

Discovery Visual Environment User Guide

G-2012.09
September 2012

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

SYNOPSYS®

Copyright Notice and Proprietary Information

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

Right to Copy Documentation

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

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

Destination Control Statement

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

Disclaimer

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

Registered Trademarks (®)

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

Trademarks (™)

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

Service Marks (sm)

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

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

ARM and AMBA are registered trademarks of ARM Limited.

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

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

Contents

1. Getting Started	1-1
Overview	1-2
Enabling Debugging	1-3
Debug Options	1-3
Required Files	1-4
Invoking DVE	1-5
64-bit Mode	1-6
Interactive Mode	1-6
Starting an Interactive Session from the DVE GUI	1-7
Post-Process Mode	1-11
Using the <code>-vpd</code> command	1-11
Loading the Design Database File in the DVE GUI	1-11
Using Session File	1-13
Using the <code>-session</code> command	1-13
Loading a Session File in the DVE GUI	1-13
Using Tcl Scripts	1-14
Passing DVE Arguments from Simulator Runtime Command Line	1-14
Saving a Session or Layout	1-16

Saving the Current View	1-18
Restoring a Saved Simulation	1-20
Closing a Database.....	1-22
Exiting DVE	1-22
DVE Log Files	1-22
DVE Licensing Queuing	1-23
DVE Setup Files	1-24
Managing User Setup Files	1-26
Usage.....	1-26
Typical Symbols Used in DVE.....	1-28
Special Symbols Used in DVE.....	1-29
Low Power Symbols Used in DVE.....	1-30
 2. Using the Graphical User Interface	2-1
Overview of DVE Window Configuration.....	2-2
Creating a Window Title for All Views and Panes	2-6
Managing DVE Panes and Views	2-8
Managing Target Views	2-9
Maximizing View	2-11
Docking and Undocking Views and Panes	2-13
Dragging and Dropping Docked Windows	2-13
The Console Pane.....	2-14
The Watch Pane	2-15
The Memory View	2-16
Setting Properties of Signal in Memory View.....	2-16
C, C++, and SystemC Code	2-19

Using the Menu Bar and Toolbar	2-20
Searching Signals or Scopes	2-20
Mapping to the Location of the Source Files	2-22
Interactive Mode	2-22
Use Model	2-22
Post-process Mode	2-23
Use Model	2-23
Editing Preferences	2-24
Using Context-Sensitive Menu	2-24
 3. Using the Hierarchy and Data Panes	3-1
The Hierarchy Pane	3-2
Scope Types and Icons	3-3
Filtering the objects in the Hierarchy Pane.....	3-5
Navigating Open Designs	3-5
Expanding and Collapsing the Scope	3-6
Rearranging Columns in the Hierarchy Pane	3-6
Populating Other Views and Panes	3-7
Displaying Variables in the Data Pane	3-7
Dragging and Dropping Scopes	3-7
Dumping Signal Values	3-9
Moving Up or Down in the Hierarchy Pane	3-10
The Data Pane	3-16
Viewing Signals and Values	3-17
Filtering the Signals	3-18
Forcing Signal Values	3-19
Viewing Interfaces as Ports	3-23
Viewing \$unit Signals	3-26

Debugging Partially Encrypted Source Code.....	3-28
4. Using the Source View	4-1
Loading Source Code	4-2
Loading a Source View from the Hierarchy Pane	4-2
Loading a Source View from the Assertion View.....	4-3
Displaying Source Code from a File	4-4
Using the Mouse in the Source View.....	4-5
Working with the Source Code	4-6
Expanding and Collapsing Source Code View	4-6
Displaying Include File as Hyperlink	4-6
Example	4-6
Editing Source Code	4-10
Selecting and Copying Text to the Clipboard.....	4-11
Color-coding the Source File	4-11
Setting Desired Color for Inactive 'ifdef `else Code in DVE.....	4-13
Usage Example	4-14
Navigating the Design from the Source View	4-17
Navigating Code in Interactive Simulation.....	4-18
Setting Breakpoints in Interactive Simulation	4-18
Managing Breakpoints	4-20
Setting Breakpoints in a Class Object	4-24
Creating Conditional Breakpoints.....	4-27
Debugging During Initialization of SystemVerilog Static Functions and Tasks	4-32
Enabling Static Debug in DVE	4-32
Debugging Static Code.....	4-33
Features Disabled in Initialization Phase	4-35

Annotating Values	4-37
5. Using Wave View	5-1
Viewing Waveform Information	5-2
Viewing a Waveform	5-2
Viewing Nanosim Analog Signals	5-4
Setting the Simulation Time	5-5
Using the Signal Pane	5-5
Expanding Verilog Vectors, Integers, Time, and Real Numbers	5-7
Adding Signal Dividers	5-8
Renaming Signals	5-8
Renaming Signal Groups	5-12
Undo and Redo Operation for Signals	5-12
Creating Multiple Groups when Adding Multiple Scopes	5-15
Creating Nested Signal Groups	5-15
Creating Nested Signal Groups in the Wave View	5-16
Creating Nested Signal Groups in the List View	5-21
Using Signal Group Manager to Create Nested Signal Groups	5-22
Deleting Signal Group	5-24
Customizing Duplicate Signal Display	5-24
Overlapping Analog Signals	5-25
Using User-defined Radices	5-26
Comparing Signals, Scopes, and Groups	5-28
Creating a Bus	5-31
Modifying Bus Components	5-33
Viewing Bus Values	5-35
Creating an Expression or a Counter	5-35
Limitations	5-39

Using the Wave View	5-40
Customizing Waveforms Display	5-41
Displaying Grid in Wave View	5-45
Example	5-46
Setting Grid Properties	5-46
Cursors and Markers	5-54
Using Cursors	5-54
Creating Markers	5-55
Extracting State Name	5-58
Example	5-59
Limitations	5-61
Zooming In and Out	5-62
Drag Zooming	5-62
Visualizing X at all Zoom Levels	5-63
Expanding and Contracting Wave Signals	5-63
Searching Value or Edge of Signal	5-64
Shifting Signals	5-65
Printing Waveform	5-66
Viewing PLI, UCLI, and DVE Forces in Wave View	5-68
 6. Using the List View	6-1
The List View	6-2
Viewing Simulation Data	6-3
Using Markers	6-3
Setting Signal Properties	6-4
Comparing Signals	6-5
Saving a List Format	6-6

7. Using Schematics	7-1
Overview	7-2
Viewing Schematic	7-2
Opening a Design Schematic View	7-3
Annotating Values	7-5
Making Modules as Black-Box.....	7-5
Mapping Symbols in Schematic.....	7-6
Generating .db or .sdb Files.....	7-10
Opening a Path Schematic View.....	7-10
Displaying Connections in a Path Schematic	7-12
Compressing Buffer and Inverter in Schematic	7-14
Following a Signal Across Boundaries.....	7-18
Finding Signals in Schematic and Path Schematic View	7-20
Highlighting Signals	7-20
Searching for Signals.....	7-21
Showing Value Annotation	7-21
Selecting and Deleting All Objects from Path Schematic View .	7-23
Back Tracing	7-24
Example.....	7-25
Setting the Back Trace Properties	7-30
Printing Schematics.....	7-33
Schematic Visualization of RTL Designs.....	7-35
Schematic Symbols	7-39
Design Analysis for RTL Symbol Creation	7-39
Default Symbol for a Process.....	7-41
Flip-Flop Schematic Symbols.....	7-43
Simple Logic Schematic Symbols	7-44
Enabling and Disabling RTL Visualization	7-45

Schematic Visualization of RTL Design Limitations.....	7-46
8. Using Smartlog	8-1
Use Model	8-2
Compile Flow	8-2
Post-processing Debug Flow	8-3
Viewing Smartlog Data in the Console Pane.....	8-3
Right-click Menu Options in Smartlog	8-4
Opening Log File	8-8
Usage Example	8-9
Post-processing Mode	8-12
Interactive Mode..	8-13
9. Tracing Drivers and Loads	9-1
The Driver Pane	9-1
Supported Functionality.....	9-3
Unsupported Functionality.....	9-3
Tracing Drivers and Loads	9-3
Active Drivers	9-5
Enabling Active Drivers	9-6
Usage Example	9-8
Visualizing Driving Signals.....	9-9
Highlighting Driving Signals in Path Schematic View.....	9-12
Tracing Signal Values over Combinational Logic.....	9-13
Incremental Active-driver Tracing in Driver Pane.....	9-16
Viewing Intermediate Drivers	9-20
Visualizing the Path Between Driver and Traced Signal	9-21
Multicycle Support for Value Tracing	9-23

Specifying Maximum Clock Cycles to Trace Value Change .	9-25
Active Drivers Support for PLI, UCLI, and DVE Forces	9-27
Driver Tracing Support for Virtual Interfaces and Clocking Blocks	9-33
Driver Tracing Support for Virtual Interfaces	9-33
Driver Tracing Support for Clocking Blocks	9-35
Limitations	9-38
Active Driver Limitations	9-39
10. Using the Assertion Pane	10-1
Compiling SystemVerilog Assertions.	10-1
Displaying Assertions	10-2
Viewing Assertion in the Wave View	10-4
Displaying Cover Properties	10-6
Debugging SystemVerilog Immediate and Concurrent Assertions .	10-7
Usage Model	10-8
11. Using the Testbench Debugger	11-1
Overview	11-2
Enabling Testbench for Debugging	11-3
Invoking the Testbench Debugger GUI	11-3
Testbench Debugger Panes	11-4
Stack Pane	11-5
Local Pane	11-8
Watch Pane	11-8
Class Browser	11-10

Usage Model	11-
10	
Dynamic Object Browser	11-
17	
Object Browser Example	11-
17	
Object Hierarchy Browser	11-
18	
Viewing Memory Size of Objects in Object Hierarchy Browser	11-
23	
Using Object Hierarchy Browser Filters	11-
27	
Viewing Objects in the Class Pane	11-
34	
Viewing Object Instance Information in the Member Pane .	11-
37	
Viewing Reference Path of an Object Instance	11-
38	
Searching for Dynamic Objects in the Local Pane.....	11-
41	
Adding Reference Paths to the Watch Pane	11-
42	
Renaming Object Name in the Watch Pane.	11-
43	
Viewing Virtual Interface Object in DVE.....	11-
44	
Testbench Debug	11-
46	
Viewing Object Identifier Values	11-
47	

Viewing Object Identifier Values in DVE.....	11-
47	
Viewing Object Identifier Values Using UCLI Commands ..	11-
49	
Viewing Object Identifier Values in Local Pane	11-
49	
Viewing Object Identifier Values in Watch Pane.....	11-
50	
Viewing Object Identifier Example	11-
53	
Creating Object Identifier Breakpoints.....	11-
54	
Creating Object Breakpoints Using UCLI Commands	11-
54	
Creating Object Breakpoints Using DVE Breakpoints Dialog	11-
55	
Creating Breakpoints at the End of a Method	11-
58	
Parameterized Class Support	11-
60	
Avoiding Stepping into VMM/UVM/OVM Code	11-
61	
Changing Dynamic Variable Values in DVE.....	11-
62	
Filtering Variables in Local Pane	11-
64	
Filtering Objects in Stack Pane	11-
64	
Viewing the Class in Class Browser from Source View and Member Pane	11-
65	

Viewing VMM/UVM Documentation	11-
68	
Viewing Struct Variables in the Local Pane	11-
69	
Struct Variables Example	11-
70	
Viewing the .size of Dynamic Arrays in Local Pane	11-
71	
Dynamic Arrays Example	11-
72	
Debugging Threads	11-
74	
Thread Debugging Example	11-
74	
Viewing Status of a Thread in the Stack Pane	11-
75	
Searching a Thread in the Stack Pane	11-
78	
Using Object ID Column in the Threads Only Display View .	11-
80	
Filtering Unnamed Scopes in the Active Call Stack View	11-
81	
Support for Thread-Specific Breakpoints in the Stack Pane . . .	11-
85	
Viewing the Console Pane Thread in the Stack Pane	11-
87	
Configuring the Background Color of a Stack Frame in the Stack Pane and Class Pane	11-
88	
Debugging UVM Testbench Designs	11-
92	

UVM Testbench Design Debug Example.....	11-
93	
UVM Resource Browser	11-
94	
Viewing the UVM Resource Browser.....	11-
94	
Using the Resource View.....	11-
96	
Using the Resource History View.....	11-
100	
Right-click Menu Options in the Resource View	11-
102	
Right-click Menu Options in the Resource History View ...	11-
103	
UVM Factory View.....	11-
103	
Right-click Menu Options in UVM Factory View.....	11-
106	
UVM Phase View.....	11-
107	
UVM Phase Breakpoints	11-
112	
Simulation Arguments Dialog Box	11-
116	
Filtering Variables in the Watch Pane	11-
117	
12. Debugging Transactions	12-1
Introduction	12-1
Transaction Debug.....	12-2

Using \$vcplusmsglog	12-
17	
Viewing Streams and Transaction Relations	12-
30	
SystemVerilog String Variables dump using \$vcplusblog() and \$vcplusmsglog()	12-
36	
Editing Transaction Debug Preferences.	12-
38	
Using tblog and msglog in DVE Command Prompt	12-
40	
Transaction Debug in SystemC Designs	12-
44	
Viewing NTB-OV Variables using tblog/msglog	12-
70	
 13. Using the C, C++, and SystemC Debugger	13-1
Getting Started	13-2
Using a Specific gdb Version	13-2
Attaching the C-Source Debugger in DVE	13-2
Detaching the C-source Debugger.	13-4
Displaying C Source Files in the Source View.	13-4
Commands Supported by the C Debugger	13-5
Changing Values of SystemC and Local C Objects with synopsys::change	13- 12
Changing SystemC Objects	13- 12
Changing Local C Variables	13- 14

Using Breakpoints	13-
16	
Set a Breakpoint from the Breakpoints Dialog Box	13-
16	
Control Line Breakpoints in the Source view	13-
16	
Set a Breakpoint from the Command Line	13-
17	
Deleting a Line Breakpoint.	13-
18	
Stepping Through C-source Code	13-
19	
Stepping within C Sources	13-
19	
Cross-stepping between HDL and C Code	13-
19	
Cross-stepping in and out of Verilog PLI Functions	13-
20	
Cross-Stepping in and out of VhPI Functions.	13-
21	
Cross-stepping from C into HDL	13-
22	
Cross-Stepping in and out of SystemC Processes.	13-
23	
Direct gdb Commands	13-
24	
Add Directories to Search for Source Files	13-
25	
Common Design Hierarchy	13-
26	

Post-processing Debug Flow	13-
30	
Interaction with the Simulator	13-
33	
Prompt Indicates Current Domain	13-
33	
Commands affecting the C domain:	13-
33	
Combined Error Message	13-
34	
Update of Time, Scope, and Traces	13-
34	
Configuring CBug	13-
35	
Startup Mode	13-
35	
Attach Mode.	13-
36	
<code>cbug::config add_sc_source_info auto always explicit</code>	13-
36	
VPD Dumping for SC_FIFO Channels	13-
37	
FIFO objects that can be Dumped or Printed	13-
37	
Displaying Data in SC_FIFO	13-
37	
Configuring Dumping of a FIFO.	13-
38	
Configuring with UCLI	13-
39	

Configuring with DVE	13-
40	
Configuring from SystemC Source Code	13-
40	
Support for Data Types	13-
41	
Native ANSI and SystemC types	13-
41	
User-defined Types	13-
42	
Change Bars in Waveform.	13-
42	
UCLI 'get' Command	13-
42	
Speed Impact.	13-
43	
Supported platforms	13-
43	
Using SYSTEMC_OVERRIDE.	13-
44	
Example: A Simple Timer	13-
46	
Viewing SystemC Source and OSCI Names in DVE.	13-
51	
Use Model	13-
51	
Source and OSCI Names	13-
53	
Displaying Source and OSCI Names in DVE.	13-
53	

Limitations	13-
56	
Using CBug to Display Instance Name of Target Instance in TLM-2.0	13-
57	
Limitations of Displaying Instance Name of Target Instance in TLM-2.0	13-
13-59	
CBug Stepping Improvements.	13-
59	
Using Step-out Feature	13-
59	
Automatic Step-through for SystemC	13-
60	
Enabling and Disabling Step-through Feature	13-
61	
Recovering from Error Conditions	13-
61	
14. Debugging Constraints	14-1
Enabling Constraint Solver for Debugging.	14-3
Invoking the Constraint Solver Debugger GUI.	14-4
Debugging Constraint-Related Problems	14-4
Breaking Execution at a Randomize Call.	14-4
Creating Solver Conditional Breakpoint at Randomize Calls	14-8
Analyzing a Randomization Call.	14-18
Constraint Browsing in Class Hierarchy Browser	14-19
Browsing Objects in Local Pane	14-23

Using the Constraints Dialog	14-
25	
Using the Solver Pane	14-
25	
Using the Relation Pane.....	14-
31	
Inconsistent Constraints.....	14-
34	
Debugging Constraints Example	14-
36	
Changing Radix Type of a Variable or Constraint Expression in Constraints Dialog Box	14-
41	
Supported Radix Types	14-
41	
Using Constraints Dialog Box to Change the Radix Type of a Variable or Constraint Expression	14-
41	
Using Tcl Command to Change the Radix Type of a Variable or Constraint Expression	14-
46	
Drag-and-Drop Support for Constraints Debug	14-
47	
Drag-and-Drop Support in Constraints Dialog Box	14-
47	
Drag-and-Drop Items from Class Browser and Member Pane to Breakpoint Dialog Box	14-
51	
Viewing Object ID Information of a Class in Solver Pane	14-
53	
Cross Probing	14-
54	

Cross Probing to Local Pane	14-
54	
Cross Probing to Class Browser from Randomize Call	14-
55	
Extracting Test Case	14-
56	
Extracting Test Cases from DVE	14-
56	
Extracting Test Cases Using UCLI Command	14-
59	
Controlling rand_mode/constraint_mode and Randomization from UCLI/DVE	14-
59	
Controlling rand_mode/constraint_mode from UCLI	14-
60	
Controlling rand_mode/constraint_mode from DVE	14-
62	
Rerandomization from DVE/UCLI	14-
66	
Constraints Debug Limitations	14-
73	
15. Debugging Macros in DVE	15-1
Enabling Macro Debug	15-2
Expanding and Collapsing the Macro Content.	15-2
Viewing Signal Value Annotations in the Macro Content	15-2
Viewing the Macro Content in a Tooltip	15-3
Viewing the Definition of a Macro in the Source Code.	15-3
Viewing Text Indentation in Expanded Macro and Tooltip	15-4
Changing Background Color of Line Attribute Area for Expanded Macros	15-6

Examples	15-8
Usage Example	15-12
Setting Breakpoints in the Macro Content	15-15
Creating Breakpoints in the Macro Content Using Breakpoints Dialog	15-18
Setting Breakpoint in the Macro Content Using DVE Tcl Command	15-20
Stepping In and Out of Macros	15-20
Tracing Drivers and Loads Inside Macro Content	15-23
Macro Expansion Location.	15-24
Nested Macro Support.	15-25
Macro Debugging Limitations	15-25
 16. DVE Interactive Rewind	16-1
Interactive Rewind Vs Save and Restore	16-2
Usage Model	16-3
Limitations	16-8
Menu Bar Options	A-2
File Menu	A-2
Edit Menu.	A-4
View Menu	A-7
Simulator Menu	A-11

Signal Menu	A-14
Scope Menu	A-17
Trace Menu	A-19
Window Menu	A-21
Help Menu	A-23
Testbench Debugger Menu Options	A-23
View Menu	A-24
.	
Signal Menu	A-24
Simulator Menu	A-24
Window Menu.	A-25
User-Defined Menu	A-25
Editing Preferences	A-26
Global Options	A-27
Assertion Debug Options.	A-28
Data Pane Options.	A-29
Design Debug Options.	A-29
Driver Pane Options	A-31
Hierarchy Pane Options.	A-31
List View Options	A-32
Memory View Options	A-32
Schematic View Options	A-33
Simulator Options	A-35
Source View Options	A-35
Testbench/CBug Options	A-37
Transaction Debug Options	A-38
Wave View Options	A-40
Toolbar Reference	A-41
File	A-41

Edit	A-42
Zoom/Zoom and Pan History	A-44
Scope	A-45
Trace	A-46
Window	A-46
Back Trace	A-48
Interactive Rewind	A-49
Signal	A-49
Simulator	A-51
Time Operations	A-52
Grid	A-53
Testbench GUI Simulator Toolbar Options	A-53
Customizing the DVE Toolbar	A-54
Adding a New Toolbar	A-54
Adding Items to a Toolbar	A-54
Deleting an Existing Toolbar	A-55
Deleting an Item in a Toolbar	A-55
Using the Context-Sensitive Menu	A-56
Hierarchy Pane CSM	A-57
Data Pane CSM	A-58
Source View CSM	A-60
Schematic View CSM	A-62
Wave View CSM	A-63
Signal Pane CSM	A-64
List View CSM	A-66
Driver Pane CSM	A-66
Watch Pane CSM	A-67
Memory View CSM	A-67
Assertion Pane CSM	A-67

Keyboard Shortcuts	A-68
File Command Shortcuts	A-69
Edit Command Shortcuts	A-69
View Command Shortcuts	A-70
Simulator Command Shortcuts	A-70
Signal Command Shortcuts	A-71
Scope Command Shortcuts	A-71
Trace Command Shortcuts	A-71
Help Command Shortcuts	A-72
Window Command Shortcuts	A-72
Tcl GUI Commands Shortcuts	A-72
Using the Command Line	A-73

1

Getting Started

This chapter provides an introduction to Discovery Visual Environment (DVE) and walks you through the basic steps of using DVE. This chapter includes the following topics:

- “Overview” on page 2
- “Enabling Debugging” on page 3
- “Invoking DVE” on page 5
- “Passing DVE Arguments from Simulator Runtime Command Line” on page 14
- “Closing a Database” on page 22
- “Exiting DVE” on page 22
- “DVE Log Files” on page 22
- “DVE Licensing Queuing” on page 23

- “DVE Setup Files” on page 24
 - “Typical Symbols Used in DVE” on page 28
 - “DVE Command-line Reference” on page 32
-

Overview

DVE is an interactive Graphical User Interface (GUI) that you can use for debugging your SystemVerilog, VHDL, Verilog, and SystemC designs. You can drag-and-drop your signals in various views or use the menu options to view the signal source, trace drivers, compare waveforms, and view schematics.

You must setup VCS to work on DVE. For more information about obtaining license information and setting up VCS HOME, see the *VCS User Guide*.

You must use the same version of VCS and DVE to ensure problem-free debugging of your simulation. You can check the DVE version using:

- The `dve -v` command-line option.
- The `gui_get_version` command.
- The **About** option from **Help** menu (**Help > About**).

Enabling Debugging

This section describes how to enable debugging options for your simulation.

Note:

If you run DVE in a directory where you do not have write privileges for files, a warning message appears to indicate that DVE is unable to write files.

Debug Options

`-debug_pp`

Gives best performance with the ability to generate the VPD/VCD file for post-process debug. It is the recommended option for post-process debug.

It enables read/write access and callbacks to design nets, memory callback, assertion debug, VCS DKI, and VPI routine usage. You can also run interactive simulation when the design is compiled with this option, but certain capabilities are not enabled. It does not provide force net and reg capabilities. Set value and time breakpoints are permissible, but line breakpoints cannot be set.

`-debug`

Gives average performance and debug visibility/control i.e more visibility/control than `-debug_pp` and better performance than `-debug_all`. It provides force net and reg capabilities in addition to all capabilities of the `-debug_pp` option. Similar to the `-debug_pp` option, with the `-debug` option also you can set value and time breakpoints, but not line breakpoints.

`-debug_all`

Gives the most visibility/control and you can use this option typically for debugging with interactive simulation. This option provides the same capabilities as the `-debug` option, in addition it adds simulation line stepping and allows you to track the simulation line-by-line and setting breakpoints within the source code. With this option, you can set all types of breakpoints (line, time, value, event etc).

Required Files

The input files required to enable the debug functionality of DVE are:

- **VPD file** — VPD files (design database files) are platform-independent, versioned files into which you can dump the selected signals during simulation. DVE gets hierarchy, value change, and some assertion information from these files. You can perform debugging in post-process mode using a VPD file.

However, VPD files are not guaranteed to contain the entire design hierarchy because you can selectively choose subsets of the design to be dumped to the VPD file.

For all DVE functionality to be available while debugging, it is essential that the VCS version used to generate the VPD, and the DVE version used to view the VPD, are identical.

- **OVA library** — DVE uses this library for advanced assertion debugging. This library is produced when a design contains OVA, SVA, or PSL assertions and the correct VCS compile options are used. The library is platform dependent.

- **Coverage databases** — In DVE, you should specify coverage databases to display coverage information. If coverage databases for different types of coverage exist, DVE automatically opens them.

You can open the coverage database simv.vdb that contains:

- Code coverage data.
- Functional (OV and SV testbench and assertions) coverage data..

Invoking DVE

You can invoke DVE:

- Without any arguments
- In 64-bit mode
- In interactive mode
- In post-process mode
- Using session file
- Using Tcl scripts

To invoke an empty DVE top-level window with no arguments, use the following command:

```
% dve
```

From this point, DVE usage can be post-process or interactive.

64-bit Mode

To invoke DVE in 64-bit mode, use the following command:

```
% dve -full64
```

To use the `-full64` option, you must download and install the 64-bit VCS binaries. By default, DVE is invoked in 32-bit mode.

Interactive Mode

In addition to loading the design database files for post-processing, you can also setup and run a simulation interactively in real-time using a compiled Verilog, VHDL, or mixed design.

You can use the following commands to invoke DVE interactively.

- `% simv -ucli`
Runs VCS/VCS MX for UCLI debugging. The DVE GUI is not displayed.
- `% simv -gui`
Opens DVE with simv attached to simulator at time 0.
- `% vcs -gui -R`
Compiles and builds simv, then opens DVE with simv attached to the simulator at time 0.
- `% dve -toolexe name -toolargs simulator args`

Invokes DVE, connects executable (name) to the simulator, and runs it with the arguments specified in args.

Starting an Interactive Session from the DVE GUI

You can rebuild the simulation in DVE either using the VCS script or using your own custom script.

Note:

The Rebuild and Start option is not supported in MX designs and is recommended only for pure Verilog designs.

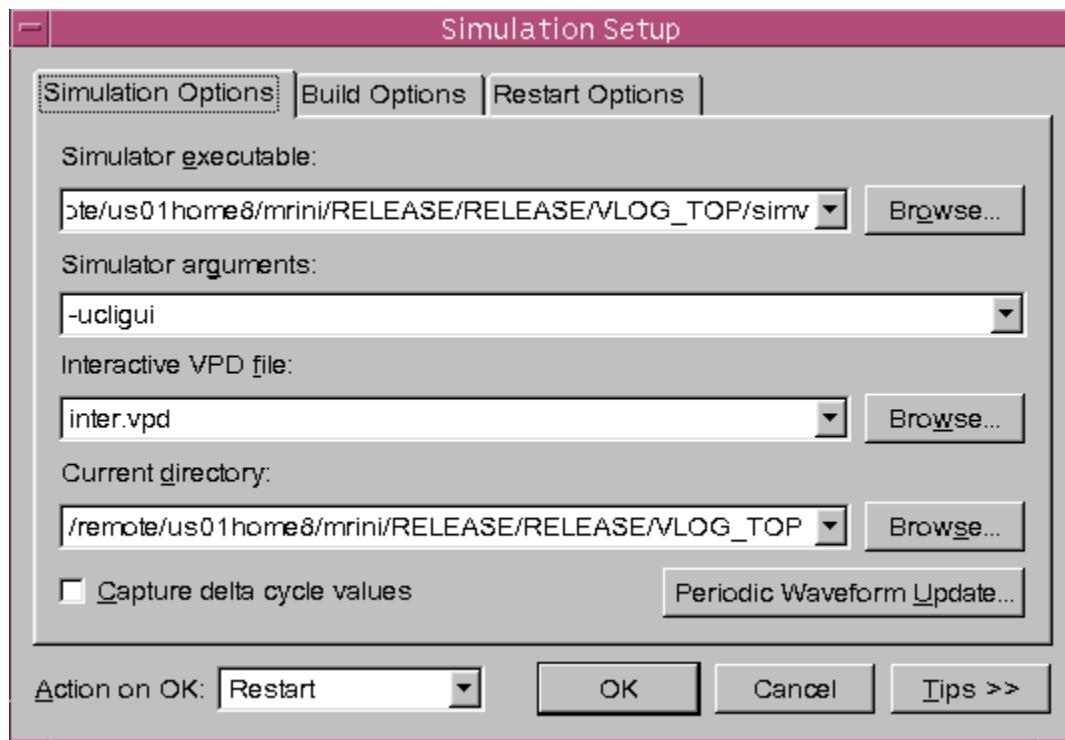
To start an interactive session from the DVE GUI

1. Invoke DVE using the following command:

```
%dve
```

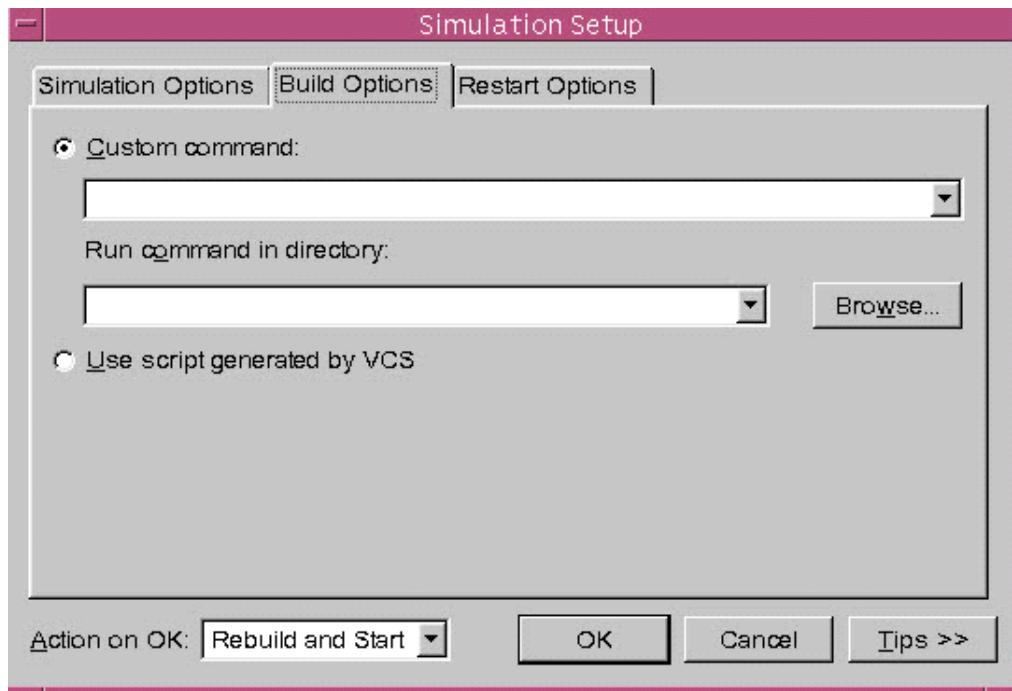
2. From the **Simulator** menu, select **Setup**.

