend_callback (vmm_timeline_callbacks *cb)
```

```
virtual void append_callback (vmm_timeline_callbacks *cb)
virtual void unregister_callback (vmm_timeline_callbacks
*cb)
~vmm_timeline () // Destructor
```

Note:

The first argument to the constructor is not a string or a `char*`,
but `sc_module_name`

vmm_simulation

Description

The `vmm_simulation` class extending from `vmm_unit` is a top-level singleton module that manages the end-to-end simulation timelines. It includes pre- and post-test timelines with predefined pre- and post-test phases. The predefined pre-test phases are `build`, `configure`, and `connect`. The predefined post-test phase is `final`.

Member Functions

You can access the following public/protected member functions. For more functions provided by this base class, see the *Doxygen* document.

<code>vmm_simulation *</code>	<code>get_sim ()</code>
<code>vmm_timeline *</code>	<code>get_pre_timeline ()</code>
<code>vmm_timeline *</code>	<code>get_top_timeline ()</code>
<code>vmm_timeline *</code>	<code>get_post_timeline ()</code>
<code>void</code>	<code>allow_new_phases (bool allow=1)</code>

Note:

The first argument to the constructor is not a `string` or a `char*`, but `sc_module_name`.

vmm_test

Description

The `vmm_test` class is an extension of `vmm_group` and handles the test execution timeline with all the default predefined phases. This is used as the base class for all the tests.

Instances of this class must be either root objects or children of `vmm_test` objects.

Member Functions

You can access the following public/protected member functions. For more functions provided by this base class, see the *Doxygen* document.

```
vmm_test (sc_module_name name="", string doc="",
vmm_object *parent=NULL) // Class constructor which takes
three arguments class name, instance name and parent
(optional).
```

virtual void	set_config ()
virtual string	get_doc ()
virtual string	get_name ()
virtual void	run (vmm_subenv *env)
~vmm_test ()	// Destructor

Note:

The first argument to the constructor is not a `string` or a `char*`, but `sc_module_name`.

vmm_data

Description

The `vmm_data` class provides some common functions required by any data payload. User-defined payload can be derived from this class to make use of these functions. The standard library provides you with the following macros to use these functions.

Macros Usage

1. `VMM_DATA` (user payload class)

Use this macro to define a payload class.

2. `VMM_DATACTOR` (user payload class) { ... }

Use this macro to define a constructor when the payload is derived from `vmm_data`. However, if your payload has multiple base classes, provide your own constructor.

3. `VMM_DATA_MEMBER_BEGIN` (user payload class)

Call this macro inside a user payload class definition. This macro is mandatory and should be used before using the following macro.

4. `VMM_DATA_MEMBER_BASE_CLASS` (base class)

Call this macro for each and every base class of user payload class with the name of the base class as its argument.

Note:

The macros from 5-13 are used based on the type of the member variable name of the user payload class.

The following arguments are used with these macros:

- *type*: gives the type of the member variable
 - *member*: member variable name
 - *length*: length of the array
 - *do_what* - is an enum with values DO_PRINT, DO_COPY, DO_COMPARE, DO_PACK, DO_UNPACK, and DO_ALL. Depending on this value, the variable is involved in the common functions.
 - *do_how* - is an enum with the following values:
 - DO_NOCOPY, DO_REFCOPY, DO_DEEPCOPY
 - HOW_TO_COPY (OR of all DO_*COPY)
 - DO_NOCOMPARE, DO_REFCOMPARE, DO_DEEPCOMPARE
 - HOW_TO_COMPARE (OR of all DO_*COMPARE)
 - DO_NONE (OR of all DO_NO*)
 - DO_REF (OR of all DO_REF*)
 - DO_DEEP (OR of all DO_DEEP*)
 - *format*- the member variable is printed based on this format. For example, <<hex, <<dec, etc. This argument can also be left blank.
5. VMM_DATA_MEMBER_SCALAR (*type*, *member*, *do_what*, *format*)
- Use this macro for any C++/SystemC scalar data types and std::string, but not char pointers.
6. VMM_DATA_MEMBER_SCALAR_ARRAY (*type*, *member*, *length*, *do_what*, *format*)

Use this macro for C++/SystemC arrays of scalar types and arrays of `std::string`.

7. `VMM_DATA_MEMBER_SCALAR_PTR` (`type, member, length, do_what, do_how, format`)

Use this macro for any C++/SystemC pointers and `std::string` pointers. You should allocate sufficient space for these pointers.

8. `VMM_DATA_MEMBER_CHAR_PTR` (`type, member, do_what, format`)

This macro can be used for C++/SystemC char pointers.

9. `VMM_DATA_MEMBER_ENUM` (`type, member, do_what, format`)

This macro can be used for enum types.

10. `VMM_DATA_MEMBER_ENUM_ARRAY` (`type, member, length, do_what, format`)

This macro can be used for an array of enums.

11. `VMM_DATA_MEMBER_VMM_DATA` (`type, member, do_what, format`)

Use this macro for any member objects of type `vmm_data` and its derivatives.

12. `VMM_DATA_MEMBER_VMM_DATA_ARRAY` (`type, member, do_what, format`)

Use this macro for any member which is an array of objects of type `vmm_data` and its derivatives.

13. `VMM_DATA_MEMBER_VMM_DATA_PTR` (`type, member, do_what, do_how, format`)

Use this macro for any member which is a pointer to type `vmm_data` and its derivatives.

14. `VMM_DATA_MEMBER_END` (user payload class)

Call this macro after all the members are covered by the above macros.

All the above macros make every member to get involved in the common functions such as, `psdisplay()`, `copy()`, `compare()`, `byte_pack()` and `byte_unpack()`. If you need to provide implementation for any of the above functions, then `implement do_psdisplay()`, `do_copy()`, `do_compare()`, `do_byte_pack()` and `do_byte_unpack()` to overwrite the corresponding default implementation provided by the standard library.

Because the `vmm_data` class is an abstract base class, it cannot be instantiated directly. The default implementations of `byte_pack()` and `byte_unpack()` are not provided if the `NON_VCS` flag is defined by a non-VCS user.

Note:

- `vmm_data` class is an abstract base class; therefore, it cannot be instantiated directly.
- The default implementations of `byte_pack()` and `byte_unpack()` are not provided if the `NON_VCS` flag is defined by a non-VCS user.
- The macros for pointer to `char array`, `array of char pointers` and containers such as `std::map`, `std::vector` and `std::list` are also not implemented. You must write the common functions for these by using `do_*` functions.

- For the `copy(vmm_data* to)` function, unlike in SystemVerilog (SV), you should either allocate the `to` object or set to `NULL` before calling the `copy` function.
- For byte packing/unpacking, the `VMMSCBytes` class creates and allocates the `bytarray`. You can get the size and pointer of the byte array by using the functions `getByteArraySize()` and `getBytesPtr()`.
- The `float`, `double`, and `sc_bit` data types are not supported.

vmm_object

Description

This class is used as a common base class for all VMM-related classes. This class helps to provide parent/child relationships for class instances. In addition, this class provides local, relative, and absolute hierarchical names.

Member Functions

You can access the following public/protected member functions. For more functions provided by this base class, see the *Doxygen* document.

```
vmm_object (vmm_object *parent, string  
name=" [Anonymous] ", bool disable_hier_insert=0)  
  
bool is_in_namespace (const string &space)  
void set_object_name (const string &name, const string  
&space="")  
virtual void set_parent_object (vmm_object *parent)  
vmm_object * get_parent_object (const string &space="")  
vmm_object * get_root_object (const string &space="")  
int get_num_children ()  
vmm_object * get_nth_child (int n)  
bool is_parent_of (vmm_object *obj, const string &space="")  
string get_object_name (const string &space="")  
string get_object_hiername (vmm_object *const root=NULL,  
const string &space="")  
vmm_object * find_child_by_name (string name, const string  
&space="")
```

```
bool is_in_namespace (const string &space)
static vmm_object* find_object_by_name(const string& name,
const string& space = "");
static int get_num_roots(const string& space = "");
static vmm_object* get_nth_root(int n, const string& space
= "");
static void print_hierarchy(vmm_object* const root=NULL ,
bool verbose = 0);
virtual void display (const string &prefix="")
string psdisplay (const string &prefix="")
virtual void kill_object ()
virtual void implicit_phasing (bool is_on)
virtual bool is_implicitly_phased ()
virtual ~vmm_object () // Destructor
```

vmm_log

Description

The vmm_log feature in VMM-SC provides an exception reporting facility to report an exceptional situation. This facility is provided with the help of macros.

Macros

- VMM_VERBOSE
- VMM_DEBUG
- VMM_TRACE
- VMM_NOTE
- VMM_WARNING
- VMM_ERROR
- VMM_FATAL

The usage of all the above macros is the same as in VMM-SV.

Controlling Log Report

1. SET_SEVERITY(severity)

Call this macro to set the severity. The expected values are,

- ERROR_MSG_TYPE: prints only VMM_ERROR messages
- WARNING_MSG_TYPE: prints VMM_WARNING and VMM_ERROR messages
- NOTE_MSG_TYPE: prints VMM_WARNING, VMM_ERROR, and VMM_NOTE messages

- TRACE_MSG_TYPE: prints VMM_WARNING, VMM_ERROR, VMM_NOTE and VMM_TRACE messages
- DEBUG_MSG_TYPE: prints VMM_WARNING, VMM_ERROR, VMM_NOTE, VMM_TRACE and VMM_DEBUG messages
- VERBOSE_MSG_TYPE : Prints all messages

Note:

By default, the severity is set to NOTE_MSG_TYPE .

2. STOP_AFTER_N_ERRORS(limit)

Call this macro to set a limit on the number of reports (ERROR types) that can cause simulation to call sc_stop () .

For example, STOP_AFTER_N_ERRORS(10) ;

This macro sets the default error count to 10.

vmm_opts

Description

VMM-SC allows configuring VMM transactors, VMM components, and verification environments using `vmm_opts`. Configuration parameters can be set from three different sources in order of decreasing priority:

1. Command-line options
2. Command-file options
3. Set_* methods

The command-line and command-file options are only supported in the SV interop flow. In pure VMM-SC, these options are not supported.

Member Functions

You can access the following public/protected member functions. For more functions provided by this base class, see the *Doxygen* document.

Table 16-1

```
bool get_bit (const string &name, const string &doc="", const
int &verbosity=0, const string &fname="", const int &lineno=0)

string get_string (const string &name, const string &dflt="",
const string &doc="", const int &verbosity=0, const string
&fname="", const int &lineno=0)

int get_int (const string &name, const int &dflt=0, const
string &doc="", const int &verbosity=0, const string
&fname="", const int &lineno=0)

void get_range (const string &name, int &min, int &max, int
dflt_min, int dflt_max, const string &doc="", const int
&verbosity=0, const string &fname="", const int &lineno=0)

vmm_object * get_obj (bool &is_set, const string &name,
vmm_object *dflt=NULL, const string &fname="", const int
&lineno=0)

bool get_object_bit (bool &is_set, vmm_object *obj, const
string &name, const string &doc="", const int &verbosity=0,
const string &fname="", const int &lineno=0)

string get_object_string (bool &is_set, vmm_object *obj,
const string &name, const string &dflt="", const string
&doc="", const int &verbosity=0, const string &fname="", const
int &lineno=0)

int get_object_int (bool &is_set, vmm_object *obj, const
string &name, const int &dflt=0, const string &doc="", const
int &verbosity=0, const string &fname="", const int &lineno=0)

void get_object_range (bool &bit, vmm_object *obj, const
string &name, int &min, int &max, int dflt_min, int dflt_max,
const string &doc="", const int &verbosity=0, const string
&fname="", const int &lineno=0)
```

Table 16-1

```
bool get_bit (const string &name, const string &doc="", const
int &verbosity=0, const string &fname="", const int &lineno=0)

vmm_object * get_object_obj (bool &is_set, vmm_object *obj,
const string &name, vmm_object *dflt=NULL, const string
&doc="", const int &verbosity=0, const string &fname="", const
int &lineno=0)

void set_bit (const string &name, const bool &val, vmm_unit
*root=NULL, const string &fname="", const int &lineno=0)

void set_string (const string &name, const string &val,
vmm_unit *root=NULL, const string &fname="", const int
&lineno=0)

void set_int (const string &name, const int &val, vmm_unit
*root=NULL, const string &fname="", const int &lineno=0)

void set_range (const string &name, const int &min, const
int &max, vmm_unit *root=NULL, const string &fname="", const
int &lineno=0)

void set_object (const string &name, vmm_object *obj,
vmm_unit *root=NULL, const string &fname="", const int
&lineno=0)

void get_help (vmm_object *root=NULL, int verbosity=0)

void check_options_usage (int verbosity=0)
```

vmm_factory

Description

This class is the utility class to generate the instances of any class through the factory mechanism.

Macro for Defining the Factory Class

`vmm_class_factory` (class name): Creates the factory class methods for the specified class. The specified class must have definitions for the `allocate()` and `copy()` functions.

Member Functions

You can access the following public/protected member functions. For more functions provided by this base class, see the *Doxygen* document .

Table 16-2

```
static T* create_instance(vmm_object* parent, string name,
string fname = "", int lineno=0);


---


static void override_with_new(string name, T* factory, string
fname="", int lineno = 0);


---


static void override_with_copy(string name, T* factory, string
fname="", int lineno = 0);
```

vmm_interop

Description

This class is available in SystemC and SystemVerilog. You can use this class to synchronize phases in SV-SC interop mode.

Member Functions

You can access the following public/protected member functions. For more functions provided by this base class, see the *Doxygen* document.

Table 16-3

```
static void run_phase(interop_dir dir, vmm_timeline*
top_timeline);
static void run_tests(interop_dir dir, string tests);
```

Use Model

This section describes how to develop a testbench using the VMM-SC library and compile the source code with VCS.

You can now use VMM-SC in a pure SystemC environment or with SystemC-SystemVerilog interoperability. The interoperability approach gives you the advantage of using verification constructs like constraints, randomization, and coverage that are available with the SystemVerilog language. Any verification component developed in SystemC or SystemVerilog is suitable for a specific project requirement.

The transfer of packet data between SystemC and SystemVerilog is achieved by the TLI adaptors available as a separate package.

The following important components for a verification environment are listed:

- Transaction data classes to specify the payload data
- Multiple verification components implemented as transactors
- Transaction-level interfaces to connect various transactors
- One or more sub-blocks instantiating lower-level transactors
- Top-level verification environment
- Multiple test cases

VMM provides base classes to implement each verification component and guidelines to implement robust and reusable testbenches. A few other base classes and utilities like `vmm_opts` are provided for ease of use and controlling the testbench behavior from the test-case level. VMM-SC provides a message service utility for issuing message reports from the testbench.

Using VMM-SC in SystemC Environment

A verification environment developed in VMM-SC can be used to verify a transaction-level reference model and reused to verify the pin-level RTL. The pin-level RTL can be in SystemC, Verilog-HDL, VHDL, or a combination of these HDLs.

Defining Transaction Data Objects Using vmm_data

VMM-SC provides a `vmm_data` base class for defining the packet data. User extensions of the `vmm_data` class specify the member fields of the payload data class. The shorthand macros available with `vmm_data` provide an implementation of basic functions such as `byte_pack`, `byte_unpack`, `print`, `compare`, and `copy` that are required for most payload classes.

You can also use the OSCI-TLM2 `tlm_generic_payload` class as a payload class in a VMM-SC environment.

Example 16-1 Transaction Class Defined Using vmm_data

```
VMM_DATA(my_payload)
{
public:
    VMM_DATACTOR(my_payload) { };
    int addr;
    int data;

    VMM_DATA_MEMBER_BEGIN(my_payload)
    VMM_DATA_MEMBER_BASE_CLASS(vmm_data)
        VMM_DATA_MEMBER_SCALAR(int,addr,DO_ALL,<<hex)
        VMM_DATA_MEMBER_SCALAR(int,data,DO_ALL,<<hex)
    VMM_DATA_MEMBER_END(my_payload)
};
```

[Example 16-1](#) shows the transaction data class extended from `vmm_data`. There are two member variables, `addr` and `data`. The use of shorthand macros, '`vmm_data_member_scalar` for each member variable provides the implementation of member functions such as, `byte_pack`, `byte_unpack`, and others.

For the complete list and functionality of the methods implemented by the shorthand macros, see the *VMM User Guide*.

Issuing Message Reports Using VMM Log Macros

VMM SystemC provides shorthand macros to issue messages/reports within the verification environment. The macros provided in VMM-SC match the different severity-level macros provided in VMM SystemVerilog. In pure VMM-SC, the messages are internally issued by `sc_report_handler`. The messages are issued on standard output by default. If the `sc_report_handler` is set to issue the messages into another log file, the VMM log messages are also redirected to the same log file.

[Table 16-4](#) lists the VMM log macros provided by VMM-SC in decreasing order of severity.

Table 16-4 VMM Log Macros Provided by VMM-SC

VMM-SC Report Macro	Severity Level
VMM_FATAL (nm,txt)	Fatal. Issuance of a fatal message causes simulation to exit.
VMM_ERROR (nm,txt)	Error. Simulation exits after the configured number of VMM_ERROR messages (default 10) have been issued.
VMM_WARNING (nm,txt)	Warning
VMM_NOTE (nm,txt)	Default setting. By default, only messages up to note severity are issued.
VMM_TRACE (nm,txt)	Trace
VMM_DEBUG (nm, txt)	Debug
VMM_VERBOSE (nm, txt)	Verbos

Each VMM log macro takes two string arguments as parameters. The first parameter is an “id”, usually the name of the component from which the log report is issued. The second string argument is the actual message string.

The default severity level is set to `VMM_NOTE`. Only messages with fatal, error, warning, and note are issued with the default severity level. You can modify the severity level by calling the `SET_SEVERITY` macro with one of the specified severity types as an argument to the macro. The severity can only be set from error down to verbose severity. The fatal and error messages are always issued.

[Table 16-5](#) lists the argument value of the `SET_SEVERITY` macro and the minimum severity message that is issued.

Table 16-5 SET_SEVERITY Argument with Minimum Severity Messages

SET_SEVERITY Argument	Minimum Severity Message Issued
<code>ERROR_MSG_TYPE</code>	Error
<code>WARNING_MSG_TYPE</code>	Warning and Error
<code>NOTE_MSG_TYPE</code>	Default (Warning, Error, and Note)
<code>TRACE_MSG_TYPE</code>	Trace (Warning, Error, Note, and Trace)
<code>DEBUG_MSG_TYPE</code>	Debug (Warning, Error, Note, Trace, and Debug)
<code>VERBOSE_MSG_TYPE</code>	Verbose (prints all messages)

Simulation is terminated when fatal severity messages are issued. Simulation can also fail when the configured number of error severity messages are issued, which defaults to 10. You can control the number of error messages issued before the simulation exits by calling `STOP_AFTER_N_ERRORS` with the specified number of error messages passed as an integral argument.

For example, STOP_AFTER_N_ERRORS(20) stops the simulation after 20 error messages are issued.

Example 16-2 Issuing Message Reports Using VMM Log Macros

```
virtual void run_ph(void)
{
    while(1) {
        First* data1;
        data1 = out->read();
        VMM_NOTE("In run_ph", data1->psdisplay("First."));
        First* data2 = out->read();
        string diff = "Equal";
        data1->compare(*data2,diff);
        if(diff == "Equal")
            VMM_NOTE("TARGET ", "Compare PASSED!!! ");
        else
            VMM_ERROR("TARGET", "Compare FAILED");
    }
}
```

Example 16-3 Changing Default Settings of VMM Log

```
int sc_main(int argc, char* argv[])
{
    vmm_timeline* t1;
    env*         e1;
    STOP_AFTER_N_ERRORS(20);
    SET_SEVERITY(WARNING_MSG_TYPE);
    t1 = new vmm_timeline("timeline", "t1");
    e1 = new env("env", "e1", t1);
    t1->run_phase();
    sc_start(50000, SC_NS);
}
```

Options and Configuration Service Using vmm_opts

Configurations can be set from the simulator command line, a command file, or from the source code itself.

```

#include "vmm.h"
Class producer: public vmm_xactor {
public:
    int num_of_trans
    virtual void run_ph() {
        for(int i=0; i< num_of_trans; i++)
            //send transactions to SV consumer
    }
    virtual void start_of_sim_ph() {
        num_of_trans =
    vmm_opts::get_object_int(is_set,this,"NUM_OF_TRANS",16);
    }
    virtual void shutdown() {}
};


```

To configure the option from the command line, use,

```
% simv +vmm_opts+NUM_OF_TRANS=5@%*producer_inst0
```

To configure the option from the command file, use,

```
% simv +vmm_opts_file+options.opt
```

`options.opt` is a command file which contains the following line:

```
+NUM_OF_TRANS=6@%*producer_inst0
```

Example 16-4 Configuring the Option From the Source Code Using `set_` Method*

```

#include "producer.h"
class env: public vmm_group {
public:
    producer* producer_inst0;
    producer* producer_inst1;

    env(string name="", string inst="", vmm_object*
parent=NULL): vmm_group(name.c_str(), inst, parent)    {}

```

```

        virtual void build_ph() {
            producer_inst0 = new producer(this, "producer_inst0");
            producer_inst1 = new producer(this, "producer_inst1");
        }
        virtual void connect_ph() {
            tli_tlm_bind_initiator(producer_inst0->in_socket,
LT, "target0", true);
            tli_tlm_bind_initiator(producer_inst1->in_socket,
LT, "target1", true);
        }
        virtual void configure_ph() {
            vmm_opts::set_int("@%*producer_inst0:NUM_OF_TRANS", 2)
        }
    };

```

Class Factory Service Using `vmm_class_factory` Macro

This factory service is the utility class to generate instances of any class through the factory mechanism. This can be achieved by using the `vmm_class_factory` macro. The factory class using this macro must have the definition for `allocate()` and `copy()` functions. If the class is derived from `vmm_data`, these functions are automatically provided when `vmm_data` macros are used. Usage is same as in SystemVerilog.

Example 16-5 Using `vmm_class_factory` Macro

```

#include "vmm.h"
class payload: public vmm_data {
public:
    // VMM DATA MEMBER MACROS
    VMM_CLASS_FACTORY(payload)
};

class generator: public vmm_xactor
{
public:
    // constructor
    payload *pyld;
}

```

```

        virtual void start_of_sim_ph()
    {
        pyld =
payload::create_instance(this,"PYLD0",__FILE__,__LINE__);
        cout << pyld->addr << endl; // Prints 10 (assigned at
override_with_copy in env class)
    }
};

class env: public vmm_group
{
    generator* gen0;
    payload *gp;
    // Constructor
    virtual void build_ph()
    {
        gen0 = new generator(this,"generator");
        gp = new payload;
    }
    virtual void configure_ph()
    {
        gp->addr = 10; // assign all the fields

payload::override_with_copy("PYLD0",gp,__FILE__,__LINE__);
    }
};

```

Note:

Use `override_with_new()` with the same arguments if the `gp` is a derived class object of `payload` class. But the values of the fields (`addr` and so on) cannot be updated at `create_instance`; only memory is allocated for the object.

Defining Transactors Using `vmm_xactor`

The term transactor is used to identify components of the verification environment that interface between two levels of abstraction for a particular protocol or to generate protocol transactions. The lifetime of a transactor is static to the verification environment.

The `vmm_xactor` class is used as the base class for defining transactors. The `vmm_xactor` is inherited indirectly from the `sc_module`.

There are three main use models of a transactor class, as follows:

- Generator of transaction data: This transactor class instantiates the packet data class it services and usually has a transaction-level interface to send the generated packet to an externally connected verification component.
- Function layer transactor: This transactor class instantiates one or more packet data classes it interfaces to and has incoming and outgoing transaction-level interfaces to connect to external verification components.
- Command layer transactor: This transactor class, usually referred to as a driver, interfaces to the DUT. The interface to the DUT can be a transaction-level interface or a pin-level interface. The command layer converts the transaction data to the relevant input of the DUT.

Transactors progress through a series of phases throughout simulations. All transactors are synchronized so that they execute their phases synchronously with other transactors during simulation execution.

VMM supports two transaction phasing usage models: implicit and explicit. In explicit phasing, the transactors are under the control of a master controller such as `vmm_env` to call the transactor phases. In implicit phasing, the transactors execute their phases automatically and synchronously.

The phases are implemented as virtual methods in the base class. You can extend any of the phase methods to define the functionality for the specific transactor.

For more details on each phase method available with the `vmm_xactor` class, see the *VMM User Guide*.

Example 16-6 Transactor Class Using vmm_xactor

```
class initiator: public vmm_xactor,
  tlm::tlm_bw_transport_if<my_payload_types>
{
public:
    my_payload* payload;
    tlm::tlm_initiator_socket<32,my_payload_types>
in_socket;

    initiator(vmm_object* parent, sc_core::sc_module_name
name) :
vmm_xactor(name, "init", 0, parent), in_socket("in_socket")
{
    in_socket.bind(*this);
    payload = new my_payload;
}
virtual void run_ph()
{
    my_payload* obj;
    obj = payload->allocate();
    sc_time delay(SC_ZERO_TIME);
    for(int i=0; i<4;i++)
    {
        obj->addr = i *4;
        obj->data = 0xFF + i;
        in_socket->b_transport(*obj,delay);
        wait(delay);
    }
}
...
};
```

[Example 16-6](#) shows a generator transactor which generates payload objects of `my_payload` class. The transactor has an output, `tlm_initiator_socket`, to send the payload data objects to an externally connected verification component. The `run_ph` method is extended in this class.

Communication Between Transactors

VMM-SC does not specifically provide any transaction-level interface. The OSCI-TLM 2.0 is the recommended transaction-level interface that you should use to connect various verification components. You can also use other transaction-level interfaces like TLM1.0 or `sc_fifo`.

Defining Verification Environment Using `vmm_group`

A verification environment or subenvironment instantiates, connects, and controls the underlying lower-level transactors. A verification environment is implemented in VMM-SC by extending the `vmm_group` or `vmm_subenv` class.

The `vmm_group` and `vmm_subenv` classes have the same phases as those in the `vmm_xactor`. The `vmm_group` is an implicitly phased environment class and the phases of the children of the `vmm_group` class are also called implicitly in a top-down or bottom-up manner depending on the phase.

The `vmm_subenv` provides both implicit and explicit phase methods. In implicit phasing, the children of `vmm_subenv` are not called implicitly and the particular phase method of the child must be called in the appropriate phase method of `vmm_subenv`.

For more details on `vmm_group` and `vmm_subenv`, see the *VMM User Guide*.

Example 16-7 Verification Environment Using vmm_group