The Simulation Setup dialog box appears.



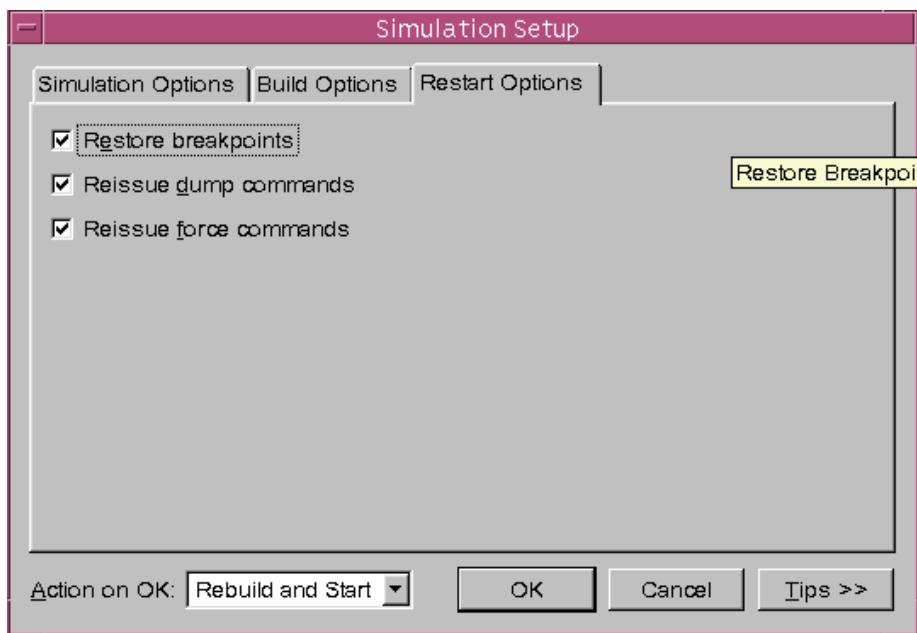
3. Click the **Simulation Options** tab and select the following options, as appropriate:
 - Simulator Executable — Specifies the name of a simulator executable. Click the **Browse** button to locate one.
 - Simulator arguments — Identifies the simulator arguments.
 - **Interactive VPD file** — Specifies the name of the VPD file. Click the **Browse** button to select an existing file that will be written during this interactive session.
 - **Current directory** — Specifies the full path of the simulator executable. Click the **Browse** button to select the path in the current directory.

- **Capture delta cycle values** — Captures the delta cycle values.
 - **Periodic Waveform Update** — Allows you to enable the waveform update and set the update interval.
4. Click the **Build Options** tab and select the following options, as appropriate:
- Use script generated by VCS — Uses the VCS script to rebuild the simulation.
 - Custom command — Uses your custom script. Enter the custom command in the text area.
 - Run command in directory — Specifies the directory name in which you run the custom script. You can browse and select the desired directory.



5. Click the **Restart Options** tab and select the following options, as appropriate.

- Restore breakpoints — Restores the breakpoints during simulation restart.
- Reissue dump commands — Reissues the dump commands while restarting the simulation.
- Reissue force commands — Reissues the force commands while restarting the simulation.



6. Select an action in the Action on OK list box to specify the action (none, start/restart, rebuild and start) that you want DVE to take when you click the OK button.
7. Click one of the following:
 - OK** to apply your specification.
 - Cancel** to close and not apply the specification.
 - Tips** to view the steps to perform in this dialog box.

The simulation is started or restarted per your selection. You can use the **Simulator** menu or the toolbar commands to further control the simulation.

Post-Process Mode

There are two ways to invoke DVE in post-process mode:

- Using the `-vpd` command
- Loading the design database in the DVE GUI

Using the `-vpd` command

The `-vpd` command invokes DVE, reads and loads the specified design database file, and opens the top-level scope for that design.

Syntax:

```
dve -vpd [filename]
```

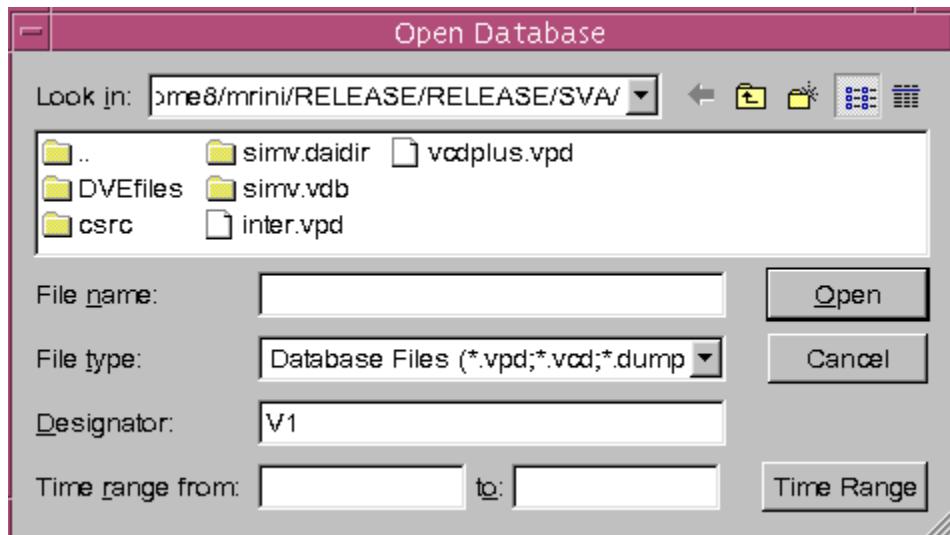
Loading the Design Database File in the DVE GUI

You can load and display any number of design database files for post-processing.

To load a design database file

1. Select **File > Open Database**.

The Open Database dialog box appears.



2. Browse and select the name of the design database file or enter the name of the file you want to open in the **File name** field.
3. Enter a designator for your design in the **Designator** field or select the default (V1).
4. Enter a time range in the **Time range** field.

The default range is the full time range for which the design database is dumped.

5. Click **Open**.

The design database file is loaded.

Note:

You need the `.sim` files to do post-process debug for VHDL and also to get statement-level drivers. Make sure to copy `simv`, `.daidir` and any other `.sim` files if you relocate your design.

Using Session File

There are two ways to invoke DVE from a session file:

- Using the `-session` command.
- Loading a session file in the DVE GUI.

For more information about how to save a session, see the section entitled “[Passing DVE Arguments from Simulator Runtime Command Line](#)” on page 14.

Using the `-session` command

The `-session` command invokes DVE with the design database file, `test.vpd`, and applies settings from the session file, `mysession.tcl`

Syntax:

```
% dve -vpd test.vpd -session mysession.tcl
```

Loading a Session File in the DVE GUI

To load a session file in the DVE GUI

1. Open DVE.
2. Select **File > Load Session**.
The Load Session dialog box appears.
3. Browse to the session and select it from the list of saved session Tcl files.

4. Click **Load**.
-

Using Tcl Scripts

You can use the following Tcl commands to invoke DVE.

`dve -cmd [tcl_cmd]`

Invokes DVE and executes the Tcl command enclosed in quotation marks. You can specify multiple commands separated by semicolons.

`dve -script [tcl_file]`

Invokes DVE and reads the Tcl script specified as argument.

`dve -session [tcl_file]`

Invokes DVE and reads the session file. If the `-session` and `-script` options are combined, the session is read first and then the script.

Passing DVE Arguments from Simulator Runtime Command Line

You can pass DVE arguments from the simv command line using the `-dve_opt` option. Instead of using the DVE options manually, you can automate the actions and pass custom Tcl scripts using the `-dve_opt` option in the simv command line.

You must precede each DVE argument by `-dve_opt`. In cases where the argument requires an additional option, the `=` sign should be used.

The following examples show the usage of `-dve_opt`.

- To print version and to log off, the DVE command is:

```
dve -v -nolog
```

The runtime command with `-dve_opt` is:

```
simv -gui -dve_opt -script=myscript.tcl -dve_opt -nolog
```

- To specify a session file, the DVE command is:

```
dve -session=mySession.tcl
```

The runtime command with `-dve_opt` is:

```
simv -gui -dve_opt -session=mySession.tcl
```

- To use the `-cmd` argument to print “Hello World”, the DVE command is:

```
dve -cmd='puts "hello world"'
```

The runtime command with `-dve_opt` is:

```
simv -gui -dve_opt -cmd='puts "hello world"'
```

The following commands cannot be passed directly to DVE, as arguments, from the `simv` command-line:

- `-vpd` – This command is not processed since the simulator already uses `inter.svpd` of file specified by `-vpd_file`, so `-vpd` is not supported.
- `-toolexe`, `-toolargs` – These command are automatically produced by `simv`.
- `-servermode` – This command is not applicable for `simv`.

- `-full64` – This command is not supported, because when simv is generated using `-full64`, DVE will be in 64-bit mode by default.
 - `-dbdir` – This command is not supported, because it is not used in interactive mode.
 - `-ucli` – This command is not supported.
-

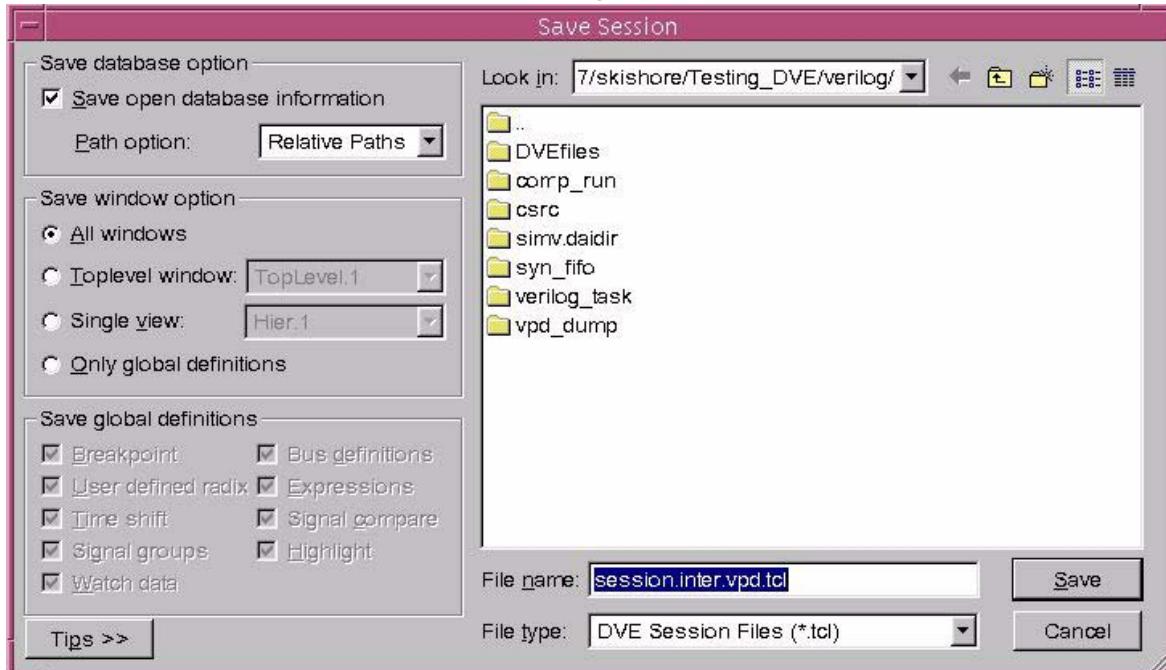
Saving a Session or Layout

You can save the current state of DVE, that is session data, display layout, and design database path options, using the Save Sessions dialog box. The options that you select in the Save Session dialog box is restored after you restart DVE.

To save a session

1. Select **File > Save Session**.

The Save Session dialog box appears.



2. Enter a file name in the **File name** field or browse for the file in your directory.
3. Select a file type in the **File type** list box.
4. Select the **Save open database information** option if you want to save the database information to the session file.
5. Select any of the following path option from the options specified under **Save window option**:
 - Relative Path — (default) Specifies path for opened design databases (relative to the directory where the session file is stored) or interactive design. If you move the directory, the session file will still work.

- Full path — Specifies fully qualified path or absolute path for opened design databases or interactive design. This allows you to reload the session file from any location, but if you move the simulation directory, the session file may no longer work.

6. Select any of the following window option from the options specified under **Save global definitions**:
 - All windows — Saves all contents in the current DVE session.
 - Toplevel window — Saves the content of only the top-level window and the contained views and panes.
 - Single view — Saves the contents of only a single view.
 - Only global definitions — Saves the data types selected in the **Save global definitions** field. This option does not save any view. For example, you can create a session file that only contains the expressions you have defined.

Enables the **Save global definitions** options. Select the data types you want to include in the session file.

7. Click **Save** to save the session or **Cancel** to close the dialog box.
8. Click **Tips** to view details about the options and fields in the Save Session dialog box.

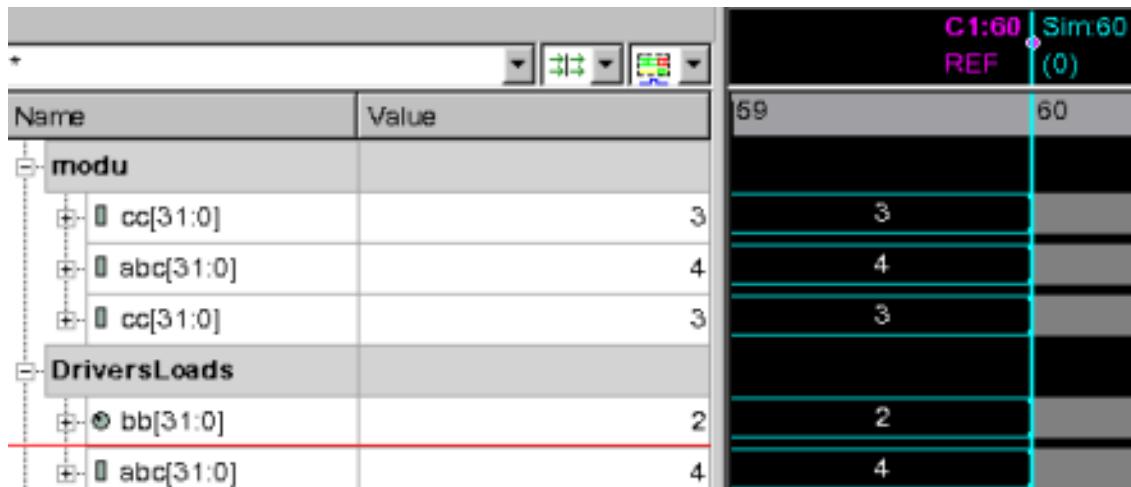
Saving the Current View

You can save your desired view instead of saving the entire session in DVE using the **Save Current View** option. The **Save Current View** option does not save any database related information and saves only the current view.

The **Save Current View** option is not enabled for all the views and is disabled for the views that are not supported.

To save the current view

1. Select a view in DVE that you want to save.



2. From the **File** menu, select **Save Current View**.

The Save Session dialog box appears.

3. Enter the file name in the **File Name** field and click **Save**.

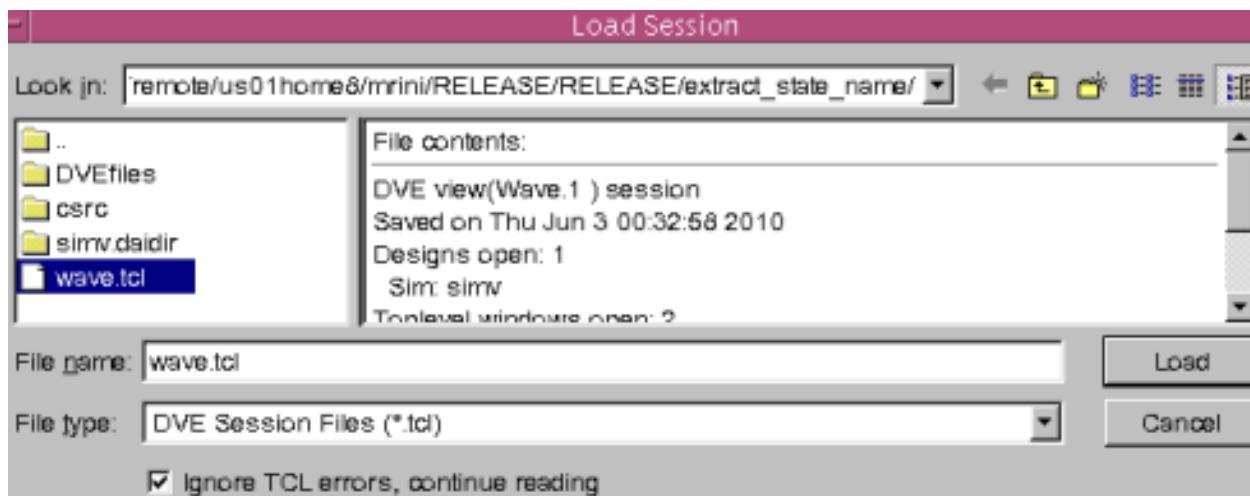
Your current view is saved.

To load the saved view

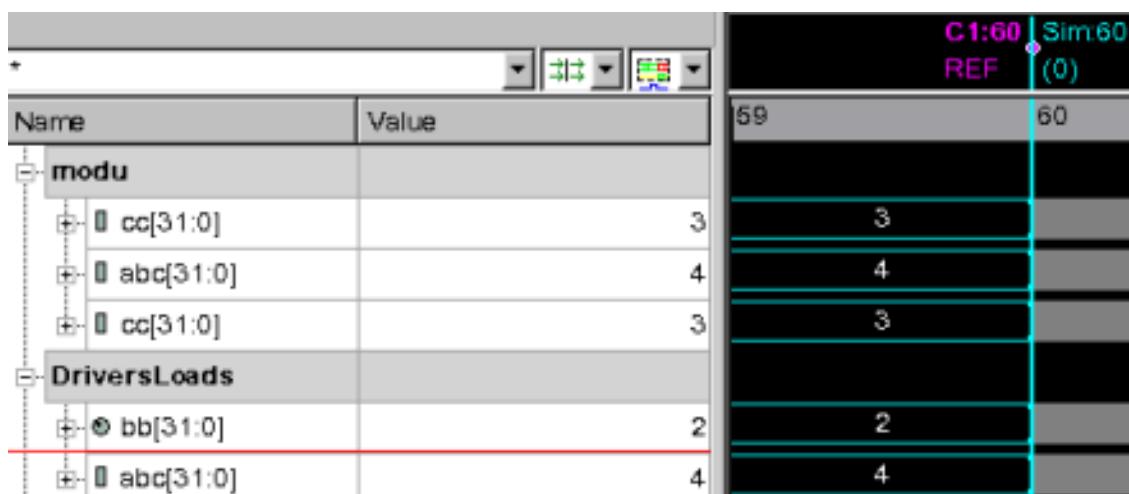
1. In DVE, select **File > Load Session**.

The Load Session dialog box opens.

2. Select the Tcl file in which you have saved your DVE view and click **Load**.



The view is loaded in DVE.



Restoring a Saved Simulation

When restoring a saved simulation, use the same technology or flow to restore that you used to save the checkpoint. For example:

- Save using UCLI commands and restore using UCLI commands.

- Save in DVE and restore in DVE.
- Save using SCL commands and restore using SCL commands.
- Save using CLI commands and restore using CLI commands.

Do not mix the technologies for saving and restoring, for example:

- Save using UCLI commands and restore using SCL commands.
- Save in DVE and restore with UCLI commands.
- Save using UCLI commands and restore using DVE.
- Save using CLI commands and restore using UCLI commands.

You must close all external applications, that communicates with VCS/VCS MX using the VHPI or PLI, before you save and reopen a simulation.

Note:

For more information about restrictions of Save and Restore commands, see the UCLI User Guide > Commands > Session Management Commands > Restore.

Closing a Database

To close a currently open database

1. Select **File > Close Database**.

The Close Database dialog box appears.

2. Make sure the correct database is selected, then click **OK**.

DVE closes the display of the selected database in the Hierarchy pane.

Exiting DVE

To exit DVE, select **File > Exit**.

DVE Log Files

DVE produces the following log files in the `DVEfiles` directory, which gets created in the current working directory. These log files are useful in the event of a problem.

- `dve_gui.log`—Contains all input and output to the console log.
- `dve_history.log`—Contains all commands that get executed during the lifetime of a debug session; useful for capturing scripts for replay.

DVE Licensing Queuing

License queuing options for DVE are as follows:

+vcs+lic+wait

Tells DVE to queue for a license if none is available. (Infinite time)

-licwait <minutes>

Tells DVE to queue for a license for specified <minutes> if none is available. (User specified time)

Post-process mode license queuing examples:

% dve +vcs+lic+wait

Queues for the license until it is available.

% dve -licwait 20

Queues for 20 minutes if the license is not available.

Interactive mode license queuing example:

% ./simv -gui -dve_opt +vcs+lic+wait

Queues for the license until it is available.

% ./simv -gui -dve_opt -licwait=1

Queues for 1 minute if the license is not available.

Examples with the UCLI flow:

% ./simv -ucli +vcs+lic+wait

% ./simv -ucli -licwait 1

DVE Setup Files

DVE sources the following files when invoked.

`.synopsys_dve_ini.tcl`

Saves the recent layout when you exit DVE.

`.synopsys_dve_default_layout.tcl`

Saves the default layout that you have selected from the menu **Window > Save Current Layout > To Default**. When you invoke DVE, this file is sourced and the layout is configured accordingly.

`.synopsys_dve_prefs.tcl`

Saves your preferences. This file is created automatically whenever you change the preferences from the menu **Edit > Preferences**.

`.synopsys_dve_usersetup.tcl` or

`.synopsys_dve_gui_usersetup.tcl`

The `.synopsys_dve_usersetup.tcl` file allows you to define GUI customization, such as creating your own shortcuts or additional menu. You create this file and it is sourced by DVE at start-up from your VCS Home directory. If the `DVE_USERSETUP_PATH` environment variable is set, the file `.synopsys_dve_usersetup.tcl` is searched (and sourced) in all the paths given in the environment variable.

If multiple paths are given, the paths are separated by ':'. For example,

```
%setenv DVE_USERSETUP_PATH /u/user/somedir  
          %/u/user/somedir/.synopsys_dve_usersetup.tcl  
  
%setenv DVE_USERSETUP_PATH /u/user/somedir:/x/y/z  
          %/u/user/somedir/.synopsys_dve_usersetup.tcl and  
          /x/y/z/.synopsys_dve_usersetup.tcl
```

The `.synopsys_dve_gui_usersetup.tcl` file contains your changes that require the GUI to be available. This file is also searched in your home directory, then in all the paths given in the `DVE_USERSETUP_PATH`. This is done when the DVE GUI is initialized.

For more information about how to create shortcut keys or hotkey, see the section “[Keyboard Shortcuts](#)” in the chapter [Menu Bar and Toolbar Reference](#).

You can also customize your menu (see “[Customizing the DVE Toolbar](#)”, or create user-defined menu and menu options (see “[User-Defined Menu](#)”).

`.synopsys_ucli_prefs.tcl`

Stores your UCLI TCL procedures, which are sourced in interactive mode, both in batch and GUI modes.

Managing User Setup Files

Use the `DVE_USERSETUP_PATH` environment variable to create multiple user setup file.

During startup, DVE reads the `.synopsys_dve_prefs.tcl`, `.synopsys_dve_usersetup.tcl`, and `.synopsys_dve_gui_usersetup.tcl` files available in the directories you specify in the `DVE_USERSETUP_PATH` variable.

You can copy the common group preference settings from `~/.synopsys_dve_prefs.tcl` and put it in `.synopsys_dve_prefs.tcl`, in a directory specified by `DVE_USERSETUP_PATH`.

If you use the `DVE_USERSETUP_PATH`, you can share the common setting, but your personal setting can still overwrite the common setting.

Usage

```
% setenv DVE_USERSETUP_PATH "dir1:dir2:dir3"
```

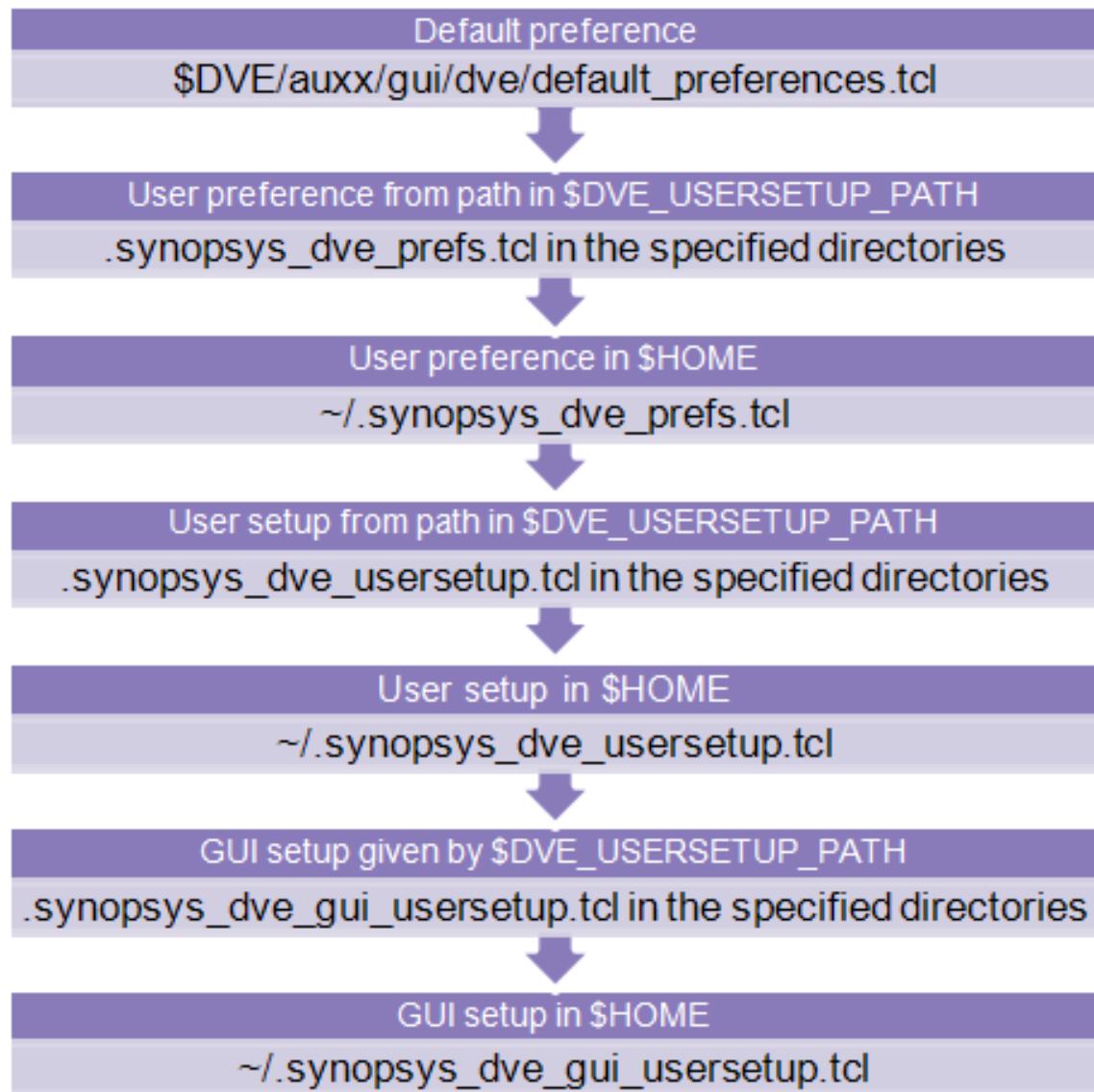
DVE reads the file in the following order:

1. `dir1/.synopsys_dve_usersetup.tcl`
2. `dir2/.synopsys_dve_usersetup.tcl`
3. `dir3/.synopsys_dve_usersetup.tcl`

DVE displays a message when the files are read.

The previous file is overridden when the current file is read, hence you need to set the priority.

Following is the order that DVE follows while reading the preferences and setup files:



The \$DVE_USERSETUP_PATH environment variable is used to specify common setting for DVE, such as in a team or group.

The preference file in the home directory is used to specify specific settings for each user. Therefore, you can use \$DVE_USERSETUP_PATH to specify the preference setting (shared by a group) in the new .synopsys_dve_prefs.tcl file, while using the setting related to GUI (like new menu or toolbar) in the .synopsys_dve_usersetup.tcl file.

Typical Symbols Used in DVE

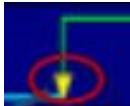
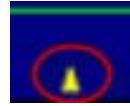
This chapter describes special symbols and Low Power Symbols used in DVE under the following sections:

- “[Special Symbols Used in DVE](#)”
- “[Low Power Symbols Used in DVE](#)”

Special Symbols Used in DVE

Table 1-1 describes special symbols used in DVE.

Table 1-1 Special Symbols Used in DVE

Symbol	Name of the symbol	Description of symbol
	Yellow Triangle Pointing Down	Indicates the point at which the driver is forced. DVE displays this symbol in Wave View.
	Yellow Triangle Pointing Up	Indicates the point at which the driver is released. DVE displays this symbol in Wave View.
[^] 1010	Carat	The carat symbol above a value “ ^{<value>} ” in the Value column of the Wave View indicates that the value is forced.
*1010	Asterisk	An asterisk symbol above a value “ ^{<value>} ” in the Source View and Wave View indicates that the value is truncated.
..1010	Three Dots	Three dots followed by a value “ ^{...<value>} ” in the Data Pane indicates that the value is truncated.

Symbol	Name of the symbol	Description of symbol
	Yellow Dot	Observed in a Wave View when multiple bits of a vector variable/signal toggle within the same delta cycle. You must enable delta cycle option to view this symbol in waveform.
	Red Bar	Highlights the X value in toggled waveform. By default it is disabled. You can enable this using the Highlight X Values right-click option. DVE displays this symbol in Wave View.
	Green circle	DVE displays this symbol in Driver Pane when the contributor is completely analyzed and detected as an active driving signal. This is usually the case when a driver is in combinational logic or a flip-flop, and no limitation is encountered.
	Yellow circle	DVE displays this symbol in Driver Pane when the contributor time is accurate but it is unable to determine if the signal really caused a value change of the traced signal. This is usually the case when a driver is in combinational logic, but DVE encounters some active driver limitations (for example, a function call on RHS). Also, this icon is used for contributors at time 0 in combinational logic and for the first clock transition in flip-flops.
	Red Circle	DVE displays this symbol in Driver Pane when it is not able to analyze the contributor. This happens when the driver is not in RTL code (for example, in testbench code) or when DVE encounters some active driver limitations.

Low Power Symbols Used in DVE

[Table 1-2](#) describes Low Power Symbols used in DVE. For more information on these symbols, see *Debugging Low Power Designs Using DVE* section in the *MVSIM Native Mode User Guide*.

Table 1-2 Low Power Symbols Used in DVE

Symbol	Name of symbol	Description of symbol
	Power Domain	Indicates power domain state is NORMAL (Power pane).
		Indicates power domain state is CORRUPT (Power pane).
	Power Switch	Power switch cell in Hierarchy pane and Power switch defined in UPF in Power pane.
	Isolation Cell/ Strategy	Isolation cell in Hierarchy pane and Isolation strategy defined in UPF in Power pane.
	Retention Strategy	Retention strategy defined in UPF in Power pane.
	LevelShifter Strategy	Level Shifter strategy defined in UPF in Power pane.
	Power Net	Primary/Isolation/Retention power nets defined in UPF (Data pane).
	Ground Net	Primary/Isolation/Retention ground nets defined in UPF (Data pane).
	Logic	Control signals like save/restore/switch control/isolation enable defined in UPF. Low power Instrumented signals are also shown with a red dot (Data pane).
	Input Supply Port	Input supply port defined in UPF for power switch (Data pane).
	Output Supply Port	Output supply port defined in UPF for power switch (Data pane).

DVE Command-line Reference

You can use the `dve -help` command at the VCS command-line to view the options supported by DVE. [Table 1-3](#) lists the options supported by DVE.

Usage

```
unix> dve [-cmd <TCL command>] [-cov] [-dbmdir <directory>]
[-full64] [-logdir <directory>] [-nolog] [-pathmap <file>]
[-replay <file>] [-script <file>] [-servermode] [-session
<file>] [-title <string>] [-toolexe <file>] [-toolargs
<options>] [-v] [-vpd <file>] [-viewlog <file>]
```

Table 1-3 Options Supported by DVE

Option	Description
<code>-cmd <TCL command></code>	Run a TCL command in DVE console when DVE is opened. Example: <code>unix> dve -cmd "puts Hello"</code>
<code>-cov</code>	Invokes DVE Coverage GUI.
<code>-dbmdir <directory></code>	Option to specify the <code>simv.daidir</code> directory path to DVE, if <code>simv.daidir</code> is in a different location from that of VPD path. This option should be used, even if <code>simv.daidir</code> is renamed. Example: <code>unix> dve -vpd <dump.vpd> -dbmdir </user/simv.daidir></code>
<code>-full64</code>	Run DVE in 64-bit mode.
<code>-logdir <directory></code>	Specify the directory where DVE log files should be saved. By default, DVE files are stored in the 'DVEfiles' directory in the present working directory.
<code>-nolog</code>	Do not generate the 'DVEfiles' directory or any DVE logs, while DVE is invoked.

Option	Description
-pathmap <file>	Provides mapping to the new location of source files. For more information, see “ Mapping to the Location of the Source Files ” section.
-script <file>	<p>Source a TCL script. DVE stops running the script on first error.</p> <p>Example: unix> dve -script <user_script.tcl></p>
-replay <file>	<p>As compared to -script, the -replay option will not stop running the script on an error.</p> <p>Example: unix> dve -replay <user_script.tcl></p>
-servermode	<p>Run DVE in server mode. This option creates a server in DVE for other tools to connect to it. This allows the tools to monitor DVE activity and send commands to DVE.</p> <p>Example: unix> dve -servermode -vpd file_name.vpd unix> cat .synopsys_dve_serverport.txt { {machine_name} {port_number} } unix> telnet machine_name port_number</p> <p>For example, if you perform the following command, then variable will be added in the Wave View. add_wave <variable> where, variable is the name of a variable with full hierarchy.</p> <p>gui_exit Connection closed by foreign host.</p>
-session <file>	<p>Option to load a DVE session file.</p> <p>Example: unix> dve -session <session_file></p>

Option	Description
<code>-title <string></code>	Set title (a string to be displayed on a top-level DVE frame caption).
<code>-toolexe <file> [-toolargs <options>]</code>	Option to invoke DVE, load simulation executable <code><file></code> , and pass simulation arguments.
	<p>Example:</p> <pre>unix> dve -toolexe simv -toolargs "+UVM_TESTNAME=test1"</pre>
<code>-v</code>	Print version information
<code>-vpd <file></code>	<p>Opens VPD <code><file></code></p> <p>Example:</p> <pre>unix> dve -vpd vpd1 -vpd vpd2</pre>
<code>-viewlog <file></code>	<p>Opens the given <code>.log</code> file generated using <code>-sml</code> option in the DVE console. This option can be specified multiple times to open multiple log files in separate console tabs. For more information, see "Using Smartlog".</p> <p>Example:</p> <pre>unix> vcs -sml -l <comp.log> unix> simv -sml -l <run.log> unix> dve -viewlog <comp.log> unix> dve -vpd vcdplus.vpd -viewlog <run.log></pre>

2

Using the Graphical User Interface

This chapter describes the basic usage of the DVE GUI and management of the windows, and includes the following topics:

- “Overview of DVE Window Configuration”
- “Managing DVE Panes and Views”
- “The Console Pane”
- “The Watch Pane”
- “The Console Pane”
- “The Memory View”
- “Using the Menu Bar and Toolbar”
- “Searching Signals or Scopes”
- “Mapping to the Location of the Source Files”

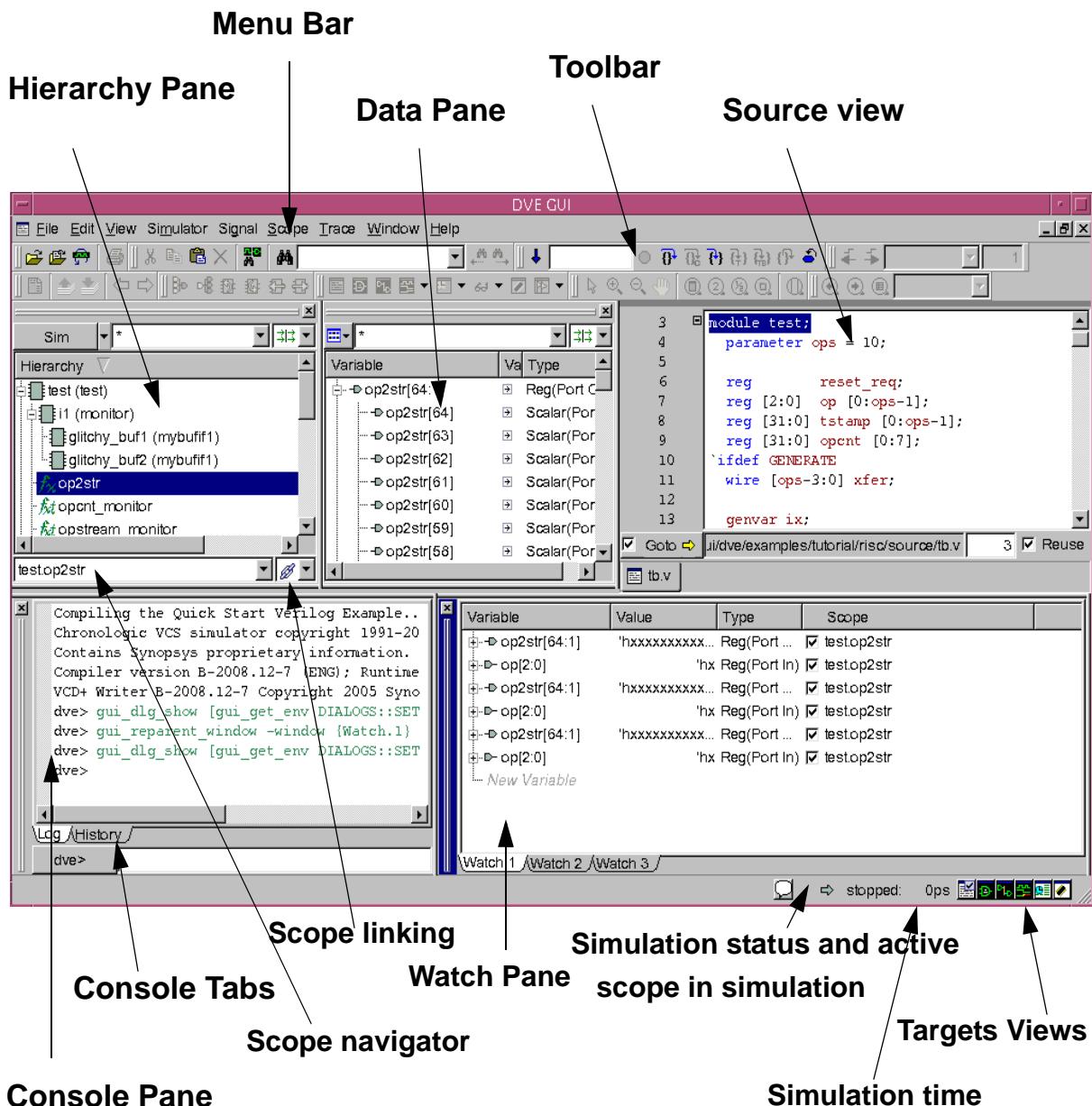
- “[Editing Preferences](#)”
- “[Using Context-Sensitive Menu](#)”

Overview of DVE Window Configuration

DVE window model is based on the concept of the TopLevel window. A TopLevel window contains a frame, menus, toolbars, status bar, and pane targets. Any number of TopLevel windows are possible, however, at startup, the default is one.

A DVE TopLevel window is a frame, which DVE uses for displaying design and debug data. The default DVE window displays the TopLevel window with the Hierarchy pane on the left, Data pane next to the Hierarchy pane, the Console pane at the bottom, and the Source view occupying the remaining space on the right. [Figure 2-1](#) shows the DVE TopLevel window.

Figure 2-1 DVE Top-level Frame



The DVE GUI contains the following panes and views:

- Hierarchy Pane — Displays the scope hierarchy of the design. For more information about the tasks you can perform in the Hierarchy pane, see the chapter “[Using the Hierarchy and Data Panes](#)” on page 1.
- Data Pane — Displays the variables of the selected scopes of the Hierarchy pane. For more information about the tasks you can perform in the Data pane, see the chapter “[Using the Hierarchy and Data Panes](#)” on page 1.
- Console Pane — Displays the simulation output and DVE messages, also allows you to type in UCLI or DVE Tcl commands. It also logs the commands executed in the History tab. For more information about the Console pane, see the section entitled “[The Console Pane](#)” on page 14.
- Watch Pane — Allows you to watch the current values of variables added to the view. Watch pane appears only when you add variables or signals to it for monitoring purposes. For more information about the Console pane, see the section entitled “[The Watch Pane](#)” on page 15.
- Source View — Displays the source code and supports source code relative features, such as tracing driver or load, and setting line breakpoints. For more information about the tasks you can perform in the Source view, see the chapter “[Using the Source View](#)” on page 1.
- Wave View — Allows you to dump the signals into a VPD file and view the value changes over time. For more information about the tasks you can perform in the Wave view, see the chapter “[Using Wave View](#)” on page 1.
- List View — Provides a table view to display the values of signals over time. For more information about the tasks you can perform in the List view, see the chapter “[Using the List View](#)” on page 1.

- Schematic View — Provides a module-based schematic to display the connectivity of the object. For more information about the tasks you can perform in the Schematic view, see the chapter “[Using Schematics](#)” on page 1.
- Path Schematic View — Provides an expandable path schematic to display the connectivity of the object. For more information about the tasks you can perform in the Path Schematic view, see the chapter “[Using Schematics](#)” on page 1.
- Memory View — Displays the value of multiple-dimension array in a table. For more information about the tasks you can perform in the Memory view, see the section entitled “[The Memory View](#)” on page 16.
- Assertions View — Displays the summary of assertion results of simulation including the success, failures, and the incomplete ones. For more information about the Assertion view, see the chapter “[Using the Assertion Pane](#)” on page 1.
- Stack Pane — Displays the current simulation testbench stack. The Stack pane appears as a new tab beside Hierarchy pane when the design contains testbench. For more information about the Testbench GUI, see the chapter “[Using the Testbench Debugger](#)” on page 1.
- Local Pane — Displays the variables of the selected frame in Stack pane. The Stack pane appears when there is a testbench in the design and is a part of the Testbench GUI. For more information about the Testbench GUI, see the chapter “[Using the Testbench Debugger](#)” on page 1.

- Coverage GUI — Provides a summary of the coverage statistics. To invoke the DVE Coverage GUI, enter the `dve` command with coverage command-line options, such as `-cov`. In the DVE Coverage GUI, open the coverage database to view the coverage statistics or reports in various views and tables. For more information about the DVE Coverage GUI, see the chapter **Viewing Coverage Reports Using the DVE Coverage GUI** in the **Coverage Technology User Guide**.
-

Creating a Window Title for All Views and Panes

You can create a common title for all the views and panes in DVE. This is useful if you open, say two designs in one DVE session, and you want to know which view or pane belongs to which design.

To create a common title, set the following environment variable:

```
% setenv DVE_CASENAME <your_title>
```

Where,

`<your_title>` — Indicates the caption that you set for all the TopLevel windows. This caption replaces the default "TopLevel" caption to "`your_title`". You can use any text or path name as title. For example, instead of specifying the title such as "mydesign", you can say `/A/B/dir1/inter.vpd`.

`DVE_CASENAME` — Adds `<your_title>` to the existing DVE windows title.

This setting is applied when you restart/reload your session.

For example, consider that you want to change the windows title to "mydesign", after setting this environment variable, the TopLevel windows will change as follows:

DVE - TopLevel.1 - [Hier.1] to DVE - mydesign - TopLevel.1
- [Hier.1]

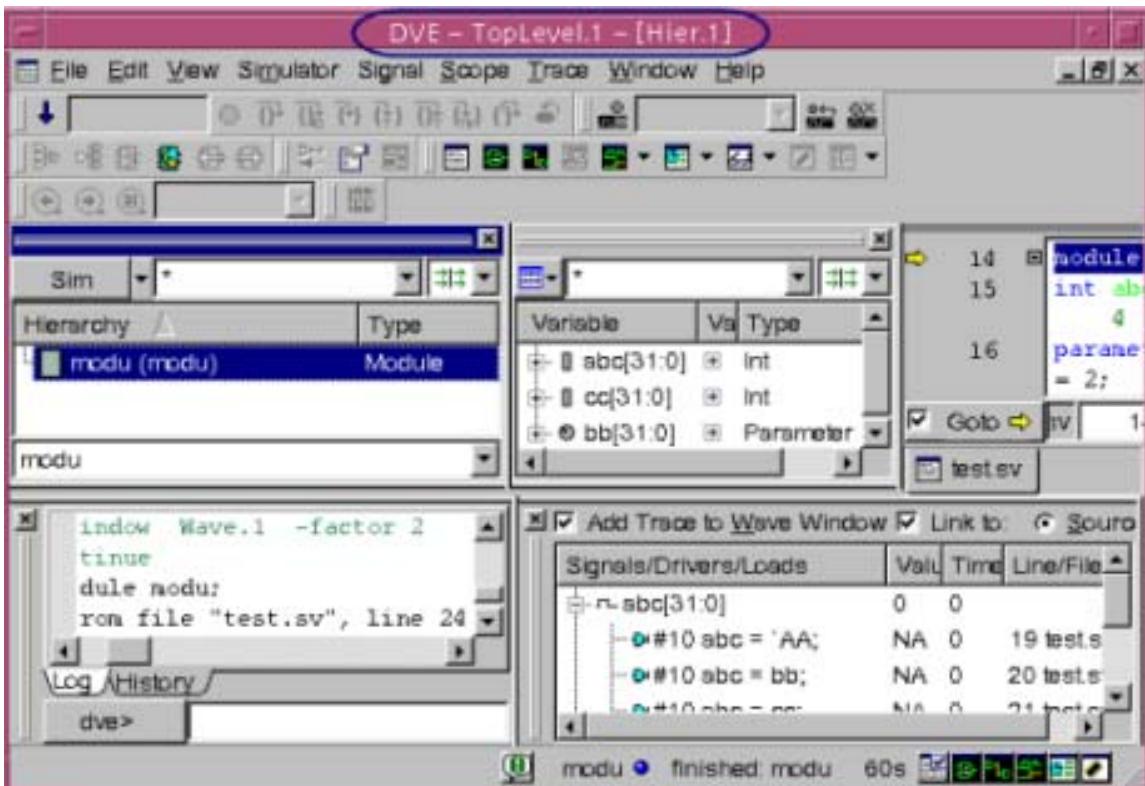
DVE - TopLevel.1 - [Data.1] to DVE - mydesign - TopLevel.1
- [Data.1]

DVE - TopLevel.1 - [Source.1 - top:design.v] to
DVE - mydesign - TopLevel.1 - [Source.1 - top:design.v]

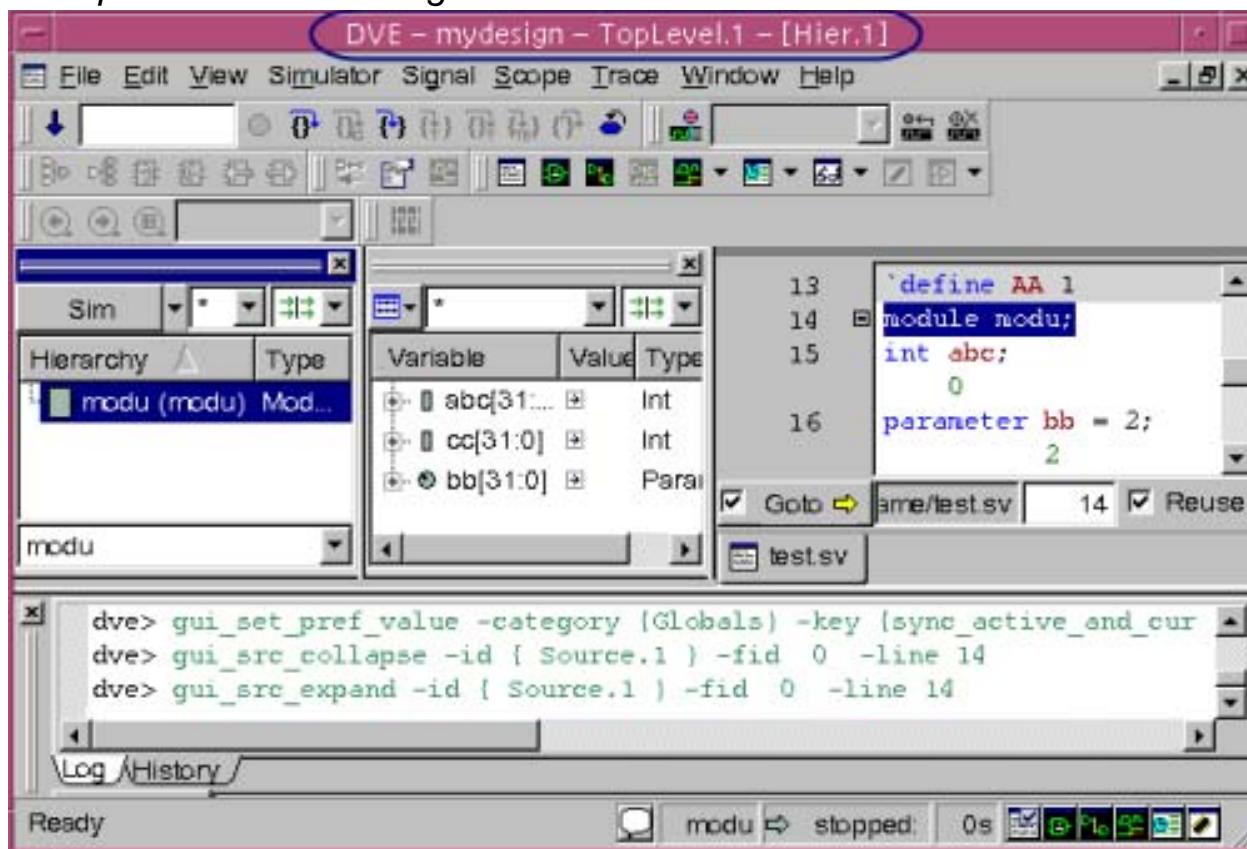
DVE - TopLevel.1 - [Console.1 - DVE Console] to
DVE - mydesign - TopLevel.1 - [Console.1 - DVE Console]

Following illustrations show the titles before and after setting the environment variable.

Example 2-1 Before setting the environment variable



Example 2-2 After setting the environment variable



Managing DVE Panes and Views

A TopLevel window is a frame that displays panes and views.

- A pane can be displayed once on each TopLevel window and it serves a specific debug purpose. Examples of panes are Hierarchy, Data, Watch, and the Console panes.

Panes can be docked on either side of a TopLevel window or remain floating in an area in the frame not occupied by docked panes (called the workspace).

- A view can have multiple instances per TopLevel window. Examples of views are Source, Wave, List, Memory, Schematic, and Path Schematic.

DVE TopLevel window can contain any number of DVE views and panes. You can choose to display data in one or many DVE windows and panes by setting defaults, using the status bar window controls, or docking and undocking windows as you work.

Managing Target Views

You can set target views to create panes either as TopLeveL window or in the existing frame. At the bottom right corner of each TopLevel window are target icons. The following table describes the icons in the their target views.

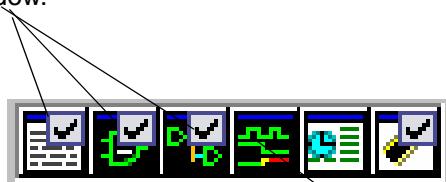
Icons	What it targets..
 Source	New Source view in a new TopLevel window.

	New Schematic view in a new TopLevel window.
	New Path Schematic view in a new TopLevel window.
	New Wave view in a new TopLevel window.
	New List view in a new TopLevel window.
	New Memory view in a new TopLevel window.

Target icons can have the following two states:

- Targeted – Indicates that a new view will be created in the current frame. This icon has a dart in it.
- Untargeted – Indicates a new TopLevel window will be created for the chosen view. This target icon has no dart in it.

Check marks indicate that targeted windows are attached to the current window.



No check exists in this targeted Wave view icon

To open a pane in a new TopLevel window

1. Click the icon in the status bar to remove the check mark.
2. Click a corresponding window icon in the toolbar to open a window of that type.

It will not be attached to the current window and will open in a new TopLevel window.

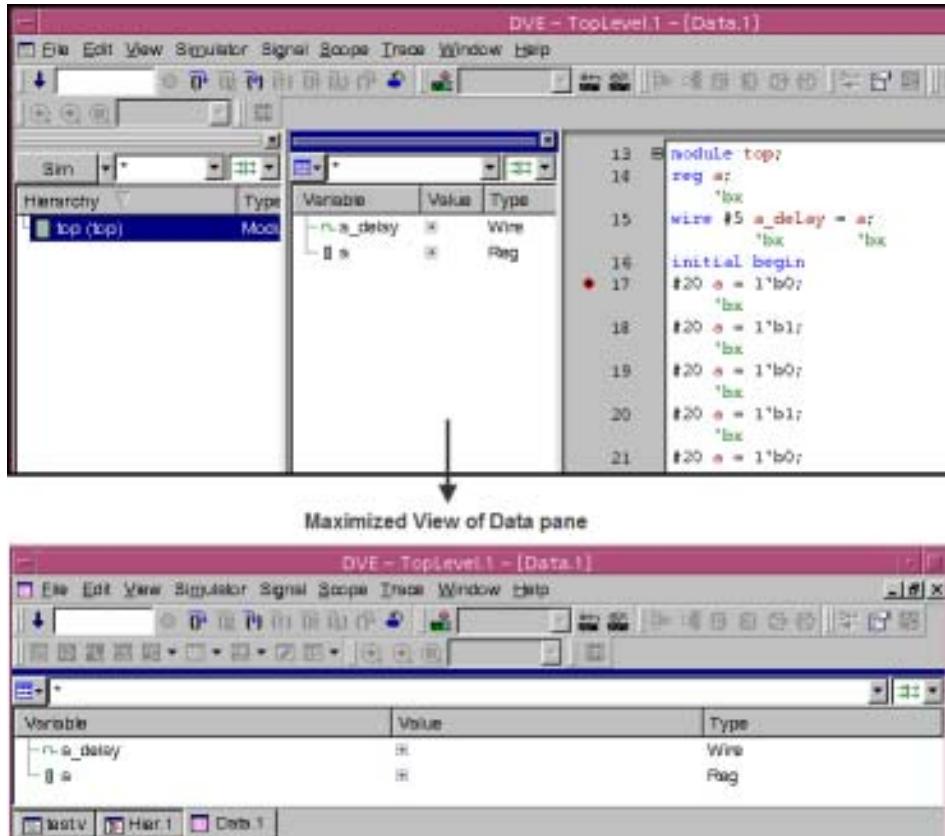
Maximizing View

You can maximize the DVE pane or view and make it a full screen view to have a larger area to work.

To maximize a pane or view

1. Select a pane or view.
2. Right-click and select **Maximize View**.

The selected pane or view is maximized to fit your screen as shown in the following figures.



3. To restore or minimize the view, double-click on the view.

To maximize a pane or view in a new toplevel window

1. Select **Edit > Preferences**.

The Application Preferences dialog box opens.

2. In the Global category, select the **“Window -> Maximize View” will expand view in new top-level window** option.
3. Double-click a pane or view that you want to maximize.

The pane or view maximizes to a new toplevel window.

Docking and Undocking Views and Panes

You can use the Windows menu to dock and undock windows and panes.

- Select **Windows > Dock in New Row**, then select the row position in which to dock the currently active window.
- Select **Windows > Dock in New Column**, then select the column position in which to dock the currently active window.
- Select **Undock** to detach the currently active window or pane.

To delete a window, click the X icon in the corner of the pane. This is the same for all dockable windows.

Dark blue color of dock handle (dock handle is the train track that connects to the X icon) indicates that this docked window is active. This is the same for all dockable windows. An action must occur such as a click to make the window active.

Dragging and Dropping Docked Windows

To drag and drop a docked window, click the dock handle and drag and drop the window to a new dock location or to a non-docked window.

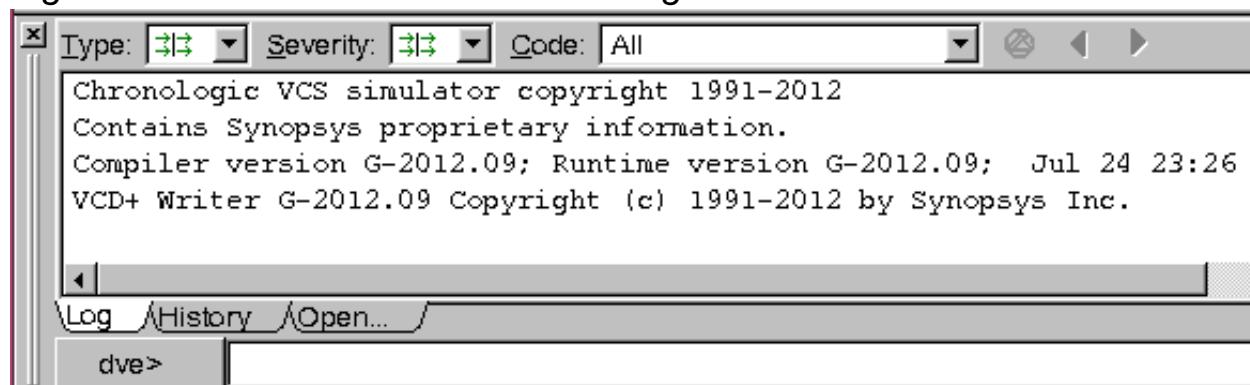
Right-clicking on the dock handle invokes a small pop-up menu:

Undock	Undocks the active window.
Dock	Left – Docks the selected window to the left wall of the TopLevel window. Right – Docks the selected window to the right wall of the TopLevel window. Top – Docks the selected window to the top wall of the TopLevel window. Not recommended. Bottom – Docks the selected window to the bottom wall of the TopLevel window.