```
class env: public vmm_group
{
public:
    initiator* initiator0;
    target* target0;

    env(string name="", string inst="", vmm_object*
parent=NULL) : vmm_group(name.c_str(), inst, parent) {}

    virtual void build_ph()
    {
        initiator0 = new initiator(this, "initiator0");
        target0 = new target(this, "target0");
    }

    virtual void connect_ph()
    {
        initiator0->in_socket(target0->target_socket);
    }

    virtual void shutdown()
    {
        wait(5000, SC_NS);
    }
};
```

[Example 16-7](#) shows how to implement a verification environment using `vmm_group`. The `vmm_group` class instantiates the lower-level initiator and target components. These are allocated in the `build_ph` method of `vmm_group`. The `connect_ph` is used to connect the initiator and target using TLM2.0 initiator and target sockets.

Extending Phase Methods Having Forever Loops

The execution of a phase in `vmm_unit` extension happens after the previous phase is completed. It is often required that some phases have an infinitely running forever loop, controlled by an external event.

You might also need to execute the next phase after forking off the loop of the previous phase. This functionality is achieved in SystemVerilog using the `fork-join_none` language construct. In VMM-SC, equivalent functionality can be achieved using the `sc_spawn` template function and defining the functionality of the loop in a separate method.

Example 16-8 Phase Method Having a Loop

```
virtual void run_ph()
{
    cout<<"IN Initiator run_ph before dynamic proc \n";
    sc_spawn(sc_bind(&initiator::forked, this));
    cout<<"IN Initiator run_ph after dynamic proc \n";
}
virtual void forked()
{
    while(1) {
        if(this->stop_run_ph)
            break;
        ...
        in_socket->b_transport( *trans, delay );
        ...
    }
}
```

Coordinating Simulation Using vmm_timeline

Use the `vmm_timeline` class to coordinate simulation through a user-defined timeline. A verification environment can have a single timeline controlling the top-level verification components or have multiple timelines in different verification subenvironments connected hierarchically to a top-level timeline.

The top-level timeline is instantiated in the verification top, which is usually an `sc_module` extension. This verification top instantiates a top-level timeline, instantiates the verification environment, connects the timeline to the verification environment, and executes the timeline by calling the `run_phase()` method of `vmm_timeline`.

For more details on `vmm_timeline`, see the *VMM User Guide*.

Example 16-9 The Usage of vmm_timeline to Coordinate Simulation

```
class sc_top: public sc_module
{
public:
    vmm_timeline* t1;
    env*          e1;
    sc_top(sc_module_name name):sc_module(name)
    {
        t1 = new vmm_timeline("timeline", "t1");
        e1 = new env("env", "e1", t1);
        t1.run_phase();
    }
};
```

Defining Test Cases Using vmm_test

You must use the `vmm_test` base class to implement test cases. For each test case, create a new class that extends `vmm_test`. VMM-SC supports only the implicitly phased test, unlike VMM-SV which has both implicitly and explicitly phased tests.

Example 16-10 Test Case Using vmm_test

```
class test_simple : public vmm_test
{
public:
    test_simple(sc_module_name name) :
vmm_test(name, "test_simple")
    {
    }
    void shutdown_ph()
    {
        wait(5000, SC_NS);
        env->stop_event->notify();
    }
};
```

Controlling Tests and Root Components Using vmm_simulation

The `vmm_simulation` class extending from `vmm_unit` is a top-level singleton module which manages the end-to-end simulation. All root-level verification components such as, `vmm_group`, `vmm_timeline` and the registered `vmm_tests` are executed implicitly by calling the `run_test` method of `vmm_simulation`.

The `vmm_simulation` class has three predefined timelines: pre-test, top-test, and post-test. The `vmm_simulation` class can execute multiple tests serially by executing the top-test timeline

repeatedly, once for each test case that must be run. The pre-test time is executed only once before the start of simulation and has the build, configure, and connect phases. The post-test time is executed only once after all the tests have completed and has the final phase.

Example 16-11 Controlling Multiple Tests Using vmm_simulation

```
class sc_top: public sc_module
{
public:
    test_simple* t1;
    sc_top(sc_module_name name):sc_module(name)
    {
        t1 = new test_simple("test_simple", "Simple testcase");
        vmm_simulation::run_tests("test_simple");
    }
};
```

Instantiating VMM-SC Environment in sc_main

You can also instantiate the top-level VMM-SC environment in `sc_main` if required by the particular verification setup. The simulation can be started from `sc_main` by the `vmm_simulation` or `vmm_timeline`. You should call `sc_start` after `vmm_simulation::run_tests()` or `vmm_timeline.run_phase()`. If you are using `vmm_timeline`, you can run up to the connect phase before calling `sc_start`.

Example 16-12 vmm_timeline in sc_main

```
int sc_main(int argc, char* argv[])
{
    vmm_timeline* t1;
    env*          e1;
    t1 = new vmm_timeline("timeline","t1");
    e1 = new env("env","e1",t1);
```

```

    t1->run_phase();
    sc_start(50000,SC_NS);
    return 0;
}

```

Example 16-13 vmm_simulation in sc_main

```

int sc_main(int argc, char* argv[])
{
    env*          e1;
    e1 = new env("env", "e1");
    vmm_simulation::run_tests();
    sc_start(50000,SC_NS);
    return 0;
}

```

Using VMM-SC With VMM-SV Interoperability

You can synchronize VMM-SC transactors with VMM-SV transactors using the interop solution (that is, you can synchronize the phases in the VMM-SC and VMM-SV worlds). The interop use model depends on whether `vmm_timeline` or `vmm_simulation` is a top level. It is recommended to use `vmm_simulation` as a top level.

Interconnects in Interop Flow

The interop flow need interconnects to communicate between the two languages (SC and SV). For this purpose you have two interconnect classes, one in SC and the other in SV.

- Interconnect class in SC: `vmm_interop`
- Interconnect class in SV: `vmm_interop`

Interconnect class in SC: vmm_interop:

Use this SC class to call phases in SV-SC interop mode using the following static methods.

1. `run_phase(interop_dir dir, vmm_timeline* top_timeline):`

This method is called when you have `vmm_timeline` as a top level and need to run SV and SC phases in sync.

The first argument is an enum which takes the direction. Valid values are,

- `vmm_interop::SV_DRV` (when the direction is SV->SC)
- `vmm_interop::SC_DRV` (when the direction is SC->SV)

The second argument is a top-level timeline.

2. `run_tests (interop_dir dir, string tests):`

This method is called when you have `vmm_simulation` as a top level and want to run SV and SC phases in sync.

The first argument is an enum which takes the direction. Valid values are,

- `vmm_interop::SV_DRV` (when the direction is SV->SC)
- `vmm_interop::SC_DRV` (when the direction is SC->SV)

The second argument is `vmm_test name`, which has to be run. By default it runs all tests registered.

Interconnect class in SV: vmm_interop:

This SV class is used to call phases in SV-SC interop mode using the following static methods.

1. `run_phase (interop_dir dir, vmm_timeline top_timeline):`

This method is called when you have `vmm_timeline` as a top level and want to run SV and SC phases in sync.

The first argument is an enum which takes the direction. Valid values are,

- `vmm_interop::SV_DRV` (when the direction is SV->SC)
- `vmm_interop::SC_DRV` (when the direction is SC->SV)

The second argument is a top-level timeline.

2. `run_tests (interop_dir dir, string tests):`

This method is called when you have `vmm_simulation` as a top level and want to run SV and SC phases in sync.

The first argument is an enum which takes the direction. Valid values are,

- `vmm_interop::SV_DRV` (when the direction is SV->SC)
- `vmm_interop::SC_DRV` (when the direction is SC->SV)

The second argument is `vmm_test name`, which has to be run. By default, it runs all tests registered.

Use Model When vmm_timeline is a Top Level

In this flow, you should call `vmm_interop::run_phase` in SV and SC with appropriate arguments. This synchronizes the phases in SV and SC.

Steps to follow in SV:

1. Call imported DPI-C function `register_dpi_scope()` in the program block where `vmm.sv` is included.
2. Call `vmm_interop::run_phase()` to run the phases implicitly, with direction and top timeline as arguments to the method.

Steps to follow in SC:

1. In the top-level SystemC module (or `sc_main`), call `vmm_interop::run_phase()` with direction and top timeline as arguments to the method.

Example 16-14 SystemVerilog Code for SV Driving SC

```
program tb;
  `include "vmm.sv"
  ...
  tb_env                         env;
  vmm_timeline                     t1;
  initial begin
    register_dpi_scope();
    t1 = new("timeline","t1");
    env = new("env","env1",t1);
    vmm_interop::run_phase(vmm_interop::SV_DRV,t1);
  end
endprogram
```

Example 16-15 SystemC Code for SV Driving SC

```
int sc_main(int argc, char* argv[])
```

```

{
    vmm_timeline* t1;
    env*          e1;
    t1 = new vmm_timeline("timeline","t1");
    e1 = new env("env","e1",t1);
    vmm_interop::run_phase(vmm_interop::SV_DRV,t1);
}

```

Example 16-16 SystemVerilog Code for SC Driving SV

```

program tb;
    `include "vmm.sv"
    ...
    tb_env                           env;
    vmm_timeline                      t1;
    initial begin
        register_dpi_scope();
        t1 = new("timeline","t1");
        env = new("env","env1",t1);
        vmm_interop::run_phase(vmm_interop::SC_DRV,t1);
    end
endprogram

```

Example 16-17 SystemC Code for SC Driving SV

```

int sc_main(int argc, char* argv[])
{
    vmm_timeline* t1;
    env*          e1;
    t1 = new vmm_timeline("timeline","t1");
    e1 = new env("env","e1",t1);
    vmm_interop::run_phase(vmm_interop::SC_DRV,t1);
}

```

Use Model When vmm_simulation is a Top Level

In this flow, you should call `vmm_interop::run_tests` in SV and SC with appropriate arguments. This synchronizes the phases in SV and SC.

Steps to follow in SV:

1. Call the imported DPI-C function, `register_dpi_scope()`, in the program block where `vmm.sv` is included.
2. Call `vmm_interop::run_tests()` to run the phases implicitly with direction as arguments to the method.

Steps to follow in SC:

1. In the top-level SystemC module (or `sc_main`), call `vmm_interop::run_tests()` with direction as arguments to the method.

Example 16-18 SystemVerilog Code for SV Driving SC

```
program tb;
  `include "vmm.sv"
  tb_env                           env;
  my_test                           test;
  initial begin
    register_dpi_scope();
    test = new();
    env = new("env", "env1");
    vmm_interop::run_tests(vmm_interop::SV_DRV);
  end
endprogram
```

Example 16-19 SystemC Code for SV Driving SC

```
int sc_main(int argc, char* argv[])
{
```

```

my_test* test;
env*      e1;
test = new my_test("my_test", "test1");
e1 = new env("env", "e1");
vmm_simulation::register_test(test);
vmm_interop::run_tests(vmm_interop::SV_DRV);
}

```

Example 16-20 SystemVerilog Code for SC Driving SV

```

program tb;
`include "vmm.sv"
...
tb_env          env;
my_test         test;
initial begin
  register_dpi_scope();
  test = new();
  env = new("env", "env1");
  vmm_interop::run_tests(vmm_interop::SC_DRV);
end
endprogram

```

Example 16-21 SystemC Code for SC Driving SV

```

int sc_main(int argc, char* argv[])
{
  vmm_timeline* t1;
  env*      e1;
  t1 = new vmm_timeline("timeline", "t1");
  e1 = new env("env", "e1", t1);
  vmm_interop::run_tests(vmm_interop::SC_DRV);
}

```

17

SystemC Features

This chapter contains the following features:

- “[Simulating Virtual Platform Models](#)” on page 415
- “[Generating Profile Reports for SystemC Designs](#)” on page 430
- “[Compiling SystemC Designs in Partition Compile Flow](#)” on page 439
- “[Integrating SystemC with Innovator](#)” on page 443

Simulating Virtual Platform Models

You can now simulate virtual platform models from SG (formerly Coware/Innovator platform models) with VCS. After simulating, you can debug your issues with full debug capabilities using SG debug tools. Limited debug capability is also provided with DVE.

This section explains how to combine VCS and the Virtualizer and Platform Architect MCO Synopsys system-level products.

Combined, these products allow you to create simulations with a mix of HDL and transaction-level models (TLMs). Virtualizer is a virtual platform product from Synopsys, aiming at the development and verification of embedded software.

Platform Architect MCO is the Synopsys architecture analysis and optimization product for multicore SoCs. Both products use SystemC TLMs as models of the (future) hardware, the former typically employing the TLM-2.0 Approximately Timed (AT) and/or Cycle-Accurate (CA) abstraction levels and the latter using the TLM-2.0 Loosely Timed (LT) abstraction level.

Using Platform Architect and Virtualizer MCO

There are multiple scenarios benefiting from the combination of HDL and TLMs. For Platform Architect MCO, a typical, but not exhaustive, list is as follows:

- Use an HDL model to replace a TLM model. This is useful when a detailed TLM model is not available, or when dealing with legacy components.
- Validate the performance of an HDL implementation of a block or subsystem in the context of a TLM performance model, by inserting the HDL component in the TLM performance models.
- Run both the TLM model and its HDL realization in tandem to compare performance. This way you can use the TLM model as a golden reference for the HDL realization and prove correct realization through simulation.

For Virtualizer, a typical (not exhaustive) list includes:

- Use an HDL model to replace a TLM model. This is useful when a detailed TLM model is not available, or when dealing with legacy components.
- Perform hardware and/or software co-verification. When you test the hardware and software together, you can verify hardware realization by exercising it with software. You can also debug the software using third-party embedded software debuggers, which are helpful for boot code development and validation (for example).
- Accelerate your HDL simulations by abstracting certain subsystems or component TLM models.
- Enable early development of testbench and test scenarios. By using TLM models as replacements for the HDL Design Under Test (DUT), you can start building the testbench environment prior to HDL availability. Similarly, development of test scenarios can be accelerated.
- Verify the consistency of TLM models and their HDL realization (model validation).
- Use a TLM as a golden reference for HDL realization, to support a top-down development flow.

What's New in Virtualizer and Platform Architect MCO

The Virtualizer and Platform Architect MCO products share a new Synopsys-provided, IEEE-1666-compatible SystemC kernel. This kernel is optimized for speed and debug ability through advanced instrumentation. It extends host OS support to include 32- and 64-bit support. The new SystemC kernel is closely integrated with VCS through its Direct Kernel Interface (DKI), thus removing the overhead of dual-kernel cosimulation.

A direct benefit to users is improved simulation performance for combined SystemC / HDL simulation. VCS now supports both the Virtualizer / Platform Architect MCO SystemC simulation kernels and the reference OSCI SystemC kernel.

Platform Architect MCO is the successor product to The Platform Architect product of CoWare. In the past, this product integrated SystemC with VCS through PLI. The new DKI-based integration boosts simulation performance and simplifies the compile flow.

Virtualizer is the next-generation Synopsys virtual prototyping product, succeeding the Innovator. While Innovator was integrated in the past with VCS through an optimized SystemC kernel, Virtualizer brings a wealth of new tools to the combined solution, including:

- more feature-rich SystemC debug solution (VPExplorer)
- more advanced hardware analysis tool (VPExplorer)
- new software analysis tool (VPViewer).

Note:

The Virtualizer / Platform Architect MCO provided SystemC simulation kernel is not a binary compatible with the reference OSCI SystemC simulation kernel, requiring (existing) SystemC models to be recompiled against the headers of the Synopsys-provided kernel. For more information, see “[Compile Flow](#)” on page 421.

Setting up Virtualizer and Platform Architect MCO

Following are the product version requirements, supported host OSs, and compilers. This section explains the setup steps required to use the products together.

Version Requirements

The capabilities and features described here require the following set of tools:

- VCS, version G-2012.09 or later.
- Virtualizer 2012.06 and/or Platform Architect MCO, version 2012.06 or later.

Note:

You cannot combine earlier releases of these products.

Platform Matrix

The following host platforms are supported:

- RHEL32 (RedHat 4, RedHat 5, SuSe 10, SuSe 11), both 32- and 64-bit variants.

Note:

The Windows and Solaris operating systems are currently not supported.

Compiler

Only the gcc v4.5.2 host compiler is supported. Note that you must use the shared variant, and not the static variant of the gcc compiler.

Important:We recommend using the GNU package that is part of the Virtualizer installation. Do not use the GNU package that is part of the VCS installation. For Virtualizer 2012.06, the GNU package contains gcc 4.5.2 and binutils 2.21. If you use a different GNU installation, then make sure to use the linker from binutils 2.21. This is recommended because Virtualizer models are quite sensitive to the order of Static CTORs/initializers. Binutils 2.21 (used by Virtualizer 2012.06) and binutils 2.22 (used by VCS 2012.09) have different orders.

Setting Up Your Environment

For VCS, set the following environment variables and values:

- Set the VCS_HOME environment variable to point to the top-level installation directory of VCS:

```
% setenv VCS_HOME <vcs_installation_directory>
```

- Set your Path to include \$VCS_HOME/bin:

```
Path=($VCS_HOME/bin $path)
```

For Virtualizer or Platform Architect MCO, set the following environment variables and values:

- Set the SNPS_VP_HOME environment variable to point to the top-level installation directory of Virtualizer or Platform Architect MCO:

```
% setenv SNPS_VP_HOME <installation_directory>
```

- Unset the VCS_INSTALLED_AT environment variable:

```
% unsetenv VCS_INSTALLED_AT
```

- For Virtualizer, source the following setup script:

```
% source $SNPS_VP_HOME/setup.csh -vauth
```

- For Platform Architect MCO, source the following setup script:

```
% source $SNPS_VP_HOME/setup.csh -pa
```

To use the gcc compiler in the Virtualizer / Platform Architect MCO installation, no extra steps are required, because the above source commands set up the compiler.

To use the VG GNU package provided by Synopsys, follow these steps:

```
% setenv VG_GNU_PACKAGE /fs/src/interfaces/vg_gnu_package\  
/TD/linux  
  
% setenv LD_LIBRARY_PATH /fs/src/interfaces/vg_gnu_package\  
/TD/linux/gcc-4.2.2_32-shared/lib  
  
% source $VG_GNU_PACKAGE/source_me_gcc4_32-shared.csh
```

Using SystemC with Virtualizer and Platform Architect MCO

The following sections explain how to compile and run:

- “Compile Flow” on page 421
- “Run Flow (No Debug)” on page 422

Compile Flow

Follow these steps:

1. Make sure the products, compilers, and valid licenses are set up and installed correctly.

2. Compile your SystemC components with `syscan` and your HDL components with VCS. You must specify use of the system-level SystemC kernel which is part of Virtualizer and Platform Architect MCO, using the `-sysc=snps_vp` options for this as follows:

```
% syscan -sysc=snps_vp ...
% vcs -sysc -sysc=snps_vp ...
```

These steps build a statically linked `simv` VCS simulation executable, identical to past SystemC support in VCS. You can use the `-cflags`, `-LINK_FLAGS` and `-LINK_LIBS` VCS switches to specify additional compile options (for instance include paths), link options, and link libraries for the SystemC TLM models that you are including.

3. If you are linking in your own SystemC components, make sure you recompile them against the SystemC simulation kernel provided with Virtualizer / Platform Architect MCO. Failure to do this can cause a simulation crash, because the provided SystemC kernel is not binary compatible with the OSCI-provided reference SystemC simulation kernel.
4. If you are using Synopsys TLM model libraries, in some cases you can implement them internally as dynamically linked shared objects loaded at runtime. For more information about the setup, see “[Run Flow \(No Debug\)](#)” on page 422.

Note:

The SystemC kernel option `-sysc=inno` introduced in an earlier version of VCS (intended for use with the Synopsys Innovator virtual platform product), is now an alias for `-sysc=snps_vp`.

Run Flow (No Debug)

Follow these steps:

1. If you want to just simulate, with no interactive debugging, do one of the following:

```
% ./simv --cwr_nosession
```

or

```
% vpsession ./simv
```

2. If you are using Synopsys TLM model libraries, you can, in some cases, implement the components internally as dynamically linked shared objects. These objects are loaded at runtime. To ensure that the TLM models are loaded correctly, do one of the following:

- Manually add the required library paths to your RHEL32 LD_LIBRARY_PATH environment variable.
- Use the build flow from the `ahb_rtl_cosim` example explained below. In the `ahb_rtl_cosim` example, the SystemC Shell tool (`scsh`) is used to generate a `Makefile` for the simulation. At the same time, `scsh` generates a `vpsession_sim.conf` configuration file. This file contains information for `vpsession` to set up the correct LD_LIBRARY_PATH to start the simulation. For this to work, the file `vpsession_sim.conf` has to be in the same directory as the simulation.

For simple simulations, the following has to be in LD_LIBRARY_PATH:

32-bit Simulation

```
 ${SNPS_VP_HOME}/common/libso-gcc-  
 4.2.2:${SNPS_VP_HOME}/gnu/gcc-4.2.2/lib
```

64-bit Simulation

```
$ {SNPS_VP_HOME}/common/libso-gcc-4.2.2-  
64:$ {SNPS_VP_HOME}/gnu/gcc-4.2.2-64/lib64
```

Debug Flow for Using SystemC with Virtualizer and Platform Architect MCO

DVE provides powerful debugging tools for both SystemC and HDL, including source code debugging and waveform generation.

Virtualizer and Platform Architect MCO provide similar powerful debug support for SystemC. You can select whatever debug tools you want to use. The different debug use flows are explained in the following sections:

- “[Using VPExplorer, VPA, or VPViewer](#)” on page 424
- “[Using DVE](#)” on page 425

Using VPExplorer, VPA, or VPViewer

Virtualizer and Platform Architect MCO contain the following debug and analysis tools:

- VPExplorer — A SystemC source code model debug and hardware analysis tool.
- Virtual Platform Analyzer (VPA) — A TLM platform runtime tool, with a built-in embedded software debugger.
- VPViewer — An embedded software analysis tool.

You can combine all these tools with a VCS simulation. There are no restrictions on feature usage when you combine them with HDL simulation. To use these debug and analysis tools, follow these steps:

1. Make sure you have `LD_LIBRARY_PATH` set or use the SystemC Shell (`scsh`) to generate a `vpsession_sim.conf` file.
2. Start VPExplorer, VPA, or VPViewer as usual.
3. Launch `simv` from within the selected tool.

Using DVE

The start mode for DVE is slightly different. Note that:

- Starting with `simv -gui` does not work.
 - Starting with `simv -gui --cwr_nosession` does not work because the simulation blocks in restart.
1. Start DVE as follows:

```
% dve -toolexe $SNPS_VP_HOME/tools/bin/vpsession \
  -toolargs simv
```

2. If you are using the `ahb_rtl_cosim` example, cd into the `export` directory and execute the following:

```
% dve -toolexe $SNPS_VP_HOME/tools/bin/vpsession \
  -toolargs Debug/simv
```

You can also first start DVE with `dve&`, select the menu Simulator/Setup..., and then set the fields in the dialog box as follows:

- Simulator Executable: full path to `vpsession`

- Simulator arguments: `./simv` (or `Debug/simv` for the `ahb` example)

Note:

CBug (DVE capabilities for SystemC, C, C++ debug) are mostly turned off because the SNPS_VP SystemC kernel is used. The SystemC models are shown in the hierarchy but you cannot select them. For example, if you double-click, DVE/CBug issues an error message.

Configure and DVE Debug Flow

You can configure the virtual platform model and use DVE interactively by invoking a shell script that starts the simulation and executes a VPA script on it. The `ahb_rtl_cosim` example contains such a script (called `sim_start`). Use this `sim_start` script instead of the simulation and/or `vpsession` in DVE:

```
% dve -toolexe sim_start
```

This script starts `vpash` (a VPA shell) with a script (`vpash_script.tcl`) and the simulation. The `vpash` script connects to the simulation, configures the platform, and then lets the simulation continue and disconnects from it. At that point, DVE can take control over the simulation. For details, see the `ahb_rtl_cosim` example.

Combined VPExplorer / VPA for SystemC & DVE for HDL Debug Flow

Simultaneous interactive debug combining VPExplorer or VPA with DVE is not supported. Instead, do a full interactive debug with VPExplorer or VPA controlling the simulation. Debug the HDL part with DVE in a post-process flow. Add a `$vcdpluson` statement to Verilog and open the resulting VPD file with DVE:

```
% dve -vpd vcdpluson.vpd
```

The VPD file shows only the Verilog / VHDL waveform output. The SystemC output is missing, but can be logged via VPExplorer or VPA. Note that the VPD waveform file is updated (flushed) anytime the simulation stops in VPExplorer or VPA. At that point you can simply reload the VPD file in DVE.

SystemC Examples

You can find a few simple examples here:

```
$VCS_HOME/doc/examples
```

sc_top_vlog_down

The following example illustrates the compile flow for SystemC-on-top with a Verilog HDL component below it in the design hierarchy. The HDL / SystemC interface is at the pin level. No models from the Synopsys System-Level Library are used, so no model licenses are required.

To build and run this example, follow these steps:

```
% cd sc_top_vlog_down
```

```
% ./comp_debug.csh  
% vlog_top_sc_down
```

vlog_top_sc_down

This next example illustrates the compile flow for Verilog-on-top with a SystemC component below it in the design hierarchy. The HDL / SystemC interface is at the pin-level. No models from the Synopsys System-Level Library are used, so no model licenses are required.

To build and run this example, follow these steps:

```
% cd vlog_top_sc_down  
% ./comp_debug.csh  
% ahb_rtl_cosim
```

This example contains a small complete virtual platform developed in SystemC which integrates an interrupt controller written as a Verilog model.

Note:

Because models from the Synopsys System-Level Library are used in this example, a model license is required.

This example illustrates the complete flow using the Virtualizer / Platform Architect MCO tools:

1. Generate a SystemC wrapper for the Verilog module using the VCS vlogan tool.
2. Import this SystemC-wrapped Verilog component into the Platform Creator Tool (PCT).
3. Build up a platform containing the imported block. Export the platform code in PCT, generating the SystemC code for the design along with its build and configuration files.

4. Instead of building the simulation, use the SystemC Shell (`scsh`) to create a makefile (`Makefile`).
 5. Create a hand-written makefile (`Makefile.vcs`) to build the simulation. This hand-written `Makefile.vcs` includes the generated one. This way, it can use all the settings and flags from the generated `Makefile`. This is important when using IP models from the Synopsys System-Level Library.
-

SystemC Restrictions

The following restrictions apply:

- RHEL32 is the only supported host platform. No current support for Solaris (Sparc and x86) or Windows.
- Simultaneous interactive debug with both DVE and Virtualizer / Platform Architect MCO debug tools (VPExplorer, VPA, or VPViewer) is not supported.
- The SystemC debug capabilities in DVE (as provided by CBug) are very limited in the above flow (when the `-sysc=snps_vp` option in VCS is set). An error is printed when one of the limitations is hit.
- VCS simulation profiling (by means of the `-simprofile` option) is not supported in the above flow (when `-sysc=snps_vp` option in VCS is set). An error is printed when this limitation is hit.
- Precompiled headers (`syscan -prec`) are not supported in the above flow (when the `-sysc=snps_vp` option in VCS is set). An error is printed when this limitation is hit.

- The VCS save/restore process is not supported in the above flow (when the `-sysc=snp_s_vp` option in VCS is set). DVE checkpointing is also not supported. VCS dies not generate an error message when you try these features with this flow, but the simulation hangs.

Generating Profile Reports for SystemC Designs

VCS now enables you to generate profiling reports for SystemC designs. Traditionally, identifying the time taken by the SystemC design during a simulation run was a problem. In the previous release, VCS provided the complete time taken by your SystemC design. VCS now extends this support by providing granular detail of the time taken by your SystemC design.

This chapter outlines the support for profiling and the use model in the following sections:

- “[Time Profiling](#)” on page 430
- “[Memory Profiling](#)” on page 432
- “[Profiler Example](#)” on page 434
- “[Profile Report Limitations](#)” on page 439

Time Profiling

Time profiling for SystemC is supported with the VCS `-simprofile` option. CPU time consumed by the SystemC model is listed in the overall profile report. This section describes how to create the profile, what kind of data is reported, and what the limitations are.

Enabling Time Profiling at Compile-time

Use the `-simprofile` compile-time option on the VCS elaboration command line.

```
% vcs -simprofile <other_VCS_options>
```

Using Time Profiling at Runtime

When you run `simv`, VCS generates a profile report. To view the report in text format, use the following command:

```
% profrpt -old -view time_all -format text -filter 0 \
simprofile_dir
```

The following different views are available in text format inside the `profileReport` directory:

6. `TimeConstructView.txt` — This is the construct view. It gives the time consumed by constructs, along with the modules that own the constructs. The main constructs for SystemC are SC processes, which could be of `SC_METHODS`, `SC_THREADS` or `SC_CTHREADS`.
7. `TimeModuleView.txt` — This is the module view. It gives details about all constructs in that particular module. If a module `bot` has two processes and is instantiated three times, there should be six processes for module `bot` in this view.
8. `TimeInstanceView.txt` — This is the instance view. It gives the hierarchical path name of the instance and the total inclusive and exclusive time for that instance. This view also displays the module corresponding to the module instance.

9. TimeSC-Spawn-OverHeadView.txt – This view provides the total time spent in SystemC overhead. The three main categories are:
 - SC-Value-OverHead — Shows the time spent in value exchanges between HDL and SystemC and vice-versa.
 - SC-Kernel-OverHead — Shows the time spent synchronizing between HDL and SystemC, VPD dumping and so on.
 - SC-Spawn-OverHead — Shows overhead relating to all the spawned processes (dynamic processes created with the `sc_spawn()` feature) in the entire SystemC design.
10. TimeSummaryView.txt — This is the summary view. It gives two numbers for SystemC:
 - Total time for SystemC.
 - Total time spent in SystemC overhead (which is a total of the time spent in all SC- Value, SC-Kernel, and SC-Spawn overheads).

Memory Profiling

Memory profiling for SystemC is limited. The memory report does not show all the memory that is allocated by SystemC objects. The entire memory allocated during SystemC elaboration is excluded from the report, including:

- Memory allocated for `SC_THREAD` and `SC_CTHREAD` stacks
- Memory allocated by SystemC module constructors
- Memory allocated during `end_of_elaboration()` methods

Only memory allocated during execution of SystemC processes is reflected in the simprofile report. To see the memory profiling report, use the following compile-time and runtime options.

Enabling Memory Profiling at Compile-time

Specify the `-simprofile` compile-time option on the VCS elaboration command line.

```
% vcs -simprofile <other_VCS_options>
```

Using Memory Profiling at Runtime

To use memory profiling at runtime, specify the following options:

```
% simv2 -simprofile mem  
% profrpt -view mem_all -format text simprofile_dir
```

VCS generates the following different report views in text format inside the profileReport directory:

1. `PeakMemConstrView.txt` — Provides the memory consumed by constructs, along with the modules that own the constructs. The main construct for SystemC is SC processes, which could be methods, threads, or cthreads.
2. `PeakMemModuleView.txt` — This is the module view. It gives details about all the constructs in that particular module. If a module `bot` has two processes and it is instantiated three times, there should be six processes for module `bot` in this view.

3. PeakMemInstanceView.txt — This is the instance view. It shows the hierarchical path name of the instance and the total inclusive and exclusive memory consumed for that instance. The module corresponding to the module instance is also displayed.
 4. PeakMemSummaryView.txt — This is the summary view. It shows total memory consumed by SystemC.
-

Profiler Example

Consider the code shown in [Example 17-1](#).

Example 17-1 Example Code for Profiler

```

Verilog
      module test_vl;
      integer a;
      integer b;
      wire [31:0] c;

      top top(.inp1(a), .inp2(b), .outp(c));
      always begin
          #10 a = 1;
          #20 a = 0;
          #1   b = 5;
          #1   b = 6;
      end
      initial begin
          #10000 $finish;
      end
      always @(a)
      begin
          $display("Initial block from verilog for a\n");
      end
      always @(b)
      begin
          $display("Initial block from verilog for b\n");
      end
  
```

```

endmodule
SystemC
Top.h
#include <systemc.h>
#include "subtractor.h"
SC_MODULE(top)
{
    SC_CTOR(top) :
        subtracter_inst1( "subtractor_inst1"),
        subtracter_inst2( "subtractor_inst2")
    {
        subtracter_inst1.ina(inp1);
        subtracter_inst1.inb(inp2);
        subtracter_inst1.outx(outp);

        subtracter_inst2.ina(inp2);
        subtracter_inst2.inb(inp1);
        subtracter_inst2.outx(outp);

        SC_METHOD(m1);
        sensitive << E1 << E2;
    }
    void m1();

    subtracter subtracter_inst1;
    subtracter subtracter_inst2;
    sc_in <int> inp1;
    sc_in <int> inp2;
    sc_out <int> outp;
    sc_event E1,E2;
};

Subtracter.h
#include "systemc.h"
SC_MODULE(subtracter)
{
public:
    sc_in<int> ina;
    sc_in<int> inb;
    sc_out<int> outx;

    SC_CTOR(subtracter):ina("ina"),inb("inb"), outx("outx")
}

```

```

    {
        SC_METHOD(m2) ;
    }
    void m2();

    sc_signal<int> value1;
    sc_signal<int> value2;
    sc_signal<int> mult_out;

};

#endif

```

Subtracter.cpp

```

#include "subtracter.h"
void subtracter::m2()
{
    static int pol=0;
    unsigned long long int i = 0;
    cout << "Inside method m2 \n";
    for (i = 0; i<=1000000000; i++) {
        pol++;
    }
}

```

Top.cpp

```

void top::m1()
{
    static int pol=0;
    unsigned long long int i = 0;
    cout << "Inside method m1 \n";
    for (i = 0; i<=1000000000; i++) {
        pol++;
    }
}

```

When you run the profiler on the code shown in [Example 17-1](#), you get the following views in the profile report (see [Figure 17-1](#)).

Figure 17-1 Profile Report

```

profileReport/TimeConstructView.txt
#####
VCS build date: Mar 26 2011 22:56:02
Compiler version: G-2012.09
Runtime version: G-2012.09
Machine Name: vgamddual150
Profile runner: msubbu
Profile data: ./simprofile_dir
Creation date: Mon Mar 28 02:34:45 2011
Profile start: 0
Command: "profrpt -view time_all -format text -filter 0 simprofile_dir"
Simulation Time: 13.7
#####
=====

Time Construct View
=====
Construct %TotalTime Module/Program Definition
/Architecture
-----
SC-Process 31.60 subtracter N/A:0
SC-Process 31.49 top N/A:0
SC-Process 31.49 subtracter N/A:0
Always 0.09 test_vl ./test.v:22
Always 0.01 test_vl ./test.v:18
Always 0.00 test_vl ./test.v:9
Initial 0.00 test_vl ./test.v:15
Initial 0.00 top Internal/Unknown source file:0
ContAssign 0.00 top ./__vcs_internal_sc_hdl_wrap__:2
Port 0.00 top ../csrc/sysc/top/top.v:7
Port 0.00 top ../csrc/sysc/top/top.v:8

profileReport/TimeInstanceView.txt
#####
VCS build date: Mar 26 2011 22:56:02
Compiler version: G-2012.09
Runtime version: G-2012.09
Machine Name: vgamddual150
Profile runner: msubbu
Profile data: ./simprofile_dir
Creation date: Mon Mar 28 02:34:45 2011
Profile start: 0
Command: "profrpt -view time_all -format text -filter 0 simprofile_dir "
Simulation Time: 13.7
#####
=====

Time Instance View
=====
Instance %TotalTime Incl|Excl Module/Program Definition
/Architecture
-----
test_vl 94.69|0.10 test_vl ../test.v:1
test_vl.top 94.58|0.00 top ../test.v:8
test_vl.top.test_vl.
    top 94.58|31.49 top Internal/Unknown source file:0
test_vl.top.test_vl.
    top.
    subtracter_inst 31.60|31.60 subtracter Internal/Unknown source file:0
test_vl.top.test_vl.
    top.
    subtracter_inst_0 31.49|31.49 subtracter Internal/Unknown source file:0
source file:0

profileReport/TimeModuleView.txt
#####
VCS build date: Mar 26 2011 22:56:02
Compiler version: G-2012.09
Runtime version: G-2012.09
Machine Name: vgamddual150

```

```

Profile runner:                               msubbu
Profile data:          ..../simprofile_dir
Creation date:           Mon Mar 28 02:34:45 2011
Profile start:            0
Command: "profrpt -view time_all -format text -filter 0 simprofile_dir"
Simulation Time:        13.7
#####
=====  

Time Module/Construct View  

=====  

63.09% subtracter (0) [N/A:0]
    31.60% SC-Process: m2
        [N/A:0]
    31.49% SC-Process: m2
        [N/A:0]

31.49% top (0) [N/A:0]
    31.49% SC-Process: m1
        [N/A:0]

0.10% test_vl (1) [./test.v:1]
    0.09% Always: NoName
        [./test.v:22]
    0.01% Always: NoName
        [./test.v:18]
    0.00% Always: NoName
        [./test.v:9]
    0.00% Initial: NoName
        [./test.v:15]

0.00% top (1) [./csrc/sysc/top/top.v:7]
    0.00% Initial: NoName
        [Internal/Unknown source file:0]
    0.00% ContAssign: NoName
        [./__vcs_internal_sc_hdl_wrap__:2]
    0.00% Port: inp1
        [./csrc/sysc/top/top.v:7]
    0.00% Port: inp2
        [./csrc/sysc/top/top.v:8]
    0.00% Port: outp
        [./csrc/sysc/top/top.v:9]
#####
VCS build date:                           Mar 26 2011 22:56:02
Compiler version:             G-2012.09
Runtime version:              G-2012.09
Machine Name:                      vgamddual150
Profile runner:                     msubbu
Profile data:          ..../simprofile_dir
Creation date:           Mon Mar 28 02:34:45 2011
Profile start:            0
Command: "profrpt -view time_all -format text -filter 0 simprofile_dir"
Simulation Time:        13.7
#####
=====  

Time Summary View  

=====
      Component          Percentage
-----
HSIM                  0.12%
DEBUG                 0.03%
SystemC                0.25%
KERNEL                 4.88%
Module                  0.10%
VERILOG                 0.10%
SystemC                94.79%
-----
TOTAL                  100.16%

```

SystemC Features

Profile Report Limitations

The following limitations apply:

- Memory profiling is not fully supported for SystemC Simprofile. However, memory allocated at simulation time is reported as mentioned above.
- No source file or line number information is available for SystemC. Therefore, line numbers are shown as 0 and source file names are shown as N/A.
- There is only one global bucket for spawn processes and the value exchange related overhead corresponding to the different module instances.

Time profiling for the SystemC component is not reported if the TLI adapter is used between SystemC and SystemVerilog.

Compiling SystemC Designs in Partition Compile Flow

You can now compile your SystemC designs with VCS in the partition compile flow. Previously, SystemC-on-top designs were not supported in the partition compile flow. If your design has SystemC modules on top and Verilog or VHDL instantiated within SystemC, you can compile your design using the `-sysc=unihier` elaboration switch. This section provides more details about how to use SystemC designs in the partition compile flow, in the following subsections:

- “[Using SystemC with Partition Compile](#)” on page 440
- “[SystemC Partition Compile Example](#)” on page 440

- “Partitioning your Design” on page 442
 - “SystemC Partition Compile Limitations” on page 443
-

Using SystemC with Partition Compile

Compile designs with SystemC modules on top and Verilog or VHDL instantiated within SystemC using the `-sysc=unihier` elaboration switch.

You can also set the `SYSC_USE_SKELETON` environment variable to `1` to compile your SystemC design in default mode (`vcs -sysc`). For example:

```
% setenv SYSC_USE_SKELETON 1
```

With this variable set, you may not need to specify `-sysc=unihier` explicitly.

SystemC Partition Compile Example

The code shown [Example 17-2](#) is first explained in terms of compiling your SystemC design and then in terms of how to partition your design and then compile.

Consider a small code snippet ([Example 17-2](#)) that contains a top-level module named `sc_top.h`. There are two child modules: a Verilog module `vlog_mod.v` and a SystemC module `sc_mod.h`.

Example 17-2 System C Partition Compile Example Code

```
//vlog child vlog_mod
module vlog_mod{.....}
```

```

//SystemC child "sc_mod"
SC_MODULE(sc_mod) {....};

//SystemC top module sc_top
SC_MODULE(sc_top) {
    //instantiate vlog_mod and sc_mod here
    vlog_mod vlog_mod_o;
    sc_mod sc_mod_o;
    SC_CTOR(sc_top) : vlog_mod_o("vlog_mod_o"),
    sc_mod_o("sc_mod_o") {.....};
}

int sc_main(int argc, char** argv) {
    " sc_top sc_top_o("sc_top);
    sc_start(100,SC_NS)
}

```

To compile the code shown in [Example 17-2](#), follow these steps:

1. Analyze the Verilog child module:

```
% vlogan -sysc sc_model vlog_mod vlog_mod.v
```

2. Compile the SystemC module:

```
% syscan sc_mod.cpp sc_top.cpp
```

3. Elaborate the design:

```
% vcs -sysc -sysc=unihier sc_main
```

Note that all SystemC-top designs start with a user-written `sc_main()` function. `sc_main` is a C function and not a SystemC module instance. This function is usually not part of the reported instance hierarchy. However, there are situations in the SystemC unihier flow where it is necessary to report `sc_main()` as part of the hierarchy. If you want your top-level module as `sc_main`, use the option `-sysc=show_sc_main`.

Partitioning your Design

To partition the design shown in [Example 17-2 on page 440](#), use a partition configuration file. This is something you need to write yourself using the following guidelines. [Example 17-3](#) shows a partition configuration file for [Example 17-2](#).

Example 17-3 SystemC Partition Compile Configuration File

```
//partcomp config file partcomp_cfg.v
config partcomp_cfg;
    design VCS_SYSC_LIB.sc_main;
    partition instance sc_main.sc_top.vlog_mod_o use
        DEFAULT.vlog_mod;
    default liblist DEFAULT;
endconfig
```

In [Example 17-3](#), `partcomp_cfg.v` is the name of the configuration file. Your configuration file can be any .v file. The configuration file must start with `config <config_modulename>`; and end with `endconfig`. The module name in [Example 17-3](#) is `partcomp_cfg`.

To compile the code shown in [Example 17-3](#) in the partition compile flow, follow these steps:

1. Analyze the Verilog child module:

```
% vlogan -sysc sc_model vlog_mod vlog_mod.v
```

2. Compile the SystemC module:

```
% syscan sc_mod.cpp sc_top.cpp
```

3. Analyze the configuration file:

```
% vlogan partcomp_cfg.v
```

4. Elaborate the design:

```
% vcs -sysc -sysc=unihier partcomp_cfg -partcomp \
-partcomp_dir="VCS_SYSC_LIB"
```

Note:

You generally specify the top-level module name to VCS in the UUM flow. Here, you must specify the configuration module name to VCS. In step 4, your configuration module name is `partcomp_cfg`.

SystemC Partition Compile Limitations

There are limitations for the SystemC `unihier` flow. It is generally available only for designs that have SystemC on top of the hierarchy and HDL instantiated below SystemC. The limitations are:

- SystemC `unihier` flow is not available for designs that have VHDL or Verilog on top and instantiate SystemC below Verilog or VHDL.
- Not available for donut designs (Verilog-SystemC-Verilog).
- Only available with UUM flow, not with old use model.

Integrating SystemC with Innovator

This section describes the integration of SystemC with Innovator. Innovator is an integrated virtual platform development environment provided by Synopsys. It supports virtual platform assembly from SystemC TLM-2 hardware models and software development on top of it.

This section contains the following topics:

- “[Introduction to Integrating SystemC with Innovator](#)” on page 444
- “[Single SystemC Compile Flow Enabling RTL Swap-in](#)” on page 445
- “[Platform Analyzer-Controlled Debug with Limited DVE Functionality](#)” on page 446

Introduction to Integrating SystemC with Innovator

The integration of SystemC with Innovator provides debugger integration that is focused mainly on Platform Analyzer. This feature enables interactive Platform Analyzer functionality in the context of VCS simulation. This allows VCS to use the Innovator SystemC kernel and to retain VCS/DVE HDL visualization capabilities.

This integration is useful when you are using Platform Analyzer for debugging and want to add an HDL model and/or VMM testbench. This is the case when you start with a full virtual platform (pure Innovator), and then replace one or more models with HDL.

A second possibility supported by this feature is when you are using Innovator-SystemC debugging capabilities (for example, in architectural tradeoffs), and when you want to incorporate an HDL model and use Platform Analyzer’s debug features.

SystemC source files can be compiled such that the object files can be used for Innovator, as well as VCS. This avoids the need to compile the same source file again.

This feature is divided into the following sub-features:

- Single SystemC Compile Flow Enabling RTL Swap-in
- Platform Analyzer-Controlled Debug with Limited DVE Functionality

Single SystemC Compile Flow Enabling RTL Swap-in

This sub-feature deals with the compile flow. Both VCS and Innovator have SystemC kernels, but they are binary incompatible, and also contain different sets of debug hooks.

This is addressed by adding a new SystemC version to Vcs-SystemC. This new version is targeted to the SystemC kernel from Innovator. All Vcs-SystemC libraries, such as `libbf*.a` that are shipped with VCS are compiled with headers from Innovator SystemC. Innovator-SystemC is not shipped with VCS; it is shipped only with Innovator.

Compilation or Elaboration Flow with Vcs-SystemC

The script `syscan` accepts a new SystemC version `inno` (innovator), as shown below:

```
% syscan -sysc=inno myfile.cpp
% vlogan -sysc=inno v_down.v -sc_model v_down
% vcs -sysc=inno top
```

As usual, `syscan` checks the consistency of the SystemC version used to compile object files. It takes the actual SystemC header files from Innovator as defined by some environment variable. The SystemC kernel is also taken from Innovator.

You must set the `$VIRTIO_HOME` environment variable for Vcs-SystemC.

VCS-SystemC uses the following option or compile flag to find the include path:

```
-I${VIRTIO_HOME}/linux/innovator/gcc-4.2.2/inno-2.2.0/  
systemc-2.2.0/include
```

VCS-SystemC uses the following option or compile flag to find the library:

```
 ${VIRTIO_HOME}/linux/innovator/gcc-4.2.2/inno-2.2.0/  
systemc-2.2.0/lib-$SYSTEMC_ARCH}/libsistema.a
```

A safety check makes sure that the SystemC version from Innovator matches the one used to build Vcs-SystemC libraries.

Vcs-SystemC and innovator generate an error message when the platform is not RHEL32, and when you use:

- gcc-3.3.6 or gcc-3.4.6 (only gcc-4.2.2 is supported)
- The -sysc=201 or -sysc=21 option as an elaboration option
- The tli/tlm option
- The -sysc=newsync option. This requires additional hooks in the Virtio SystemC kernel
- The -sysc=adjust_timeres option. This requires additional hooks in the Virtio SystemC kernel

Platform Analyzer-Controlled Debug with Limited DVE Functionality

This sub-feature deals with displaying virtual platform transactions in DVE.

DVE can debug only HDL (Verilog, VHDL) in post-process mode. Therefore, the following DVE features are supported:

- Display Verilog, VHDL design hierarchy, and signal tracing, only if tracing was initiated with \$vcpluson.
- Display and analysis of transactions captured with tblog and msglog.

The following DVE features are not supported:

- Interactive control. There is no step, run, or stop
- VPD tracing of SystemC objects
- CBug
- Active statement drivers, loads in HDL

Note:

VCS generates an error message if you start simv with interactive control (simv -gui or simv -ucli).

18

Using Verification Planner HVP Interactive Editor

The interactive HVP editor helps to create a Verification Planner verification plan. You can then annotate that plan with live verification data using the Unified Report Generator (URG) tool.

This chapter takes you through the basic steps required to create a new Verification Planner verification plan. It contains the following sections:

- “Working with HVP Files”
- “Incompleteness Checks”

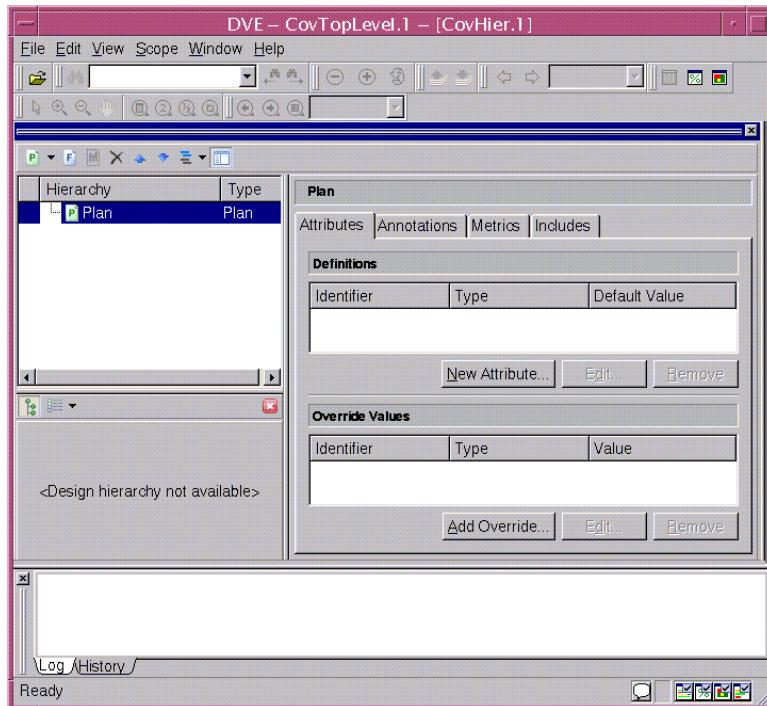
Working with HVP Files

This section describes how to work with HVP files. You create the HVP files in the DVE Coverage GUI. For more information about DVE (Discovery Visual Environment) GUI and DVE Coverage GUI, see the DVE User Guide and DVE Coverage GUI document in the VCS Online Documentation.

Creating and Editing an HVP File

To create and edit an HVP file

1. Open the DVE Coverage GUI.



2. (Optional) Click **File > Open Database** to open your functional coverage database (.vdb file).

The Open Coverage Database dialog box appears.

3. Click **File > New HVP File** to create a new file or **File > Load HVP File** to open an existing one.

If you create a new file, the root plan node appears in the HVP Hierarchy pane. In that navigation pane, you will be prompted to enter the name of the plan. You can choose the default or enter a name.

For an existing HVP file, the hierarchy is displayed.

You can also load an existing HVP file by using the -plan option to the DVE coverage command line along with the -dir option.

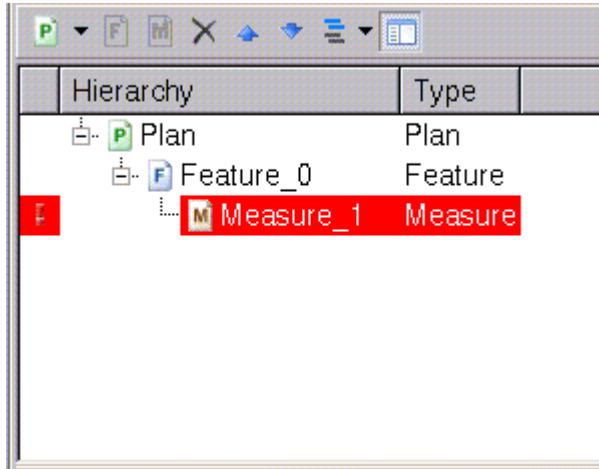
```
dve -cov -dir foo.vdb -plan plan.hvp
```

4. Do one of the following steps to add a feature:
 - Right-click any plan or feature node and select **New Feature**.
 - Select any plan or feature node and click the **F** (new feature) icon above the HVP navigation pane.

Once a feature is added you can give it any valid identifier name. You can change the name of the plan, feature, and measure at any point of time.

5. Do one of the following steps to add a measure node to the feature that you just added:
 - Right-click any plan or feature node and select **New Measure**.

- Select any plan or feature node and click the **M** (new measure) icon above the HVP navigation pane.



Once a measure is added, you can give it any valid identifier name. Unless you have multiple measure nodes attached to a feature, the name may be somewhat redundant, so you may simply want to name all of your measures with a simple name like "m".

You can change the measure at any time.

6. Select the measure and click the **Metrics** tab in the right pane to designate the metrics that you want to measure.

You can rename the measure at any time.

7. Do one of the following steps if you do not know the valid region names for your covergroups:

- Use a URG report and then copy and paste names from the URG report. Note that you can use wildcards in source strings to match more than one source region. For example, a::b::C* matches both a::b::chicago and a::b::cupertino.

- If you had loaded a coverage database, locate the covergroup in the coverage database navigation pane to the left and just below the HVP navigation pane. Once you have located a covergroup, drag and drop it into the Sources area of the measure node's middle pane. A new source entry is displayed in the Sources area.
8. Repeat steps 6-8 for each feature in your verification plan.

You can attach features to features creating an arbitrary hierarchy in your verification plan. Note that not all features need to have measure statements. Higher level features often only serve to add structure to the verification plan and those do not need measure sources, as they inherit verification metrics from their child features.

9. Click **File > Save as HVP file** to save your verification plan.

You can now use the HVP file to annotate live verification data and create a URG report. See the *Unified Coverage Reporting User Guide* for more information. You can regenerate the URG report as your verification data evolves throughout the project. There is no need to modify your verification plan unless the plan itself requires change.

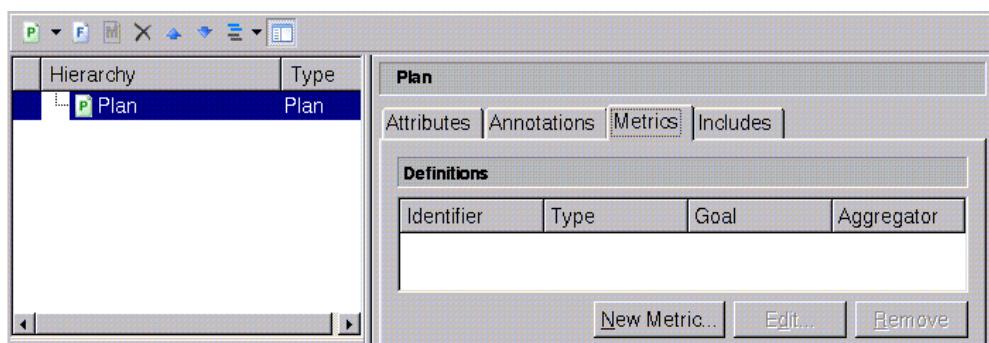
Creating User-Defined Metrics

Verification Planner allows not only built-in metrics like the Group metric used in the previous example, but also user defined metrics.

To define and reference user defined metrics in the DVE editor

1. Highlight the "Plan" (root) node of your verification plan, then click the **Metrics** tab in the right pane.

2. Click New Metric.



3. Enter the name of your new metric in the "identifier" field. For example, bugRate.
4. Select the metric type.
5. Enter a default goal for the metric. For example, "bugRate <= 0".
6. Enter the aggregator associated with this metric. The aggregator determines how the parent node's score is calculated from its child feature nodes.
7. Click **OK**.

The metric is created and saved.

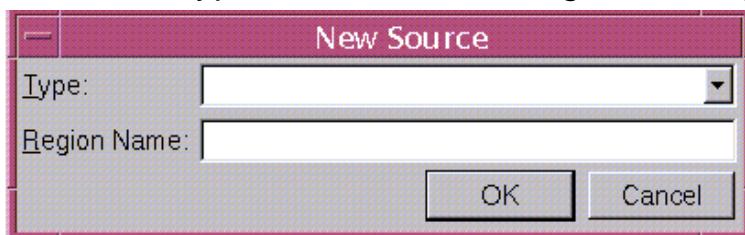
Creating Feature Nodes

Now that you have defined the metric type, you may reference it in as many feature nodes as required.

To create a feature node

1. Select the feature node.
2. Select a measure node under that feature.

3. Click the **Select** button in the metrics subwindow in the right pane of the HVP navigation pane.
4. Select the box to the left of the new metric (bugRate) and then click **OK**.
5. click **New Source** in the sources subwindow.
A pop-up window appears.
6. Clear the **Type** field (user-defined metrics have no type) and then type in the source string in the "region name" field.



Note:

- You can use wildcard characters.
- Alternatively, steps 5 and 6 can be achieved via drag and drop. You first need to create a user-defined metrics data file like this:

```
HVP metric = bugRate protocolA = 1 protocolB = 4 dmaA =
1 dmaB = 1 dmaC = 0 ...
```

and then load that file into DVE via the **File > Manage VE Data** command. Once loaded, you can browse the data in that file by clicking on the VE Data icon (labeled VE Data when you mouse over).

- The drag and drop measure source population works exactly the same as with built-in coverage data.
- When running URG, you can annotate user-defined data with the -annotate flag:

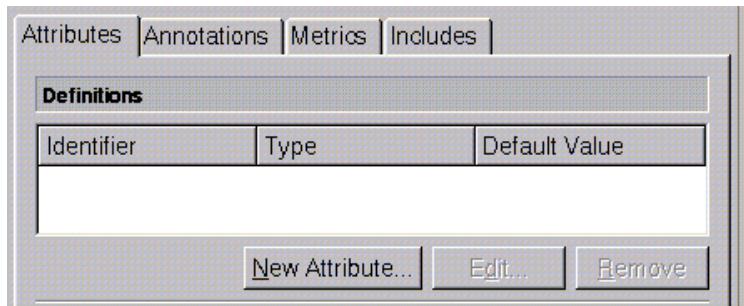
```
urg -plan yourPlan.hvp -dir yourCoverageDB.vdb -annotate  
bugRate.txt
```

Adding User-Defined Attributes

You may add as many user-defined attributes as you like to features. These attributes may be used for filtering and they are also listed on URG generated reports.

To add a new attribute

1. Select the Plan node in the HVP Hierarchy pane.
2. Select the **Attributes** tab in the right pane of the HVP Hierarchy



pane, then click **New Attribute**.

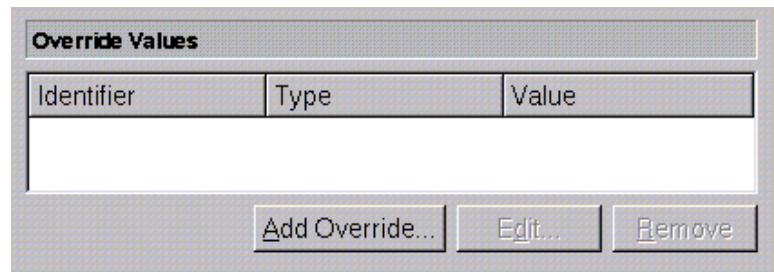
3. Enter the attribute name in Identifier.
4. Select an attribute type.
5. Enter a default value.
6. Click **OK**.

Overriding an Attribute

Now that you have defined an attribute, you may override the default value.

To override an attribute

1. Select a feature node.
2. Select the **Attributes** tab in the right pane..



3. Click **Add Override**.
4. Select the attribute name, enter its value, and click **OK**.

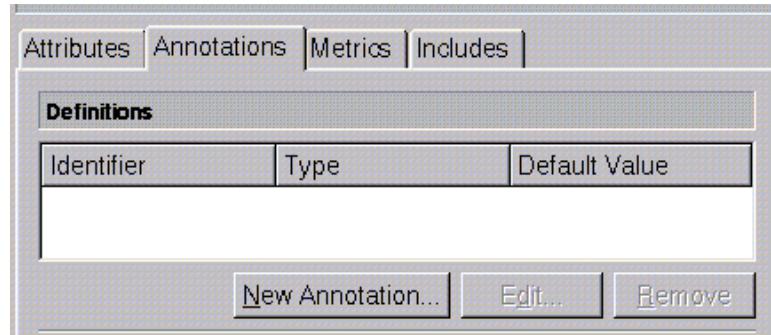
Adding Annotations

Annotations are added exactly the same as attributes except that you will be entering data in the "annotation" tabs of the appropriate nodes instead. The difference between annotations and attributes is that attributes are inherited by child features where annotations are not inherited at all.

For example, if you add an attribute to feature f1 and f1 has children f1.f11 and f1.f12, then f1.f11 and f1.f12 automatically inherit f1's attributes. However, if you add an annotation to f1, that annotation is not inherited by f1.f11 or f1.f12.

To add an annotation

1. Select the Plan node in the HVP Hierarchy pane.
2. Select the **Annotation** tab In the right pane of the HVP Hierarchy pane..



3. Enter the attribute name in Identifier.
4. Select an attribute type.
5. Enter a default value.
6. Click **OK**.

The annotation is added.

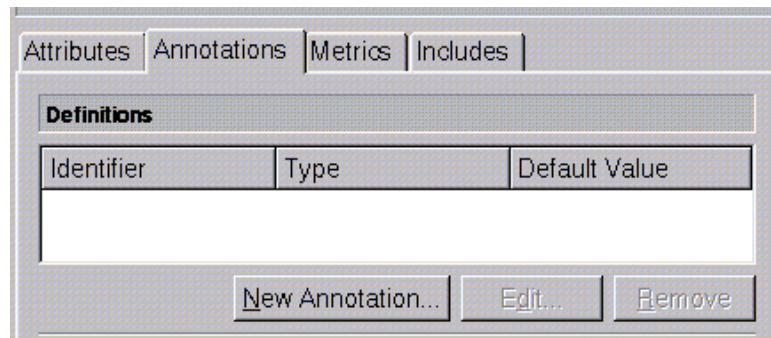
Goal Overrides

Normally, metrics have default goals that are used to color the URG report outputs. Features which meet goals will have their value cells colored green. Features which fail goals will show red.

You may override the value of any goal at any feature.

To override the goal value

1. Select a feature.
2. Select the **Metrics** tab in the right pane of the HVP Hierarchy pane.



3. Click **Add Override**.
4. Select the metric for which you would like to override the goal in the Identifier field.
5. Type in the new goal expression in the "goal" field.
6. Click **OK**.

Creating a subplan

You can create, edit, include, and remove a subplan from the top-level plan.

To create a subplan

1. Select a plan in the HVP navigation pane.
2. Click the drop-down arrow beside the Plan button above the HVP Navigation pane.

3. Select **Create a Subplan**.
4. Type a name for the subplan.

The subplan is created.

5. Select a subplan and enter the parameters, such as attribute or annotation value, in the **Definition** field in the right pane of the HVP Hierarchy pane.

The subplan is edited. The parameter that you enter in the Definition field is a comma separated list of "attribute/annotation = value" expression to override in subplan.

For example,

```
owner="QA team", phase=2, weight=2
```

The attributes/annotations that are overrided in the subplan parameter fields must be defined in the plan.

6. Click the Plan under which you have created a subplan and click the **Includes** tab in the right pane of the HVP Hierarchy pane.

7. Enter the subplan filename in the text box and click **Add**.

The subplan is included.

8. Select the subplan from the include file list and click **Remove**.

The subplan is removed.

Incompleteness Checks

DVE contains a feature that allows you to compare your verification plan against the actual coverage data. By comparing the two, you can determine whether there are any errors or inconsistencies that need to be corrected. Examples of errors and inconsistencies are:

Typos in metric source names

- Unimplemented functional coverage checks. For example, a covergroup was expected to be implemented, but not yet coded in the test bench or DUT code.
- Coverage objects not referenced in the plan. These coverage objects could be uncovered and not affect the plan results at all.

Incompleteness checks are automatically turned on when you load a verification plan and at least one coverage database or user-defined metric file. You will see the effects of the incompleteness checks in the form of orange-red marks placed to the far left of the navigation panes.

In the HVP navigation pane, a red-orange exclamation mark (!) indicates that something is wrong with the definition of a feature. You can expand that feature's child feature and measure nodes to isolate which nodes are causing the warning. If you mouse over the "!", then you should see a pop-up tooltip indicating the specific nature of the problem.

In the coverage data pane, there is a similar incompleteness mark asterisk (*) that indicates that the particular coverage object is not referenced anywhere in your verification plan. In other words, that object could assume any value and not affect the plan's results.

19

Fast Compilation

VCS has new compile-time performance optimizations called Fast Compilation that you can use to reduce compile-time for your design (and therefore overall turnaround time). You can enable Fast Compilation using the `-fastcomp` compile-time option.

There are two levels of Fast Compilation, specified by the `-fastcomp=0` and `-fastcomp=1` compile-time options.

- `-fastcomp=0` is the same as `-fastcomp` (without the argument) and is the generally preferred option.
- `-fastcomp=1` applies more aggressive compile-time performance optimizations.

20

The Unified Simulation Profiler

The unified simulation profiler reports the amount of CPU time and machine memory used by the Verilog or SystemVerilog (and for VCS MX the VHDL) part of the design. For SystemC parts the unified profiler reports just the CPU times.

This information can be in summary form and per module definition and module instance. It also can be based on constructs such as always procedures. There is a snapshot mechanism for writing additional reports when the profile data changes.

The reports written by the profrpt profile report generator. These reports can be in text or HTML files or both.

The major sections in this unified profiler documentation are as follows:

- “The Use Model”

- “HTML Profiler Reports”
- “Constraint Profiling Integrated in the Unified Profiler”
- “PLI/DPI/DirectC Enhancement in the Unified Profiler”
- “Limitations”

Fox examples of SystemC profiler reports see “[SystemC Views](#)” .

The Use Model

The use model for the unified simulation profiler is as follows:

1. Compile your Verilog design using the `-simprofile` compile-time option. The `-simprofile` option requires the `-lca` option.

Important:

If this is not the first compilation of your Verilog design, delete the `csrc` and `simv.daidir` directories and `simv` executable file before this step. Incremental compilation is not yet supported for the unified profiler.

2. At runtime you can enter the `-simprofile` runtime option with a keyword argument or sub-option to specify the type of data VCS collects during the simulation. These keyword arguments or sub-options are as follows:

`time`

Specifies collecting data on the CPU times.

`mem`

Specifies collecting data on the machine memory.

`noprof`

Tells VCS not to collect profiling information at runtime. Synopsys recommends entering this runtime option and keyword argument or sub-option instead of simply omitting the `-simprofile` runtime option. See “[Omitting Profiling at Runtime](#)”.

After simulation, but before you run the `profrpt` profile report generator there is a small amount of profile information available to you, see “[Post Simulation Profile Information](#)”.

3. Run the `profrpt` profile report generator using the `profrpt` command and its command-line options.
4. Then review the reports created by the `profrpt` profile report generator; see “[Running the profrpt Profile Report Generator](#)” for more information.

Omitting Profiling at Runtime

If you compiled the design to collect profile data, by entering the `-simprofile` compile-time option, but decide to forgo the performance cost of collecting profile data during simulation, you enter the `-simprofile noprof` runtime option and keyword argument or sub-option.

When you do, VCS does not create the `profileReport.html` file or the `profileReport` directory, but does create the `simprofile_dir` directory, however this `simprofile_dir` directory will be empty.

Omitting the `-simprofile` runtime option after compiling with the `-simprofile` compile-time option is not recommended.

If you compile your design with the `-simprofile` compile-time option, but omit the `-simprofile` runtime option when you run the simulation, VCS, by default, creates the `simprofile_dir` and `profileReport` directories and write the `profileReport.html` in the current directory that only contains information about the simulation time.

The `simprofile_dir` directory never contains any information that you can read. It does contain non-readable files that come with a performance cost.

If, when you omit the `-simprofile` runtime option when you run the simulation, the `profileReport` directory and `profileReport.html` file already exists, VCS renames the existing directory and file `profileReport.integer` and `profileReport.integer.html`, where the `integer` differentiates the old file and directory from the new ones.

Post Simulation Profile Information

After simulation, but before you run the `profprt` profile report generator, VCS provides a limited amount of profile information.

At the end of simulation VCS writes the `simprofile_dir` and `profileReport` directories and the `profileReport.html` file.

The `simprofile_dir` directory contains the databases that are read by the `profprt` profile report generator to write profile reports in a separate step after the simulation.

the `profileReport.html` file can tell you the total simulation time and the location of the profiler databases. For mixed signal and mixed HDL simulations there is component information about the CPU time of the various components.

Running the profrpt Profile Report Generator

You run the profrpt profile report generator with the `profrpt` command line. The syntax of this command line is as follows:

```
profrpt simprofile_dir -view view1 [+view2 [...]]
[-h | -help] [-format text | html | ALL] [-output name]
[-filter percentage] [-snapshot [delta | incr | delta+incr]]
[-timeline [dynamic_memory_type_or_class +...]]
```

Where:

`simprofile_dir`

Specifies the profile database directory that VCS or VCS MX writes at runtime. The default name is `simprofile_dir`. You enable the writing of this database with the `-simprofile` compile-time option and specify the kind of data in the database with the `-simprofile` runtime option.

`-view view1 [+view2 [...]]`

Specifies the views you want to see in the reports, see “[Specifying Views](#)”. You must specify this option.

`-h | -help`

Displays help information about the `profrpt` command-line options.

`-format text | html | ALL`

Specifies whether the report files are text files, HTML files, or in both formats (by specifying the `ALL` keyword). The default format is HTML.

`-output name`

Specifies the name of the directory for the profile reports and, if `profrpt` is writing HTML reports (which is the default format), the name of the HTML index file that contains hypertext links to the HTML files in that directory.

If you omit the `-output` option, the default name of the output directory is `profileReport` and the default name of the HTML index file is `profileReport.html`.

Any currently existing `profileReport` directory and `profileReport.html` file is renamed by `profrpt` to `profileReport.integer` directory and `profileReport.integer.html`. The integer value is incremented to differentiate it from the current `profrpt` output.

See “[The Output Directories and Files](#)” for more information on the `-output` option.

`-filter percentage`

Specifies the minimum percentage of machine memory or CPU time that a module, instance, or construct needs to use before `profrpt` enables reporting about it in the output views and reports. The default limit is 0.5%. For a more granular report enter a small percentage, for example: `-filter 0.0001`

`-snapshot [delta|incr|delta+incr]`

Specifies writing snapshot reports for SystemVerilog dynamic memories. It writes a snapshot report each time a dynamic memory uses a specified different amount of machine memory. For information on specifying this amount, and more on the snapshot mechanism, see “[The Snapshot Mechanism](#)”.

`-timeline [dynamic_memory_type_or_class +...]`

Specifies two things:

- Timeline reports for SystemVerilog dynamic memories.
- Snapshot reports using the default delta threshold of 5%.

If you omit the `dynamic_memory_type_or_class +...` argument or arguments, profrpt writes all the dynamic class timeline views.

For information on the keyword arguments or sub-options for specifying the types of SystemVerilog dynamic memories in the timeline reports, see “[Specifying Timeline Reports](#)”.

Specifying Views

You must enter the `-view` option on the profrpt command line.

The views you can specify with the `-view` option depend on the type of report that profrpt is writing, which depends on the argument to the `-simprofile` runtime option.

The arguments and the views they specify are as follows:

CPU Time views:

`time_summary`

To specify writing the time summary view.

`time_inst`

To specify writing the time instance view that shows the CPU time used by the various module, program, and interface instances in a design. For VHDL this view also reports the CPU time used by the various entity/architecture instances in a design.

`time_mod`

To specify writing the time module view that shows the CPU time used by the various module, program, and interface definitions in a design. For VHDL this view also reports the CPU time used by the various entity/architecture definitions in a design.

`time_constr`

To specify writing the time construct view that shows the CPU time used by constructs such as the `always` procedures.

`time_solver`

To specify generating the Time Constraint Solver view.

`time_all`

Specifies writing all the supported CPU time views.

Machine memory views:

`mem_summary`

To specify writing the peak memory summary view which is when your design used the most machine memory.

`mem_inst`

To specify writing the peak memory instance view.

`mem_mod`

To specify writing the peak memory module view.

`mem_constr`

To specify writing the peak memory construct view.

`dynamic_mem`

To specify writing the dynamic memory peak view.

`dynamic_mem+stack`

To specify writing the dynamic memory peak view, and machine memory stack traces. The stack traces can help you determine which callers consume the most memory, see “[Stack Trace Report Example](#)”

`mem_solver`

To specify generating the Memory Constraint Solver view.

`mem_all`

To specify writing all supported machine memory views. This argument also enables machine memory stack traces.

Both memory and time profiler:

`ALL`

To specify writing all supported views. The profrpt output is the HTML or text files for all these views, including machine memory stack traces.

The Snapshot Mechanism

The keyword arguments, or sub-options, that you include after the `-snapshot` option control the snapshot mechanism. They are as follows:

delta

A numerical value (not a keyword) specifying the delta threshold for another snapshot, for example `-snapshot 8.5` specifies a delta threshold of 8.5%, so profrpt writes another snapshot report when a dynamic memory uses 8.5% more machine memory or 8.5% less machine memory.

incr

A keyword specifying the generation of another snapshot only when the machine memory for a SystemVerilog dynamic memory increases by 5%.

delta+incr

Specifies another snapshot when the amount of machine memory used by a SystemVerilog dynamic memory increases, but not decreases, by the specified delta threshold.

If you enter no arguments or sub-options, the profiler uses the default delta threshold of 5%, and enables a new snapshot when the amount of machine memory used by a SystemVerilog dynamic memory increases or decreases by that 5%.

Specifying Timeline Reports

The `-timeline` option specifies writing timeline reports. The keyword arguments or sub-options that you include after the `-timeline` option specify the types of SystemVerilog dynamic memories are in the timeline reports. You can also specify a SystemVerilog class by name and it's dynamic memories are included in the timeline reports.

The arguments or sub-options for the `-timeline` option are as follows:

`vcs_ST`

keyword for string dynamic memories

`vcs_ET`

keyword for event dynamic memories

`vcs_DA`

keyword for dynamic arrays

`vcs_SQ`

keyword for queues

`vcs_AA`

keyword for associative arrays

class

a class name, not a keyword, specifying a class

`ALL`

keyword specifying all types of dynamic memories

If you enter the `-timeline` option without an argument or sub-option, `profrpt` writes timeline reports for all dynamic memories, so the keyword `ALL` as an argument or sub-option is the same as entering no argument or sub-option.

Recording and Viewing Memory Stack Traces

You can use the unified profiler to record stack traces whenever machine memory is allocated. The stack traces can help you determine which callers consume the most memory.

You enable memory stack traces with the `dynamic_mem+stack`, `mem_all`, or `ALL` arguments to the `profrpt -view` option. See “[Stack Trace Report Example](#)” .

The Output Directories and Files

The `-output name` option and argument specifies two things:

- The name of the directory for the profiler reports
- If the profiler is writing HTML reports (the default format), the name of the HTML index file that the `profrpt` writes in the current directory. This index file is `name.html`.

If you omit the `-output` option, the default name of the output directory is `profileReport` and the default name of the HTML index file is `profileReport.html`.

As explained in “[Post Simulation Profile Information](#)” VCS or VCS MX write the `profileReport` directory and HTML index file `profileReport.html` at the end of simulation. So if you omit the `-output` option, profrpt renames this directory and file `profileReport.integer` and `profileReport.integer.html` and then writes a new `profileReport` directory and `profileReport.html` file. This new directory and file contain post-processing information from the database.

If the specified directory and file already exists you will see a warning message, and the profrpt creates a new output directory and file and renames the older output `name.integer` and `name.integer.html` to differentiate them from the new directory and file.

HTML Profiler Reports

Profiler reports are by default in HTML format.

What follows are examples of these reports based on the following SystemVerilog code:

Example 20-1 Profiler SystemVerilog Code Example

```
program tb_top;

    logic [255:0]           Squeue_data_info[$];
    logic [255:0]           temp;

    class PACKET;
        rand reg [255:0] packet_val;
    endclass
```

```

initial
begin

    for(int y = 0 ; y < 1000 ; y++)
        begin
            PACKET packet_inst;

            packet_inst = new();
            packet_inst.randomize();
            #1;

            Squeue_data_info.push_back(packet_inst.packet_val);
            #1;

        end

repeat(10)
$display("DEBUG==> Pushed 1000");

for(int y = 0 ; y < 500 ; y++)
begin

    #1;
    temp = Squeue_data_info.pop_front();
    #1;

end
repeat(10)
$display("DEBUG==> Popped 500");

for(int y = 0 ; y < 10000 ; y++)
begin

    PACKET packet_inst;

    packet_inst = new();
    packet_inst.randomize();
    #1;

    Squeue_data_info.push_back(packet_inst.packet_val);

```

```

#1;

end

repeat(10)
$display("DEBUG==> Pushed 10000");

for(int y = 0 ; y < 5000 ; y++)
begin
    PACKET packet_inst_2;

#1;
    temp = Squeue_data_info.pop_front();
#1;

    end
repeat(10)
$display("DEBUG==> Popped 5000");

for(int y = 0 ; y < 100000 ; y++)
begin
    PACKET packet_inst;

packet_inst = new();
packet_inst.randomize();
#1;

Squeue_data_info.push_back(packet_inst.packet_val);
#1;

end

repeat(10)
$display("DEBUG==> Pushed 100000");

for(int y = 0 ; y < 50000 ; y++)
begin
    PACKET packet_inst_2;

#1;

```

```

        temp = Squeue_data_info.pop_front();
        #1;

    end
repeat(10)
$display("DEBUG==> Popped 50000");

for(int y = 0 ; y < 1000000 ; y++)
begin
    PACKET packet_inst;

    packet_inst = new();
    packet_inst.randomize();
    #1;

    Squeue_data_info.push_back(packet_inst.packet_val);
    #1;

end

repeat(10)
$display("DEBUG==> Pushed 1000000");

for(int y = 0 ; y < 500000 ; y++)
begin
    PACKET packet_inst_2;

    #1;
    temp = Squeue_data_info.pop_front();
    #1;

end
repeat(10)
$display("DEBUG==> Popped 50000");

$finish;

end

```

```
endprogram
```

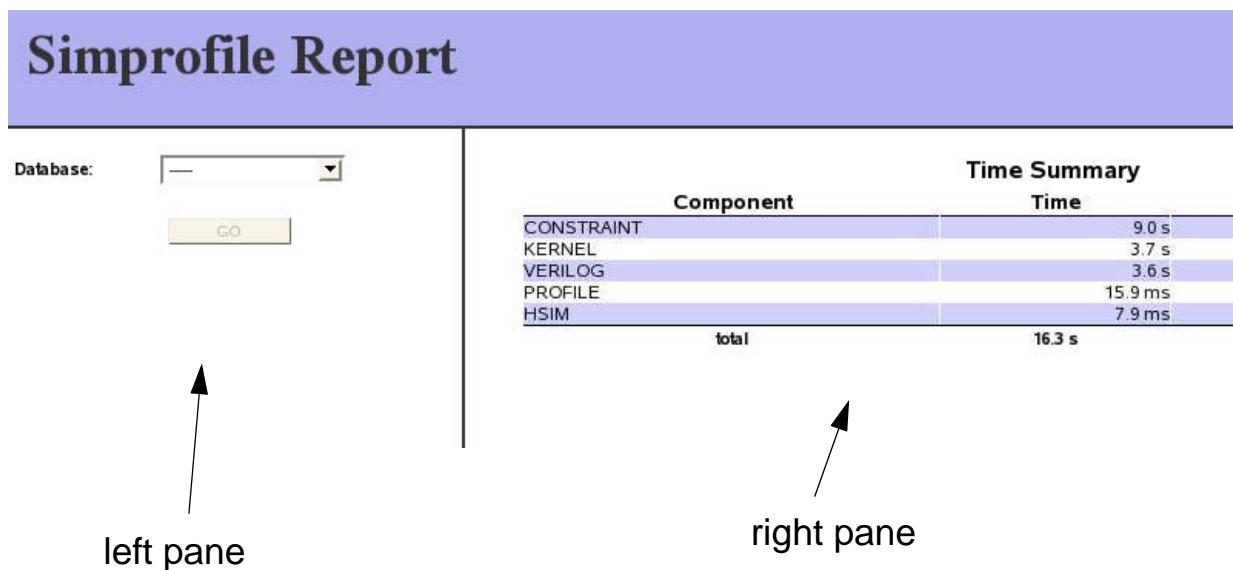
This code was compiled and simulated for CPU time profile information with the following command lines:

```
vcs smart_queue.v -lca -simprofile -sverilog  
simv -simprofile time
```

The profrpt command line was as follows:

```
profrpt simprofile_dir -view time_all -timeline ALL
```

Figure 20-1 The *profileReport.html* File for CPU Time Profile Information



The *profileReport.html* file contains two panes:

- The left pane is for specifying the profile database and the view you want to see.

- The right pane is for displaying profile information.

Figure 20-2 The Left Pane of the simprofileReport.html file

Database:

The Database: field is a pulldown menu. We can select the only database in this example so far, the simprofile_dir directory. Doing so adds the View: field to the left pane and the default view, which in this case is the Time Summary view. Then click the GO button.

Figure 20-3 The Left Pane of the simprofileReport.html file

Database:

View:

Figure 20-4 The Right Pane of the simprofileReport.html file for CPU Time Summary Information

Time Summary View		
Component	Time	Percentage
CONSTRAINT	9.03 s	55.40 %
KERNEL	3.65 s	22.42 %
VERILOG	3.59 s	22.03 %
Program	3.59 s	22.03 %
total	16.30 s	100%

Components, in this case, are consumers of CPU time during simulation. The components in this example are as follows:

CONSTRAINT

The CPU time needed to solve and simulated SystemVerilog constraint blocks.

Also the CPU time used for calls to the `randomize()` method, like in this example, are included in this component. These calls to `randomize()` are taking most of the CPU time reported for this component, and this component used most of the CPU time.

KERNEL

The CPU time used by the VCS or VCS MX kernel. This CPU time is separate from the CPU time needed to simulated your Verilog or SystemVerilog, VHDL, SystemC, or C or C++ code for your design and testbench.

VERILOG

The CPU time VCS or VCS MX needed to simulate this example's SystemVerilog code, which is a program block. For Verilog and SystemVerilog there are sub-components. In this example there is only one sub-component named Program.

This example consists of a SystemVerilog program block that used 22.03% of the CPU time.

Possible other sub-components are Module, Interface, UDP, and Assertion, for the CPU time used by Verilog and SystemVerilog definitions for module, interface, user-defined primitive, package and assertion.

Other possible components are as follows:

DEBUG

The CPU time VCS or VCS MX needed to simulate this example with the debugging capabilities of DVE and the UCLI or to write a simulation history VCD or VPD file.

Value Change Dumping

The CPU time VCS or VCS MX needed to write a simulation history VCD or VPD file. This component is always accompanied by the DEBUG component. This component has the following sub-components:

VPD

The CPU time VCS or VCS MX needed to write a VPD file.

VCD

The CPU time VCS or VCS MX needed to write a VCD file.

VHDL

For VCS MX only, the CPU time needed to simulate a design's VHDL code.

PLI/DPI/DirectC

The CPU time VCS or VCS MX needed to simulate the C/C++ in a PLI, DPI, or DirectC application.

HSIM

This is about the CPU time used by HSOPT optimizations.

COVERAGE

The CPU time needed for functional coverage (testbench and assertion coverage). Code coverage is not part of this component.

SystemC

The CPU time needed for SystemC simulation.

If we select the Time Module view in the View: field in the left pane, then click the GO button again, the right pane changes to show this view.

Figure 20-5 The CPU Time Module View

Time Module View		
Module	Time	Percentage
tb_top	3.59 s	22.03 %
total	3.59 s	22.03 %
Page: 1		

click here

As explained earlier, modules not only include Verilog and SystemVerilog modules, but can also include SystemVerilog programs and interfaces, and for VHDL, entity/architectures.

In this example program block tb_top used 3.59 seconds of CPU time, which was 22.03% of the CPU time used by the simulation.

The program name is a hypertext link to expand the display in this view. If we click on it we see the scopes inside the program block.

Figure 20-6 The Expanded CPU Time Module View

Time Module View		
Module	Time	Percentage
tb_top	3.59 s	22.03 %
NoName	3.45 s	21.16 %
total	3.59 s	22.03 %

Page: 1

In this example the scopes inside the program block are begin-end blocks that are not named, so profprt calls them all NoName. These begin-end blocks use most of the CPU time used by the program block. In this example NoName is not a hypertext link.

Other possible scopes inside a module are fork-join blocks and user-defined tasks and functions.

If we select the Time Construct view in the View: field in the left pane, then click the GO button again, the right pane changes to show this view.

Figure 20-7 The CPU Time Construct View

Time Construct View		
Name	Time	Percentage
Initial	3.45 s	21.16 %
total	3.51 s	21.55 %
Page: 1 click here		

In this example the only construct is an initial block. This initial block uses 21.16% of the CPU time.

For Verilog and SystemVerilog, the constructs in this view can include initial procedures, always procedures (including the SystemVerilog always procedures such as `always_comb`), SystemVerilog final procedures, user-defined tasks, and user-defined functions.

For VHDL the constructs in this view are processes in architectures.

The initial keyword is a hypertext link to expand the display in this view. If we click on it we see the scopes inside the initial procedure.

Figure 20-8 The Expanded CPU Time Construct View

Time Construct View		
Name	Time	Percentage
▼ Initial	3.45 s	21.16 %
NoName	3.45 s	21.16 %
total	3.51 s	21.55 %

Page: 1

In this example the scopes inside the initial procedure are begin-end blocks that are not named, so the profiler calls them all NoName. These begin-end blocks use most of the CPU time used by the program block. In this example NoName is not a hypertext link.

If we select the Time Instance view in the View: field in the left pane, then click the GO button again, the right pane changes to show this view.

Figure 20-9 The CPU Time Instance View

Time Instance View					
Instance	Inclusive Time	Percentage	Exclusive Time	Percentage	
► tb_top	3.59 s	22.03 %	3.59 s	22.03 %	
total	3.59 s	22.03 %	3.59 s	22.03 %	

Page: 1

This view shows CPU times and percentages for the instances in the design. These are instances of Verilog and SystemVerilog modules and also instances of SystemVerilog interfaces and VHDL entity/architectures.

This view shows for an instance the inclusive and exclusive time and percentage values.

The inclusive time and percentage is for the percentage of CPU time used by this instance and all instances that are hierarchically under it in the design hierarchy.

The exclusive time and percentage is for the CPU time used by this instance alone, not counting the instances that are hierarchically under this instance.

In this example there is only one instance of program tb_top, so the inclusive and exclusive values are the same, which are 3.59 seconds and 22.03% of the CPU time.

The instance name tb_top is not a hypertext link.

In this example there is no PLI, DPI, or DirectC code so there is no information in the Pli/DPI/DirectC view. There also is no information in the Dynamic Timeline view because this view is for machine memory information and we do not collect machine memory profile information in the profile database.

We can now simulate for machine memory profile information:

```
simv -simprofile mem
```

The profrpt command line is as follows:

```
profrpt simprofile_dir -view mem_all -timeline ALL
```

The `-timeline` option specifies snapshot reports.

The profile report generator, profrpt, rewrites the `profileReport.html` file for machine memory information, so re-open this file.

In the left pane, in the Database: field select again the `simprofile_dir` profile database directory. Doing so adds the following fields to the left pane:

- the View: field which is at the default selection of the Memory Summary view
- the Snapshot: field which is at the default selection of the peak machine memory snapshot, in this example the 33rd snapshot at simulation time 2000000.

Then click the GO button.

Figure 20-10 The Machine Memory Summary View for the Peak Snapshot

simulation time of the peak snapshot

Component	Size	Percentage
VERILOG	116.16 MB	73.93 %
Program	116.14 MB	73.92 %
Package	658.48 KB	0.41 %
KERNEL	29.99 MB	19.08 %
CONSTRAINT	8.96 MB	5.70 %
HSIM	2.01 MB	1.28 %
COVERAGE	316 B	0.00 %
total	157.11 MB	100%

Components, in this view, are consumers of machine memory during simulation. This view reports the amount machine memory used by each component and their percentage of the total machine memory used in the snapshot. In this example this is the peak snapshot.

In this snapshot the preponderance of machine memory is used by the VERILOG and KERNEL components.

The components in this view are as follows:

VERILOG

The machine memory VCS or VCS MX needed to simulate this example's SystemVerilog code, which is a program block, at the peak snapshot. There are the following sub-components:

Program

The machine memory needed to simulate the SystemVerilog program block in the code example.

Package

Usually the machine memory needed to simulate a SystemVerilog package.

In this case this is an anomaly, reporting a small amount of machine memory for a package when there is no package in the code example. You can ignore these anomalies.

Possible other sub-components are Module, Interface, UDP, and Assertion.

KERNEL

The machine memory used by the VCS or VCS MX kernel. This is separate machine memory from the machine memory needed to simulate the code in the code example.

CONSTRAINT

The machine memory needed to solve and simulate SystemVerilog constraint blocks, but also counted in this component are calls to the `randomize()` method.

HSIM

This is about the machine memory used by HSOPT optimizations.

COVERAGE

This component is for functional coverage (SystemVerilog testbench or assertion coverage). A small percentage of machine memory is reported here even though there is no functional coverage code in the design. This is the machine memory needed for functional coverage enabling optimizations, which are default optimizations.

Code coverage is not reported in this component. The machine memory used for code coverage is in the VERILOG (or VHDL) component(s).

Other possible components, were there different source code and compile-time and runtime options, are as follows:

DEBUG

Like for CPU time, this component is for the machine memory VCS or VCS MX needed to simulate this example with the debugging capabilities of DVE and the UCLI or to write a simulation history VCD or VPD file.

PLI/DPI/DirectC

Like for CPU time, this component is for the machine memory VCS or VCS MX needed to simulate the C/C++ code in a design.

SystemC

The machine memory needed for SystemC simulation.

In VCS MX, for VHDL and mixed-HDL designs, there is an additional possible component:

VHDL

The machine memory needed to simulated a design's VHDL code.

So far we have looked at the machine memory summary view for the peak snapshot. There is a summary view for other snapshots.

For example, if we select the 10th snapshot.

Figure 20-11 Selecting the 10th Snapshot

Database:	<input type="text" value="simprofile_dir"/>
View:	<input type="text" value="Memory Sumr"/>
Snapshot:	<input type="text" value="#10 (clock:193)"/>
<input type="button" value="GO"/>	

Then click the GO button, the right pane shows the machine memory summary view for this snapshot.

Figure 20-12 The Machine Memory Summary View for the 10th Snapshot

Memory Summary View (clock:193131)		
Component	Size	Percentage
KERNEL	29.99 MB	67.64 %
VERILOG	11.59 MB	26.14 %
Program	11.57 MB	26.09 %
Package	153.64 KB	0.34 %
HSIM	2.01 MB	4.53 %
CONSTRAINT	768.03 KB	1.69 %
COVERAGE	316 B	0.00 %
total	44.33 MB	100%

This view shows the machine memory used by the various components in the 10th snapshot.

Now, back in the left pane, we can return to the peak snapshot, the 33rd, in the Snapshot: field and select the Memory Module view in the View: field, then click the GO button. The right pane changes to the machine memory module view for the peak snapshot.

Figure 20-13 The Machine Memory Module View for the Peak Snapshot

Memory Module View (clock:2000000)		
Module	Size	Percentage
tb_top	116.14 MB	73.92 %
total	116.78 MB	74.33 %
Page: 1		

Page: 1

click here

As explained earlier, modules not only include Verilog and SystemVerilog modules, but can also include SystemVerilog programs and interfaces, and for VHDL, entity/architectures.

In this example program block tb_top used 116.14 MB of machine memory, which is 74.33% of the machine memory used to simulate the peak snapshot.

The program name is a hypertext link to expand the display in this view. If we click on it we see the scopes inside the program block.

Figure 20-14 The Expanded Machine Memory Module View for the Peak Snapshot

Memory Module View (clock:2000000)		
Module	Size	Percentage
tb_top	116.14 MB	73.92 %
NoName	116.14 MB	73.92 %
total	116.78 MB	74.33 %

Page: 1

In this example the scope inside the program block is a begin-end blocks that is not named, so profrpt calls it NoName. This begin-end block uses most of the machine memory used by the program block. In this example NoName is not a hypertext link.

Other possible scopes inside a “module” are fork-join blocks and user-defined tasks and functions.

Like the machine memory summary views, there is a machine memory module view for each snapshot.

If, in the left pane, we select the Memory Constant view and then click the GO button, the right pane changes to the machine memory construct view for the peak snapshot.

Figure 20-15 The Machine Memory Construct View for the Peak Snapshot

Memory Construct View (clock:2000000)		
Name	Size	Percentage
Initial	116.14 MB	73.92 %
Total	116.14 MB	73.92 %

Page: 1

click here

In this example the only construct is an initial block. This initial block uses, at the peak snapshot, 116.14 MB of machine memory, which is 73.92% of the total machine memory use at the peak snapshot.

For Verilog and SystemVerilog, the constructs in this view can include initial procedures, always procedures (including the SystemVerilog always procedures such as `always_comb`), SystemVerilog final procedures, user-defined tasks, and user-defined functions.

For VHDL the constructs in this view are processes in architectures.

The initial keyword is a hypertext link to expand the display in this view. If we click on it we see the scope inside the initial procedure.

Figure 20-16 The Expanded Machine Memory Construct View for the Peak Snapshot

Memory Construct View (clock:2000000)		
Name	Size	Percentage
▼Initial	116.14 MB	73.92 %
NoName	116.14 MB	73.92 %
total	116.14 MB	73.92 %

Page: 1

In this example the scope inside the initial procedure is a begin-end block that is unnamed, so profprt calls it NoName. This begin-end blocs use all of the machine memory used by the initial procedure. In this example NoName is not a hypertext link.

Like the machine memory summary and module views, there is a machine memory construct view for each snapshot.

If we select the Memory Instance view in the View: field in the left pane, then click the GO button again, the right pane changes to show this view.

Figure 20-17 The Machine Memory Instance View for the Peak Snapshot

Memory Instance View (clock:2000000)				
Instance	Inclusive Size	Percentage	Exclusive Size	Percentage
►tb_top	116.14 MB	73.92 %	116.14 MB	73.92 %
total	116.16 MB	73.93 %	116.16 MB	73.93 %

Page: 1

This view shows the machine memory used and percentages for the instances in the design at the peak snapshot. These are instances of Verilog and SystemVerilog modules and also instances of SystemVerilog programs and interfaces and VHDL entity/architectures.

This view shows for an instance the inclusive and exclusive machine memory used and percentage values for the peak snapshot.

The inclusive machine memory amount and percentage is the percentage of machine memory used by this instance and all instances that are hierarchically under it in the design hierarchy.

The exclusive machine memory amount and percentage is the machine memory used by this instance alone, not counting the instances that are hierarchically under this instance.

In this example there is only one instance of program tb_top, so the inclusive and exclusive values are the same, which are 116.14 MB and 73.92% of the machine memory.

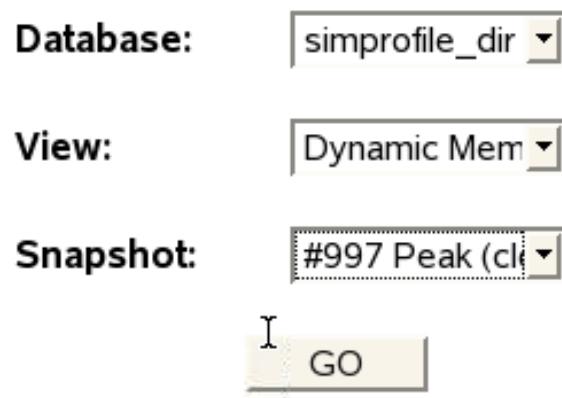
The instance name tb_top is not a hypertext link.

Like the machine memory summary, module, and construct views, there is a machine memory instance view for each snapshot.

In this example there is no PLI, DPI, or DirectC code so there is no information in the Pli/DPI/DirectC view.

If we select the Dynamic Memory view in the View: field in the left pane, the Snapshot: field to automatically change to snapshot #997.

Figure 20-18 The Left Pane After Selecting the Dynamic Memory View



Snapshot #997 is the peak snapshot for dynamic objects.

If we click the GO button again, the right pane changes to show this view.

Figure 20-19 The Dynamic Memory View for the Peak Snapshot

Dynamic Memory View (clock: 1998612)				
Dynamic Object	Instance Number	Memory	Percentage	
► PACKET	31552	29.68 MB	87.12 %	
► SmartQueue	N/A	4.39 MB	12.88 %	
► String	1	48 B	0.00 %	
		34.07 MB	100%	

Page: 1

click here

The peak machine memory dynamic view shows the machine memory that was used by dynamic objects at their peak machine memory consumption. This is not the peak machine memory consumption of the entire design and testbench, just the peak machine memory consumption of their dynamic objects.

The dynamic objects include dynamic and associative arrays and queues.

In this view is a SystemVerilog queue and string.

Smart Queues are a concept in the *OpenVera Language Reference Manual: Testbench*. The profrpt profile report generator lists SystemVerilog queues as Smart Queues. In this example there is only one SystemVerilog queue. It is declared as follows:

```
logic [255:0] Squeue_data_info[$];
```

Squeue_data_info, in this peak machine memory dynamic view, used 4.39 MB of machine memory, which is 12.88% of the machine memory used at this peak by the this queue.

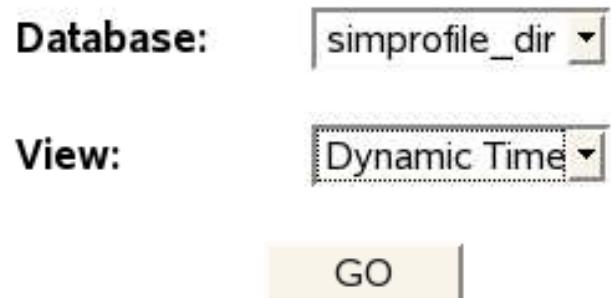
The profrpt profile report generator cannot report the number of instances of this queue.

The string entry is for a small amount of machine memory and can be ignored.

There is a dynamic object machine memory view for each snapshot.

If we select the Dynamic Timeline view in the View: field in the left pane, the Snapshot: field disappears.

Figure 20-20 The Left Pane After Selecting the Dynamic Timeline View



If we click the GO button again, the right pane changes to show this view.

Figure 20-21 The Machine Memory Dynamic Timeline View

Column for snapshots

Dynamic Memory Timeline									Display percentage
Clock	Assoc-Aarry	Dynamic Array	Smart Queue	Event	Mailbox	String	Class	Total	
#0	0	0 B	0 B	0 B	0 B	48 B	0 B	48 B	
#1	1	0 B	0 B	288 B	0 B	0 B	48 B	104 B	440 B
#2	36	0 B	0 B	200 B	0 B	0 B	48 B	1.93 KB	2.17 KB
#3	1182	0 B	0 B	3.83 KB	0 B	0 B	48 B	60.13 KB	64.01 KB
#4	1246	0 B	0 B	3.83 KB	0 B	0 B	48 B	63.38 KB	67.26 KB
#5	1314	0 B	0 B	3.83 KB	0 B	0 B	48 B	66.84 KB	70.71 KB
#6	1384	0 B	0 B	3.83 KB	0 B	0 B	48 B	70.39 KB	74.27 KB
#7	1458	0 B	0 B	3.83 KB	0 B	0 B	48 B	74.15 KB	78.02 KB
#8	1536	0 B	0 B	3.83 KB	0 B	0 B	48 B	78.11 KB	81.98 KB
#9	1618	0 B	0 B	3.83 KB	0 B	0 B	48 B	82.27 KB	86.15 KB
#10	1704	0 B	0 B	3.83 KB	0 B	0 B	48 B	86.64 KB	90.52 KB
#11	1794	0 B	0 B	3.83 KB	0 B	0 B	48 B	91.21 KB	95.09 KB
#12	1888	0 B	0 B	3.83 KB	0 B	0 B	48 B	95.98 KB	99.86 KB
#13	1921	0 B	0 B	11.41 KB	0 B	0 B	48 B	97.61 KB	109.06 KB
#14	2104	0 B	0 B	7.58 KB	0 B	0 B	48 B	106.95 KB	114.58 KB
#15	2218	0 B	0 B	7.58 KB	0 B	0 B	48 B	112.74 KB	120.37 KB
#16	2338	0 B	0 B	7.58 KB	0 B	0 B	48 B	118.84 KB	126.46 KB
#17	2464	0 B	0 B	7.58 KB	0 B	0 B	48 B	125.23 KB	132.86 KB
#18	2596	0 B	0 B	7.58 KB	0 B	0 B	48 B	131.94 KB	139.56 KB
#19	2734	0 B	0 B	7.58 KB	0 B	0 B	48 B	138.95 KB	146.57 KB

Page:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 4

hypertext links for page numbers

This view, unlike the previous machine memory views, is not for a specific snapshot, but for all snapshots in the profile database.

In this example there are multiple pages and the page numbers, at the bottom of the view, are hypertext links to show the different pages. In this view there are many pages because there are hundreds of snapshots in the database.

Notice that there is a significant increase in the machine memory for the queue in snapshot 13.

We can scroll to the right and click on page 50, which includes the dynamic object machine memory peak snapshot, and the right pane changes to show this page.

Figure 20-22 Page 50 of the Machine Memory Dynamic Timeline View

Dynamic Memory Timeline								Display percentage
Clock	Assoc-Aarry	Dynamic Array	Smart Queue	Event	Mailbox	String	Class	Total
#980	1936896	0 B	0 B	4.39 MB	0 B	0 B	48 B	1.83 MB 6.22 MB
#981	1936896	0 B	0 B	4.39 MB	0 B	0 B	48 B	1.52 MB 5.91 MB
#982	1936896	0 B	0 B	4.39 MB	0 B	0 B	48 B	1.23 MB 5.61 MB
#983	1936896	0 B	0 B	4.39 MB	0 B	0 B	48 B	968.30 KB 5.33 MB
#984	1936896	0 B	0 B	4.39 MB	0 B	0 B	48 B	695.20 KB 5.07 MB
#985	1936896	0 B	0 B	4.39 MB	0 B	0 B	48 B	435.70 KB 4.81 MB
#986	1936896	0 B	0 B	4.39 MB	0 B	0 B	48 B	189.21 KB 4.57 MB
#987	1945230	0 B	0 B	4.39 MB	0 B	0 B	48 B	423.31 KB 4.80 MB
#988	1950072	0 B	0 B	4.39 MB	0 B	0 B	48 B	669.20 KB 5.04 MB
#989	1955156	0 B	0 B	4.39 MB	0 B	0 B	48 B	927.37 KB 5.29 MB
#990	1960494	0 B	0 B	4.39 MB	0 B	0 B	48 B	1.17 MB 5.56 MB
#991	1966098	0 B	0 B	4.39 MB	0 B	0 B	48 B	1.45 MB 5.84 MB
#992	1971982	0 B	0 B	4.39 MB	0 B	0 B	48 B	1.74 MB 6.13 MB
#993	1978160	0 B	0 B	4.39 MB	0 B	0 B	48 B	2.05 MB 6.43 MB
#994	1984648	0 B	0 B	4.39 MB	0 B	0 B	48 B	2.37 MB 6.76 MB
#995	1991460	0 B	0 B	4.39 MB	0 B	0 B	48 B	2.71 MB 7.09 MB
#996	1998612	0 B	0 B	4.39 MB	0 B	0 B	48 B	3.06 MB 7.45 MB
#997	1998612	0 B	0 B	4.39 MB	0 B	0 B	48 B	29.68 MB 34.07 MB

Hypertext Links to the Source Files

The pathnames of source files in any of the HTML views are hypertext links. Clicking on one of these links opens a new window of the browser to display that source file. This section describes and illustrates this feature.

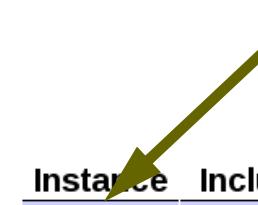
Note:

The hypertext link to the source files feature is not implemented for SystemC/C/C++ source files.

To use this feature do the following:

1. Compile a design with the `-lca` and `-simprofile` options.
2. Run the simulation with the `-simprofile time/mem/ time+mem` option and keyword argument to enable VCS or VCS MX to collect time/memory/time&memory profile information.
3. Run the `profprt` utility to create the HTML views.
4. Open the `profileReport.1.html` file.
5. Select a profile database in the left pane.
6. Select the Time Instance view.

Figure 20-23 Time Instance View



click here

Instance	Inclusive Time	Percentage	Exclusive Time	Percentage
►tb_top	3.52 s	23.46 %	3.52 s	23.46 %
total	3.52 s	23.46 %	3.52 s	23.46 %

Page: 1

7. Click on an instance in the view.

This adds this information about the instance to the bottom of the HTML page:

Instance Name	a reiteration of the instance name
Exclusive Time	the CPU time used by the instance
Exclusive Percentage	the percentage of the total CPU time that was used by this instance
Inclusive Time	the CPU time used by the instance and all instances under it in the design hierarchy
Inclusive Percentage	the percentage of the total CPU time that was used by this instance and all instances under it in the design hierarchy
Master Module	the name of the top-level module in the design hierarchy

Child Instance Number the number of instances under this instance in the design hierarchy

Source Information the path to the source file and line number of the header of the module, interface, or program definition

The Source Information is in blue text in this expanded view because it is a hypertext link to the source code. See [Figure 20-24 on page 507](#).

Figure 20-24 Time Instance View Expanded

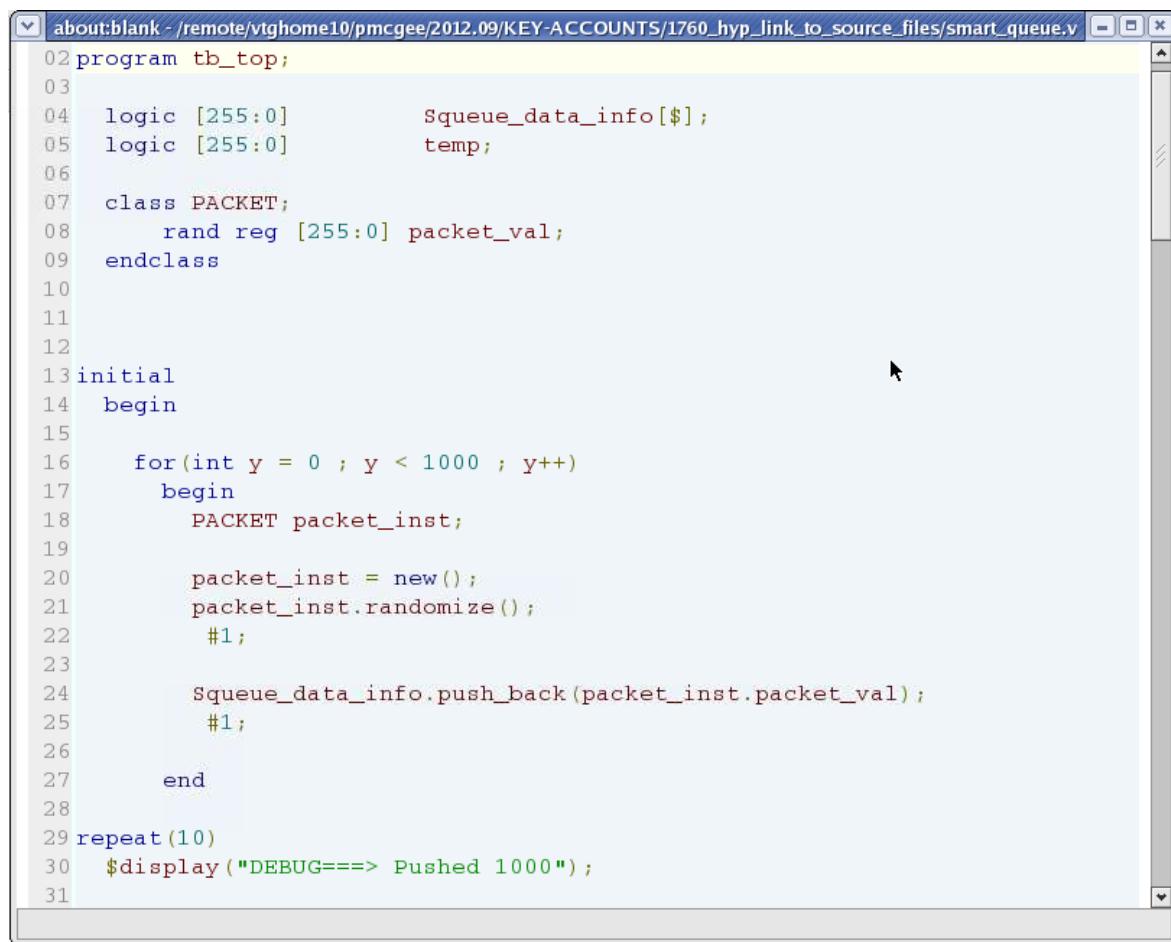
Instance Information		
Instance Name		tb_top
Exclusive Time		3.52 s
Exclusive Percentage	click here	23.46 %
Inclusive Time		3.52 s
Inclusive Percentage		23.46 %
Master Module		tb_top
Child Instance Number		0
Source Information	/file_system/big_design/VCS_user_files/smart_queue.v:2	

In this example source information for the instance is the program definition for instance tb_top is in:

/file_system/big_design/VCS_user_files/smart_queue.v on line 2.

8. Click on the blue path name of the source file and line number, this is a hypertext link. The browser opens a new window to display the source file, see [Figure 20-25 on page 508](#).

Figure 20-25 New Source File window



```
about:blank - /remote/vtghome10/pmcgee/2012.09/KEY-ACCOUNTS/1760_hyp_link_to_source_files/smart_queue.v
02 program tb_top;
03
04     logic [255:0]           Squeue_data_info[$];
05     logic [255:0]           temp;
06
07     class PACKET;
08         rand reg [255:0] packet_val;
09     endclass
10
11
12
13 initial
14 begin
15
16     for(int y = 0 ; y < 1000 ; y++)
17         begin
18             PACKET packet_inst;
19
20             packet_inst = new();
21             packet_inst.randomize();
22             #1;
23
24             Squeue_data_info.push_back(packet_inst.packet_val);
25             #1;
26         end
27
28
29 repeat(10)
30     $display("DEBUG====> Pushed 1000");
31
```

The program header is `program tb_top;` in line 2, has a lighter background.

The lines in this source file window also shows the line numbers.

Single Text Format Report

Text format views are merged together into a text file named `profileReport.txt` in the current directory.

You specify text format reports with the `-format text` or `-format all` option and argument on the `profrpt` command line.

If you run the `profrpt` report generator more than once, the utility overwrites the `profileReport.txt` file in the current directory so that its profile information is from the last run.

When you specify text format reports the `profrpt` utility also creates separate text files for each view in the profile report directory. These separate text files for each view have names such as `PeakMemInstanceView.txt` or `TimeConstr.txt`.

Stack Trace Report Example

The following file, named `check.v`, is used to produce a sample stack trace report.