The Console Pane

Use the command line at the bottom of the DVE top-level window to enter DVE and UCLI commands. [Figure 2-2](#) shows the command line where you enter commands. The results are displayed in the Log tab above the command line. The History tab displays the list of all commands or actions that you have taken while working in the GUI.

Figure 2-2 Command Line with the Log tab



To view the list of DVE commands, use the following command:

```
help -gui
```

To quickly view the UCLI commands and their usage, enter one of the following commands at the DVE prompt:

`help -ucli` — Displays a list of UCLI commands and a short description.

`help -ucli [argument]` — Displays a description and the command syntax.

DVE provides log analysis (diagnostic information) for each line in the log file. It provides the diagnostic information in a separate log file known as a smartlog file. For more information, see “[Using Smartlog](#)”.

The Watch Pane

Watch pane monitors the status of a specific signal, a group of signals, or an object regardless of the active thread. You can drag and drop the object or signal from the Hierarchy and Data panes into the Watch pane to view its behavior.

The Watch pane displays the selected item, its value, type, and the scope in which it belongs.

The Watch pane, by default, contains three tabs labeled Watch 1 through Watch 3. There is no limit to the number of tabs you can add. Using the check box in the scope column, you can tie the variable to a given thread throughout simulation or tie the variable to the currently selected thread.

To open the Watch pane

1. Select an object or signal from the Hierarchy or Data pane.

2. Right-click and select **Add to watches**.

The Watch pane is displayed with the selected signals.

3. To add a Watch tab, go to the menu **View > Watch > Add New Page**. You can also delete the watch tabs.

The Memory View

The Memory view displays the values of MDA in a tabular list. You can add signals to the Memory view from the Data pane.

You can perform the following tasks in the Memory view:

- Add signals to Wave view
- Add signals to List view
- Create a group of signals to display in the current Wave view
- Edit the property of the signals to display in the Wave view
- Set Radix to display the signals values in the chosen notation

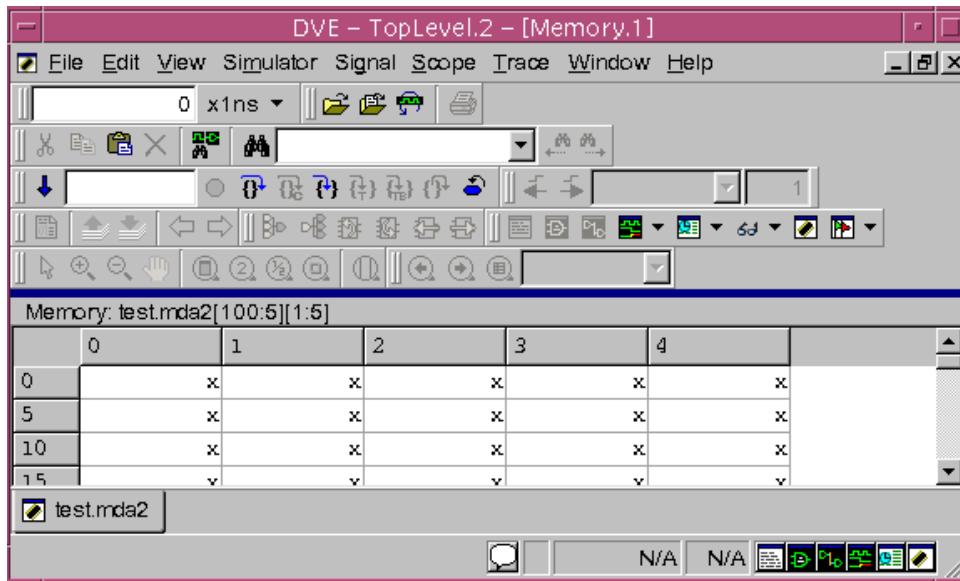
Setting Properties of Signal in Memory View

You can set the properties of signals to display the values in the Wave view.

To set the property of the signal

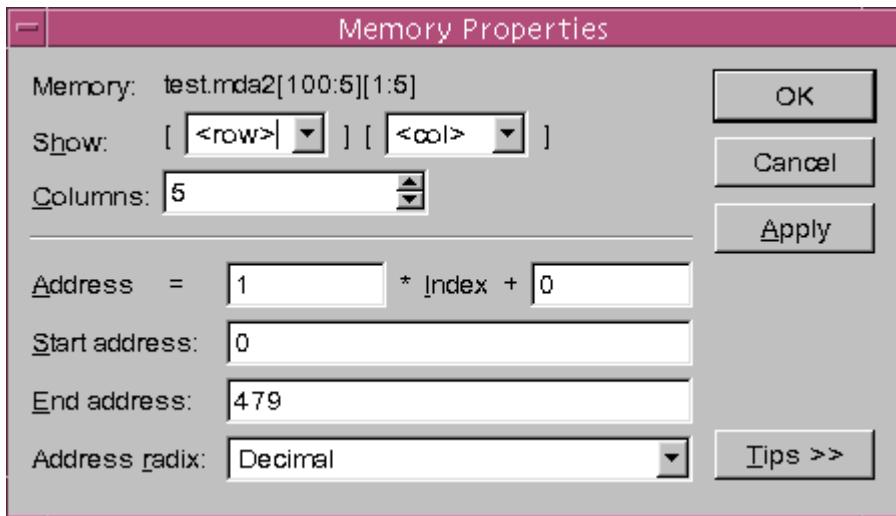
1. Select an MDA from the Data pane.
2. Click the Signal menu and select Show Memory option.

The Memory view appears with all the signals and its variables in a tabular form.



3. Select a variable in the table, right-click and select Properties.

The Memory Properties dialog box appears and displays the property of the MDA.



4. Select the following memory properties, as appropriate.

- Show — Specifies which elements to show for MDAs. For any index, you can select <row> or <col> from the list box or enter a number. The number should be a valid index for the corresponding array dimension.
 - Columns — Specifies how much table columns to use for displaying the memory.
 - Address — Specifies the formula for address computation based on index.
 - Start address — Specifies the start address of the memory.
 - End address — Specifies the end address of the memory.

The array element will be displayed in the range specified in the Start and End Address fields.
 - Address radix — Specifies radix that will be used for displaying addresses in the table.
5. Click the **Tips <<** button to view more information about the fields.
 6. Click **OK**.

The memory properties are saved.

C, C++, and SystemC Code

The following steps outline the general flow for using UCLI to debug VCS or VCS MX (Verilog, VHDL, and mixed) simulations containing C, C++, and SystemC source code.

To start the C debugger

1. Compile your VCS or VCS MX with C, C++, or SystemC modules as you normally would, making sure to compile all the C files you want to debug.

For example, for a design with Verilog on top of a C or C++ module:

```
gcc -g [options] -c my_pli_code.c  
vcs +vc -debug_all -P my_pli_code.tab my_pli_code.o
```

Or for a design with Verilog on top of a SystemC module:

```
syscan -cflags -g  
syscan -cpp g++ -cflags "-g" my_module.cpp:my_module  
vcs -cpp g++ -sysc -debug_all top.v
```

Note:

You must use `-debug` or `-debug_all` to enable debugging.

2. Open the simulation in DVE using the following command:

```
simv -gui
```

3. Select **Simulator > C/C++ Debugger** to start the C debugger.

Using the Menu Bar and Toolbar

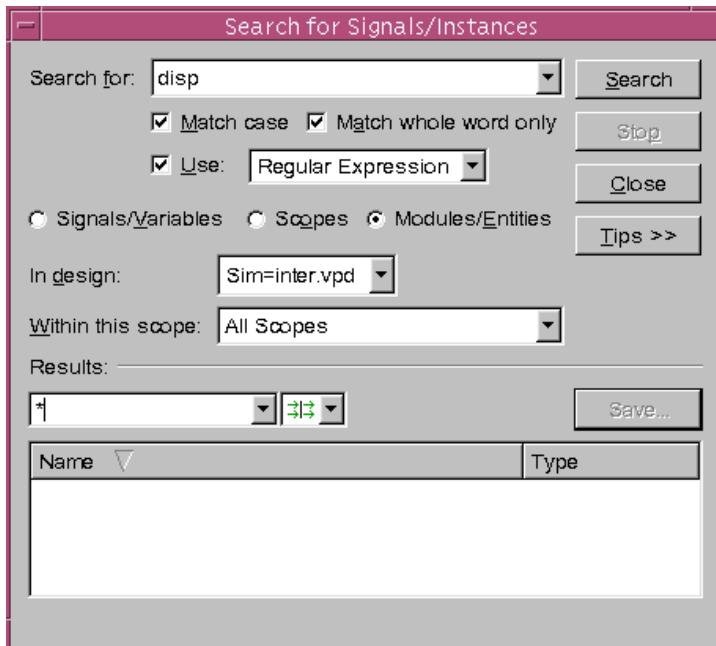
The menu bar and toolbar allow you to perform standard simulation analysis tasks, such as opening and closing a database, moving the waveform to display different simulation times, or viewing HDL source code. For more information about the menu bar and toolbar options, see the [Appendix A, "Menu Bar and Toolbar Reference"](#)

Searching Signals or Scopes

To search scopes and signals

1. Click the Search for Signals/Instances icon  on the toolbar.

The Search for Signals/Instances dialog box appears.



2. Enter the search criteria as described in the following table:

Field Name	Description
Search for:	Specifies the signal name that you want to search.
Match case:	Searches specific to the specified text case. For example, if you enter <code>disp</code> as the signal name in the "Search for:" text box, and you select this check box, DVE would search only those signals that match the case <code>disp</code> .
Match whole word only:	Searches the signals containing the whole word. For example, if you enter <code>clk reset</code> in the "Search for:" text box, and select this check box, you will not find signals <code>clk</code> or <code>reset</code> .
Use:	Finds a signal using wildcard or Regular Expression.
Signal type:	Identifies signal, instances, scope, modules, or entities.
In design:	Specifies the design in which you want to search.
Within the scope:	Specifies the scope in which you want to search.

3. Click **Search**.

All the signals matching the specified criteria are displayed in the Results text area.

4. Enter the text string to filter items or select the filter type from the pull-down menu.

5. Click **Save**.

The results are saved as a text file.

Mapping to the Location of the Source Files

DVE uses source file location for "Show source" query, schematic, and driver tracing operations. The default directory for the source files is the one they were in when last compiled.

If source files are moved to a new location, use the `-pathmap` runtime option to provide mapping to the new location of source files. You can pass this option to simv or DVE as follows:

```
-pathmap <mapfile>
```

Where, `<mapfile>` is the path map file which contains the mapping related information. Following is the syntax of `<mapfile>`:

```
<Full_Path_To_Old_Location> : <Full_Path_to_New_Location>
```

Interactive Mode

To run the simulator after the design is moved to a new location, create a path map file `<mapfile>` to relink the design directory. The simulator provides resolved path for every source file from the map file. DVE locates source file, and VPD file records the resolved path for file information.

Use Model

Perform the below steps to create a mapping from your current view to the new location where the source files are moved:

1. Create the `<mapfile>` to specify information related to mapping from the old directory to the new directory.

2. Start simulator with `-pathmap <mapfile>` option to load the path map file.

```
%simv -gui -pathmap <mapfile>
```

Post-process Mode

To debug VPD file generated before the design is moved, use path mapping mechanism to locate the source files.

Use Model

Use either of the following two ways to create a mapping from the current view to the new location where the source files are moved:

1. Use `-pathmap <mapfile>` to load map file. For example:
`%dve -vpd vpdfile -pathmap mapfile`
2. Use `gui_pathmap` to load map file dynamically:
 - a. Start DVE.
 - b. Open VPD file.
 - c. Run `gui_pathmap -add mapfile` on the DVE command-line to load map file.

Note:

- If there are path maps added by both `gui_pathmap -add` and `<mapfile>`, then path map rules loaded from `<mapfile>` will be used first.
- Use single `<mapfile>` for providing mapping information for all the source files.

- Mapping can be done for a directory or an individual file.
-

Editing Preferences

You can edit preferences to customize the display of DVE views and panes. For more information about the preferences option for all the panes and views in DVE, see the section entitled, “[Editing Preferences](#)” on page 26.

Using Context-Sensitive Menu

You can perform several actions using the context-sensitive menu (CSM) in all the DVE panes and views. For details about CSM of each pane or view, see the section entitled, “[Using the Context-Sensitive Menu](#)” on page 56.

3

Using the Hierarchy and Data Panes

This chapter describes using the DVE Hierarchy and Data panes to:

- Display the static design structure in a tree view.
- Navigate the design to view results in other DVE windows and panes.
- Display signal data.

This chapter includes the following sections:

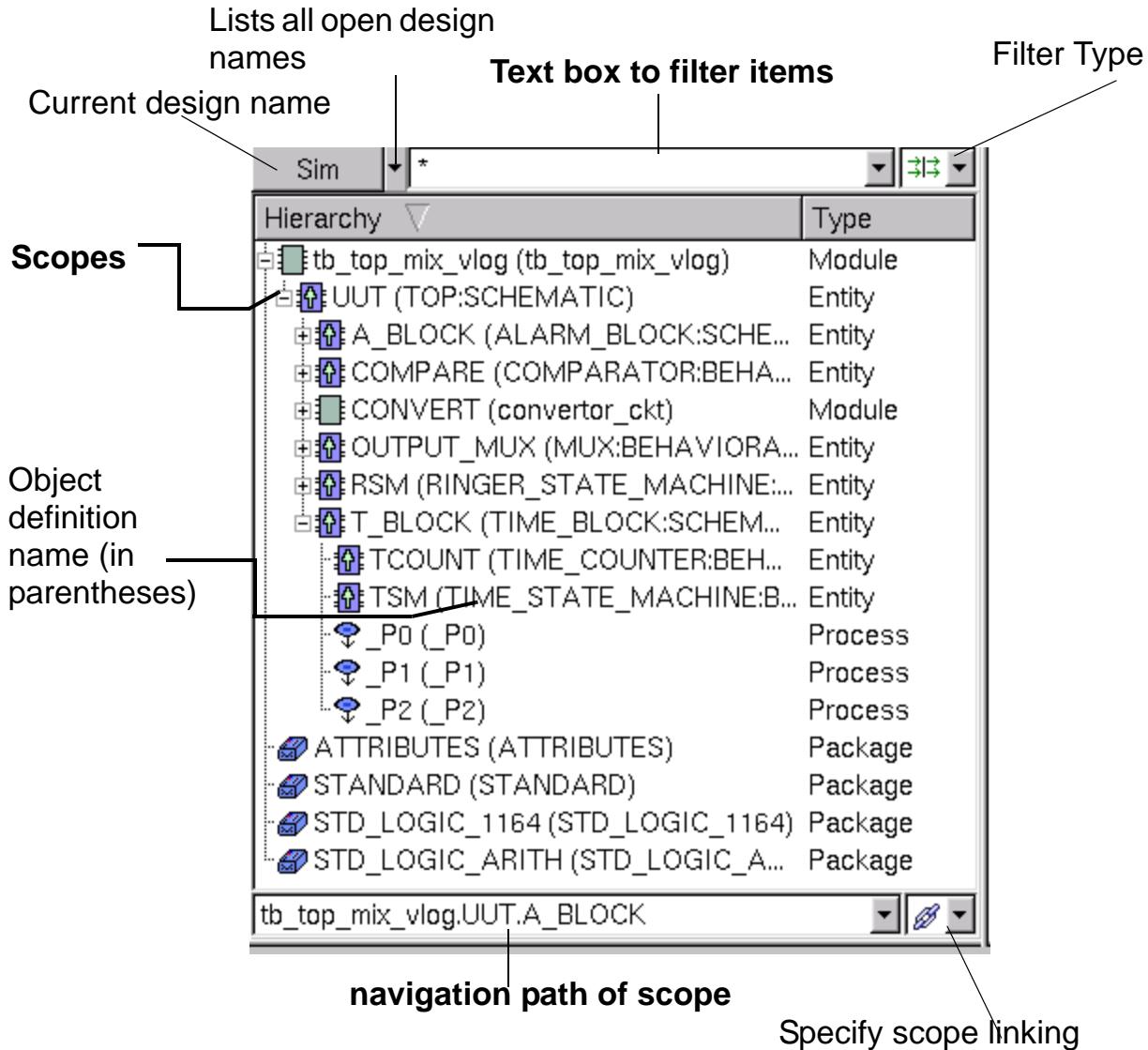
- “[The Hierarchy Pane](#)” on page 2
- “[The Data Pane](#)” on page 16

The Hierarchy Pane

The Hierarchy pane, shown in [Figure 3-1](#), is a tree view composed of the following:

- Hierarchy and Type columns.
 - The Hierarchy column displays the static instance tree. The names in the instance tree are in the instance name (definition name) format. Top modules (or scopes) are at the top-level of the tree.
 - The Type column displays the type of hierarchical object.
- A drop-down list box on the left, which is the design selection list, and contains a list of currently open designs with the current design at the top.
- A drop-down list box on the right is for filtering object types.
- A text box in the middle to input text string for searching objects.
- At the bottom of the Hierarchy pane, you can view the Scope Navigator and the Scope Linking box, when you select the Scope Navigator check box from the CSM. The Scope Navigator displays the path traversed by you with the scope and the Scope Linking box allows you to specify the scope linking.

Figure 3-1 Hierarchy Pane



Scope Types and Icons

There are various scope types in the Hierarchy pane. Each scope type is represented by a specific icon. The following table provides an overview of the scope types and their corresponding icons.

Table 3-1 Scope Types

Scope Type	Icon
Tasks (Verilog)	
Functions	
Named Blocks (Verilog)	
Packages (Verilog)	
SV Unit Packages	
Class Definition	
Interfaces (Verilog)	
Packages (VHDL)	
Blocks (VHDL)	
Processes (VHDL/SystemC/Unnamed)	
Leaf VHDL Cells	
Leaf Verilog Cells	
Leaf SystemC Cells	
OVA Unit	
All	

Filtering the objects in the Hierarchy Pane

You can filter object types such as Tasks, Functions, Blocks, Packages, Processes, Interfaces, and Unnamed Processes in the Hierarchy pane based on the object/scopes types mentioned in the table [Table 3-1](#).

To filter the data based on scope types, click the Type filter list and select or clear the desired object types.

You can also use the Text search box to filter objects. You can either use regular expressions or wildcard (*) character to search for objects.

Navigating Open Designs

In DVE, more than one design can be open, but only one of them can be active at any point of time. This active design is the "current design".

Designs are identified by designator strings, so that in cases where objects from more than one design are allowed (for example, in the Wave view), it is possible to relate object names to their designs. By default, the designators are V1, V2, V3, and so on.

For example, if a design A contains an object called `top.a` and design B also contains an object called `top.a`, these objects would be shown as `V1:top.a` and `V2:top.a`, by default. You can also choose your own design designators from the Open Database dialog box.

You can open many VPD files, but you can open only one interactive session. The designator for the interactive session is `simv`.

Expanding and Collapsing the Scope

If the scope has subscopes, a plus sign (+) appears to the left of the scope.

To expand and collapse the scope

1. In the Hierarchy pane, click the plus sign (+) beside the scope name.

All the subscopes are displayed.

2. Click the minus sign (-) .

The expanded child scopes collapse.

Rearranging Columns in the Hierarchy Pane

You can sort the Hierarchy column or rearrange the order in which the column headings appear in the Hierarchy pane.

To rearrange and sort the columns in the Hierarchy pane

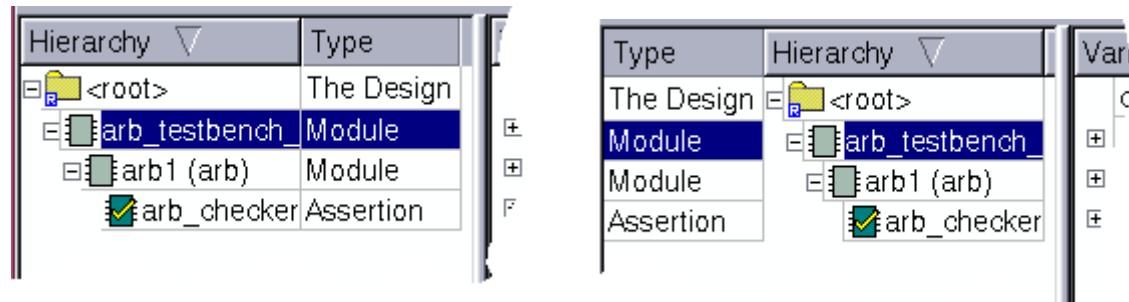
1. Click and hold the left mouse button on the column heading.
2. Drag the column to the desired location and release the mouse button.

The column moves to the desired location in the Hierarchy pane.

3. Click the arrow in the column heading.

The scopes are sorted in alphabetical order.

Figure 3-2 Moving a Column Heading



Populating Other Views and Panes

Use the Hierarchy pane to view data in other DVE windows and panes.

Displaying Variables in the Data Pane

To display variables in the Data pane, select an object in the Hierarchy pane.

Dragging and Dropping Scopes

You can drag and drop a selected object into any other DVE pane or window (such as the Source view, the Wave view, and the List view).

The following points should be noted while dragging the objects in various panes:

- Dropping a scope into the Source view displays the definition of that object in the Source view and selects the definition line.

- Dropping a scope into the Data pane causes the scope to be selected in the Hierarchy pane and displays the scope variables in the Data pane.
- Dropping a scope into the Wave view adds all the scopes signals to a new group or puts them under the insertion bar of the wave signal list.
- Dropping a scope into the Schematic view displays the design schematic for that scope.
- Dropping a scope into the Path Schematic view has no useful results.
- Dropping a scope into the Memory view is not allowed.
- Dropping a scope into the List view displays the simulation results in a tabular format.
- Dropping a scope into a text area, such as the DVE command line, drops the full hierarchical text. However, one exception is to drop a scope in the Find Dialog text entry area (either dialog or toolbar area). In this special case, just the leaf string is dropped. For example, dropping "top.c.b.a" results in just "a" in the Find text area.

If you select more than one hierarchy object (you can do this by pressing the Control key and clicking the mouse button), the object closest to the linear top of the list is dropped. For example:

```
top
  top.a
    top.a.b
  top.b
    top.b.b
```

In this example, if you select, drag, and drop both `top.a.b` and `top.b` into a text area, DVE drops only `top.a.b`.

Dumping Signal Values

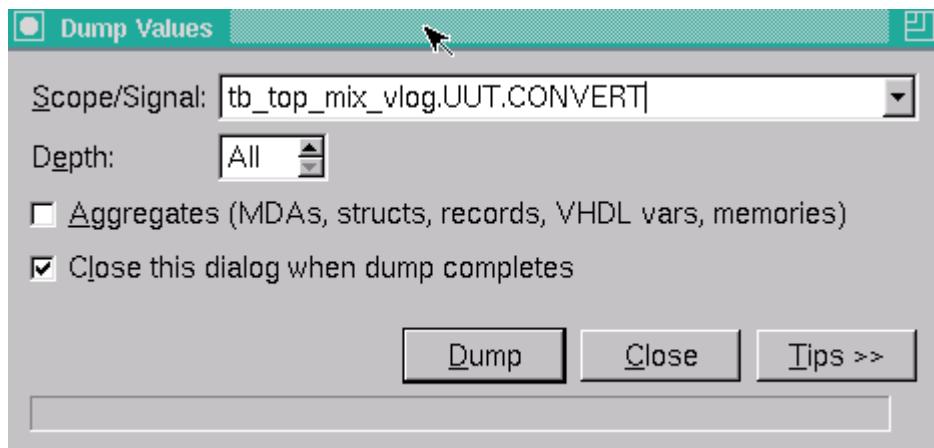
To dump signal values

1. Select the scope in the Hierarchy pane, right-click and select **Add Dump**.

or

Select **Simulator > Add Dump**.

The Dump Values dialog box appears.



2. Select the Scope/Signal that is to be dumped.
3. Select the **Depth** to specify the level in the hierarchy for which the objects are to be dumped. The default depth is All.
4. Select **Aggregates** to dump all complex data types.
5. Select the **Close this dialog when dump completes** check box to close the database after dumping is completed.
6. Click **Dump**.

The values are dumped recursively to a VPD file.

Moving Up or Down in the Hierarchy Pane

In designs with multiple scopes, when you have expanded the scopes to view the full hierarchy, it is time consuming to scroll up and down to find the desired scope. In case you have the same instance in many scopes, while searching for the desired instance, you might look into the wrong scope.

With this feature, you can limit the display of scopes in the Hierarchy pane to view only the desired scope.

Example

The following example contains multiple scopes.

test.v

```
`define fadd_s fadd_primgates_1
`define fadd_b fadd_pblock_arith
module fadd_pblock_arith (co, sum, a, b, ci);
    output co, sum;
    reg    co, sum;
    input   a, b, ci;

    wire [1:0] isum = a + b + ci;

    always @(isum)
        begin
            co = isum[1];
            sum = isum[0];
        end

    endmodule

module fadd_primgates_1 (co, sum, a, b, ci);
    output co, sum;
    input   a, b, ci;
```

```

        wire axorb, aandb, aandci,
              bandci;

        xor      g1 (axorb, a, b);
        xor #1 g2 (sum, axorb, ci);

        and     g3 (aandb, a, b),
              g4 (aandci, a, ci);

        and     g5 (bandci, b, ci);

        or   #1 g6 (co, aandb, aandci, bandci);

endmodule

module add4_bsBS (co, sum, a, b, ci);
    output          co;
    output [3:0] sum;

    input [3:0] a, b;
    input          ci;

    wand [2:0] icar;

    `fadd_b m0 (icar[0], sum[0], a[0], b[0], ci);
    `fadd_s m1 (icar[1], sum[1], a[1], b[1], icar[0]);
    `fadd_b m2 (icar[2], sum[2], a[2], b[2], icar[1]);
    `fadd_s m3 (co,         sum[3], a[3], b[3], icar[2]);

endmodule
module test_add4;
    reg [3:0] a, b;
    reg          ci;

    wire [3:0] sum;

    add4_bsBS m0 (yY, sum, a, b, ci);

    initial
        begin : stim
            integer i;

```

```

a = 0; b = 0; ci = 0;

forever
begin

repeat (100)
#10 {a, b, ci} = $random;

#10
$finish;

end
end
endmodule

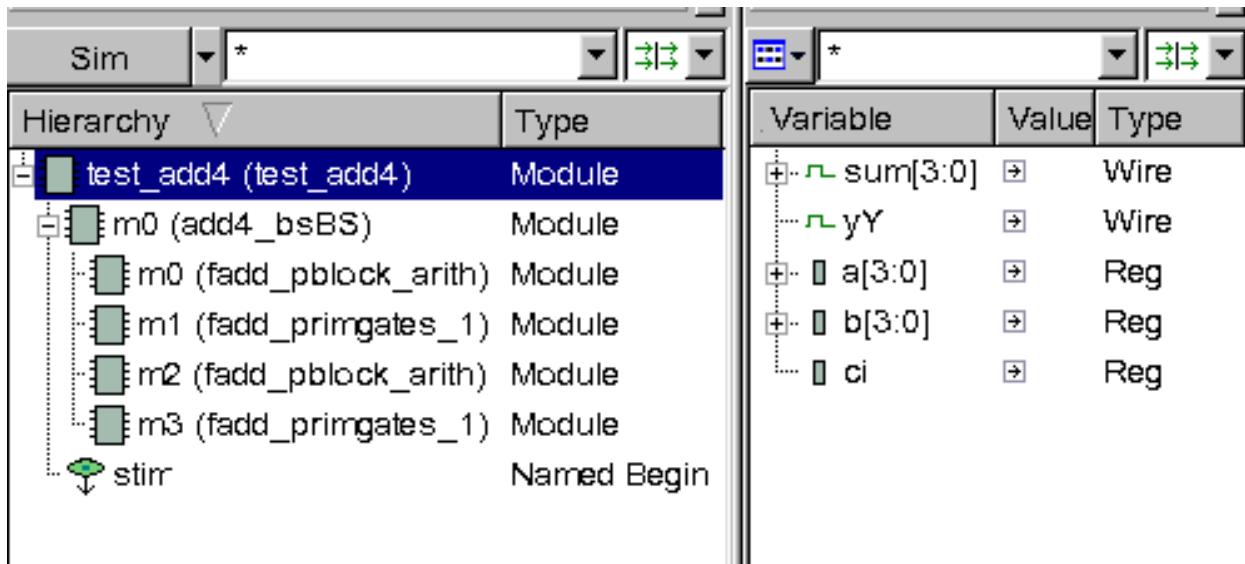
```

To compile this example, use the following commands:

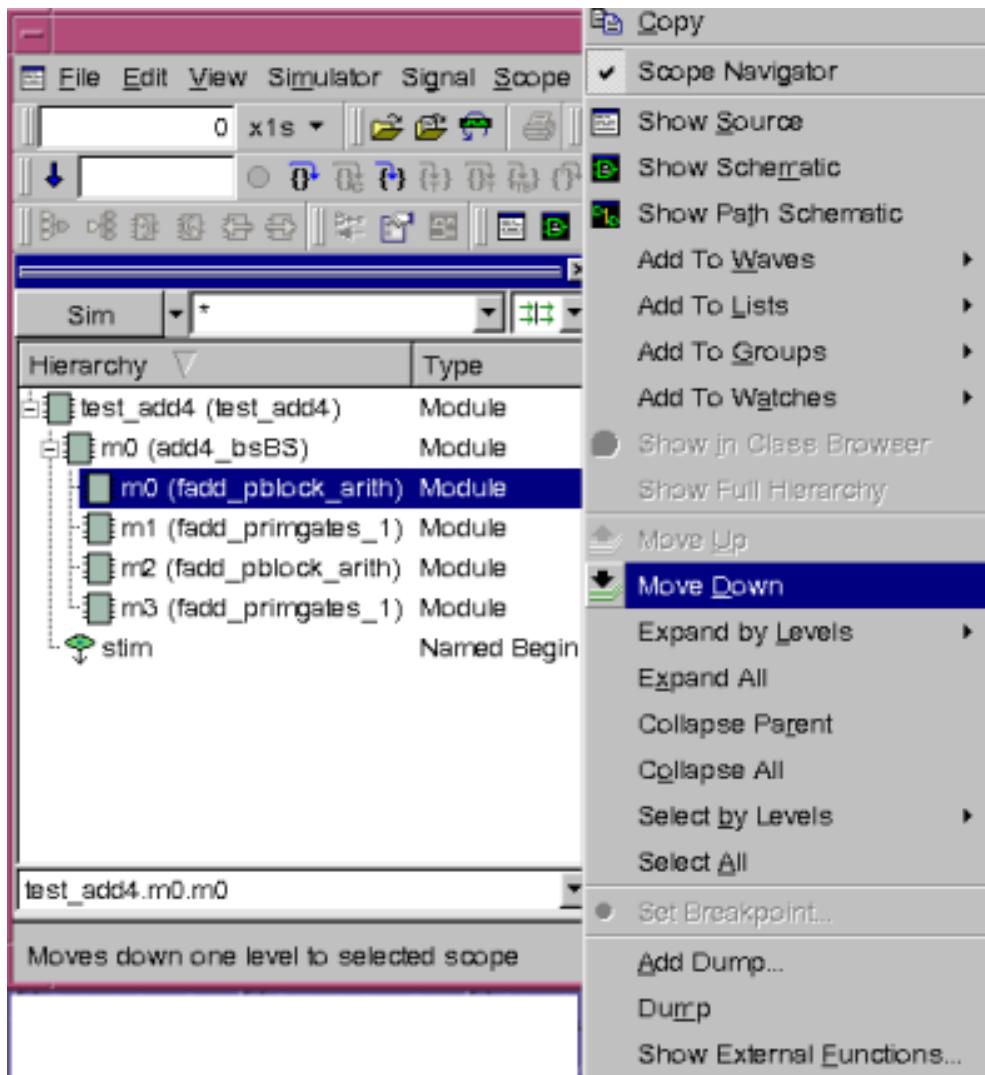
```
vcs -debug_all test.v
```

To move up and down in the Hierarchy Pane

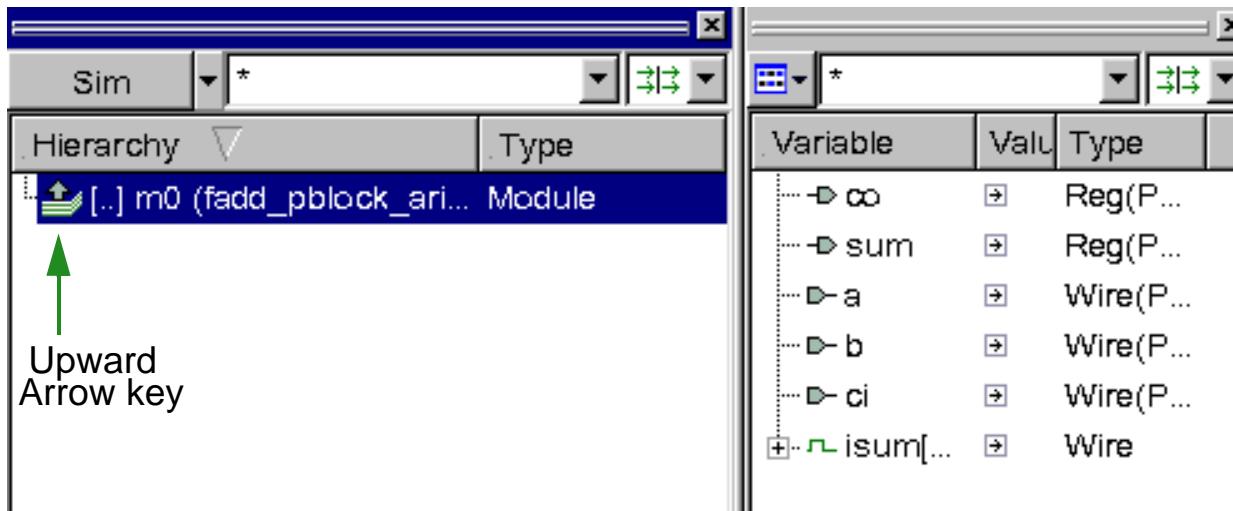
1. Run the design and load it in DVE.