```
class Packet;
    bit[100000:0] b;
    function new();
        b = 0;
    endfunction
endclass
```

```

Packet pp[int];
    int cindex = 0;
reg r;

program p;
    function Packet AllocPacket();
        begin
            AllocPacket = new;
        end
    endfunction

task A;
    begin
        fork
            B();
            C();
        join
    end
endtask

task B;
    int i;
    Packet lpp[int];
    begin
        $display("B called");
        for (i=0; i < 100000; i++)
            pp[i] = AllocPacket();
    end
endtask

task C;
    int i;
    Packet lpp[int];
    begin
        $display("C called");
        for (i=0; i < 10000; i++)
            lpp[i] = AllocPacket();
    end
endtask

initial
begin

```

```
A();  
end  
endprogram
```

The following command sequence generates the stack trace report for the check.v example:

```
vcs check.v -lca -simprofile -sverilog  
  
simv -simprofile mem  
  
profrpt simprofile_dir -view dynamic_mem+stack
```

[Figure 20-26](#) shows the HTML stack trace report for the check.v example. The stack trace information is at the bottom of the view.

Figure 20-26 The Machine Memory Dynamic Object View for the Peak Snapshot

Dynamic Memory View (clock: 0)				
Dynamic Object	Instance Number	Memory	Percentage	
▼ Packet	110000	1315.92 MB	100.00 %	
► AllocPacket	100000	1196.29 MB	90.91 %	
► AllocPacket	10000	119.63 MB	9.09 %	
► AssociativeArray	N/A	128 B	0.00 %	
► String	1	48 B	0.00 %	
		1315.92 MB	100%	

Page: 1

Stack Information			
#0	AllocPacket	/file_system/big_design/VCS_user_files/stack.sv:15	
#1	C	stack.sv:44	
#2	A	stack.sv:23	
#3	p	stack.sv:50	

SystemC Views

The following views are from a SystemC cosimulation after running the profrpt profile report generator.

The code examples for these views is in the \$VCS_HOME/doc/examples/systemc/vcs/vcs_profiler. There is a minor change to one of the files to show the name for a begin-end block in sv_mod.sv as follows:

```
module sv_mod(iclk);
    input iclk;
    static int count=0;
    int i;

    always @(posedge iclk)
begin: be1
    count++;
    $display("SV:Executing on pos edge @%d",count);
    for(i=0;i<1000*100000000;i++)
    ;
end

endmodule
```

Figure 20-27 The Time Summary View

Time Summary View		
Component	Time	Percentage
VERILOG	282.53 s	77.85 %
Module	282.53 s	77.85 %
Package	999.89us	0.00 %
SystemC	79.85 s	22.00 %
KERNEL	505.94 ms	0.14 %
HSIM	12.00 ms	0.00 %
PLI/DPI/DirectC	999.89us	0.00 %
total	362.90 s	100%

As you would expect from reading the SystemVerilog and SystemC files in this example, most of the CPU time was used by the SystemVerilog and SystemC modules.

A small amount of CPU time was used by The VCS or VCS MX kernel.

A small amount of CPU time was reported used by a SystemVerilog package, writing a VPD file, and PLI, DPI, or a DirectC application, even though these are not present in this example. Notice that they all take 0.00% of the CPU time. You can ignore these anomalies.

If our example wrote a VPD file or contained a PLI, DPI, or DirectC application, we would see significant values for the CPU times in this view.

Figure 20-28 The Time Module View

Time Module View

Module	Time	Percentage
►sv_mod	282.53 s	77.85 %
►sc_mod	79.85 s	22.00 %
►sv_top	2.00 ms	0.00 %
►std	999.89us	0.00 %
►global_	0.00us	0.00 %
total	282.53 s	77.85 %

Page: 1

This view shows the CPU times and percentages for the main consumers of CPU times, the SystemVerilog module sv_mod and the SystemC module sc_mod. A small amount of time is used by the top level module sv_top.

The modules std and _global_ are from the internals of VCS or VCS MX and when seen in this view should be ignored.

If you click on these module names the view expands to show scopes inside these module definitions.

Figure 20-29 The Expanded Time Module View

Time Module View		
Module	Time	Percentage
sv_mod	282.53 s	77.85 %
be1	282.53 s	77.85 %
NoName	0.00us	0.00 %
iclk	0.00us	0.00 %
sc_mod	79.85 s	22.00 %
mythread	79.85 s	22.00 %
sv_top	2.00 ms	0.00 %
NoName	2.00 ms	0.00 %
NoName	0.00us	0.00 %
std	999.89us	0.00 %
global	0.00us	0.00 %
total	282.53 s	77.85 %

Page: 1

In SystemVerilog module sv_mod:

- The begin-end block named be1 consumes all of the CPU time of the module.
- There is an extraneous process call NoName that consumed no CPU time and can be ignored.

- The input port named `iclk` in `sv_mod` is shown as a process, like a begin-end block of code, because it also attached to `sc_mod` and in SystemC the clock signal is a process. If `sv_mod` had other ports that were not clock signals, `profrpt` would not show them as processes.

In SystemC module `sc_mod`, `mythread()` is the SystemC variant of a named block in Verilog or SystemVerilog, and represents code (like in a SystemVerilog always procedure, but is shown in this view rather than the Time Construct view). The implementation of this function is in the `.cpp` file.

Figure 20-30 The CPU Time Construct View

Time Construct View		
Name	Time	Percentage
► Always	282.53 s	77.85 %
► Initial	2.00 ms	0.00 %
► Task	0.00us	0.00 %
► Function	0.00us	0.00 %
► Port	0.00us	0.00 %
total	282.53 s	77.85 %

Page: 1

The CPU time construct view shows the CPU times and percentages used by the always and initial procedures in the design, and also the port in the design.

In this example Task and Function do not refer to a user-defined task and function, but rather refer to the internals of VCS or VCS MX and did not consume CPU time. If this example contained user-defined tasks or functions, they would be listed as a Task or Function here.

Figure 20-31 The Time Instance View

Time Instance View				
Instance	Inclusive Time	Percentage	Exclusive Time	Percentage
» sv_top	282.53 s	77.85 %	2.00 ms	0.00 %
» sv_top.sc_mod_inst	79.85 s	22.00 %	79.85 s	22.00 %
» std	999.89us	0.00 %	999.89us	0.00 %
» _global_	0.00us	0.00 %	0.00us	0.00 %
total	282.53 s	77.85 %	282.53 s	77.85 %

Page: 1

In this view, as it initially appears, we see the SystemVerilog top-level instance sv_top. We also see the SystemC instance sv_top.sc_mod_inst because it is a SystemC instance in this SystemVerilog on top example.

As in previous views, std and _global_ are from the internals of VCS and VCS MX and can be ignored.

If we click on the top-level module sv_top we see instance sv_mod_inst.

Figure 20-32 The Expanded CPU Time Instance View

Time Instance View				
Instance	Inclusive Time	Percentage	Exclusive Time	Percentage
sv_top	282.53 s	77.85 %	2.00 ms	0.00 %
sv_mod_inst	282.53 s	77.85 %	282.53 s	77.85 %
sv_top.sc_mod_inst	79.85 s	22.00 %	79.85 s	22.00 %
std	999.89us	0.00 %	999.89us	0.00 %
global	0.00us	0.00 %	0.00us	0.00 %
total	282.53 s	77.85 %	282.53 s	77.85 %

Page: 1

Figure 20-33 The PLI/DPI/DirectC View

Time PLI/DPI/DirectC View		
Name	Time	Percentage
PLI	999.89us	0.00 %
\$sc_mod_init	999.89us	0.00 %
\$vcplusplusfilter	0.00us	0.00 %
\$msglog	0.00us	0.00 %
\$lsi_dumpports	0.00us	0.00 %
\$countdrivers	0.00us	0.00 %
\$vcsmemprof	0.00us	0.00 %
\$start_toggle_count	0.00us	0.00 %
\$report_toggle_count	0.00us	0.00 %
\$set_toggle_region	0.00us	0.00 %
\$toggle_start	0.00us	0.00 %
\$toggle_stop	0.00us	0.00 %
\$toggle_reset	0.00us	0.00 %
\$toggle_report	0.00us	0.00 %
\$read_lib_saif	0.00us	0.00 %
\$read rtl_saif	0.00us	0.00 %
\$set_gate_level_monitoring	0.00us	0.00 %
DPI	0.00us	0.00 %
DirectC	0.00us	0.00 %
total	999.89us	0.00 %

This view, for the PLI, shows both VCS and VCS MX internal functions and user written PLI functions.

In this example all functions are VCS and VCS MX internal functions. You can look for ones that consume significant CPU time. The `$vcplusmsglog` system function, not in this example, can consume significant CPU time.

For SystemC there is an additional CPU time view, the SC (SystemC) OverHead View.

Figure 20-34 The SC OverHead View

Time SC-OverHead View

Name	Time	Percentage
SC-Value-OverHead	0.00us	0.00 %
SC-Kernel-OverHead	0.00us	0.00 %
SC-Spawn-OverHead	0.00us	0.00 %
total	0.00us	0.00 %

Page: 1

This data depends on the test case. It can be that kernel overhead becomes an issue and/or it can be compared against the Verilog kernel overhead.

The sc-value overhead is time taken to transfer data from one domain to another, like to or from SystemC to or from Verilog, SystemVerilog, or VHDL. This can be expensive when there is large amounts of data such as with a large vector signal or a large multi-dimensional array. Also spawning of processes can take time and accumulate sc-overhead.

Kernel overhead from SystemC, but also from Verilog, SystemVerilog or VHDL, can become an issue when your code doesn't consume much CPU time and there is significant overhead to keep the cosimulation running.

Usually you want these CPU time values to be low.

Constraint Profiling Integrated in the Unified Profiler

Constraint profiling is integrated in the unified profiler. This integration adds the following views to the profile reports:

- the Time Constraint Solver view
- the Memory Constraint Solver view

These views tell you, in detail, the calls to the `randomize()` method that use the most CPU time or the most machine memory. With this information you can consider revising your constraints on the random variables to use less of these resources.

Changes to the Use Model for Constraint Profiling

To tell profrpt to generate these views the following is added to the use model:

The `profrpt -view` option's arguments now include:

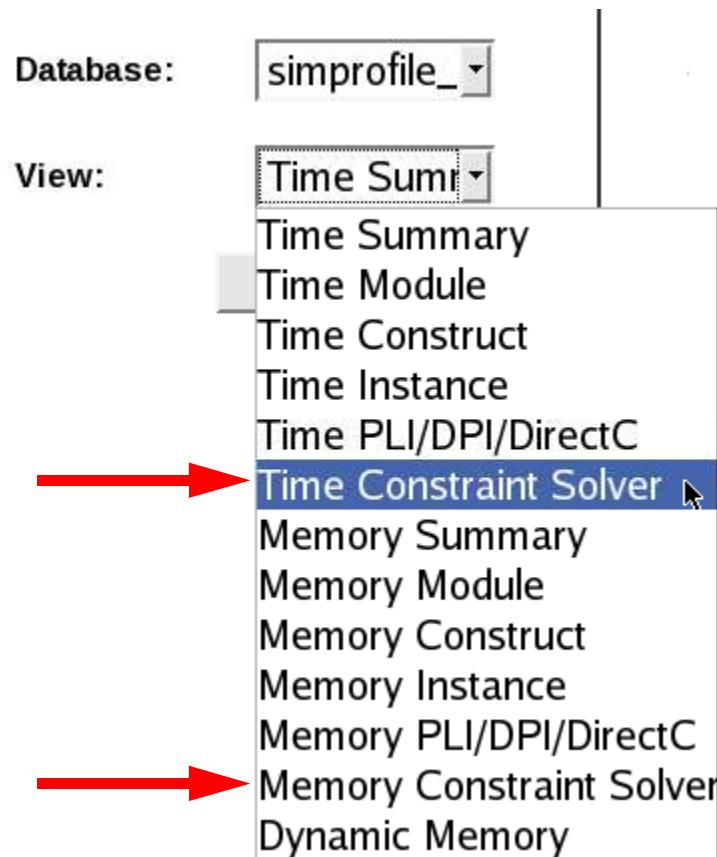
- `time_solver` to specify generating the Time Constraint Solver view
- `mem_solver` to specify generating the Memory Constraint Solver view.

The `time_all` and `mem_all` arguments also generate these views.

As in previous releases, the left pane of the `profileReport.html` file, after selecting a profile database, contains a drop down menu for views. This menu now contains the the following for constraint profiling:

- the Time Constraint Solver view
- the Memory Constraint Solver view

Figure 20-35 New Constraint Views



The following sections describe these views.

The Time Constraint Solver View

The following is an example of the Time Constraint Solver View.

Figure 20-36 Example Time Constraint Solver View

Time Constraint Solver View						
Total user time: 11.670seconds						
Total system time: 0.120seconds						
Total randomize time: 0.030seconds						
Total randomize count: 2						
Top randomize calls based on cpu runtime						
File:line@visit	serial#	time (sec)	variables	constraints	cnst blocks	
./env/nvs_atapi_env.sv:118@1	1	0.030	27	37	9	
./env/nvs_atapi_env.sv:120@1	2	0.000	3	3	1	
Top randomize calls based on cumulative cpu runtime						
File:line	calls	time (sec)				
./env/nvs_atapi_env.sv:118	1	0.030				
./env/nvs_atapi_env.sv:120	1	0.000				

Top partitions based on cpu time

File:line@visit	Rand.Partition	cpu time (sec)	variables	constraints	cnst blocks
/env/nvs_atapi_env.sv:118@1		1.1	0.03	2	4 2
/env/nvs_atapi_env.sv:118@1		1.2	0.00	1	1 1
/env/nvs_atapi_env.sv:118@1		1.3	0.00	7	12 3
/env/nvs_atapi_env.sv:118@1		1.4	0.00	5	10 3
/env/nvs_atapi_env.sv:118@1		1.5	0.00	2	1 1
/env/nvs_atapi_env.sv:118@1		1.6	0.00	1	1 1
/env/nvs_atapi_env.sv:118@1		1.7	0.00	1	1 1
/env/nvs_atapi_env.sv:118@1		1.8	0.00	2	1 1
/env/nvs_atapi_env.sv:118@1		1.9	0.00	2	1 1
/env/nvs_atapi_env.sv:118@1		1.10	0.00	2	1 1

Constraint solver profile

Solver	Time (sec)
Core Solver (default)	0.030
Core Solver (mode=1)	0.000
Core Solver (FAST)	0.000
Problem Generation	0.000

Top partitions based on BDD size

File:line@visit	Rand.Partition	peak BDD size	final BDD size	variables	constraints	cnst blocks
-----------------	----------------	---------------------	----------------------	-----------	-------------	----------------

Parts of this view, [Figure 20-36](#) are described in detail below in [Figure 20-37](#), [Figure 20-38](#), [Figure 20-39](#), [Figure 20-40](#), [Figure 20-41](#), and [Figure 20-42](#).

Figure 20-37 Introductory information is at the top of the view

Time Constraint Solver View	
Total user time:	11.670seconds
Total system time:	0.120seconds
Total randomize time:	0.030seconds
Total randomize count:	2

Total user time:

Specifies the total CPU time to simulate the design and testbench. In this example it is 11.670 seconds.

Total system time:

Specifies the total CPU time used by VCS when not simulating the design or testbench. In this example it is 0.12 seconds.

Total randomize time:

Specifies the CPU time VCS needed to execute the `randomize()` method calls in the design. In this example it is 0.03 seconds.

Total randomize count:

Specifies the number of entries of the `randomize()` method are in the SystemVerilog source code. In this example it is 2.

Figure 20-38 Top randomize calls based on cpu time

hypertext link [Top randomize calls based on cpu runtime](#)

File:line@visit	serial#	time (sec)	variables	constraints	cnst blocks
./env/nvs_atapi_env.sv:118@1	1	0.030	27	37	9
./env/nvs_atapi_env.sv:120@1	2	0.000	3	3	1

This section of the view is for the `randomize()` entries in the source code that use the most CPU time. There is a separate line for each entry. There are two such entries in this code example, so they are listed here.

The columns in this section are for the following values:

File:line@visit

Specifies the following three things:

File

Specifies the path name for the source file that contains the entry. In this example view the first line is for a source file with the path name `/env/nvs_atapi_env.sv`.

line

Specifies the line number in the source file that contains the entry. In this example the entry is on line 118.

@visit

A visit is an execution of the entry. There can be multiple executions of the same entry throughout a simulation. In this example view, the first line is for the first execution, or visit, of the entry.

If the code example had VCS execute the entry three times, there could have been a line in this section that began with:

[/env/nvs_atapi_env.sv:118@3](#)

Important:

The File:line@visit part of a line is blue because this part is a hypertext link. When you click on it, the browser opens a new window showing the source file with the line specified at the top.

serial#

The series in this column is the order in which VCS executes the calls to the `randomize()` method. In this example line 118 contained the first call and line 120 contained the second call.

Note:

This section of the view is for the calls that used the most CPU time, and these top users are not always the first or second `randomize()` calls that VCS executes.

time (sec)

The amount of CPU time used by the call.

variables

The number of `rand` or `randc` variables randomized by a call. Not all such variables in a class are randomized by a call.

constraints

The number of constraints in the class that are randomized by a call.

cnst blocks

The number of constraint blocks that contain these constraints.

Note:

In the following example:

```
constraint reasonable_on_latencies {
    dior_to_data_place_time < 10;
    data_prepare_time       < 10;
    dior_to_data_place_time > 0;
    data_prepare_time       > 0;
} //end constraint reasonable_on_latencies
```

There is one constraint block and four constraints.

Figure 20-39 Top randomize calls based on cumulative cpu runtime

hypertext link



File:line	calls	time (sec)
/env/nvs_atapi_env.sv:118	1	0.030
/env/nvs_atapi_env.sv:120	1	0.000

VCS can execute, or visit, a call to the `randomize()` method in a specific location of the source code more than once. If it does so, VCS keeps track of the cumulative CPU time used by these multiply executed calls and profrpt reports this cumulative time in this section.

This section reports:

- The location of the call in a hypertext link that opens a new window displaying the source code.
- The number of calls or visits to this location.
- The cumulative CPU time used by the calls.

Figure 20-40 Top partitions based on cpu time

Top partitions based on cpu time						
File:line@visit	Rand.Partition	cpu time (sec)	variables	constraints	cnst blocks	
./env/nvs_atapi_env.sv:118@1	1.1	0.03	2	4	2	
./env/nvs_atapi_env.sv:118@1	1.2	0.00	1	1	1	
./env/nvs_atapi_env.sv:118@1	1.3	0.00	7	12	3	
./env/nvs_atapi_env.sv:118@1	1.4	0.00	5	10	3	
./env/nvs_atapi_env.sv:118@1	1.5	0.00	2	1	1	
./env/nvs_atapi_env.sv:118@1	1.6	0.00	1	1	1	
./env/nvs_atapi_env.sv:118@1	1.7	0.00	1	1	1	
./env/nvs_atapi_env.sv:118@1	1.8	0.00	2	1	1	
./env/nvs_atapi_env.sv:118@1	1.9	0.00	2	1	1	
./env/nvs_atapi_env.sv:118@1	1.10	0.00	2	1	1	

VCS has a constraint solver to determine the possible values that conform to your constraints. To solve these problems the constraint solver divides its work into partitions. This section reports the number of partitions in a problem.

In this example this section reports on the one visit to the `randomize()` method in the example source file at `/env/nvs_atapi_env.sv` on line 118.

The constraint solver divided its work into 10 partitions. The `profprt` utility reports, for each partition:

- the CPU time needed to solve the partition
- the number of random variables in the partition
- The number of constraints
- The number of constraint blocks that contained these constraints

Figure 20-41 Constraint solver profile

Constraint solver profile	
Solver	Time (sec)
Core Solver (default)	0.030
Core Solver (mode=1)	0.000
Core Solver (FAST)	0.000
Problem Generation	0.000

The total randomize time is further broken down into the different internal solvers and problem generation. This information might indicate where you can revise your constraints and randomize calls to improve the total CPU time.

Figure 20-42 Top partitions based on BDD size

Top partitions based on BDD size				
File:line@visit Rand.Partition	peak BDD size	final BDD size	variables	constraints
			cnst	blocks

This part of the constraint profile report is empty unless VCS uses the solver (mode=1) in the randomization. When it uses mode=1, this section shows some memory footprint information of different

randomize calls executed under this solver (mode=1). You specify using the mode=1 solver with the `+ntb_solver_mode=1` runtime option and argument.

No information is in this example section because the default solver is doing the constraint solving for this example.

The Memory Constraint Solver View

The following is an example of the memory constraint solver view.

Figure 20-43 Example Memory Constraint Solver View

Memory Constraint Solver View						
Largest memory increment: 640KB						
Top randomize calls based on memory increment						
File:line@visit	serial#	mem incr (KB)	variables	constraints	cnst blocks	
/env/nvs_atapi_env.sv:118@1	1	640	27	37	9	
/env/nvs_atapi_env.sv:120@1	2	8	3	3	1	

Top recurring randomize calls based on memory increment		
File:line	calls	mem incr (KB)
/env/nvs_atapi_env.sv:118	1	640
/env/nvs_atapi_env.sv:120	1	8

Parts of this view, [Figure 20-43](#), are described in detail below in [Figure 20-44](#) and [Figure 20-45](#).

The view begins with the size of that largest increase of machine memory during the simulation:

Figure 20-44 Largest memory increment

Memory Constraint Solver View

Largest memory increment: 640KB

In this example the largest increase in machine memory was an increase of 640 KB.

Next are the `randomize()` entries that cause the largest increases in the use of machine memory:

Figure 20-45 Top randomize calls based on memory increment

Top randomize calls based on memory increment						
hypertext link	File:line@visit	serial#	mem incr (KB)	variables	constraints	cnst blocks
./env/nvs_atapi_env.sv:118@1		1	640	27	37	9
./env/nvs_atapi_env.sv:120@1		2	8	3	3	1

The columns in this section are for the following values:

File:line@visit

Specifies the following three things:

File

Specifies the path name for the source file that contains the entry. In this example view the first line is for a source file with the path name /env/nvs_atapi_env.sv (just like it was with .

line

Specifies the line number in the source file that contains the entry. In this example the entry is on line 118.

@visit

A visit is an execution of the call. There can be multiple executions of the same call throughout a simulation. In this example view, the first line is for the first execution, or visit, of the call.

If the code example had VCS execute the call three times, there could have been a line in this section that began with:

[/env/nvs_atapi_env.sv:118@3](#)

Important:

The File:line@visit part of a line is blue because this part is a hypertext link. When you click on it, the browser opens a new window showing the source file with the line specified at the top.

serial#

The series in this column is the order in which VCS executes the calls to the randomize() method. In this example line 118 contained the first call and line 120 contained the second call.

Note:

This section of the view is for the calls that used the most machine memory, and these top users are not always the first or second randomize () calls that VCS executes.

mem incr (KB)

The amount of additional machine memory VCS needs when it executes the call.

variables

The number of rand or randc variables randomized by a call.
Not all such variables in a class are randomized by a call.

constraints

The number of constraints in the class that are randomized by a call.

cnst blocks

The number of constraint blocks that contain these constraints.

hypertext link

Top recurring randomize calls based on memory increment		
File:line	calls	mem incr (KB)
./env/nvs_atapi_env.sv:118	1	640
./env/nvs_atapi_env.sv:120	1	8

The next section is for the randomize() calls that VCS executes the most. There are two randomize() entries in this example, and each are executed only once. These executed once calls are in this section because the code example does not contain calls that execute more frequently during the simulation.

This section reports:

- The path to the source file, and the line number of the call.
- The number of times VCS executes a call
- The amount of additional machine memory VCS needs to execute the call.

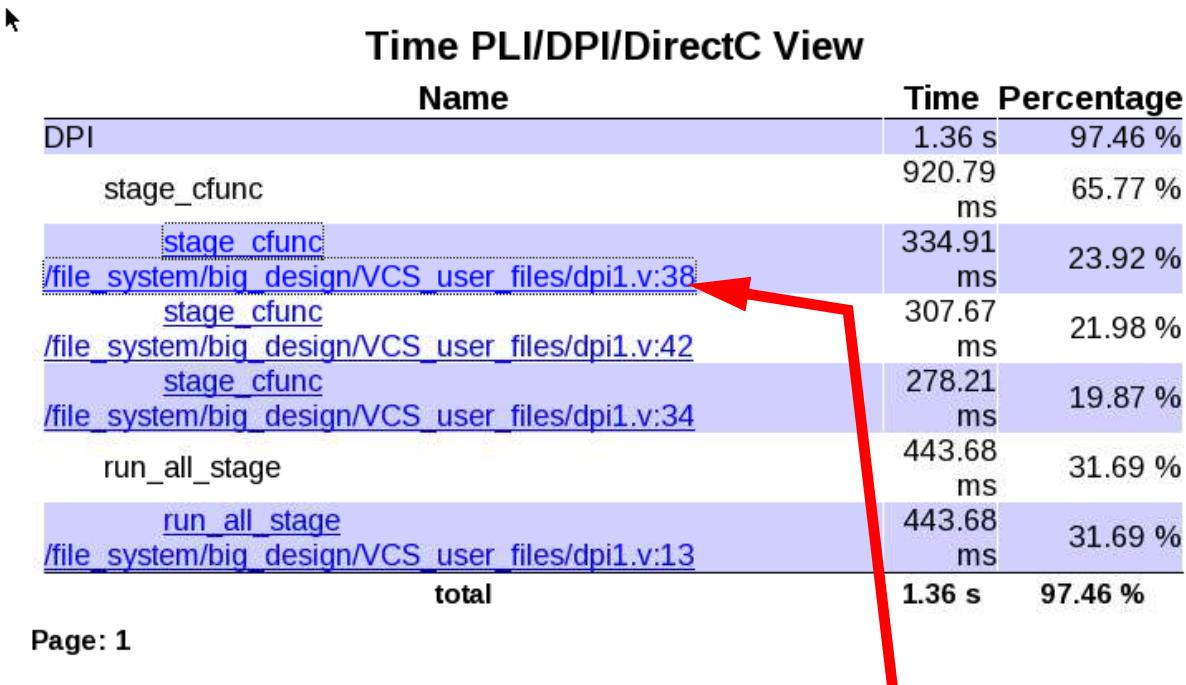
PLI/DPI/DirectC Enhancement in the Unified Profiler

The Time and Memory PLI/DPI/DirectC views are enhanced in the G-2012.09 release to include, along with the name of the PLI user-defined system task or DPI/DirectC function call, the following:

- the path name of the Verilog or SystemVerilog source file that contains that system task or function call
- the line number in that source file that contains that system task or function call

The pathname and line number are blue because they are a hypertext link, as shown in [Figure 20-46 on page 535](#).

Figure 20-46 The Time PLI/DPI/DirectC View for a DPI Function Call



The screenshot shows a table titled "Time PLI/DPI/DirectC View". The table has three columns: "Name", "Time", and "Percentage". The "Time" column includes units like "s" and "%". A red arrow points from the text "hypertext link" to the source file path in the third row of the table.

Name	Time	Percentage
DPI	1.36 s	97.46 %
stage_cfunc	920.79 ms	65.77 %
stage_cfunc /file_system/big_design/VCS_user_files/dpi1.v:38	334.91 ms	23.92 %
stage_cfunc	307.67 ms	21.98 %
stage_cfunc /file_system/big_design/VCS_user_files/dpi1.v:42	278.21 ms	19.87 %
run_all_stage	443.68 ms	31.69 %
run_all_stage /file_system/big_design/VCS_user_files/dpi1.v:13	443.68 ms	31.69 %
total	1.36 s	97.46 %

Page: 1

hypertext link

This view reports the CPU time used by:

- Three calls of the DPI function named `stage_cfunc`, the view reports the total CPU time and percentages of all three of these calls collectively and individually. The hypertext links are for the individual calls.
- One call of the DPI function named `run_all_stage`

When you click on one of these links, the browser opens another window containing the source file with the selected line. So if, for example, we click the hypertext link for:

[**/file_system/big_design/VCS_user_files/dpi1.v:38**](#)

The browser opens another window to show the contents of dpi1.v, highlighting line 38, as shown in [Figure 20-47 on page 536](#).

Figure 20-47 Source Code Window with Highlighted Line



A screenshot of a web browser window displaying a source code file. The title bar shows the URL: "about:blank - /remote/vtghome10/pmcgee/2012.09/KEY-ACCOUNTS/1758_break_down_PLI_fun". The code is written in Verilog-like syntax. Line 38, which contains the statement "stage_cfunc(c, b, d);", is highlighted with a yellow background. The rest of the code is in a standard monospaced font.

```
34     stage_cfunc(a, b, c) ;
35
36     b = 10 ;
37     @e2 ;
38     stage_cfunc(c, b, d) ;
39
40     @e3 ;
41
42     stage_cfunc(c, d, e) ;
43
44 end
45
46 endtask
47
48 initial
49 begin
50     run_all_stage() ;
51     $display(c) ;
52     $display(d) ;
53     $display(e) ;
54 end
55
56 initial
57 begin
58     #10 -> e1 ;
59     #20 -> e2 ;
60     #30 -> e3 ;
```

Similarly, this hypertext link feature is now in the Memory PLI/DPI/ DirectC view as shown in [Figure 20-48 on page 537](#).

Figure 20-48 The Memory PLI/DPI/DirectC View for a DPI Function Call

Memory PLI/DPI/DirectC View (clock:60)		
Name	Size	Percentage
DPI	66.76 MB	60.38 %
stage_cfunc	57.22 MB	51.76 %
<u>stage_cfunc</u> /file_system/big_design/VCS_user_files/dpi1.v:34	19.08 MB	17.25 %
<u>stage_cfunc</u> /file_system/big_design/VCS_user_files/dpi1.v:38	19.07 MB	17.25 %
<u>stage_cfunc</u> /file_system/big_design/VCS_user_files/dpi1.v:42	19.07 MB	17.25 %
run_all_stage	9.54 MB	8.63 %
<u>run_all_stage</u> /file_system/big_design/VCS_user_files/dpi1.v:13	9.54 MB	8.63 %
total	66.76 MB	60.38 %

Page: 1

hypertext link

This view reports the machine memory used by:

- Three calls of the DPI function named stage_cfunc. The view reports the peak machine memory amount and percentages of all three of these calls collectively and individually. The hypertext links are for the individual calls.
- One call to the DPI function named run_all_stage

When you click on one this links, the browser opens another window containing the source file with the selected line in reverse video. So if, for example, we click the hypertext link for:

[**/file_system/big_design/VCS_user_files/dpi1.v:34**](#)

The browser opens another window to show the contents of dpi1.v, highlighting line 34, as shown in [Figure 20-49 on page 538](#).

Figure 20-49 Another Source Code Window with Highlighted Line



A screenshot of a web browser window titled "about:blank - /remote/vtghome10/pmcgee/2012.09/KEY-ACCOUNTS/1758_break_down_PLI_fur". The window displays a Verilog-like source code. Line 34, which contains the assignment "b = 10 ;", is highlighted in yellow. The code includes declarations for stage_cfunc, assignments for b, c, d, and e, and tasks for run_all_stage and \$display.

```
34     stage_cfunc(a, b, c) ;
35
36     b = 10 ;
37     @e2 ;
38     stage_cfunc(c, b, d) ;
39
40     @e3 ;
41
42     stage_cfunc(c, d, e) ;
43
44 end
45
46 endtask
47
48 initial
49 begin
50     run_all_stage() ;
51     $display(c) ;
52     $display(d) ;
53     $display(e) ;
54 end
55
```

In previous releases the Time and Memory PLI/DPI/DirectC views did not have the hypertext link feature to open another window for the source code. They also reported only the collective CPU time and peak machine memory use of a PLI user-defined system task or DPI/DirectC function. There was no reporting on individual calls of the same DPI function, for example.

The following figures show these views for the same code example in previous releases.

Figure 20-50 Time PLI/DPI/DirectC View in a Previous Release

Time PLI/DPI/DirectC View		
Name	Time	Percentage
DPI	1.47 s	97.79 %
stage_cfunc	1.05 s	69.68 %
run_all_stage	421.75 ms	28.12 %
total	1.47 s	97.79 %

Page: 1

Figure 20-51 Memory PLI/DPI/DirectC View in a Previous Release

Memory PLI/DPI/DirectC View (clock:30)		
Name	Size	Percentage
DPI	66.76 MB	59.75 %
stage_cfunc	57.22 MB	51.21 %
run_all_stage	9.54 MB	8.54 %
total	66.76 MB	59.75 %

Page: 1

Limitations

The following technologies are not supported in the unified profiler:

- Multicore — Both for Application Level Parallelism (ALP) and Design Level Parallelism (DLP is an LCA feature).
- The behavior would be unpredictable if you fork child processes or threads in your C code which might be called through PLI/DPI/ DirectC interfaces.
- Incremental compilation — Not supported yet for the unified profiler.
- OpenVera is not officially supported, VCS provides some information for reference but the name of the programs and constructs might be a bit different from the original one.
- Code coverage is not supported yet, the time and memory used by code coverage would be counted to corresponding HDL code.
- Only black box information (no break down of information for instances, module, constructs, and so forth) for SystemVerilog constraints and assertions and functional coverage.

21

Partition Compile

Partition Compile is a VCS feature that allows you to compile portions of the design and get significantly faster turnaround time during the iterative process of compile and recompile.

The Partition Compile feature requires the VCS MX installation, however you don't need any additional VCS MX licenses or special license. Verilog only users can use their existing licenses for partition compile.

With Partition Compile you specify partitions in your design that you expect to revise and recompile often. VCS MX recompiles only the modified partitions.

You can specify these partitions two ways:

- In an `+optconfigfile` configuration file (for Verilog/SV designs)

- In a V2K or SV configuration (for Verilog/SV/VHDL/SystemC and MX designs)

Some of the partition compile features are:

- You can compile partitions independently and in parallel
- Support for cross-module references (XMRs) across partitions
- Reduced disk space across multiple partitions
- Faster IP integration in SOC environments

The Partition Compile use model is very similar to the existing VCS MX regular compile use model. No changes are required to the source code of the design and testbench when migrating to partition compile and runtime performance is not compromised when compared to the regular VCS MX compilation use model.

With Partition Compile, you designate separate partitions for various parts of the design and testbench, making sure to designate a partition or partitions for the part that you frequently revise and compile. You then see faster turnaround times from recompiling only some partitions and not needing to recompile other partitions.

You can also improve the compile-time performance by compiling these partitions in parallel on multicore machines instead of scratch compiling in regression mode.

Partition Compile also helps reduce the disk space for multiple tests if the design and test are in separate partitions.

This chapter explains how to use Partition Compile in the following sections:

- “[Partition Compile Use Model](#)” on page 544

- “Partition Compile Example” on page 546
- “Cell or Instance Based Partitions” on page 549
- “Package Based Partitions” on page 550
- “Limitations of the `+optconfigfile` Configuration File” on page 551
- “The Partition Compile Three Step Flow” on page 551
- “Specifying Partitions in a V2K or SystemVerilog Configuration” on page 553
- “Specifying a Location for the Partition Data Generation” on page 556
- “Parallel Compilation of Tests” on page 557
- “Parallel Compilation of Partitions” on page 559
- “Cross-Module References (XMRs) ” on page 560
- “Partition Compile Flow for SystemC-on-Top Designs” on page 561
- “Specifying Partitions in a VHDL Configuration File” on page 563
- “MVSIM Native Mode in Partition Compile” on page 564
- “Partition Compile Limitations” on page 566
- “Scenarios Causing a Recompilation of a Partition” on page 564
- “Achieving the Best Turnaround Time (TAT) with Partition Compile” on page 565

Partition Compile Use Model

The primary use model for Partition Compile follows these steps:

1. Create a partition configuration file.

For an elementary design that has a module named `mod` in a Verilog library named `MYLIB` (filename `mod.v`) and another module named `tb` (filename `tb.v`). To create a partition for each module definition, the configuration file is in [Example 21-1](#).

Example 21-1 Configuration File for Partition Compile

```
// configuration file config.txt
partition cell MYLIB.mod;
partition cell tb;
```

In [Example 21-1](#) one partition is for all instances of module `mod` and another partition is for all instances of module `tb`.

You can specify different partitions for different instances (see “[Cell or Instance Based Partitions](#)” on page 549).

For V2K or SV packages (see “[Package Based Partitions](#)” on page 550).

You can also use a V2K or SV configuration to specify partitions instead of using an `+optconfigfile` configuration (see “[Specifying Partitions in a V2K or SystemVerilog Configuration](#)” on page 553).

Note:

In addition to the partitions you specify, VCS MX creates internal or default partitions for the following parts of the design:

- a partition for the top-level module

- a partition for all SystemVerilog packages
 - a partition for all SystemVerilog interfaces
2. Compile for Partition Compile with the -partcomp option:

```
% vcs tb.v -y MYLIB +libext+.v -partcomp \
+optconfigfile+config.txt
```

The +optconfigfile option specifies the configuration file.

3. Run the simulation.

```
% simv
```

There is no runtime option for Partition Compile.

4. Based on the simulation results, modify the source code. Let's say you only modify tb.v.

```
% vi tb.v
```

5. Compile the design for Partition Compile again entering the same exact compilation command line:

```
% vcs tb.v -y MYLIB +libext+.v -partcomp \
+optconfigfile+config.txt
```

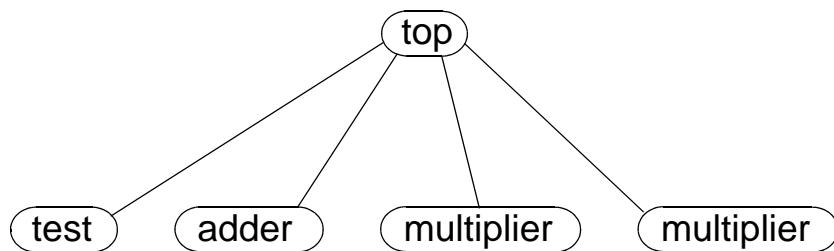
6. Run the simulation again.

```
% simv
```

Partition Compile Example

[Figure 21-1](#) shows the hierarchy in [Example 21-2](#).

Figure 21-1 Partition Compile Example Module Hierarchy



The top-level module named `top` instantiates the following (see [Example 21-2](#)):

- a testbench program block named `test`
- a module named `adder`
- two instances of a module named `multiplier`

Example 21-2 Partition Compile Example Source Files

```
// file top.v
module top;
    logic [7:0] log1, log2, log3, log4;
    wire [31:0] sum, prod1, prod2;
    adder a1 (sum,log1,log2);
    multiplier m1 (prod1, log1, log2);
    multiplier m2 (prod2, log3, log4);

    test t1 (log1, log2, log3, log4);
endmodule
```

```

// file test.v
program test (output [7:0] log1, log2, log3, log4);
...
endprogram

// file add_mult.v
module adder (output [31:0] sum, input [7:0] addend1,
addend2);
...
endmodule

module multiplier (output [31:0] product, input [7:0]
multiplier, multiplicand);
...
endmodule

```

Assuming multiple revisions to the program block named `test` and the module named `multiplier`, the configuration file to create the partitions is shown in [Example 21-3](#).

Example 21-3 Partition Compile +optconfigfile Configuration File

```

// +optconfigfile configuration file cfg.txt
partition cell test;
partition cell multiplier;

```

You specify a partition in this file with a line beginning with the keyword `partition`. The keyword `cell` specifies that the partition is for all instances of the module name or program block name that follows.

This configuration file specifies:

- one partition for program block `test`
- one partition for both instances of module `multiplier`.

The default and unspecified partition contains the top-level module `top` and module `adder`.

Compile the design and testbench with the `-partcomp` option and specifying the configuration file with the `+optconfigfile` option, for example:

```
% vcs -partcomp top.v test.v add_mult.v \
+optconfigfile+cfg.txt [other options]
```

You must repeat this same command line precisely each time you compile and elaborate the design and testbench.

You then simulate the design and testbench by running the `simv` executable as follows:

```
% simv [other options]
```

There is no runtime option for partition compile.

We now can modify the source code for the `multiplier` module and `test` program block, by editing the `add_mult.v` and `test.v` files.

We now compile these partitions by entering the same `vcs` command line:

```
% vcs -partcomp top.v test.v add_mult.v \
+optconfigfile+cfg.txt [other options]
```

VCS MX recompiles only the revised source code for the `multiplier` module and `test` program block.

You can then rerun the updated `simv` executable:

```
% simv [other options]
```

Cell or Instance Based Partitions

As shown in [Example 21-2 on page 546](#), you can specify a partition for all instances of a module definition or program block with the keyword `cell` following the keyword `partition`.

You can specify the Verilog library for a module or program block by prepending the library to the module name. [Example 21-4](#) is an example of specifying a library for a module definition in an `+optconfigfile` configuration file.

Example 21-4 Cell Based +optconfigfile Configuration File

```
partition cell MYLIB.mod;
partition cell tb;
```

In the first partition in [Example 21-4](#), the module definition named `mod` is in the Verilog Library directory named `MYLIB`.

You can also specify a partition for an instance of a module or program block with the keyword `instance` instead of the keyword `cell` as shown in [Example 21-5](#).

Example 21-5 Instance Based +optconfigfile Configuration File

```
partition instance top.t1;
partition instance top.m1;
partition instance top.m2;
```

[Example 21-5](#) is for the SystemVerilog code in [Example 21-2](#) and specifies three partitions:

- one for the instance `t1` of program block `test`
- one for instance `m1` of module definition `multiplier`

- on for instance m2 of module definition multiplier

A line in the +optconfigfile configuration file beginning with the partition and instance keywords can only have the hierarchical name for one instance. You cannot use instance based configurations to specify a partition for multiple instances, for that you need to use a module or cell based partition.

Package Based Partitions

Partition Compile allows you to create separate partitions for SystemVerilog packages, using the keyword package. For example:

```
partition package library1.package1
partition package library2.package2;
```

This example creates one partition for package package1 and one partition for package2. This example prepends the SystemVerilog library name to the package. Prepending a library is not required.

```
partition package library1.package1 library2.package2;
```

This example creates a single partition for both package1 and package2.

Note:

- Partitions for VHDL packages are not supported.
- VCS MX also creates an internal or default partition for all SystemVerilog packages, but this internal partition does not effect a package partition you specify.

Limitations of the `+optconfigfile` Configuration File

You can only use an `+optconfigfile` configuration file to specify partitions for Verilog and SystemVerilog code.

Partitions for VHDL code must use a VHDL configuration (see “[Specifying Partitions in a VHDL Configuration File](#)” on page 563).

Partitions for SystemC code must use a V2K or SV configuration (see “[Partition Compile Flow for SystemC-on-Top Designs](#)” on page 561).

There is no limitation prohibiting multiple `+optconfigfile` entries on the `vcs` command line, for example:

```
% vcs +optconfigfile+configfile1_specifying_a_partition \
+optconfigfile+configfile2_specifying_a_partition
```

This example used multiple `+optconfigfile` entries and configuration files to specify different partitions.

The Partition Compile Three Step Flow

Partition Compile supports the three step flow too (analysis, elaboration, and simulation).

Here are the steps to run the partition compile in the three step flow:

Analysis of Code

- Analyze the files using the regular VCS MX commands. There is no extra option needed at the analysis stage for Partition Compile.

- Verilog/SV designs and testbenches use the `vlogan` command.
- VHDL designs and testbenchesw use the `vhdlan` command. For more information see “[Specifying Partitions in a VHDL Configuration File](#)” on page 563.
- MX (mixed HDL) designs and testbenches use the `vlogan` and `vhdlan` commands.
- SystemC parts of a design or testbench use the `syscan` command. See “[Partition Compile Flow for SystemC-on-Top Designs](#)” on page 561

Here is the `synopsys_sim.setup` file for example [Example 21-2](#):

Example 21-6 synopsys_sim.setup File

```
WORK > DEFAULT
DEFAULT : ./rtllib
rtllib : ./rtllib
```

The example `vlogan` command line to analyze the source code in [Example 21-2](#) is as follows

```
% vlogan -work rtl -sv test.v add_mult.v top.v \
[other options]
```

After analyzing the files, the next step is elaboration. The example `vcs` command to compile the design in the Partition Compile flow is as follows :

```
% vcs -partcomp top +optconfigfile+cfg.txt [other options]
```

Specifying Partitions in a V2K or SystemVerilog Configuration

There are two ways to specify a partition in Partition Compile:

- with a `+optconfigfile` configuration file
- with a V2K/SV configuration.

This section describes using a V2K/SV configuration.

Partition Compile adds an extension keywords to V2K/SV configurations to specify partitions. [Example 21-7](#) is a V2K/SV configuration for the example source code in [Example 21-2](#).

Example 21-7 Partition Compile V2K/SV Configuration File

```
// topcfg.v
config topcfg;
design top; // top-level module
partition instance top.t1; // partition for program test
instance top.m1 use multiplier;
instance top.m2 use multiplier;
partition instance top.m1; // partition for multiplier
                         //instance m1
partition instance top.m2; // partition for multiplier
                         // instance m2
endconfig
```

The new keyword `partition` begins a partition specification.

The keyword `instance` specifies that the partition is for an instance of a module or a program block. The keyword `cell` specifies that the partition is for all instances of a module or a program block. For a discussion of which to use see “[Instance or Cell Based Partitions in a V2K/SV Configuration](#)”

The above configuration specifies:

- one partition for program block test
- one partition for the m1 instance
- one partition for the m2 instance.

The default and unspecified partition contains the top-level module top and module adder.

2 step commands for partition compile

Add the topcfg.v file and -top config_name option to the vcs command line, for example:

```
% vcs -partcomp -top topcfg topcfg.v top.v test.v \
add_mult.v [other options]
```

Then run the simv executable.

```
% simv
```

3 step commands for partition compile

Analyze the partition configuration file topcfg.v file, for example:

```
% vlogan -work rtllib -sverilog test.v add_mult.v top.v
topcfg.v [other options]
```

Then you enter only the configuration name on the vcs command line, for example:

```
% vcs -partcomp topcfg [other options]
```

Then run the simv executable.

```
% simv
```

Instance or Cell Based Partitions in a V2K/SV Configuration

You can specify the partitions in the V2K/SV configuration file based on the `instance` or `cell` keyword. The configuration shown in [Example 21-8](#) uses `cell` entries, following the keyword `partition`, to specify a partition for all instances of a module or testbench.

Example 21-8 Partitions By Module or Cell

```
// topcfg.v
config topcfg;
design rtllib.top; // top-level module name
default liblist rtllib;
partition cell test; // partition for test cell
instance top.m1 use multiplier;
instance top.m2 use multiplier;
partition cell multiplier; // partition for multiplier cell
endconfig
```

In this configuration separate partitions are specified for:

- all instances of the cell `multiplier`
- all instances of the cell `test`.

The configuration shown in [Example 21-9](#) uses `instance` entries, following the keyword `partition`, to specify partitions for specific instances.

Example 21-9 Partitions By Instances

```
// topcfg.v
config topcfg;
design rtllib.top; // top-level module name
default liblist rtllib;
partition instance top.t1; // partition for
```

```

                // testbench instance t1
instance top.m1 use multiplier;
instance top.m2 use multiplier;
partition instance top.m1; // partition for
                           // multiplier instance m1
partition instance top.m2; // partition for
                           // multiplier instance m2
endconfig

```

In this configuration separate partitions are specified for:

- the instance `top.t1` of the testbench program `test`
- the instance `top.m1` of module definition `multiplier`
- the instance `top.m2` of module definition `multiplier`

Specifying a Location for the Partition Data Generation

Use the `-partcomp_dir=dir_path` option to specify a directory for the partition data, for example:

```
% vcs -partcomp top.v test.v add_mult.v \
+optconfigfile+cfg.txt -partcomp_dir=./PARTCOMP
[other options]
```

Using this example, VCS MX creates a directory named `PARTCOMP` to contain the partition data.

For MX or VHDL designs, a directory named `vhdl_objs_dir` is created for data related to VHDL partitions. This directory is located in the current path by default. The `-partcomp_dir` option does not change the location of the `vhdl_objs_dir` location.

Parallel Compilation of Tests

Partition compile allows you to compile and simulate multiple tests (modeled as programs or modules) in parallel.

Each test case should be in a separate directory and have its own simulation executable.

These simulation executables are only small wrapper executables containing the test case information, so they use far less disk space than the regular VCS MX executables if they ran in parallel.

Compilation of each test case is faster because the executable contains only the test case. They share the compiled data from other portions of the design.

Because the test cases are compiled and simulated simultaneously, make sure that compilation of one test case does not overwrite the compiled data of another test or trigger compilation of other portions of the design shared by multiple tests. To ensure this, adopt the following use model for parallel compile of tests.

1. The `synopsys_sim.setup` file should be set up so that the physical to logical library mapping for the test is configurable using an environment variable. Each test sets its own physical path to the environment variable for the logical test library. For example:

```
WORK > DEFAULT
DEFAULT : work
DUT_LIB : /u/design/lib_dut
TEST_LIB : $TEST_RUN_DIR/lib_test
```

2. All the tests use the same configuration file.

The following is the `+optconfigfile` configuration file:

```
//cfg.txt
partition instance top.m1;
partition instance top.p1;
```

If you prefer, you can use a V2K or SV configuration, for example:

```
config topcfg;
    design top;
    partition instance top.m1;
    partition instance top.p1;
endconfig
```

3. Compile and elaborate the tests in parallel from different terminals in their respective directories.

Source file `test1.v` is in directory `/usr/design/test1` and `test2.v` in `/usr/design/test2`. We compile the entire design (with a sample test) in the directory `/usr/design`.

For `test1`:

```
% cd /usr/design/test1
% setenv TEST_RUN_DIR `pwd` 
% vcs -partcomp top.v test.v add_mult.v \
+optconfigfile+cfg.txt [other options]
```

In parallel, for `test2`:

```
% cd /usr/design/test2
% setenv TEST_RUN_DIR `pwd` 
% vcs -partcomp top.v test.v add_mult.v \
+optconfigfile+cfg.txt [other options]
```

Note:

You can generate the partitions by running the first test and then reusing the common partition data for the remaining tests running in parallel with the `-partcomp_sharedlib=dir_path` option.

The vcs command line for test1 is:

```
% vcs -partcomp top.v test.v add_mult.v \
-partcomp_dir=./PARTCOMP_test1 [other options]
```

The command line for test2 is:

```
% vcs -partcomp top.v test.v add_mult.v \
-partcomp_dir=./PARTCOMP_test2 \
-partcomp_sharedlib=./PARTCOMP_test1 [other options]
```

VCS MX reuses the common DUT partitions from the PARTCOMP_test1 directory and issues messages that tell you the partitions being reused in the current compilation.

Parallel Compilation of Partitions

When scratch compile times are too long, partition the DUT and compile the partitions in parallel. Parallel compilation of partitions can significantly improve scratch compile times.

To enable the parallel compilation of partitions, use the -fastpartcomp=j option on the vcs command line. For example:

```
% vcs top.v test.v add_mult.v -partcomp -fastpartcomp=jN
[other options] [other partcomp options]
```

where N is the number of parallel processes. For example:

```
% vcs top.v test.v add_mult.v -partcomp -fastpartcomp=j4
[other options] [other partcomp options]
```

This command allows the compilation of a maximum of 4 partitions in parallel on a 4 core (or above) machine.

Cross-Module References (XMRs)

When your design is partitioned there is a chance that there are cross module references (XMRs) across partitions; that is, signals and variables from one partition being accessed from another partition. You can handle XMRs as follows when using the partition compile flow:

- Partition compile allows XMRs across partitions with the use of a XMR configuration file. The configuration file contains information on all the XMRs that could be potentially used in all tests for the design. Specify the XMR configuration file at compilation time using the `+optconfigfile+config_file` option. Use the following syntax for the configuration file:

```
xmr {module_or_program_name} {signal_name};
```

- Use the `-partcomp=noxmrconfig` elaboration option so that only the changed partition is recompiled. This option comes with a small runtime performance cost.
- If none of the above options are used and:
 - No XMRs change, then there is no extra recompile.
 - XMRs change, both the referrer and referee partitions are recompiled. There is no negative impact on runtime.

Note:

Cross-module references between partitions or to or from subroutines, methods, tasks, or functions do not require either option.

Partition Compile Flow for SystemC-on-Top Designs

Partition compile works with SystemC-on-top designs. Consider [Example 21-10](#), which contains a top-level module named `sc_top.h` and two child modules: a Verilog module named `vlog_mod.v` and a SystemC module named `sc_mod.h`.

Example 21-10 SystemC-on-Top with Partition Compile