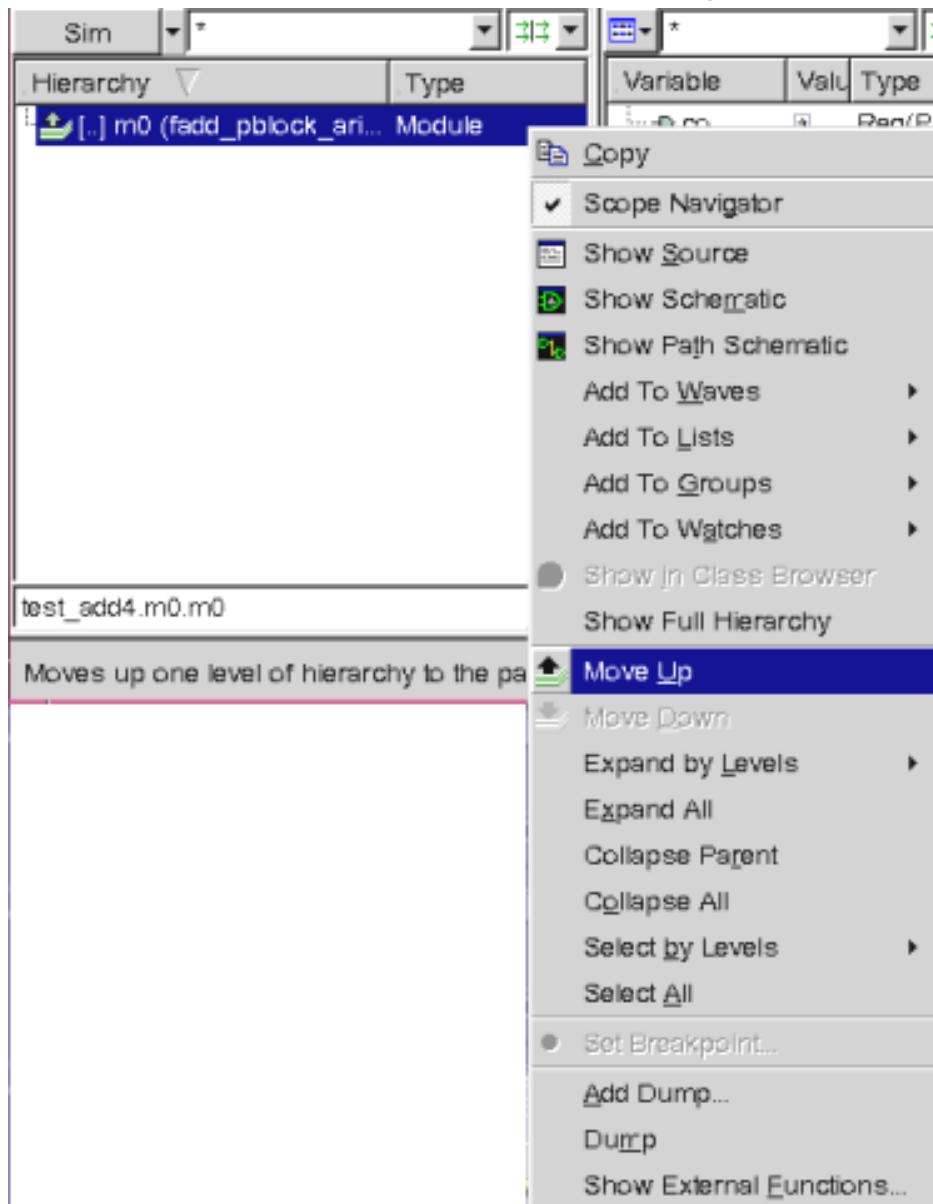
2. Select your desired scope or module in the Hierarchy pane, right-click and select **Move Down**.



The hierarchy descends to the selected scope. The scope or module name is appended with a “[.]” sign and an upward arrow key can be seen. The **Move Up** option gets enabled.



3. Select the descended scope, right-click and select **Move Up**.



The control moves one level up in the hierarchy.

4. Type "/" in the Scope Navigator or right-click and select **Show Full Hierarchy** to see the full hierarchy.

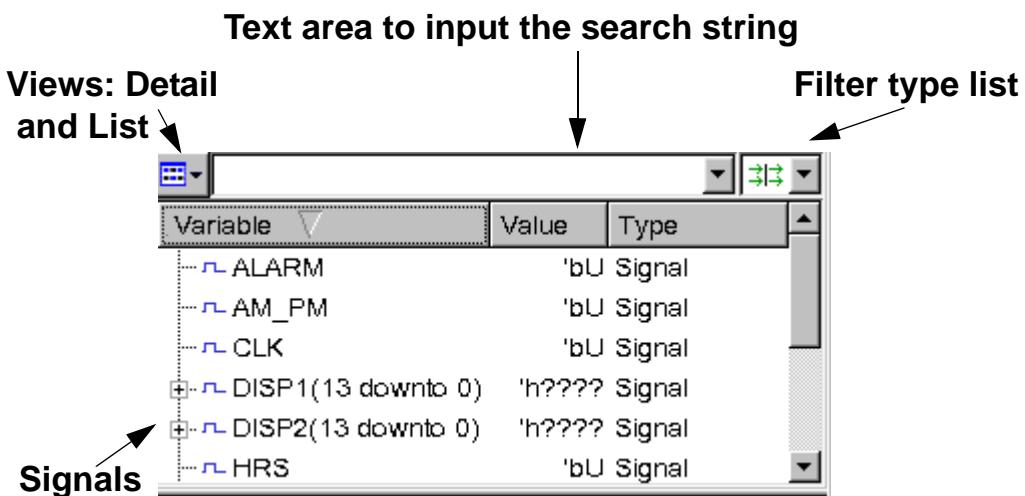
You can also press CTRL + double-click to descend or ascend in the hierarchy, or view the full hierarchy.

You can use the text filter and type filter in the Hierarchy pane to filter the flat hierarchy. Text filter only works for the current level of hierarchy; it is reset when you go down into a new scope, while it remains if you ascend in the hierarchy.

The Data Pane

The Data pane displays the signals and values of the corresponding scope that you select in the Hierarchy pane. You can view the signal data either in Detail mode or List mode.

Figure 3-3 Data pane



Similar to the tasks that you perform in the Hierarchy pane, you can also perform in the Data pane, such as:

- View Signals and their values

- Filter the signals
 - Rearrange the columns
 - Dump Signals
 - Add signals to the views or panes, such as Wave view or Watch pane,
-

Viewing Signals and Values

To view signals and their values in the Data pane

1. Click the arrow  next to the object in the Data pane.

The values at the current simulation time of the selected scope are displayed. You can also click the Annotate Values



icon on the toolbar to view the signal values.

2. Click the arrow in the **Variable** column.

The signals get sorted in ascending, descending, and by declarations order.

3. Click the down arrow to display the type filtering pull-down menu.

4. Select or clear the check boxes against each filter type.

The signals are filtered based on your selection.

5. Select a signal in the Data pane, then select **Source > Show Source**.

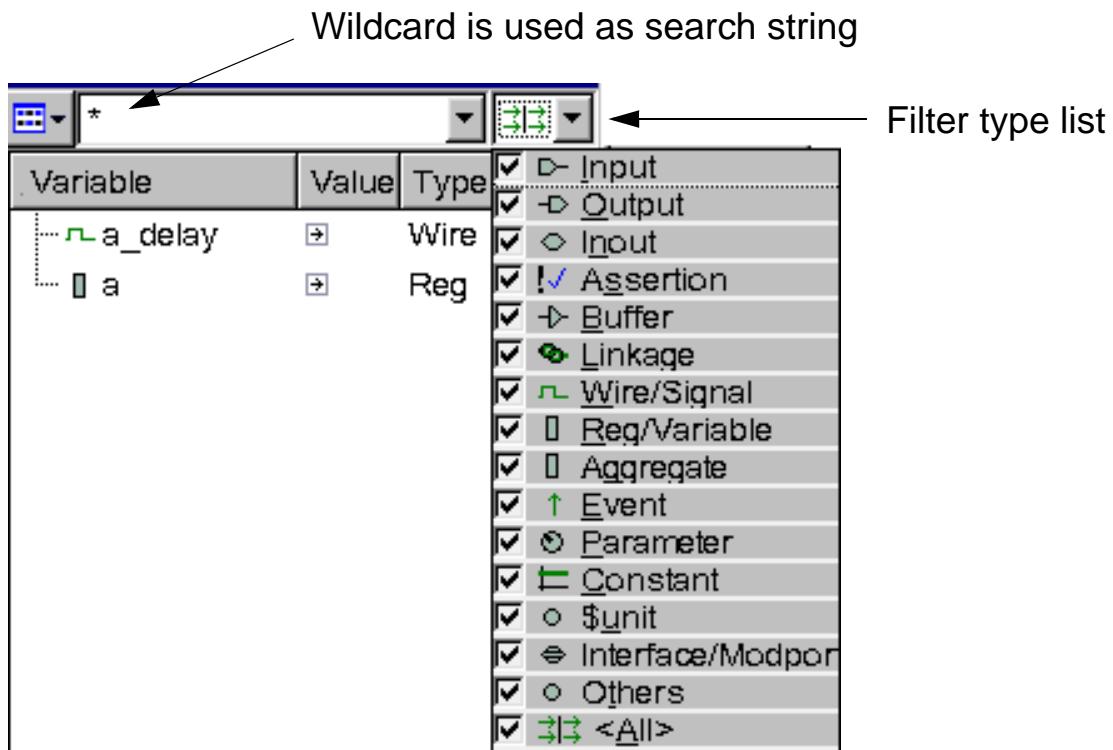
The source code of the selected signal is displayed in the Source view.

Filtering the Signals

You can filter the signals based on their types in the Data pane. To filter the signals based on their types, click the Filter type list and select or clear the desired signal types.

You can also filter the signals based on the text string. For example, type the search string in the form of regular expressions or wildcards (*) in the Text box to filter the signals.

Following are the available filter types:

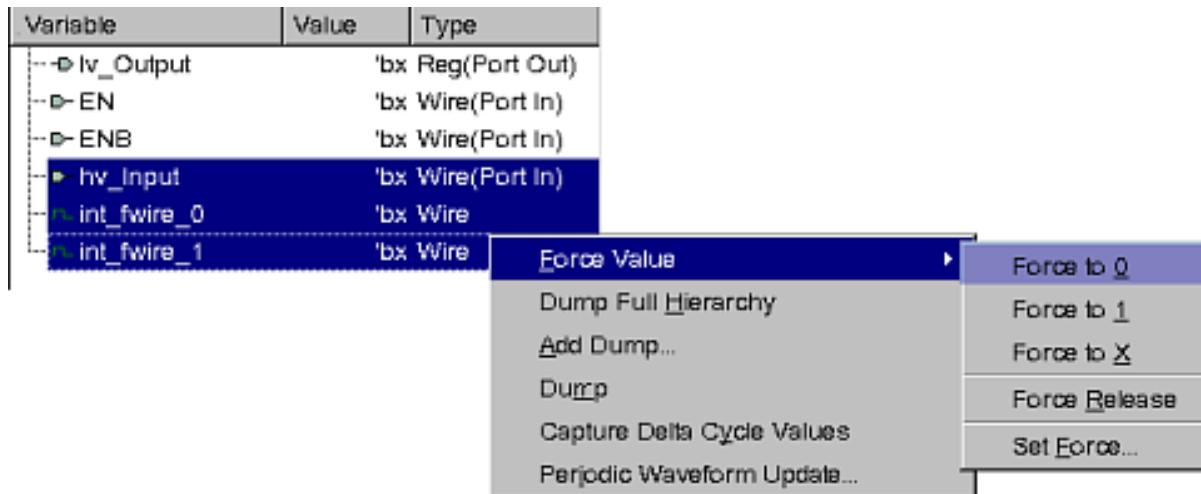


Forcing Signal Values

You can force single or multiple signals to a certain values (say 0/1/X) and also release the forces on those signals without opening the Force Values dialog box. You can create force as clock signals in the Force Values dialog box.

To force signal values without opening the Force Values dialog box

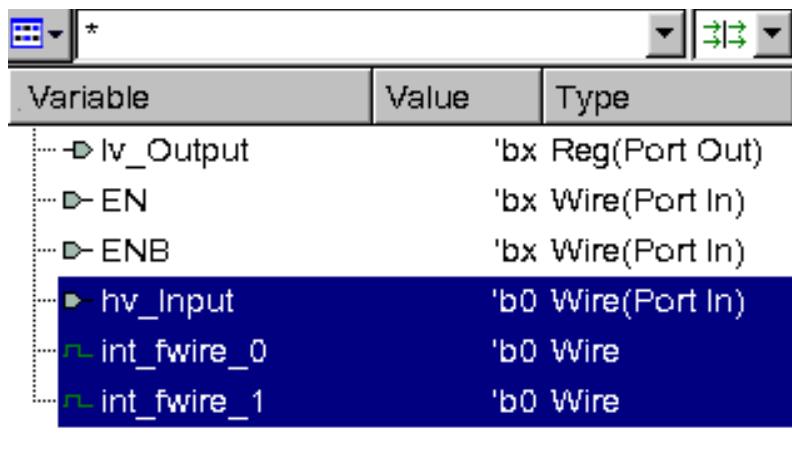
1. Select one signal or multiple signals in the Data pane.



2. Right-click and select **Force Value**.
3. Select one of the following options from the Force Value submenu:
 - Force to 0 — Forces the value of signal to zero.
 - Force to 1 — Forces the value of signal to 1.
 - Force to X — Forces the value of signal to X.

- Force Release — Releases the forces from the selected signals.
- Other Force — Opens the Add Force dialog box.

For example, if you want to force the value zero, select the **Force to 0** option. The signal values are changed to zero.



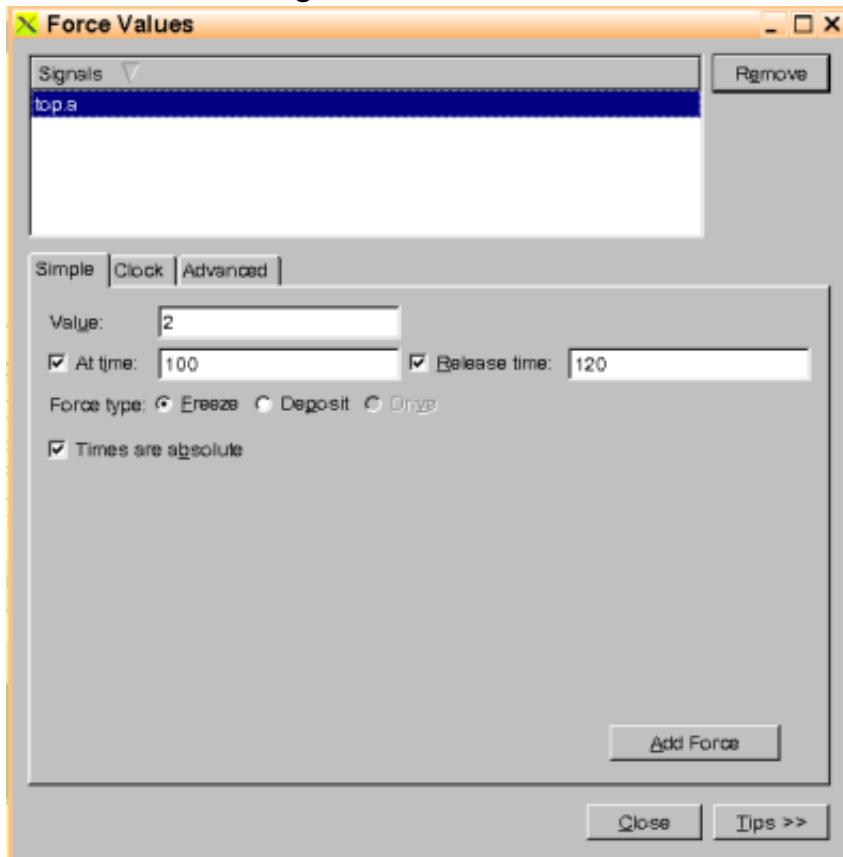
The screenshot shows a software interface for setting signal values. At the top, there are buttons for 'File', 'Edit', 'Search', and 'Help'. Below is a table with three columns: 'Variable', 'Value', and 'Type'. The table lists several variables:

Variable	Type
lv_Output	'bx Reg(Port Out)
EN	'bx Wire(Port In)
ENB	'bx Wire(Port In)
hv_Input	'b0 Wire(Port In)
int_fwire_0	'b0 Wire
int_fwire_1	'b0 Wire

To force signal values from the Force Values dialog box

1. Select **Set Force** from the **Force Values** submenu, as specified in the previous procedure.

The Force Values dialog box appears with all the signals listed in the Signals list box.



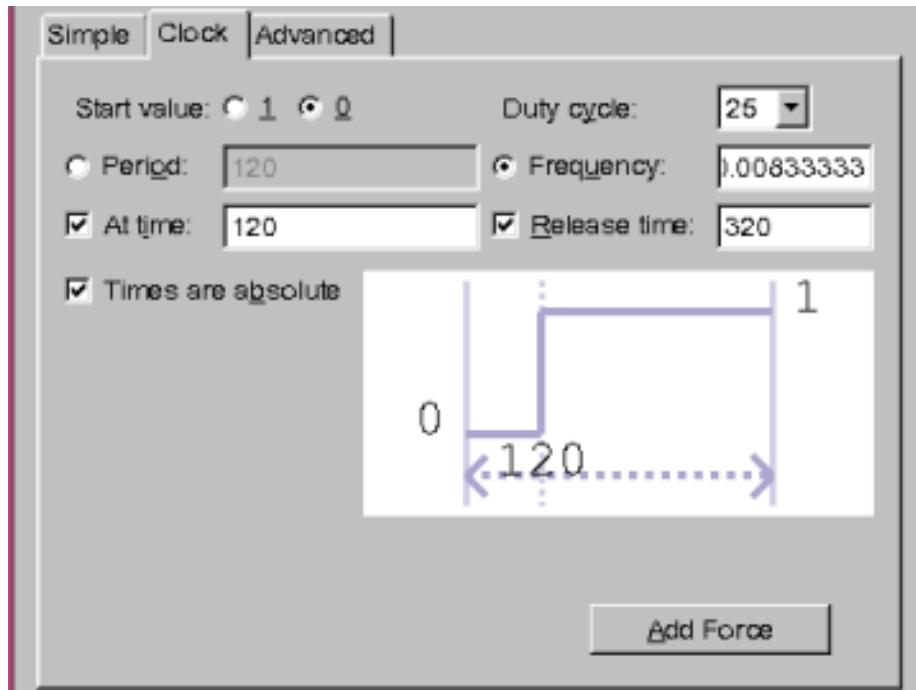
2. (Optional) Select a signal from the Signals list box and click **Remove**.

The signal is removed.

3. Enter appropriate values in the following tabs:

- Simple tab — Enter the forced value in the **Value:** field and select the **At time:** and **Release time:** to set the time (with time unit) when the forced value takes effect or is released. These two fields only accept numbers (digit + ".") and time units. The **drive** option is not available for non-VHDL signals.

- Clock tab — Specify the start value and the end value of the clock. Set the Period or the Frequency to specify the clock period. If the frequency is used, the clock period equals “1s/frequency”.



The duty cycle is 50% by default; you can select any value from the list. The clock waveform displays the start value, end value, period, and the duty cycle as you select the values in various fields.

- Advanced tab — Use the Time-Value pair table to enter the value and time to generate more complex force values.

The screenshot shows the 'Advanced' tab of a dialog box. At the top, there are three tabs: 'Simple', 'Clock', and 'Advanced'. The 'Advanced' tab is selected. Below the tabs is a table with two rows:

Value	Time
20	100
30	200

Below the table are three buttons: 'Insert', 'Delete', and 'Clear'. Underneath these buttons is a section labeled 'Force type:' with three radio buttons: 'Freeze' (selected), 'Deposit', and 'Drive'. There are also two checkboxes: 'Times are absolute' (checked) and 'Repeat period:' (unchecked). Another checkbox 'Release time:' is also present. At the bottom right is a button labeled 'Add Force'.

4. At any point, click the **Tips>>** button to view descriptions of all the fields.
5. Click **Add Force**.

The values are forced onto the selected signals.

Viewing Interfaces as Ports

You can view Interface/Modport in the Data pane when it is passed as port. You need to select the module name in the Hierarchy pane to view the port in the Data pane. You can add the interface/modport port to the Wave view, List View, or Watch view.

To view the interface port in Data pane

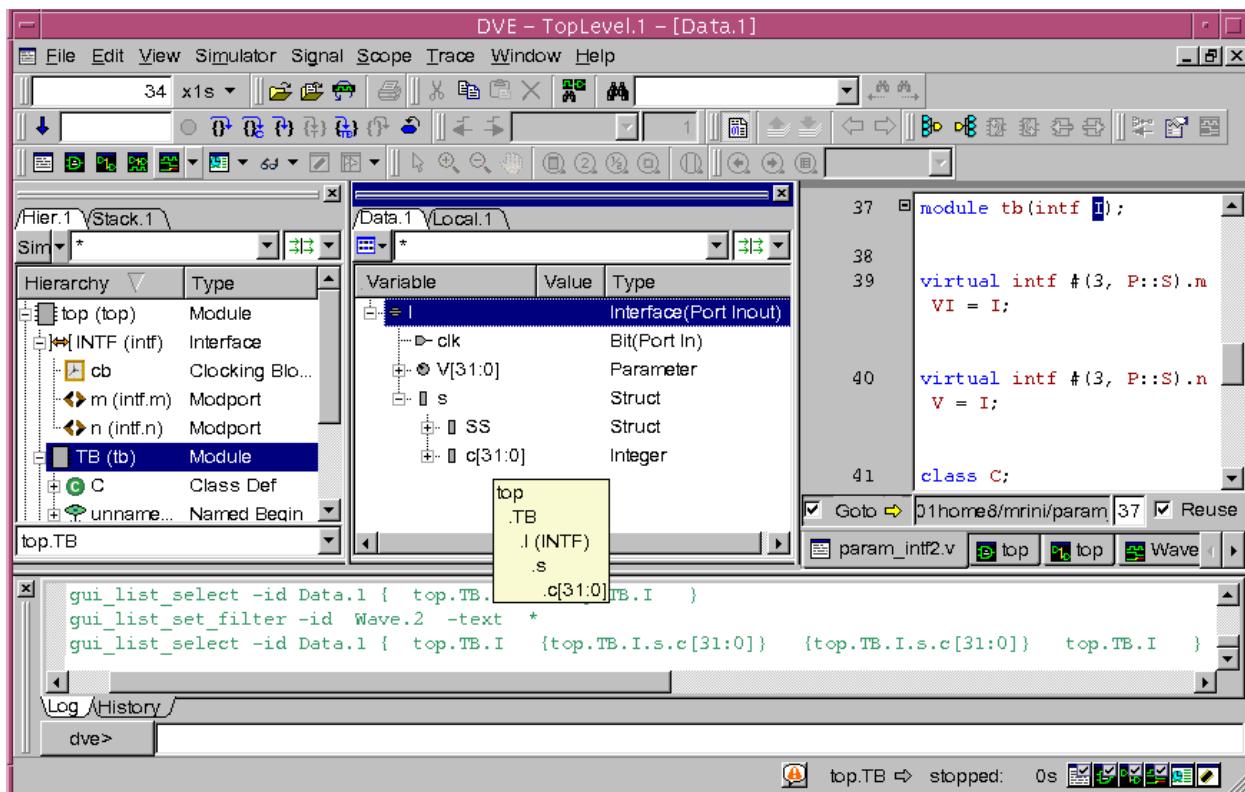
1. Load the database in DVE.

The module is displayed in the Hierarchy pane.

2. Select the module.

The interface/modport and its type is displayed in the Data pane.

The tooltip shows the interface/modport used.



3. Click the “+” button under the Variable column in the Data pane to expand the interface/modport port.

The signals under the interface/modport port are displayed. You can also sort the signals by declaration.

4. Right-click the interface/modport port in the Data pane and select **Show Source**.

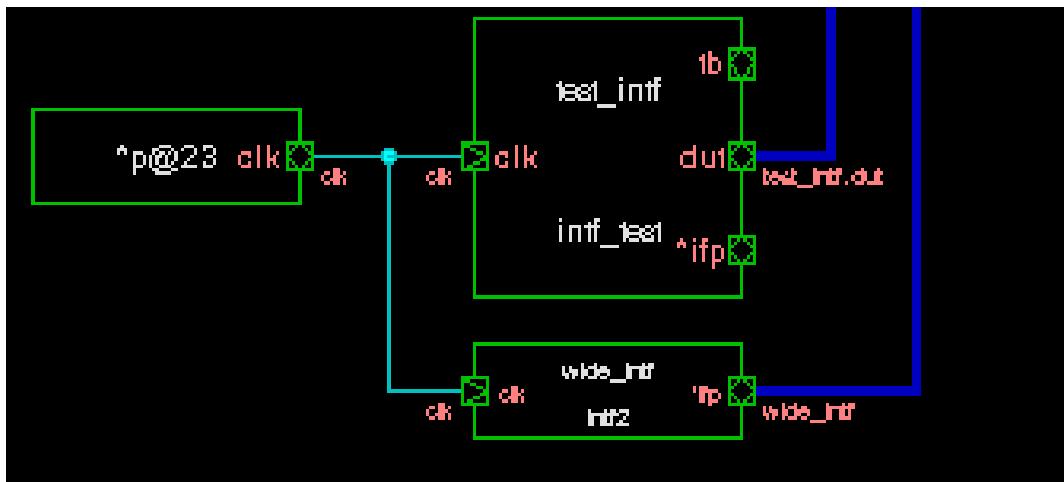
The source of interface/modport is shown in the Source view. You can also drag and drop the interface/modport from the Data pane to the Source view.