```
module vlog_mod{.....}      //vlog child vlog_mod

SC_MODULE(sc_mod) {....} ;   //SystemC child "sc_mod"

// SystemC top module sc_top
SC_MODULE(sc_top) {
    // instantiate vlog_mod and sc_mod here
    vlog_mod vlog_mod_o;
    sc_mod sc_mod_o;
    SC_CTOR(sc_top) : vlog_mod_o("vlog_mod_o"),
                      sc_mod_o("sc_mod_o") {.....}
};

int sc_main(int argc, char** argv)
{
    sc_top sc_top_o("sc_top");
    sc_start(100,SC_NS);
}
```

Partitioning Your Design

Partition your design using a V2K or SV partition configuration file, as shown in [Example 21-11](#). This example configuration file works for the code shown in [Example 21-10](#).

Example 21-11 SystemC Partition Compile V2K or SV Configuration File

```
config partcomp_cfg;
```

```
design LIB.sc_top; // systemC top module
instance sc_top.vlog_mod_o use DEFAULT.vlog_mod;
partition instance sc_top.vlog_mod_o;
default liblist DEFAULT;
endconfig
```

Compiling the SystemC Partition Compile Example

To compile the SystemC partition compile example, follow these steps:

1. Analyze the Verilog child module:

```
% vlogan -sysc sc_model vlog_mod vlog_mod.v
```

2. Compile the SystemC module:

```
% syscan sc_mod.cpp sc_top.cpp
```

3. Analyze the configuration file:

```
% vlogan partcomp_cfg.v
```

4. Elaborate the design using the following options:

```
% vcs -sysc partcomp_cfg -partcomp \
-partcomp_dir=".my_partitionlib"
```

Specifying Partitions in a VHDL Configuration File

VHDL and VHDL-on-top, mixed-HDL designs require a VHDL configuration file to specify the partitions (see [Example 21-12](#)). The syntax and semantics of the configuration file are the same as for a regular VHDL configuration file, with the addition of the new keyword `partition`.

Example 21-12 VHDL Configuration File for Partition Compile

```
//file topcfg.vhd
library IEEE;
library work;
USE STD.TEXTIO.all;
USE IEEE.STD_LOGIC_TEXTIO.all;
use IEEE.STD_LOGIC_1164.all;
use work.all;
configuration topcfg of top is
for top_arch
    for partition i1: child use entity work.child(module);
    end for;
end for;
end;
```

Note:

The `partition` keyword with `for all` and `for other` clauses of a VHDL configuration is not allowed in partition compile.

Analyze the VHDL configuration file along with other VHDL source files using the `vhdlan` command. For example:

```
% vhdlan top.vhd topcfg.vhd
```

MVSIM Native Mode in Partition Compile

You can apply the incremental changes to the design source or IEEE 1801, also known as the Unified Power Format (UPF) files, or you can modify the low power options.

Make sure the `synopsys_sim.setup` file has the UPF library mapping:

```
upf: $VCS_HOME/$platform/packages/upf
```

Where, `$platform` is Linux for a 32-bit VCS build and amd64 for a 64-bit VCS build.

Scenarios Causing a Recompilation of a Partition

The following cases cause a recompilation of a partition:

- Change of a module definition in a partition, either instance- or cell-based.
- Change of a vcs command-line option.
- Change in the `synopsys_sim.setup` or `+optconfigfile` file.
- Global design property changes like adding `$dumpvars` to a partition, in which case all partitions need to be recompiled.
- Changes to the ports or parameters that affect the ports of a partition. If there is a parent and child partition, and this change is only to the child partition, the parent need not be recompiled.

- Changes to \$unit scope. Any change (addition of new signal, type, module, or interface) to a \$unit causes recompilation of all the partitions associated with the \$unit.
- Changes to an interface require the recompilation of all modules connected to that interface.
- Changes to a shared package. In this case all the partitions sharing this package require recompilation.
- Reanalysis of source code (`vlogan` or `vhdlan`) with different command-line options causes recompilation of the source code and all partitions that share the logical library.

Achieving the Best Turnaround Time (TAT) with Partition Compile

To get the maximum gain in turnaround time (TAT) from the partition compile flow, note the following:

- Fresh compilation and recompilation of very large partitions take the most time. Modules or test cases being modified frequently should be made separate partitions.
- A high number of partitions increases the initial compilation time.
- Some coding styles mentioned in the section on “[Scenarios Causing a Recompilation of a Partition](#)” on page 564 can also lead to recompilation of multiple partitions, leading to loss in TAT productivity.
- Set up your test cases so that multiple test cases can be compiled in parallel and do not over-write the compiled data or other test cases (see “[Parallel Compilation of Tests](#)” on page 557).

- Make external VIPs or IPs separate partitions; they don't need to be frequently recompiled.
- If a module has many instances, then all instances of the module should be made partitions, or the module made as a partition by specifying the module as a cell, so that the module is not compiled in multiple partitions. Partition compile makes a single partition if multiple instances of a module are specified as `partition` in the configuration file.

Partition Compile Limitations

The following technologies are not supported with partition compile:

- OpenVera testbench shell and shared object files, as specified with the `-ntb_cmp` and `-ntb_vl` elaboration or compile-time options.
- The timing optimizer (`+timopt`).
- Searching for the `PATHPULSE$ specparam` in specify blocks with `+pathpulse`.

You cannot use Partition Compile with self instances of virtual interfaces `interface::self()`

For SDF backannotation with partition compile, the following limitations apply:

- The delays in any SDF file must be for a single partition; they cannot cross partition boundaries.
- Two or more instances of a partition cannot have different SDF files.

- Conditional and delayed SDF annotation are not supported.
- For mixed-HDL designs, the VHDL part cannot have SDF backannotation.

Code coverage has the following limitations:

- Detecting coverable objects (statements, conditions, toggles) that can never execute and automatically excluding them from coverage with `-cm_noconst`.

AMS is not supported, but mixed-signal simulation with FastSPICE, NanoSim, or XA is supported.

Using the `+optconfigfile` method of specifying partitions is only enabled for Verilog and SystemVerilog code.

For mixed HDL designs:

- A partition that includes VHDL inside Verilog or Verilog inside VHDL cannot be specified with the `+optconfigfile` method.
- A partition for Verilog that is inside VHDL must be specified with a VHDL configuration to specify the Verilog partitions.
- A partition for VHDL that is inside Verilog must be specified with a Verilog configuration to specify the VHDL partitions.

MVSIM Native Mode Limitations

The following limitations apply in MVSIM native mode:

- Partitioning of instances in an instance array is not supported.
- The `bind_checker` command is not supported.

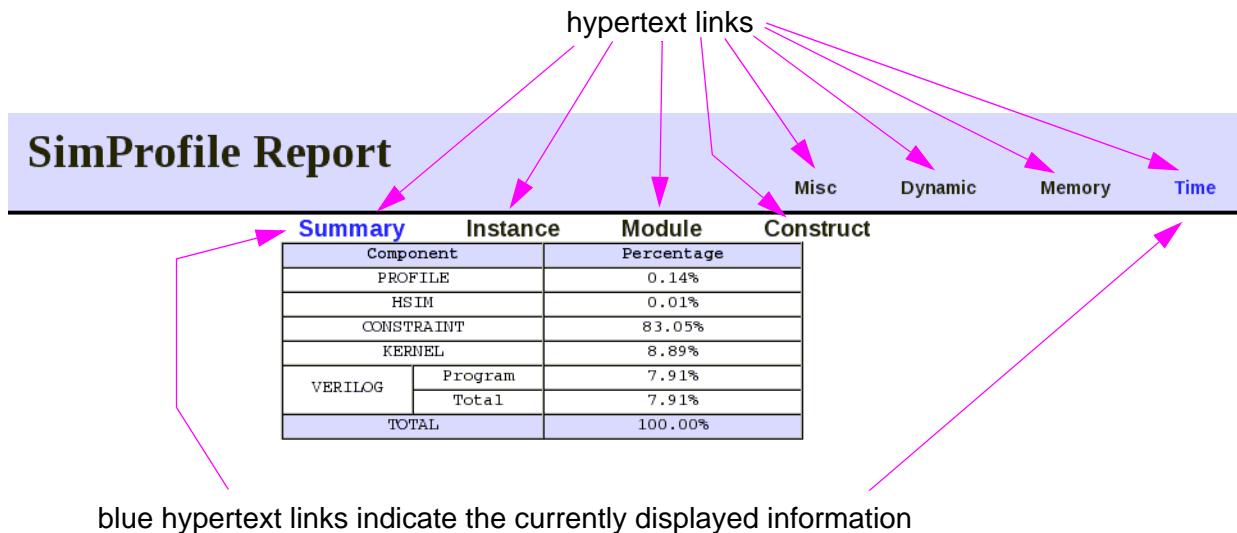
- Level Shifter inference and 30-70 corruption is not supported.

22

Profiling the SystemC Portion of a Design

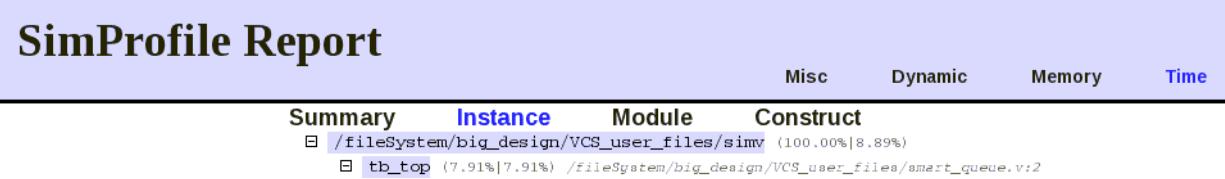
If the simulation contains SystemC modules, then the unified simulation profiler shows CPU time specific to SystemC. The following views are from a SystemC cosimulation. The first figures are of views created by VCS or VCS MX after simulation, the next set of views are created by the profrpt utility. The code examples for these views is in the \$VCS_HOME/doc/examples/systemc/vcs/vcs_profiler.

Figure 22-1 The ProfileReport.html



A single CPU time number is reported for all SystemC modules together. This is the "SystemC 9.93%" in figure 18-20. This single number contains the CPU time used by all SystemC processes together and also includes CPU time spend within the SystemC kernel as well as CPU time needed to exchange value updates between SystemC and Verilog/VHDL.

Figure 22-2 Simulation Time Instance View



SystemC module instances are listed (figure 5-2), however, their individual CPU time is not known and always printed as 0% in this instance-based report. Memory used by SystemC is not reported at all.

If the simulation happens to have SystemC on top and is elaborated with syscsim, then a surplus profileReport directory is created during the elaboration. Please ignore this directory. No surplus directory is created when the simulation is compiled in UUM flow with "vcs -sysc sc_main ...".

For more information on Unified Simulation Profiler, see ["The Unified Simulation Profiler" on page 465](#).

Limitations

The following are the limitations of profiling the SystemC portion of a design:

- VCS -simprofile does not report memory allocated by SystemC.
If a Donut or SystemC-on-top design (using non-UUM flow) is invoked during elaboration, VCS generates multiple profile reports.

23

Multi-Driver Support for Wreal

This chapter describes the multi-driver support provided by VCS for wreal nets.

This chapter consists of the following sections:

- “Introduction”
- “Features Supported in this VCS Release”
- “Use Model”
- “Limitations”

Introduction

There is increased usage of digital real numbers to develop analog or mixed-signal functional model, and to speed up the simulation performance using a digital simulator. This is useful to:

- Investigate the system characteristics, before starting transistor-level design.
- Verify connectivity of an entire system.
- Enable a top-down design methodology.

If the real valued model is entirely simulated by the digital simulator, it improves the simulation performance tremendously, and uses digital simulator and advanced digital methodology like test bench, assertion, and coverage to reduce the time required to perform intensive verification. However, there is a trade-off between performance and accuracy. If a SPICE model is available, you can swap a real valued model by SPICE model to check for more accuracy and finding circuit problem, then you can reuse the same test bench and may use both the digital top or SPICE top design style.

The digital methodology is used to develop real-valued behavioral model, using VHDL real number. VHDL language allows a user-defined resolution function, which in-turn allows a real-type or sub-type with multiple drivers. This real-type or sub-type with multiple drivers is powerful to develop the real-valued behavioral model. Other languages such as Verilog, SystemVerilog, and Verilog-AMS support real number (wreal), but without any resolution function, multiple drivers, or INOUT port. Due to this limitation, you cannot develop analog functional model using SV real number.

Note:

The multi-driver support for wreal is available from VCS version 2010.06.

Features Supported in this VCS Release

This VCS release supports:

- Resolution functions such as default, min, max, and sum.
- The wreal z and x states are represented using predefined macros `wrealZState and `wrealXState.
- Wreal and SV real connection.
- Array of wreal, including at (parameterized) port declaration.
- Wreal to SPICE and SPICE to wreal connection in mixed-signal verification.
- The \$table model in digital domain using wreal.
- The INOUT wreal port.

Multiple Drivers and Resolution Function on Wreal Net

A resolution function is required when there are multiple drivers. VCS provides a built-in resolution function, which can be selected by you using the `wreal <res_func>` compile-time option, where “`res_func`” can be either `res_def`, `res_sum`, `res_min`, or `res_max`.

- `res_def` – Single active driver only. Support for x and z state
- `res_sum` – Resolved value is the sum of all the drivers value

- `res_min` – Resolves to the minimum value of all the drivers value
- `res_max` – Resolves to the maximum value of all the drivers value

Example for the Driver Resolution:

DR#1	DR#2	default	sum	min	max
1.5	2.5	x	4.0	1.5	2.5
1.5	z	1.5	1.5	1.5	1.5
1.5	X	x	x	x	x
z	z	z	z	z	z
x	X	x	x	x	x

Wreal X and Z states

Predefined macros are used to describe x (`~wrealXState`) and z (`~wrealZState`) states.

Example:

```
module myMod(r);
parameter xval = 1.25;
parameter zval = 1.5;
inout r;
wreal r;
real r1;
reg clk;
initial clk = 0;
```

```

always #5 clk = ~clk;

assign r = r1;
always @(clk) begin
    If ( clk == 1'b1)
        r1 = 3.3;
    else
        r1 = `wrealZState;
end

always @(r) begin
    if (r == `wrealXState) begin
        $display($time,,,"%m: Wreal net r has value X");
        r1 = xval;
    end
end
endmodule;

```

Expressions

Expressions using wreal x and z states results in wreal x state (consistent with digital behavior). For a wreal net w1, the resulting expressions are as follows:

- $w1 (+, -, *, /) `wrealXState = `wrealXState$
- $w1 (+, -, *, /) `wrealZState = `wrealXState$

If the argument is `wrealXState and `wrealZState, then the resultant value should be `wrealXState.

Arithmetic expressions involving 4-state integers that have x or z bits and real numbers will also result in `wrealXState. The \$realtobits for `wrealXState and `wrealZState should result in 0 (similar to 2-state conversion).

Operators

The following conditional expressions are supported on wreal:

- `>`, `<`, `==`, `==`, `>=`, `<=`
- Wreal and SV real numbers can be compared with signed and unsigned integers.
- The operations involving `x` or `z` values results in `x`, similar to integers.
- The ternary operator is supported.
- The logical operators `&&` and `||` are supported, and are similar to integers.

Unsupported Operators

The following operators are not supported on wreal:

- Bitwise operators: `&`, `|`, `^`, `~`, `~&`, `~|`, `~^`
- Reduction operators: `&`, `|`, `^`, `~`, `~&`, `~|`, `~^`
- Shift operators: `>>`, `<<`, `>>>`, `<<<`

Array of wreal

- The wreal arrays can be connected to real or wreal arrays of the exact size. The size mis-match results in an error.
- Wreal array cannot be connected to a wire array.
- Dynamic arrays of wreal type are not supported.

Initial Value of Wreal

As per LRM, a wreal net is initialized to 0.0. However, with multiple driver support, 0.0 may not be the correct initialized value. So, the uninitialized wreal nets will have Z STATE like other net type.

Use Model

- VCS allows wreal usage for pure digital simulation with the `-realport` compile-time option.
- For mixed-signal verification with Nanosim or XA, the `-ams` option enables wreal usage.
- The `wreal <res_func>` compile-time option is required to:
 - Enable multiple drivers on wreal net
 - Choose the resolution function

```
% vcs -wreal res_def test.v
```

Limitations

- User-defined resolution functions are not supported.
- Connecting wreal array to real VHDL array is not supported.
- PLI force Z and X states.
- Handle value translation of X and Z states from wreal net to SPICE (dependency on analog solver) is not supported.
- Real x and z states are not supported in VCD, EVCD, and FSDB.
- INOUT wreal ports are not supported at MX boundary.

- INOUT wreal ports are not supported at SPICE boundary in the SV-SPICE flow.
- You cannot apply force or release on wreal.

24

MVSIM Native Mode Features

This chapter contains the following features:

- “[Support for Isolation of SVD Constructs](#)” on page 581
- “[Partition Compile Support](#)” on page 582
- “[Isolating Interface Modports](#)” on page 583

Support for Isolation of SVD Constructs

MVSIM native mode supports the virtual isolation of the following SystemVerilog Design (SVD) constructs:

- logic, int, struct, enum, reg, shortint, and union
- multidimensional arrays

Limitations

- Real data type is not supported.
- Supply0, supply1, and tri are not supported.

Partition Compile Support

You can apply the incremental changes to the design source or IEEE 1801, also known as the Unified Power Format (UPF) files, or you can modify the low power options.

Make sure the `synopsys_sim.setup` file has the UPF library mapping:

```
upf: $VCS_HOME/$platform/packages/upf
```

Where, `$platform` is Linux for a 32-bit VCS build and amd64 for a 64-bit VCS build.

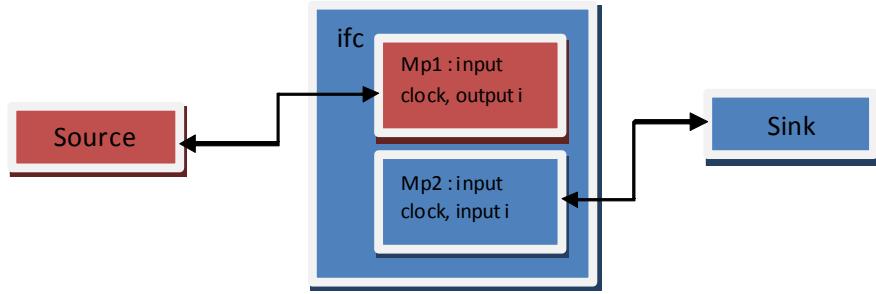
Limitations

The following limitations apply in MVSIM native mode:

- Partitioning of instances in an instance array is not supported.
- The `bind_checker` command is not supported.
- Level Shifter inference and 30-70 corruption is not supported.

Isolating Interface Modports

You can isolate input/output of interface modports in SVD designs.
This feature is enabled by default.



Limitations

- Isolation on interfaces without modports will not work, as there is no direction associated with the signals.
- Only `-self` and `-parent` locations are supported.
- Clamp value 0/1/z are supported.
- Simple directional ports in the interfaces are not supported.

25

SAIF Support for SystemVerilog Data Types

SAIF supports monitoring of SystemVerilog data types in SystemVerilog designs.

This chapter consists of the following sections:

- “Use Model”
- “Usage Examples”
- “Supported SystemVerilog Data Types”
- “SAIF File Format for Unpacked Struct”

Use Model

You must pass the `sv` keyword to the `$set_gate_level_monitoring` system task to monitor SystemVerilog datatypes in SystemVerilog designs. If you want to dump SAIF file for SystemVerilog through the UCLI command, you must pass the `sv` string to `power -gate_level` command.

Note:

You must use the `-debug_pp` compile-time option at vcs command-line to dump SystemVerilog object.

Usage Examples

Enabling SystemVerilog SAIF Dumping Using System Task

The following example shows how to dump SAIF file for SystemVerilog using the `$set_gate_level_monitoring` system task:

```
initial begin
    $set_gate_level_monitoring("rtl_on", "sv");
    $set_toggle_region(top);
    $toggle_start;
    a = 0;
    #2 a = 1;
    $toggle_stop;
    $toggle_report("new.saif", 1e-9, top);
end
```

Enabling SystemVerilog SAIF Dumping Using UCLI Command

The following example shows how to dump SAIF file for SystemVerilog using the UCLI command:

```
% simv -ucli  
ucli% power -gate_level on sv  
ucli% power <scope>  
ucli% power -enable  
ucli% run 100  
ucli% power -disable  
ucli% power -report <saif_filename> <timeUnit> <modulename>  
ucli% quit
```

Supported SystemVerilog Data Types

The following SystemVerilog data types are supported in the SAIF file:

- Integer data types: byte, int, shortint, longint
- Logic, bit
- Enum type
- Packed struct
- Unpacked struct
- Packed union
- Interface
- Interface with modport

SAIF File Format for Unpacked Struct

The following is an example for SAIF file format for unpacked struct:

```
typedef struct {
    bit [1:0] bv;
    byte byteval;
} UPS2state;
UPS2state s1;

in SAIF file it shows like:
(s1\[bv\]\[1\])
    (T0 5) (T1 95) (TX 0)
    (TC 1) (IG 0)
)
(s1\[bv\]\[0\])
    (T0 92) (T1 8) (TX 0)
    (TC 2) (IG 0)
)
(s1\[byteval\]\[7\])
    (T0 92) (T1 8) (TX 0)
    (TC 2) (IG 0)
)
(s1\[byteval\]\[6\])
    (T0 5) (T1 95) (TX 0)
    (TC 1) (IG 0)
)
```

Index

Symbols

+optconfigfile [564](#)
+optconfigfile+config_file [560](#)
+pathpulse [566](#)
+timopt) [566](#)
\$dumpvars [564](#)
\$scope [210](#)
\$unit [565](#)
\$vcdblusion [210, 427](#)
\$VIRTIO_HOME [445](#)

A

About the HVP Editor [450](#)
About VMM Planner [450](#)
adaptive exclusion [168](#)
Adding User-Defined Attributes in HVP Editor
[456](#)
ahb_rtl_cosim [426](#)
-assert concfail [354](#)
-assert failonly [354](#)
-assert svvunit [352](#)

B

backannotation [566](#)

C

CBug [426](#)
-cflags [422](#)
checksum [151](#)
-cm_noconst [567](#)
compilation
 fast [463](#)
concfail [354](#)
coverage
 report-time exclusion [149](#)
covergroup
 stand-alone [165](#)
CoWare [418](#)
Cowaare [415](#)
Creating a Subplan in VMM Planner Editor [459](#)
--cwr_nosession [425](#)

D

Direct Kernel Interface [417](#)
DKI [417](#)
DPI-C [411](#)
DVE [446](#)
 detail pane [189](#)
dve [426](#)

E

echo procedural sampling 159
EVCD 210
excl_bypass_check 150
-excl_resolve on 171
-expand_dumpport_scope 210

F

fail-only 354
failonly 354
fast compilation 463
-fastpartcomp=j 559
FastSPICE, 567
fileevent 213, 214
fileevent.tcl. 216

G

gcc v4.2.2 419

I

Innovator 415, 443
interop flow 408
IP 566

L

LD_LIBRARY_PATH 423
-LINK_FLAGS 422
-LINK_LIBS 422

M

mem_solver
argument to the profrpt -view option 473, 520
Memory Constraint Solver view
in profiler reports 473, 520, 530

multicore 416
multiple drivers 163

N

NanoSim 567
navigation pane 176
non-rand case 165
-ntb_cmp 566
-ntb_vl 566

O

OSCI-TLM 2.0 402

P

parallel compilation 557
of partitions 559
-partcomp 548, 558
-partcomp_dir=dir_path 556
-partcomp_sharedlib=dir_path 558
-partcomp=noxmrconfig 560
partially dumped vpd 222
partition 563, 566
partition compile 541
best TAT 565
configuration file 561
example 546
limitations 566
MVSIM native mode 564
using 544
VHDL configuration 563
with SystemC 439
with SystemC-on-Top 561
with XMRs 560
partitions
instance or cell 549
package-based 550
PATHPULSE\$ 566

Platform Analyzer. 444
Platform Architect 416
Platform Architect MCO 417
Platform Creator Tool 428
PLI 418
procedural sampling
 echo 159
 sampling non random cover point, standalone
 covergroup, passing rand through
 function or task 159
profiler
 limitations with SystemC 439
 simulation 465
profiling
 SystemC 430
profrpt 431
PSL 352

R

rand case 165
RedHat 4 419
RedHat 5 419
reuse of exclusion file 168
review markers 176

S

SC_CTHREADS 431
sc_main 407, 441
SC_METHODS 431
sc_spawn 404
sc_start 407
SC_THREADS 431
scsh 425
SDF 566
Signature of exclusion 183
sim_start 426
-simprofile 429, 571
–simprofile 430, 431, 433

simv -gui 425
SNPS_VP_HOME 420
SoCs 416
Solaris 429
Sparc 429
SuSe 10 419
SuSe 11 419
SVA 352
SVA PP, SystemVerilog Assertions Post-processing, SVAPP, Post-processing SVA 57
SVAPP Limitations, Limitations SVAPP 63
SVAPP syntax, syntax for SVAPP 58
synopsys_sim.setup 557, 564, 582
-sysc 440
SYSC_USE_SKELETON 440
-sysc=adjust_timeres 446
-sysc=inno 422
-sysc=newsync 446
-sysc=show_sc_main. 441
-sysc=snps_vp 422, 429
-sysc=unihier 439
syscan 422
SystemC 444
 partition compile 439
 profiling 430
 standard library classes 365
SystemVerilog 352

T

Tcl
 socket-based communication 214
Time Constraint Solver view
 in profiler reports 472, 520, 522
time_solver
 argument to the profrpt -view option 472, 520
TLI 392, 439
TLM-2 443
-toolexe 426

transaction-level models 416

U

UCAPI 150

-ucli2Proc 222

ucliCore

tempPauseResume 214

tempPause 214

unified simulation profiler 465

unmappable exclusions 192

Using the VMM Planner Editor 449

UUM 443

V

VCS_INSTALLED_AT 420

VCS-SystemC 446

VG GNU 421

vhdl_objs_dir 556

view full hierarchy 222

VIP 566

Virtual Platform Analyzer 424

virtual platform models
simulating 415

Virtualizer 416, 417

vlogan 445

vmm_data 368

vmm_factory 368

vmm_group 370, 406

vmm_log 371

vmm_object 387

vmm_opts 387

vmm_simulation 377, 406

vmm_subenv 377

vmm_test 377, 406

vmm_timeline 390, 391, 405

vmm_unit 390, 391, 406

vmm_xactor 390, 391

VMM-SC 366, 392, 402

base class library 367

VMM-SystemC 366

VPA 424

vpash 426

vpash_script.tcl 426

VPD 427

-vpd 427

vpd2vcd 210

VPExplorer 418, 424

VPI callback 354

vpsession_sim.conf 425

VPViewer 418, 424

vunit 352

W

Windows 429

Working with HVP Files 450

X

x86 429

XA 567