5. Use the Text filter or Type filter drop-down and select the Interface/Modport port filter to filter the signals.
6. Select the interface/modport port in the Data pane and select **Signal > Show Definition** from the menu or right-click the signal and select **Show Definition**.

The definition is shown in the Hierarchy pane, signals of interface/modport port in the Data pane, and the definition location is shown in the Source view. You can also drag and drop the interface/modport port from the Data pane to the Wave view.

7. Select the interface/modport port in the Data pane, right-click and select **Show Schematic** or **Show Path Schematic**.

The schematic or path schematic is shown. You can also trace



drivers or loads for the interface signals.

Note:

- Interface array port is not displayed in the Data pane.

- Follow signal does not work for interface port and signals of interface port.
 - Modport clocking port is not shown in the Data pane.
-

Viewing \$unit Signals

\$unit is the name of the scope that encompasses a compilation unit. Its purpose is to allow the unambiguous reference to declarations in the compilation unit scope. This is done through the same class scope resolution operator used to access package items. For more information about compilation units, see the chapter “Hierarchy” in the *IEEE P1800 SystemVerilog LRM*.

You can view the \$unit signals in the Data pane.

To view the \$unit signals

1. Load the design in DVE.

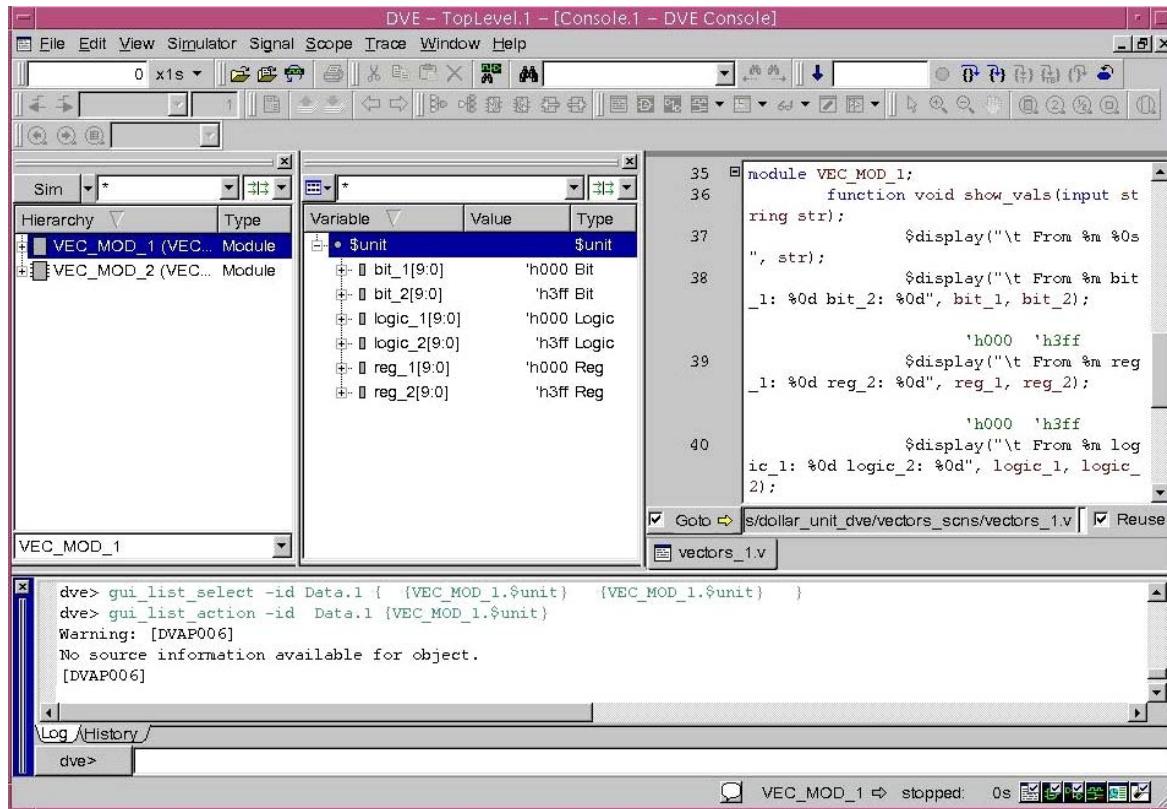
The design appears in the Hierarchy pane.

2. Select a scope in the Hierarchy pane.

The \$unit folder is visible in the Data pane with the Type field \$unit.

3. Expand the \$unit folder.

The signals under the \$unit folder are visible.



4. Filter or sort the signals under the \$unit folder, if required.
5. Add and view the \$unit signals in Waveform, Watch, and List window, as required.

Note:

- \$unit is not visible in the Hierarchy pane because it is not a global scope.
- The task, function, class definition defined in \$unit are not visible in the hierarchy pane or data pane. You can view the task, function, or class if you put them in a global package.
- Drivers or loads, schematic, path schematic, or back trace schematic are not supported for \$unit signals.

Debugging Partially Encrypted Source Code

You can debug the partially encrypted source code using the +object_protect option. In the following example of ‘protect’ and ‘endprotect’, when in full protect, both the objects ‘r’ and ‘l’ are not visible for any debug operations like VPI, VPD etc. But in partial protect, the object ‘r’ gets debug visibility and the object ‘l’ is invisible and you can’t access it.

test.sv

```
module m ();
    wire r;
    assign r = 1'b0;

    `protect
        logic l;
        assign l = 1'b1;
    `endprotect

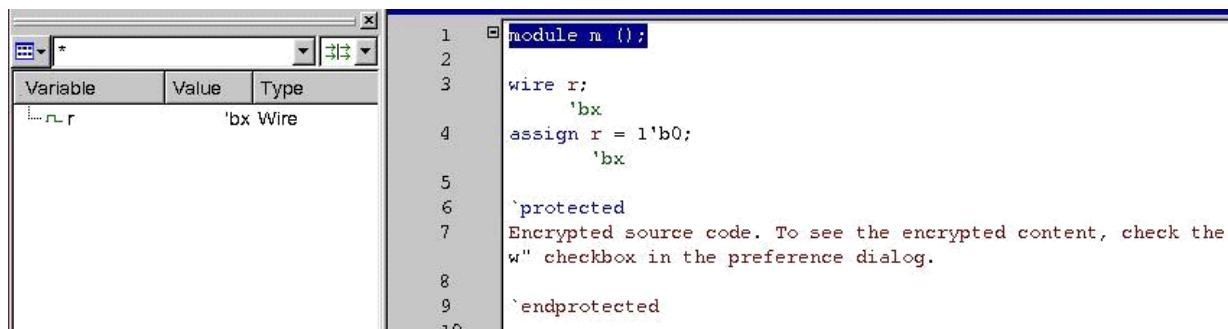
    initial #10 $finish();

endmodule
```

To compile this example code, use the following commands:

```
vcs test.sv -sverilog +protect +object_protect
vcs -sverilog -debug_all test.svp
simv -gui &
```

The following illustration display the variable 'r' in the Data pane.



The screenshot shows a software window with two panes. The left pane is titled 'Data' and contains a table with three columns: 'Variable', 'Value', and 'Type'. There is one entry: 'r' with value ''bx' and type 'Wire'. The right pane is titled 'Source' and displays Verilog source code:

```
1 module m ();
2
3   wire r;
4   assign r = 1'b0;
5
6   `protected
7   Encrypted source code. To see the encrypted content, check the
8   "w" checkbox in the preference dialog.
9   `endprotected
10
```

A note at the bottom of the source code pane states: 'Encrypted source code. To see the encrypted content, check the "w" checkbox in the preference dialog.'

4

Using the Source View

The Source view displays the HDL, any foreign language (C, C++, SystemC or OV) or assertion source code of your design. You can open as many Source views as you need to perform your analysis by selecting **Window > New > View > Source View**. You can also set the number of Source views that DVE should display in the TopLevel window.

This chapter includes the following topics:

- “[Loading Source Code](#)” on page 2
- “[Using the Mouse in the Source View](#)” on page 5
- “[Working with the Source Code](#)” on page 6
- “[Navigating the Design from the Source View](#)” on page 17
- “[Navigating Code in Interactive Simulation](#)” on page 18

- “Setting Breakpoints in Interactive Simulation” on page 18
 - “Annotating Values” on page 37
-

Loading Source Code

This section includes the following topics:

- “Loading a Source View from the Hierarchy Pane”
 - “Loading a Source View from the Assertion View”
 - “Displaying Source Code from a File”
-

Loading a Source View from the Hierarchy Pane

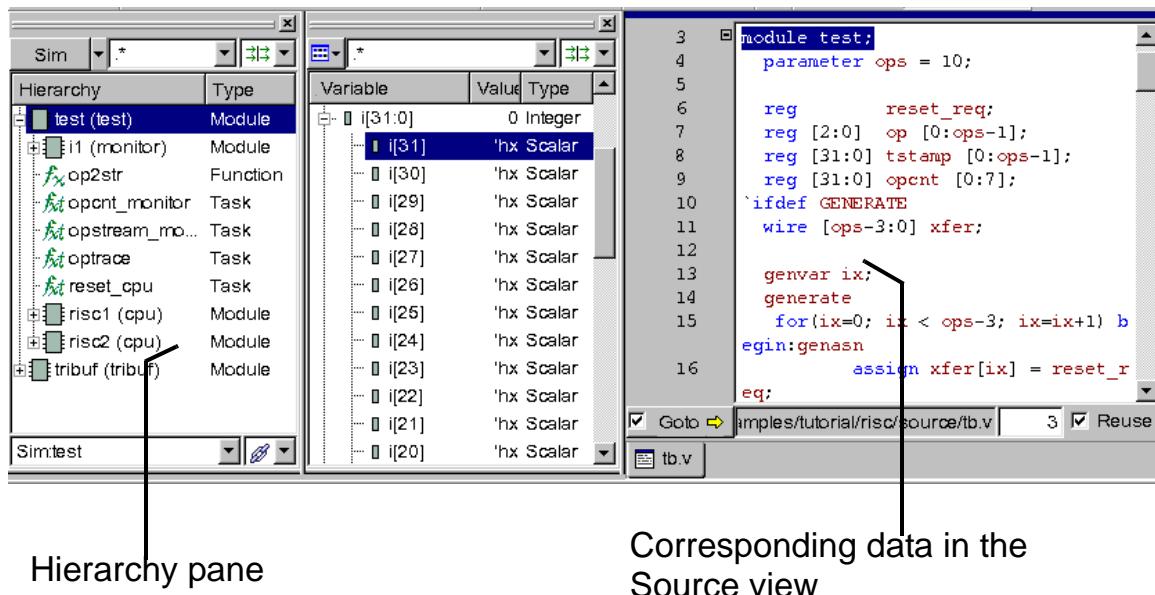
Ensure that a database is currently loaded in the Hierarchy pane.

To load the Source view from the Hierarchy pane

1. In the Hierarchy pane, perform one of the following:
 - Select a scope, then select the menu **Scope > Show Source**.
 - Select a scope, right-click and then select **Show Source**.
 - Double-click on a scope icon.
 - Drag and drop scope from the Hierarchy pane to the Source view.

The Source view loads the data corresponding to the selected scope.

Figure 4-1 Loading the Source View



Loading a Source View from the Assertion View

If your design contains assertions, the Assertion view loads results when you open the simulation database.

Note:

The Assertion view is not loaded by default, but you can choose to open the Assertion view automatically by selecting the **Automatically open assertion window** option in the Application Preferences dialog box.

To load assertion code into a Source view via the Assertion view

- Select an assertion in either tab, then select **Scope > Show Source**.

- In the Assertion Summary tab or the Assertions tab, double-click the variable or assertion you want to display in the Source view.
- In the Assertion Summary tab or the Assertion tab, drag and drop the item to the Source view.
- Select **File > Open File**, then select an assertion file.

DVE loads and displays the source file.

Displaying Source Code from a File

You can open a source file in the existing Source view, or you can open a new window.

To display the source code from a file

1. Select **File > Open File**.

The Open Source File dialog box appears.

2. Select the name of the design file you want to display from the browser, and then click **Open**.

DVE loads and displays the selected source file.

Using the Mouse in the Source View

The following table describes the different mouse actions in the Source view:

Mouse Action	Command Operations
Left-click	Clears the current selection and selects a signal or an instance.
Drag-left	Selects area for multiple selection.
Click on the line number	Selects the whole line.
Double-click on a signal name	Traces the signal's drivers.
Double-click on an instance	Pushes down into the instance's definition module.
Double-click on a module name	Displays the upper hierarchy and locates the module's instantiation.
Double-click on an architecture	Jumps to the entity definition of selected entity_name or jumps to an instance definition of the entity.
Double-click on an entity (after double-clicking on an architecture)	Jumps to the architecture that was previously double-clicked.
Right-click on a signal name or anywhere in the Source view	Displays a CSM or Source view menu.
Position the mouse cursor on any signal name.	Displays ToolTip with the current value.

Working with the Source Code

This section describes how to use the Source view to examine the source code while debugging it. It allows you to expand and collapse required portions of the code, display line attributes for specific lines, and edit the source code using a text editor.

Expanding and Collapsing Source Code View

To expand or collapse the source code view

1. Click  in the Line Attribute area, or right-click and select **Expand Source** to view code that is folded.
2. Click  in the Line Attribute area, or right-click and select **Collapse Source** to hide code.

Displaying Include File as Hyperlink

The include file in your design is now shown as a hyperlink in the DVE Source view. You can click the hyperlink to view the include file separately in the same Source view. The hyperlink display is enabled by default when you load your design in DVE.

However, you can choose to view the collapsed view using a preference option **Enable 'include file expansion in source code**.

Example

The following example code top.v contains two include files.

top.v

```
`include "mynand.v"
`include "mynor.v"
module top;
wire na,nb,ny,nra,nrb,nry;
norgate nor1(.nra(nra), .nrb(nrb), .nry(nry));
nandgate nand1(.na(na), .nb(nb), .ny(ny));
endmodule
```

mynand.v

```
`include "myand.v"
module nandgate (na,nb,ny);
input na,nb;
output ny;
wire a_r,b_r;
andgate and1(.a(na), .b(nb), .y(a_r));
assign ny=! (a_r);
endmodule
```

mynor.v

```
`include "myor.v"
module norgate (nra,nrb,nry);
input nra,nrb;
output nry;
wire a_r;
orgate or1(.a(nra), .b(nrb), .y(a_r));
assign nry=! (a_r);
endmodule
```

myor.v

```
`include "myand.v"
module nandgate (na,nb,ny);
input na,nb;
output ny;
wire a_r,b_r;
```

```

andgate and1(.a(na), .b(nb), .y(a_r));
assign ny=!(a_r);

```

myand.v

```

module andgate (a,b,y);
input a,b;
output y;

assign y=a&b;
endmodule

```

Steps to compile the example code

```

% vcs -debug_all top.v +incdir+./
% ./simv -gui&

```

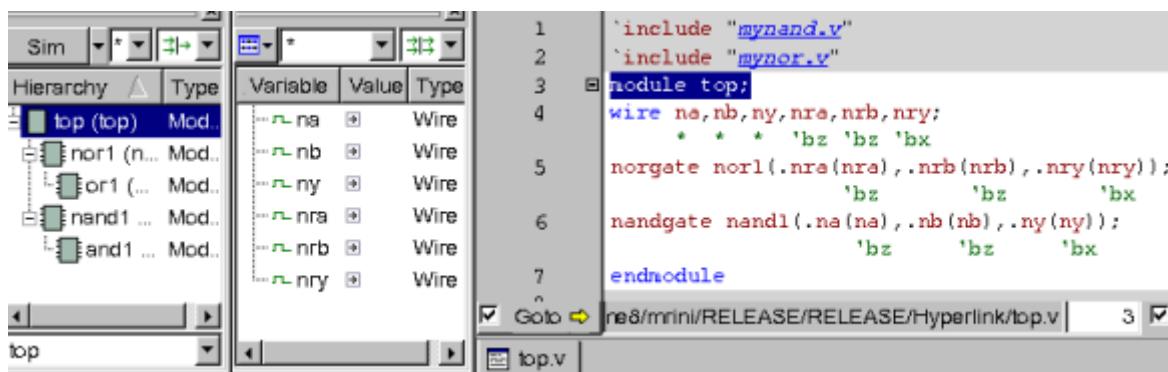
To view the include file as a hyperlink

1. Compile the design file and open in DVE.

The design is loaded in DVE.

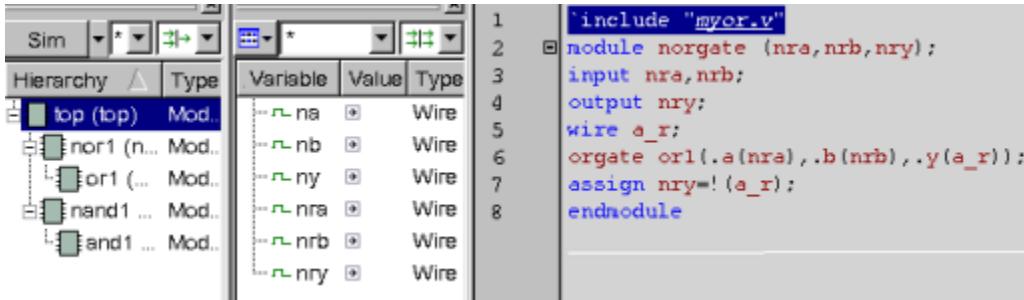
2. Select the scope in the Hierarchy pane.

The source file is displayed in the Source view with the include files as hyperlinks.



3. Click any of the hyperlinks, say mynor.v, in the Source view.

The include file is displayed in the Source view.



The screenshot shows the ModelSim interface with the 'Sim' tab selected. On the left, the Hierarchy viewer displays a tree structure of modules: top (top), nor1 (n...), or1 (...), nand1 ..., and1 In the center, there's a variable viewer showing columns for Variable, Value, and Type. On the right, the Source View window is open, displaying the following Verilog code:

```
1 'include "myor.v"
2 module norgate (nra,nrb,ny);
3   input nra,nrb;
4   output ny;
5   wire a_r;
6   orgate orl(.a(nra),.b(nrb),.y(a_r));
7   assign ny=~(a_r);
8 endmodule
```

4. From the **Scope** menu, use the **Back** or **Forward** options to move back or forward in the source code.

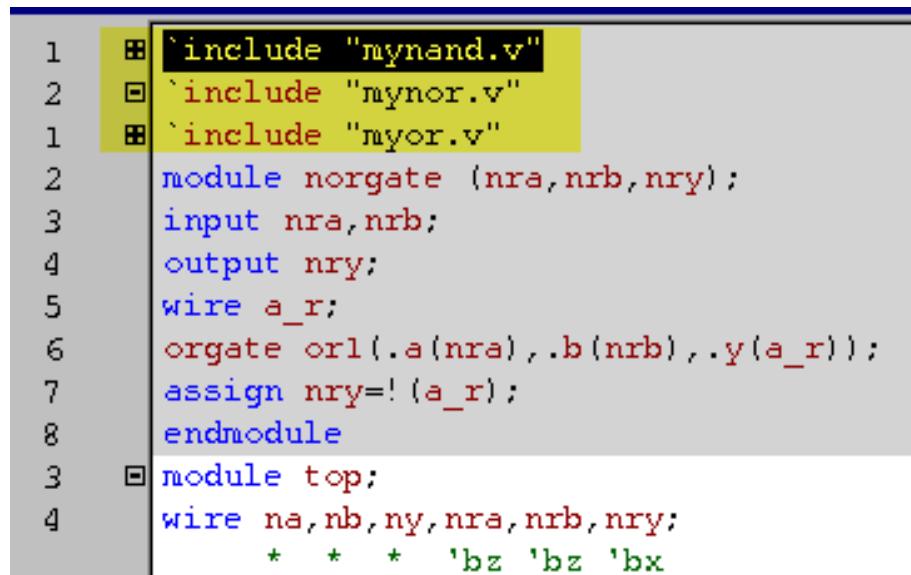
To view the include file as expanded

1. Select **Edit > Preferences**.

The Applications Preferences dialog box opens.

2. Click the **Source View** category and select the check box **Enable 'include file expansion in source code**.

The hyperlinks in the include files are removed and you can expand the files in the Source view.



A screenshot of a Verilog source code editor. The code is as follows:

```
1 `include "mynand.v"
2 `include "mynor.v"
1 `include "myor.v"
2 module norgate (nra,nrb,nry);
3   input nra,nrb;
4   output nry;
5   wire a_r;
6   orgate or1(.a(nra),.b(nrb),.y(a_r));
7   assign nry=! (a_r);
8 endmodule
3 module top;
4   wire na,nb,ny,nra,nrb,nry;
      * * * 'bz 'bz 'bx
```

The first three lines of code, which are includes, are highlighted with a yellow background. The entire code block is enclosed in a light gray border.

Editing Source Code

To edit source code

1. Select the text editor by setting the \$EDITOR environment variable.

```
%>setenv EDITOR vi [OR]
```

2. Select **Edit > Preferences**, select **Source view**, choose the editor you prefer and save from the editor pull-down menu.

The default editor is Vi.

3. In the source code area, right-click and select **Edit Source** or **Edit Parent** to open the source code in the default editor and edit the same.

Selecting and Copying Text to the Clipboard

You can select some or all text displayed in a Source view, and copy it to your clipboard.

To select all text or copy text in a Source view

1. Drag your mouse across the text to select a portion of text in the Source view.

DVE highlights the selected text.

2. Right-click and select **Copy** from the CSM.

You can paste this text in any text area or in the source editor.

Color-coding the Source File

You can set the background color and distinguish the active and inactive scopes in the DVE Source view. For an inactive scope, the annotations are not displayed.

The scopes in the Source view can be distinguished as follows:

- Active scope — Default background color is white, color is configurable.
- Inactive scope — Default background color is gray, color is configurable.

The code that is conditionally compiled is highlighted. The uncompiled code is shown in plain text without any syntax highlighting, as an inactive scope. For example,

```
117
118 'ifdef LOUD
119     initial
120         forever @(negedge test.risc1.fetch)
121             if (test.risc1.ireg[8:6] == 3'h7 )
122                 $display(test.risc1.mem1.memory[5'h1a]);
123 'endif
124
125 endmodule
```

In this example, since LOUD is not defined at compile time, it is not color-coded.

You can select the background color for the active and inactive scopes from the Application Preferences dialog box.

To set preferences for background color of the scope

1. Select **Edit > Preferences**.

The Application Preferences dialog box opens.

2. Select **Source Color** under the **Source view** category.

The options are displayed in the right pane.

3. Select the colors for the active and inactive scopes from the options **Background for active scope** and **Background for inactive scope**.

The active and inactive scopes are shown in the chosen color in the Source view.

```

1  `define cycle 5
2
3
4  `ifndef NODESIGN
5  module designA(clk,a,b);
6  `endif // NODESIGN
7
8
9  `ifndef NOTESTBENCH
10 module testbench;
11 reg clk;
12 reg [4:0] a;
13 reg b;
14
15 designA d0(clk,a,b);
16
17 initial
18 begin
19 #50;@(posedge clk) a = 5'b11111;
20 #20;@(posedge clk) b = ~b ;
21 #30;@(posedge clk) a = 5'b10101 ;
22
23

```

Inactive Scope

Active Scope

Goto ASE/SVA/operator_preced_and_or_delay1.v 67 Reuse

operatorpreced_and...

Note:

- Color-coding is not supported for dynamic scopes.
- If you select signals defined in the inactive scopes to perform some operation, such as “Add to Waves”, a dialog box lists all the scopes. You can select one or multiple scopes to perform the operation on the signals.

Setting Desired Color for Inactive 'ifdef `else Code in DVE

You can set the desired color for inactive 'ifdef `else code. The default color of this code is light gray, as shown in [Figure 4-2](#).

Usage Example

Consider the following example testcase test.v:

```
`define path top
module top;

    reg in1 , in2,clk;
    wire out1, out2, out3;
    wire macro_w;

    and1 inst (in1, in2, clk, out1);

    initial
        begin
            clk = 0;
            `ifdef OPP
                in1 = 0;
                in2 = 0;
                #5 in2 = 1;
                #5 in1 = 1;
                in2 = 0;
                in1 = 0;
            `else
                in1 = 1;
                in2 = 1;
                #5 in2 = 0;
                #5 in1 = 0;
                in2 = 1;
                in1 = 1;
            `endif
            #25;
            in2 = 1;
            #5;
            $finish;
        end
    endmodule

module and1(input a,b,c,output d);
    and as(a,b,c,d);
endmodule
```

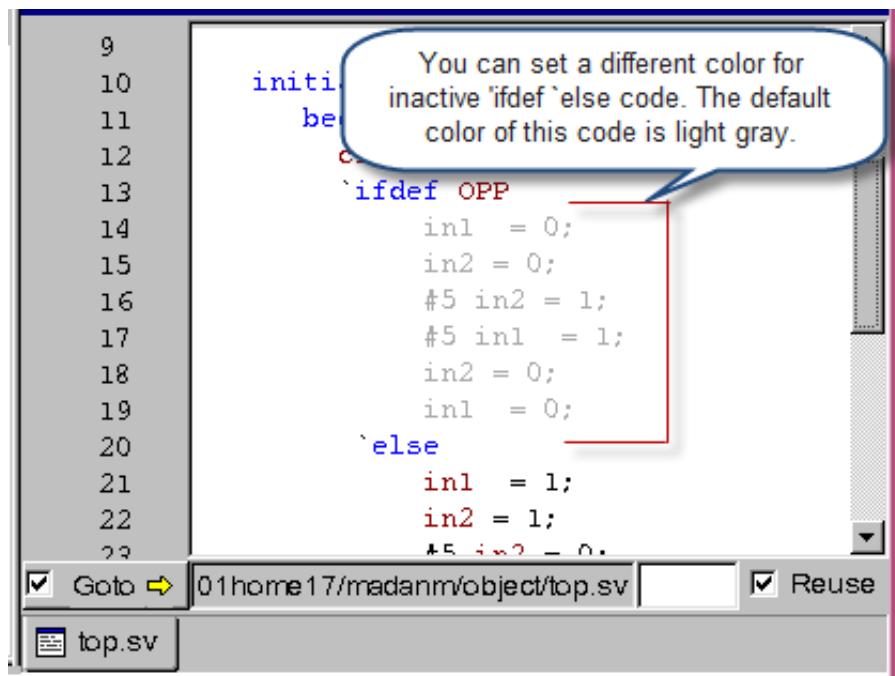
Compile the test.v example:

```
% vcs -debug_all test.v
```

Invoke the DVE GUI:

```
% ./simv -gui&
```

Figure 4-2 Default color of inactive 'ifdef `else code



To change the color of inactive 'ifdef `else code:

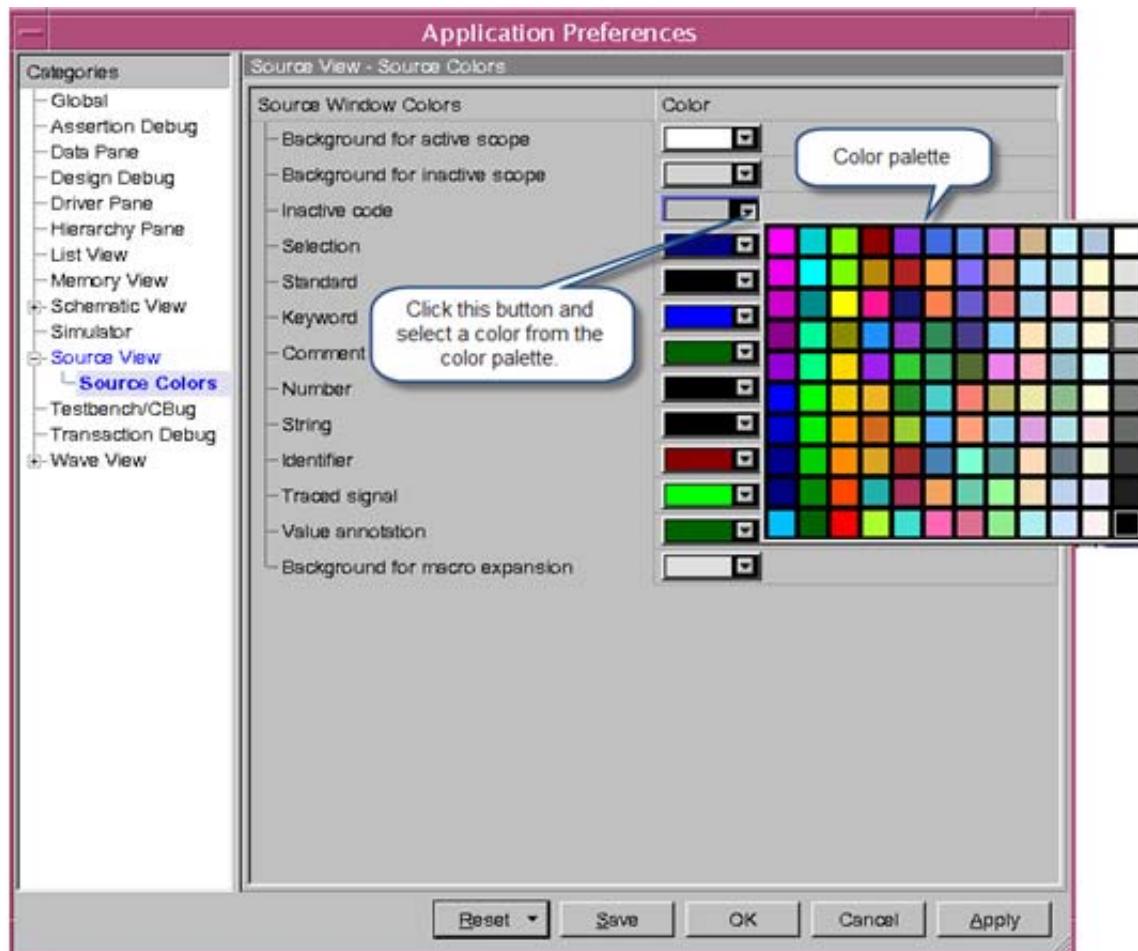
1. Select **Edit > Preferences**.

The Applications Preferences dialog box appears.

2. In the **Source View > Source Colors** category, click the **Inactive Code** drop-down, as shown in [Figure 4-3](#).

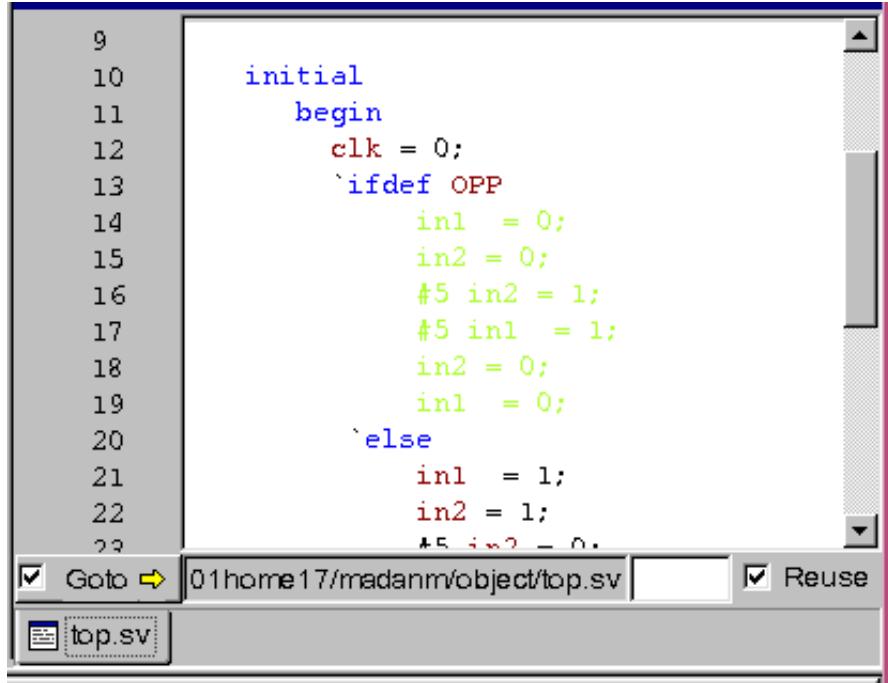
3. Select a color from the color palette and click **Apply**.
4. Click **OK**.

Figure 4-3 Selecting a color from the color palette



For example, if you select green color from the palette, then the color of the inactive 'ifdef `else code changes to green, as shown in [Figure 4-4](#).

Figure 4-4 Changing the color of inactive 'ifdef `else code



The screenshot shows a Source View window with the following Verilog code:

```
9
10    initial
11        begin
12            clk = 0;
13            `ifndef OPP
14                in1 = 0;
15                in2 = 0;
16                #5 in2 = 1;
17                #5 in1 = 1;
18                in2 = 0;
19                in1 = 0;
20            `else
21                in1 = 1;
22                in2 = 1;
23                #5 in1 = 0;
```

The code uses color coding: blue for keywords like `initial` and `begin`, red for `ifndef` and `else`, green for variable names like `clk`, `in1`, and `in2`, and yellow for comments like `#5`. The `ifndef` block is inactive, indicated by the red color.

At the bottom of the window, there is a toolbar with a 'Goto' button and a 'Reuse' checkbox, and a status bar showing the file path '01home17/madanm/object/top.sv'.

Navigating the Design from the Source View

Use the Source view to navigate through the design and view results in other DVE windows by dragging and dropping signals, scopes, and objects.

To navigate the design from the Source view

1. Select the required object, signal or scope, in the Source view.
The text is highlighted.
2. Right-click and add to the desired view from the CSM.

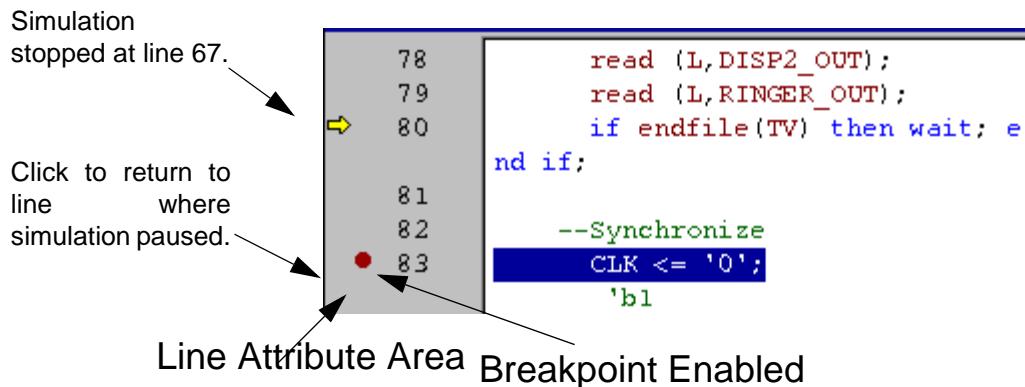
You can view the object from the source code in the Wave view, List view, add to Groups, or add to the Watch pane.

Navigating Code in Interactive Simulation

Use the line attribute area to control line breakpoints when running interactive simulation. To display line attributes, right-click in the line attribute area, then select **Line Number**.

When you run a simulation interactively, the line where the simulation stopped is marked by a yellow arrow in the Source view. However, you can search and review any code in the design during a pause in the simulation. You can return to the line where the simulation paused by clicking the yellow arrow at the bottom of the Source view as shown in the following illustration.

Figure 4-5 Navigating an interactive simulation



Setting Breakpoints in Interactive Simulation

You can set breakpoints to stop the simulation. Note the following points regarding breakpoints:

- Line breakpoints execute each time a specified line is reached during simulation (see the section Displaying Line Attributes and Managing Breakpoints from the dialog box for more information) about line breakpoints. You can also specify an instance to have the tool stop only at the line in the specified instance.
- Time breakpoints stop at a specified absolute or relative time in the simulation.
- Signal breakpoints trigger when a specified signal rises, falls, or changes.
- Assertion breakpoints stop at a specified assertion event.
- Task/Function breakpoints stop at the specified task or function.

To set and delete a breakpoint from the CSM

1. Click in the line attributes area of the Source view next to an executable line.

A solid red circle indicates that a line breakpoint is set.

Note:

A line breakpoint can only be set on an executable line. If a line is not executable, no breakpoint will be set when you click next to it.

OR

Right-click in the attributes area of the Source view, then select **Set Breakpoint**.

A plus sign (+) appears when you set more than one breakpoint on one executable line.

2. Select the solid red breakpoint circle to disable it.

- The solid red circle changes to an empty red circle.
3. Right-click on an enabled or disabled breakpoint, then select **Delete Breakpoint** or **Delete All Breakpoints**.

The red circle disappears indicating that the breakpoint is deleted. You can also delete a breakpoint by double-clicking on the solid red circle/clicking the empty red circle.

The following table describes the breakpoint icons:

Breakpoint Icon	Description
	Denotes a line breakpoint was set on this line, and it is enabled.
	Denotes a line breakpoint that was set more than once on the same line, and it is enabled.
	Appears when you disable one breakpoint on an executable line.
	Denotes a disabled line breakpoint.

Managing Breakpoints

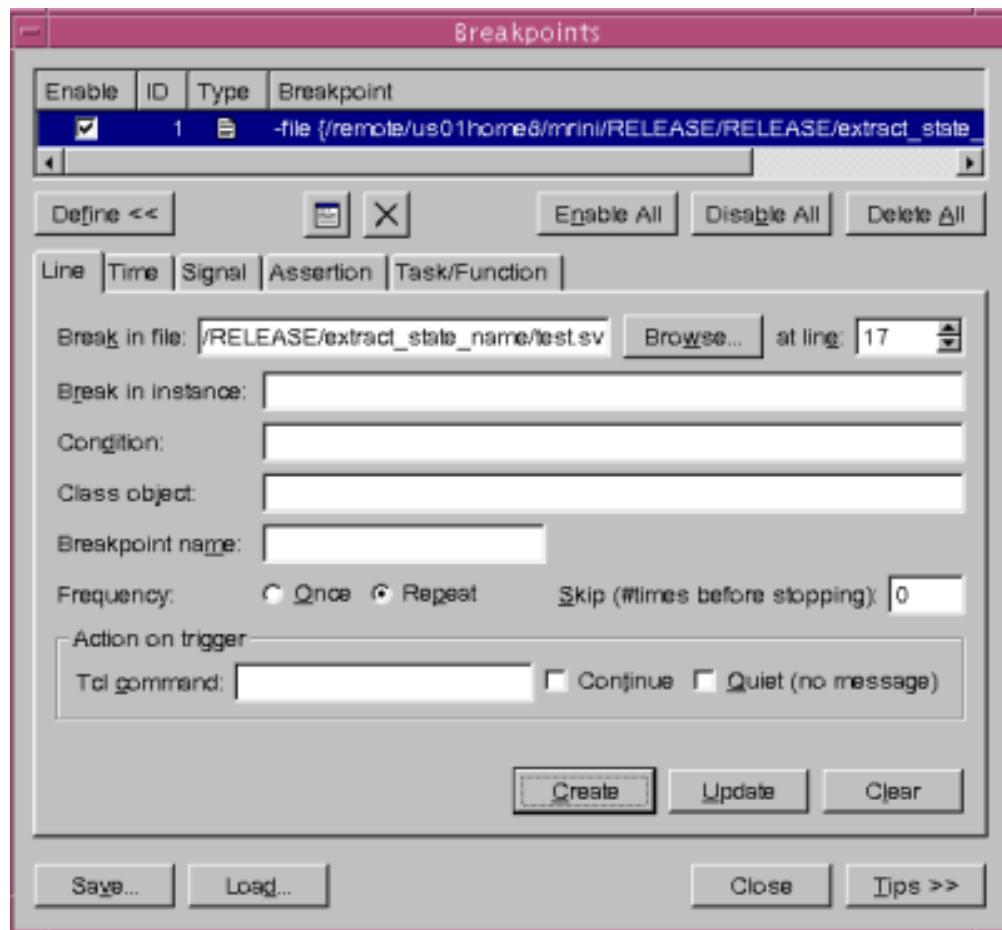
You can manage all types of breakpoints in an interactive simulation from the Breakpoints dialog box.

To create or update breakpoints using the Breakpoints dialog box

1. Select a line in the Source view line attribute area, right-click and select **Properties**.

To create a class object breakpoint for a task or function, select one method in the Hierarchy pane or Stack pane and then right-click and select **Set Breakpoint**.

The Breakpoints dialog box appears.



2. Click **Define** to display the breakpoint creation tabs.
3. Select the **Line** tab and enter the following information:
 - Break in file — Enter the file name or browse to the file where you want to create the breakpoint.
 - At line — Enter the line number for the breakpoint.

- Break in instance — Enter the instance where the breakpoint will fire.
4. Select the **Time** tab and enter the following information:
 - Select **Absolute** or **Relative** time reference, then enter the time to set the breakpoint.
 5. Select the **Signal** tab and enter the following information:
 - Enter the desired signal in the Break on signal text.
 - Select Any, Rising, or Falling Edge to define the breakpoint event.
 6. Select the **Assertion** tab and enter the following information:
 - Enter the full path to the Assert in the Break on Assertion text field.
 - Select an event type to trigger the breakpoint from any, start, end, failure, or success.
 7. Select the **Task/Function** tab and enter the following information:
 - Enter the full path to the task or function in the Break in Task/ Function field.
 8. (Optional) Enter a condition for VHDL objects to be met for the breakpoint to fire.

Note:
Condition is not supported for Verilog objects.
 9. Enter the class object in the **Class Object** field if you want to create a breakpoint for a class object.
 10. Select the frequency. Select **Once** if you want to fire the breakpoint once, else select **Repeat**.

11. Provide a name for the breakpoint in the **Name** field.
12. Define Tcl commands to execute when breakpoint triggers in the **Command** field.
13. Enter the skip time before stopping in the **Skip** field. Select the **Continue** check box to prevent breakpoint to stop. Selecting the **Quiet** check box will not print any error message when breakpoint triggers.

14. Click **Create**.

The breakpoint is created and appears in the Breakpoint list box.

15. Select a breakpoint by clicking on it from the Breakpoint list box and click **Save**.

The Save Breakpoint dialog box appears.

16. Provide a file name in the File Name field and click **Save**.

The breakpoint is saved in the file with a .tcl extension. Repeat the steps to save more breakpoints.

17. Select the breakpoint from the Breakpoint list box, change the settings and click **Update**.

The breakpoint is updated.

To load the breakpoints

1. In the Source view line attribute area, right-click and select **Properties**.

The Breakpoints dialog box appears.

2. Click **Load**.

The Load Breakpoints dialog box appears.

3. Select the tcl file in which the breakpoints are saved and click **Open**.

The breakpoints are loaded in DVE and you can view the red circles against the line that has the breakpoint in the Source view. Loading breakpoint doesn't replace the existing breakpoints, rather it adds the breakpoint from the file in the existing list.

Setting Breakpoints in a Class Object

You can now set breakpoints in the individual class objects without modifying the contents of the class. Use any of the following methods to set breakpoint on a class object:

- Specifying the breakpoint using -object UCLI command.
- Specifying the breakpoint in the DVE Breakpoint dialog box.

The breakpoints set on the class objects are not stored in the DVE Session file.

Note:

Usage of -object with System-C code is not supported.

Example

```
program p;

    class A;

        int id;
        task my_method();
            $display("Inside A::my_method(%0d)", id);
        endtask
        function new (int i);
            id = i;
        endfunction
```

```

endclass

initial
begin

    A c0, c1; // First create all our objects
    c0 = new(0); // id = 0
    c1 = new(1); // id = 1
    $stop; // When stopped here, enter these commands:
            // stop -file obp_doc.v -line 6 -object c1
            // (stops in method of c1 only, not c0)
            // Now call all the objects' method to test if
            //the simulation stops
    c0.my_method();
    $display("BP 1 should trigger after next stmt");
    c1.my_method();

end

endprogram

```

The following figures show how the breakpoint is triggered. First at \$stop and then after entering the UCLI command and running:

```
stop -file obp_doc.v -line 6 -object c1
```

Figure 4-6 Class Object Breakpoint Set

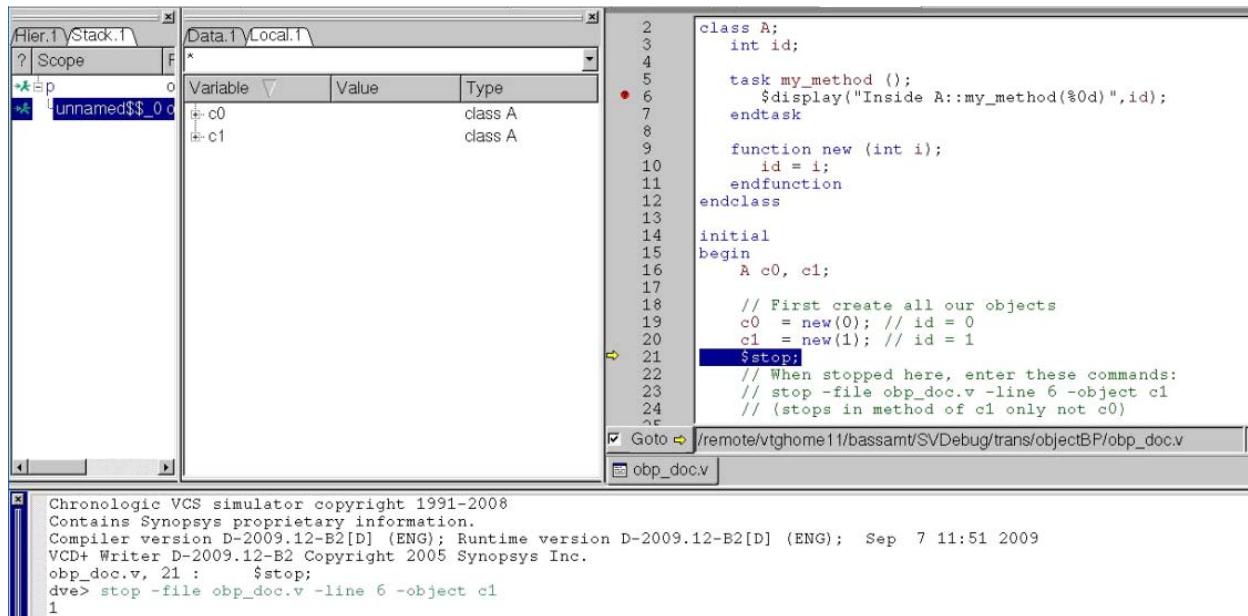
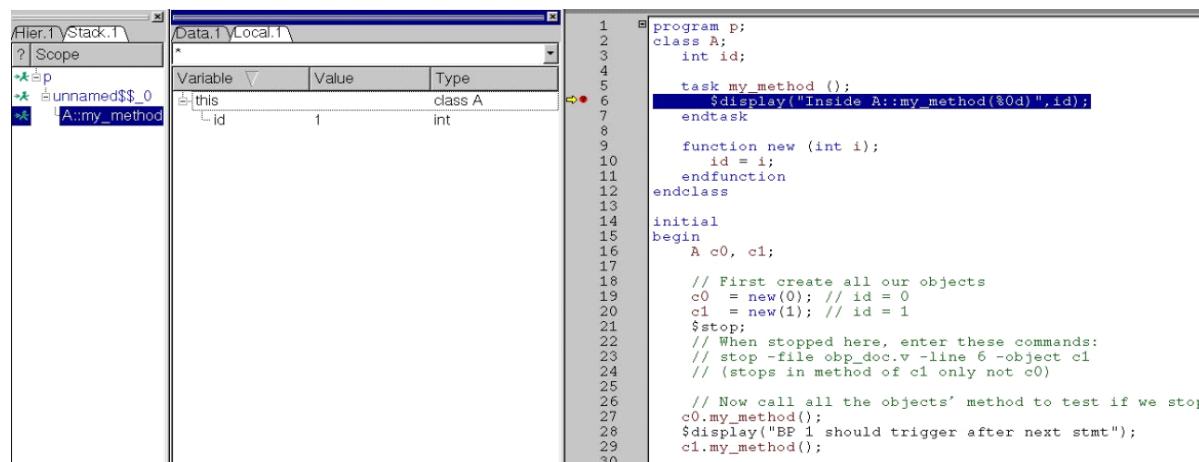


Figure 4-7 Class Object Breakpoint Hit



Using the Source View

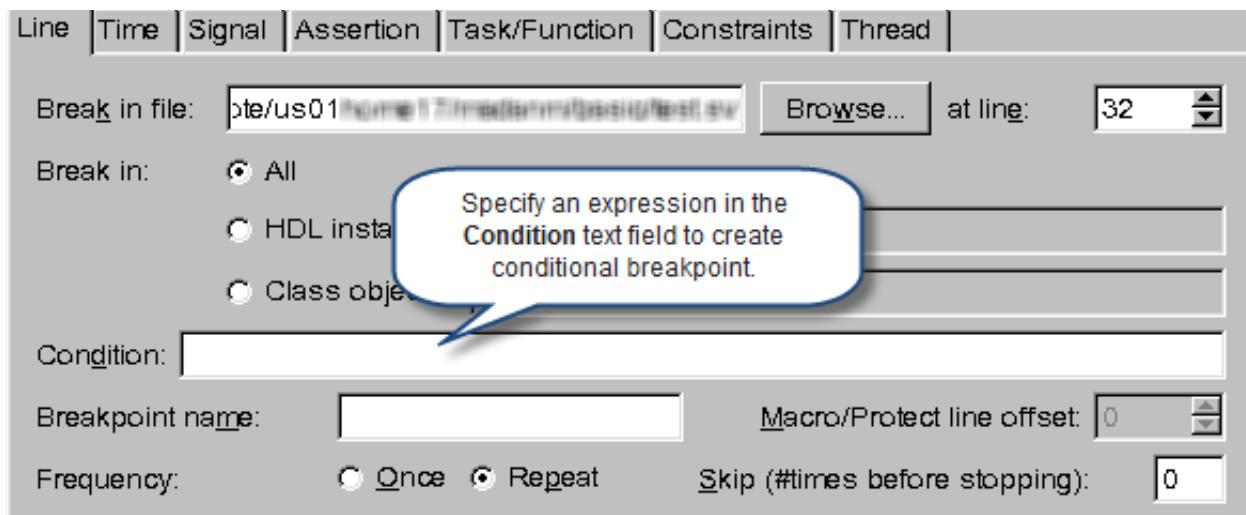
Creating Conditional Breakpoints

You can use the `-condition <expression>` option (see [Table 4-1](#)) or the DVE Breakpoints dialog box (see [Figure 4-8](#)), to create conditional breakpoint.

Table 4-1 Command to create conditional breakpoint

UCLI Command	Description
<code>stop -file <file> -line <lineno> -condition <expression></code>	Stops the execution at the specified conditional expression.

Figure 4-8 Creating conditional breakpoint using DVE Breakpoints dialog



Using Object IDs in Conditional Expression

You can specify object IDs in conditional expression. DVE displays a unique object ID for every class instance in the following format:

`<classname>@<instance number>`

Where,

- `<classname>` is the name of a class.

- <instance number> is the instance number of <classname>. Instance number is an unsigned number.

For more information on Object ID, refer to “[Viewing Object Identifier Values](#)” section.

Key points to note:

- You can use Object ID in the conditional expressions only in Line and Task/Function breakpoints.
- To set conditional breakpoint in combination with the -line option, you must compile your designs with -debug_all.

A simple class object will look like C @1. However, the object ID will have special characters like “::”, “#”, or “\$”, in case of scenarios involving nested classes, parameterized classes, packages and so on. Hence these ObjectID names must be treated as escaped identifiers. You should use a preceding back slash (\) and terminate the <class_name> with a white space followed by @<unsigned_number>, as shown below, while using these object IDs in the stop command or Breakpoints dialog.

```
\<package_name>::<class_name> @<unsigned_number>
```

Examples:

```
\C @1 // object ID of a simple class object
```

```
\C1::C2 @2 //object ID of a nested class
```

```
\Base#(2) @1 // object ID of a parameterized class
```

Class Defined within a Package

Following is the syntax of object ID, if a class is defined within a package:

```
\<package_name>::<class_name> @<unsigned_number>
```

Example:

```
stop -file dynamic.sv -line 19 -condition {inst  
==\pkg::C @1}
```

Note:

The following operators are supported with Object IDs:

`==, !=, =, >=, <=`

Arithmetic operators are not supported.

Usage Example

[Example 4-1](#) illustrates the usage of Object IDs in conditional expression.

Example 4-1 Using Object IDs in Conditional Expression

top.sv

```
package pkg;  
class Base#(int size=5);  
    bit [size:0] a;  
    task disp();  
        $display("Package:Size of the vector a is $d", $size(a));  
    endtask  
endclass  
  
class C1;  
    class C2;  
        function foo;  
            $display("Package Nested class");  
        endfunction  
    endclass  
endclass
```

```

        endfunction
    endclass
    C2 c2 = new();
endclass

class C;
    int a=1;
    task main(int x = 0);
        begin
            $display("Package:Message");
        end
    endtask
endclass
endpackage // pkg

program class_scenario;

import pkg::*;

pkg::C1 pkg_c1 = new();
pkg::C1::C2 pkg_cc2=new();
pkg::C pkg_inst = new;
pkg::Base#(5) pkg_B3=new();

initial
begin //: A1
    pkg_inst.a=9;
    pkg_inst.main();
    pkg_c1.c2.foo();
    pkg_cc2.foo();
    pkg_B3.disp();
    #1 $finish;
end // : A1
endprogram

```

Compile the `top.sv` example shown in [Example 4-1](#):

```
% vcs -debug_all -sverilog top.sv
```

Invoke the DVE GUI:

```
% simv -gui&
```

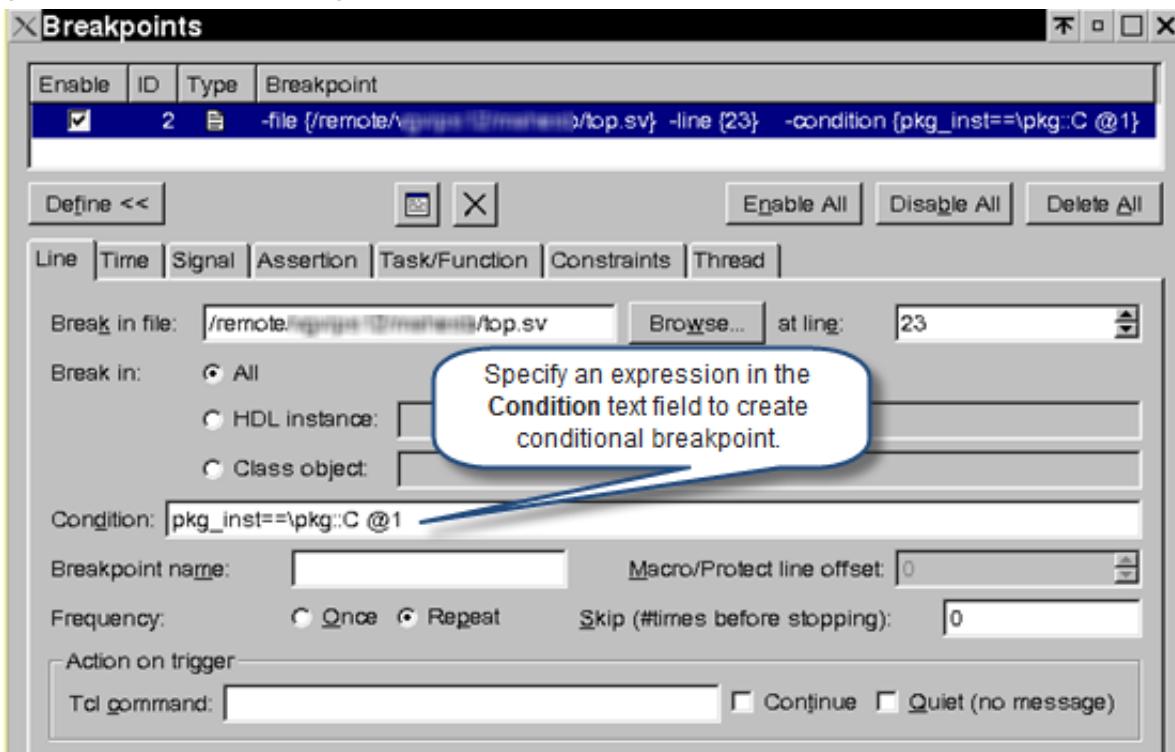
Enter the following commands in the DVE command-line:

```
dve>stop -line 23 -file top.sv -cond {pkg_inst==\pkg::C @1}  
dve>run
```

Or,

Set the conditional breakpoint using Breakpoints dialog, as shown in [Figure 4-9](#).

Figure 4-9 Specifying an expression in the Condition field



The simulation stops at line 23, as shown in [Figure 4-10](#).

Figure 4-10 Breakpoint hit at a specified condition

The screenshot shows the DVE Source View window. The code is as follows:

```
18 class C;
19     int a=1;
20     task main(int x = 0);
21         begin
22             $display("Package:Message");
23         end
24     endtask
25 endclass
26 endpackage // pkg
27 /////////////////////////////////// Program /////
28 ///////////////////////////////////
29 program class_scenario;
30
31 import pkg::*;
32     pkg::C1 pkg_c1 = new();
33     pkg::C1::C2 pkg_cc2=new();
34     pkg::C pkg_inst = new();
35     pkg::Base#(5) pkg_B3=new();
```

A red dot indicates a breakpoint at line 22. The status bar at the bottom shows "Goto /remote/vgvips.../top.sv" and "29".

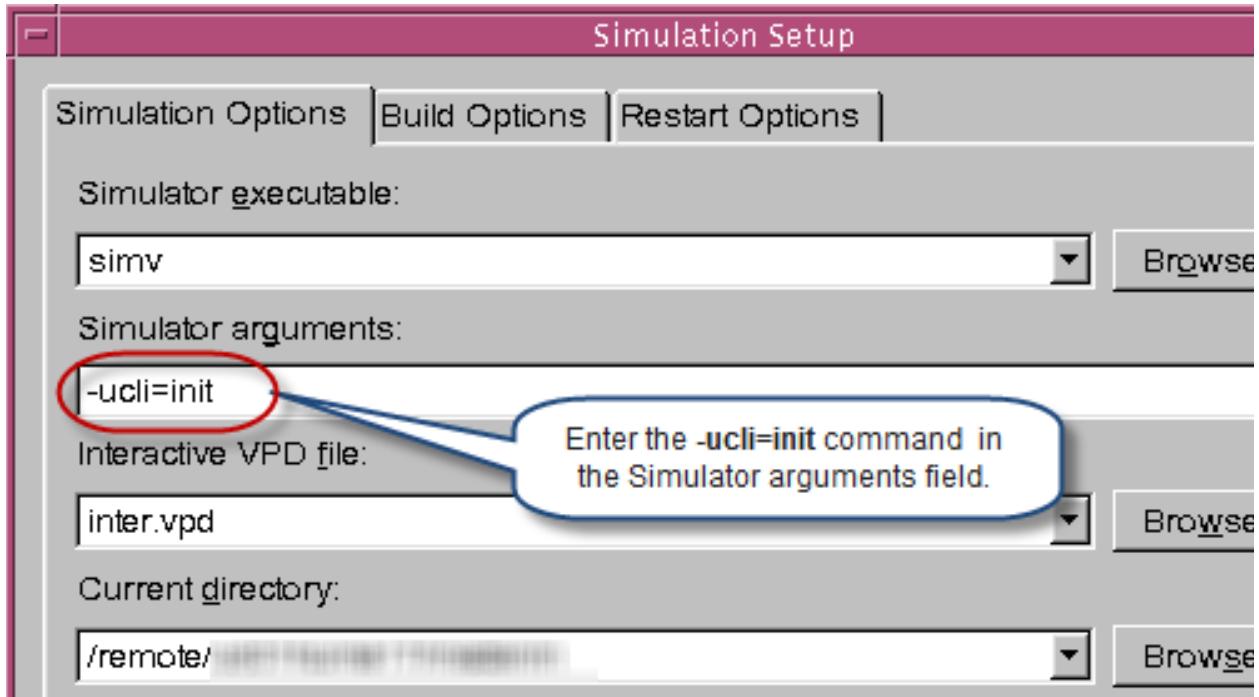
Debugging During Initialization of SystemVerilog Static Functions and Tasks

Enabling Static Debug in DVE

You can use one of the following three methods to enable static debug in DVE:

- Using the `simv -ucli=init -gui` command
- Set environment variable `ENABLE_SVINIT_DEBUG` and start DVE using the `simv -gui` command
- Loading `simv` using the Simulation Setup dialog, as shown in the following figure:

Figure 4-11 Loading simv Using Simulation Setup Dialog



Debugging Static Code

Consider the following example code:

test.sv

```
module top ();
reg a;
class bp;
    static int a = do_int();
    static function int do_int();
        $display("TOTOT");
        return 3;
    endfunction
endclass : bp
bp my_bp=new;
initial
begin
    a = 1;
```

```
#10 $display("End %d....",my_bp.a);
end
endmodule
```

Perform the following steps to debug static code:

1. Compile **test.sv**

```
% vcs -sverilog -debug_all test.sv
```

2. Invoke the DVE GUI

```
% simv -ucli=init -gui
```

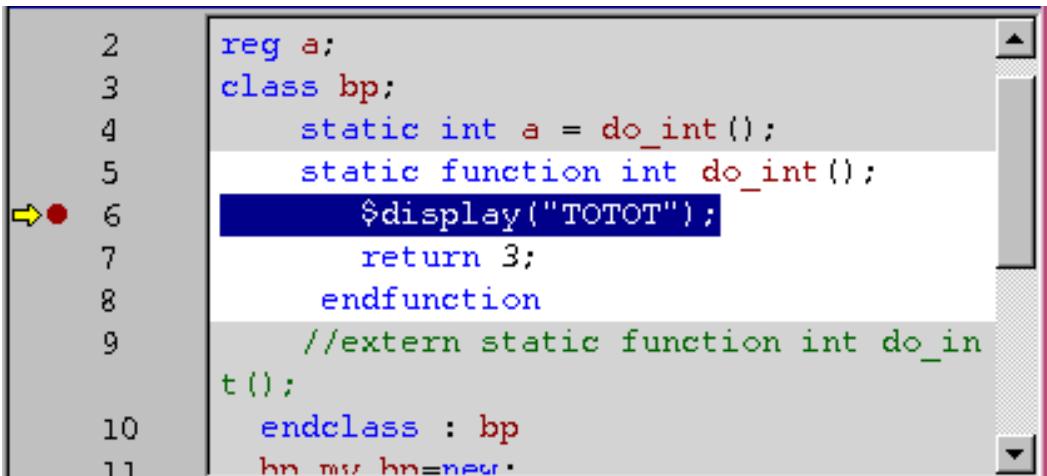
3. Set breakpoint in static function, as shown in the following figure, to debug the code:

Figure 4-12 Debugging Static Code

```
1  module top ();
2    reg a;
3    class bp;
4      static int a = do_int();
5      static function int do_int();
6        $display("TOTOT");
7        return 3;
8      endfunction
9      //extern static function int do_in
10     t();
11   endclass
```

4. Run the simulation. The simulator stops at the breakpoint, as shown in the following figure:

Figure 4-13 Debugging Static Code



The screenshot shows a source code editor window for SystemVerilog. The code is as follows:

```
2 reg a;
3 class bp;
4     static int a = do_int();
5     static function int do_int();
6         $display("TOTOT");
7         return 3;
8     endfunction
9     //extern static function int do_in
10    t();
11 endclass : bp
12
13
```

A yellow arrow and a red dot indicate a breakpoint is set on line 6. The line containing the breakpoint is highlighted in blue.

The DVE prompt (`dve>`) remains unchanged. You can use the following tcl command to check whether simulation is in init phase (static debug). It will return 1 if simulation is in init phase.

```
gui_check_init_debug_state
```

Features Disabled in Initialization Phase

All features that are not supported in UCLI during initialization phase (see “*Debugging During Initialization of SystemVerilog Static Functions and Tasks*” section in *UCLI User Guide*) are also not supported in DVE. Menu items for the following operations in DVE are disabled in initialization phase:

- Add signals/scopes into waveform/list window
- Dump
- Force
- Interactive Rewind
- save/restore

- C/C++ Debugging

Note:

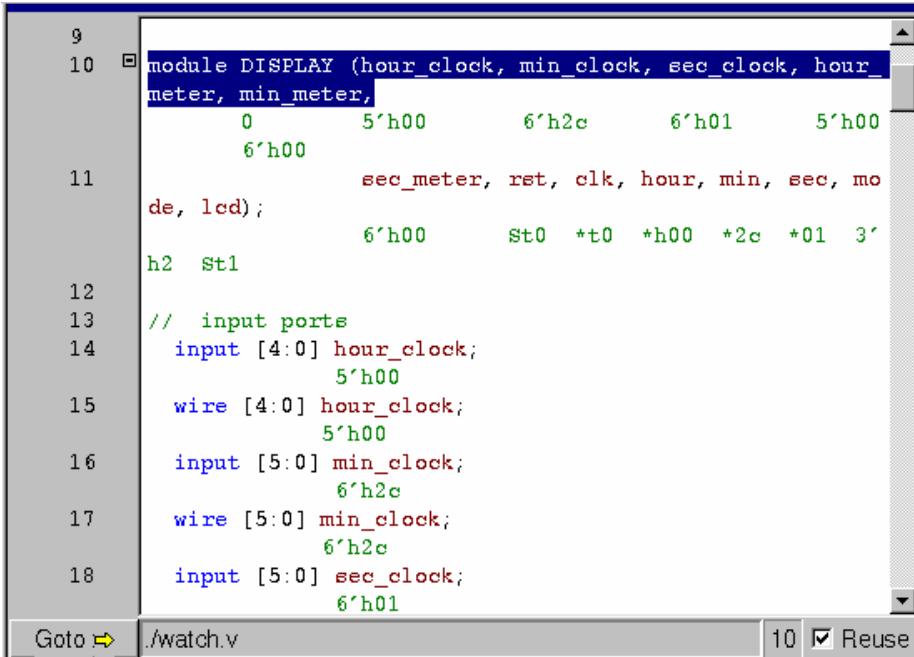
You can execute `run 0`, `run -nba`, or `run -delta` to step out of static debug. Once the simulation steps out of the init phase (static debug), all the above mentioned features are enabled.

Annotating Values

To enable value annotation for variables or signals in the Source view

1. Click the Annotate Values Icon  in the toolbar.
2. Select **Scope > Annotate Values**.
3. In the Source view, right-click and select **Annotate Values**.

The annotated values are displayed in the Source view.



A screenshot of a software interface showing a source code editor. The code is written in Verilog and defines a module named DISPLAY. The annotated values for variables are shown in red text. The annotated code is as follows:

```
9
10 module DISPLAY (hour_clock, min_clock, sec_clock, hour_
11 meter, min_meter,
12     0      5'h00      6'h2c      6'h01      5'h00
13     6'h00
14     sec_meter, rst, clk, hour, min, sec, mo
15     de, lcd);
16     6'h00      St0 *t0 *h00 *2c *01 3'
17     h2 St1
18 //  input ports
19     input [4:0] hour_clock;
20     5'h00
21     wire [4:0] hour_clock;
22     5'h00
23     input [5:0] min_clock;
24     6'h2c
25     wire [5:0] min_clock;
26     6'h2c
27     input [5:0] sec_clock;
28     6'h01
```

The annotated values are: hour_meter (0, 6'h00), min_meter (6'h00), sec_meter (6'h00), rst (St0), clk (*t0), hour (*h00), min (*2c), sec (*01), mo (3'), and lcd (h2). The annotations are placed directly next to their respective variable declarations.

If there is not enough space to show the values, the value is shown as * (asterisk character). You can see the exact value when you hover your mouse on the variable.

5

Using Wave View

The Wave view displays waveforms for signals, traced assertions, and signal comparison.

This chapter includes the following topics:

- “Viewing Waveform Information”
- “Using the Signal Pane”
- “Using the Wave View”
- “Viewing PLI, UCLI, and DVE Forces in Wave View”

For information about using the Wave view to view and debug assertions, see [Using the Assertion Pane](#).

Viewing Waveform Information

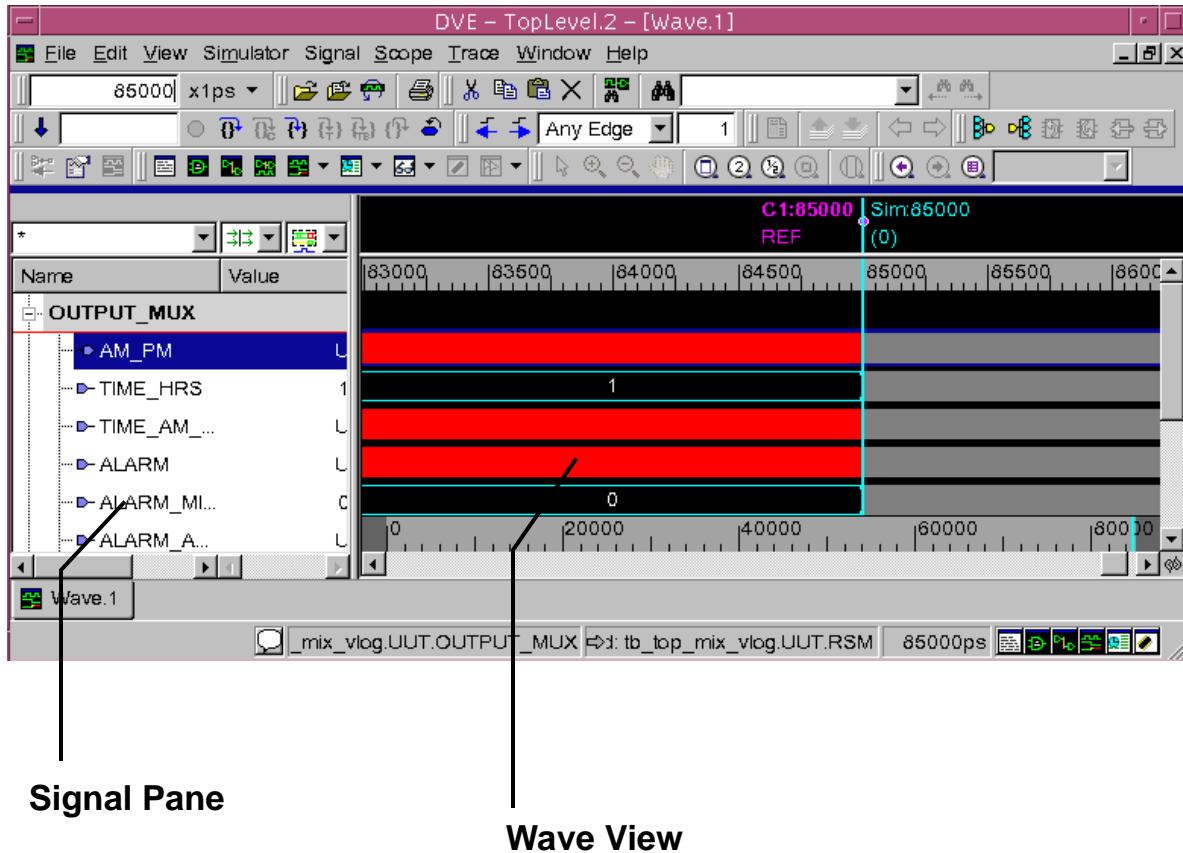
To view waveform information in the Wave view, set the target window and choose the waveform you want to view. You can customize how DVE displays the waveform by changing the settings in the Wave view.

Viewing a Waveform

To view waveform information for signals in the Wave view

1. Select a scope or object from the Hierarchy pane, Data pane, Source view, List view, Schematic view, or Assertion view.

2. Click the **Add to Waves** icon in the toolbar .



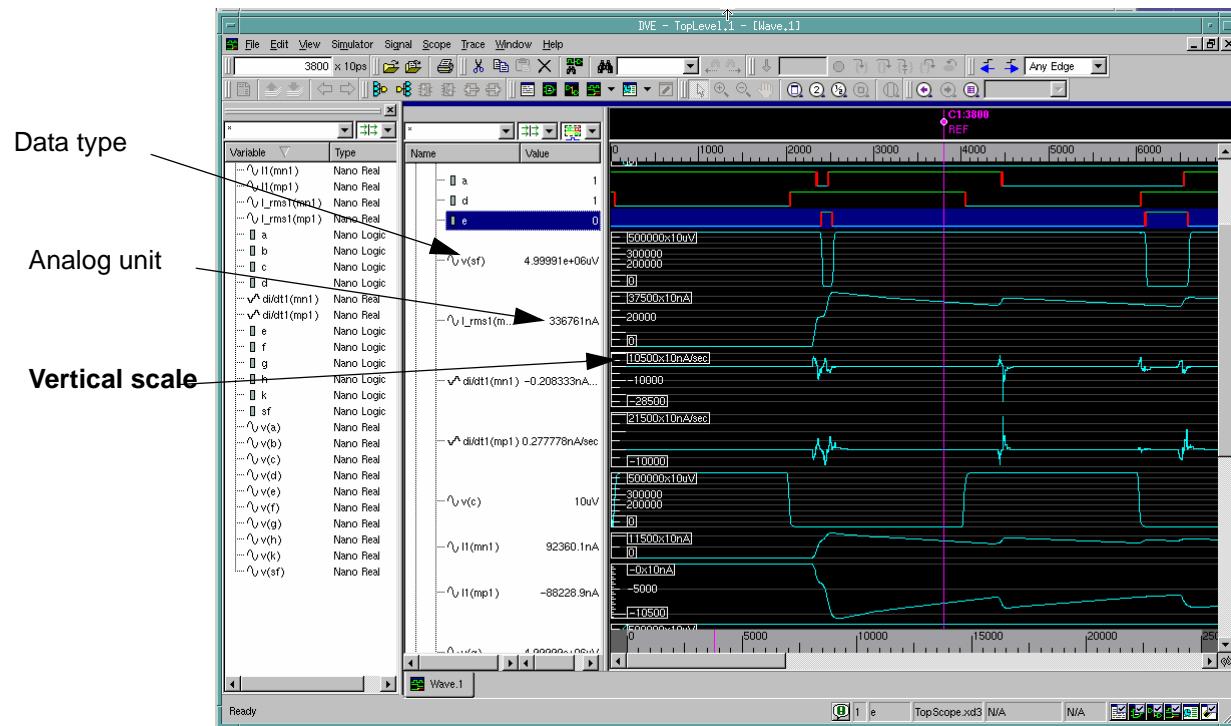
The selected signal is added to the recently used Wave view if it exists, else it is added to a new view.

You can also select a signal and use the CSM to add it to either a new Wave view or a recently used Wave view.

Viewing Nanosim Analog Signals

DVE supports display of Nanosim signals dumped to a VPD file. The Wave view displays the signals with units and resolution appended to the values. [Figure 5-1](#) shows the Wave view display of Nanosim data.

Figure 5-1 Displaying Nanosim Signals from a VPD File



For information about dumping and debugging Nanosim signals, see the *Discovery AMS: NanoSim-VCS User Guide* and the *Discovery AMS: NanoSim-VCS-MX User Guide*.

Setting the Simulation Time

To set the simulation time display in the waveform

1. Select **View > Go To Time**.

The Go To Time dialog box appears.



2. Enter a value and click **Apply** or **OK**.

The waveform display moves to the specified simulation time. The corresponding values will also be visible in the Source view and List view.

Using the Signal Pane

The Signal pane displays signals in groups:

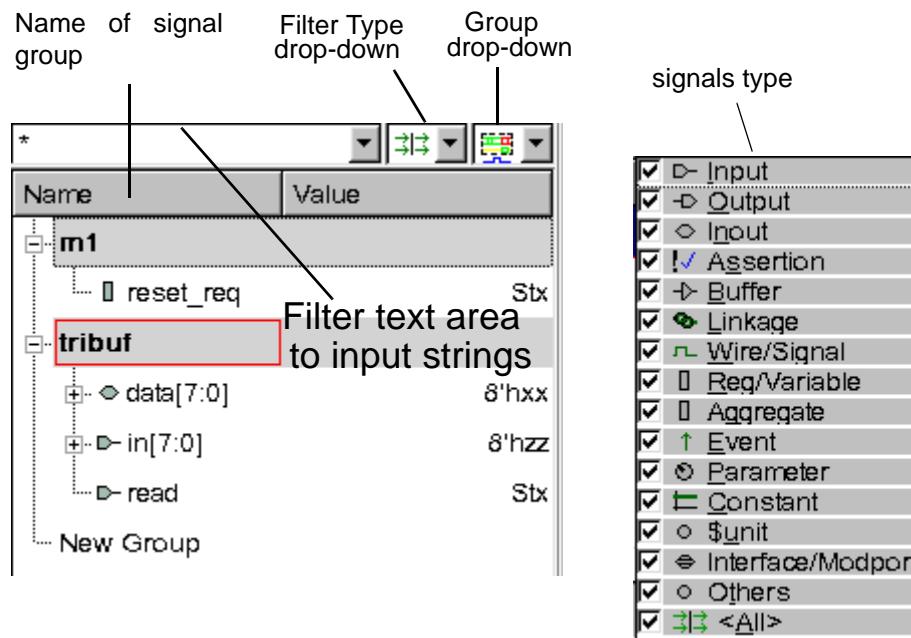
- Scalar signals have their value displayed in binary radix.
- Vector signals have their values displayed in hexadecimal radix.
- Integers, real numbers, and times are displayed in the floating point radix.
- Signed numbers are displayed in 2's compliment.

The Signal pane consists of:

- Name column — Displays signal names and their groups.
- Value column — Displays the value of signals at the simulation time selected by the C1 cursor (which is also the value in the TopLevel Window Time field).
- Filter text area — Allows you to input a string to filter items. You can use wildcard as your search string.
- Group filter drop-down — Displays the group name to filter the signals based on their group.
- Filter type drop-down — Displays the signals types to filter the signals. You can select or clear the checkbox beside each signal type to filter them.

See [Figure 5-2](#) for an example of the Signal pane.

Figure 5-2 The Signal Pane



Expanding Verilog Vectors, Integers, Time, and Real Numbers

To expand the Vector signals to their individual bits, click the plus icon to the left of the signal name.

After you expand the display, each bit is added to the Signal pane and waveforms for these bits are added to the Wave view.

DVE represents integers in 32 bits, so you can expand an integer in the Signal pane to display separate waveforms for each of these bits. Similarly, DVE represents the time data type with 64 bits, and you can expand a time to display a waveform for each of these 64 bits.

You cannot expand a real data type.

You can also expand assertions. Upon expanding an assertion, its children will include the assertion clock and the signals and events (or sequences and properties for SVA) that make up the assertion.

Adding Signal Dividers

A divider, inserted into a Signal Group, displays in every instance of that signal group when opened in Wave views. Dividers are saved in the session TCL file and are restored when the session is opened.

To separate signals in the Wave view, click **Signal > Insert Divider**. Dividers are added between signals.

There is no limit to number of dividers you can add between signals.

Renaming Signals

You can change the name of the signal and its bit range in the Wave view. This is useful when you want to give unique names to each signal so they can be efficiently compared with other signals or modified versions (using expressions) of the same signal.

Note the following guidelines while renaming the signals:

- The new name should be composed of a sequence of letters, digits, and underscore characters.
- Renaming only affects the current item (not all items of the same signal in the Wave view).

- The alias name (new signal name) does not support Tcl commands. The Tcl log uses the original signal name; therefore, you should use the original name to access a signal using a Tcl command.
- The new signal name is displayed only in the Wave view.
- All types of signals (Verilog/VHDL/SC/Analog and so on) are supported. Expressions, bus, time shift, and compare objects are not supported.

To rename the signal

1. Compile your design and open in DVE.

Select a scope or signal in the Hierarchy or Data pane, right-click and select **Add to Waves > New Wave view**.

2. Click on a signal that you want to rename and type the name.

OR

Right-click on a signal, select **Edit > Rename, and type a name**.

3. Press **Enter**.

The signal is renamed. This way, you can only change the signal name and not the range.

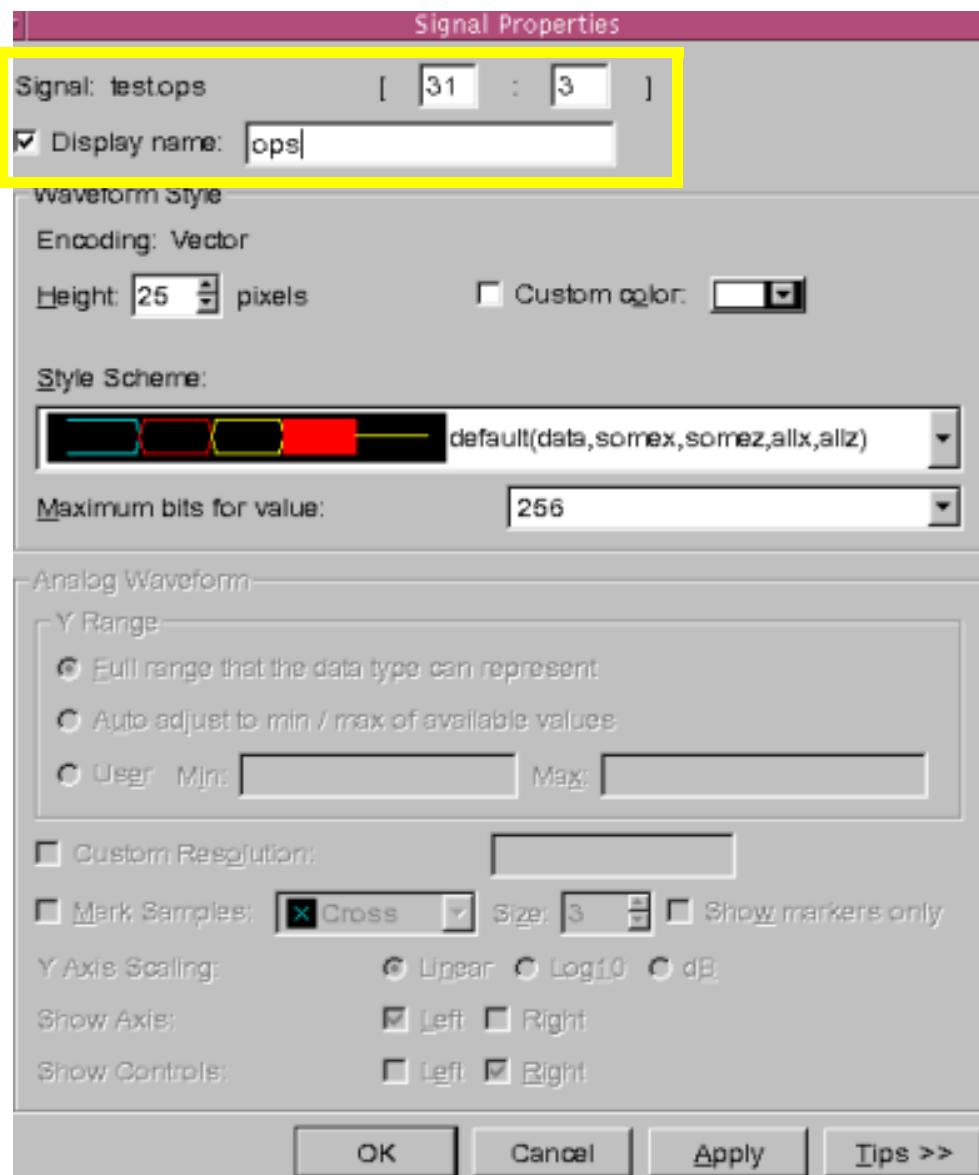
To change the signal name and range

1. Select a signal in the Signal pane.

Name	Value
Group 1	
obp[30:3]	0
cc[31:0]	3
bb[31:0]	2
New Group	

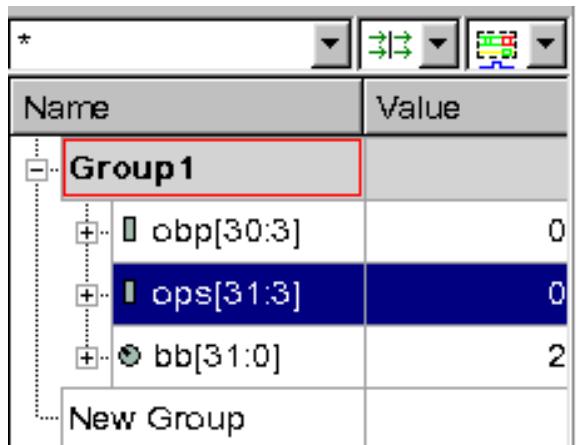
2. Right-click and select **Properties**.

The Signal Properties dialog box opens.



3. Type a name in the **Display Name** field and edit the range.
4. Click **OK**.

The signal name and range are changed.



A screenshot of the Signal pane in a software interface. The pane has a header with buttons for file operations. Below the header is a table with two columns: 'Name' and 'Value'. A red box highlights the first row, which contains the text 'Group1'. An arrow points from the text 'Signal name after rename' to this row. The table also lists three signals under 'Group1': 'obp[30:3]' with value 0, 'ops[31:3]' with value 0, and 'bb[31:0]' with value 2. At the bottom of the table is a row labeled 'New Group'.

Name	Value
Group1	
obp[30:3]	0
ops[31:3]	0
bb[31:0]	2
New Group	

Renaming Signal Groups

To rename a signal group

1. Double-click the signal group in the Signal pane.

The signal group is selected.

2. Type a new name.

The signal group is renamed.

Undo and Redo Operation for Signals

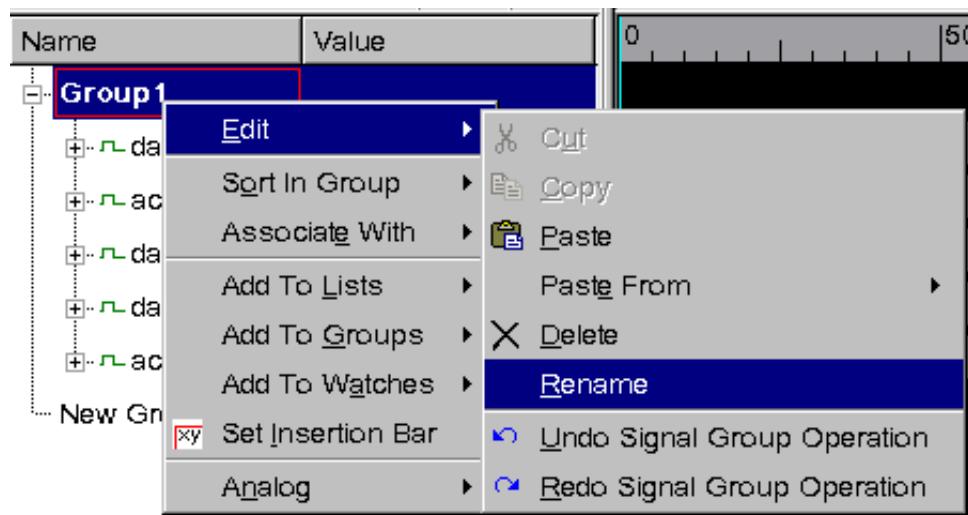
You can undo (revert the operation) and redo (redo the reverted operation) the following actions on the signal groups in the Wave view:

- Adding signals to the signal group
- Deleting signals and signal groups from the Wave view

- Reordering signals and signal groups in the Wave view
- Renaming signal groups in the Wave view
- Creating and deleting signal groups

For example, to undo and redo renaming of signal group in Wave view:

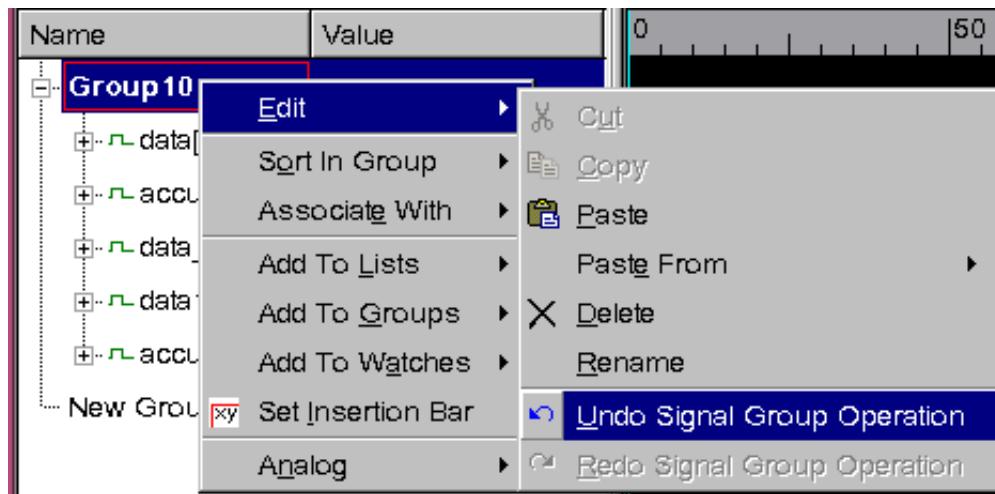
1. Right-click on the signal group and select **Edit > Rename**, as shown in the following figure:



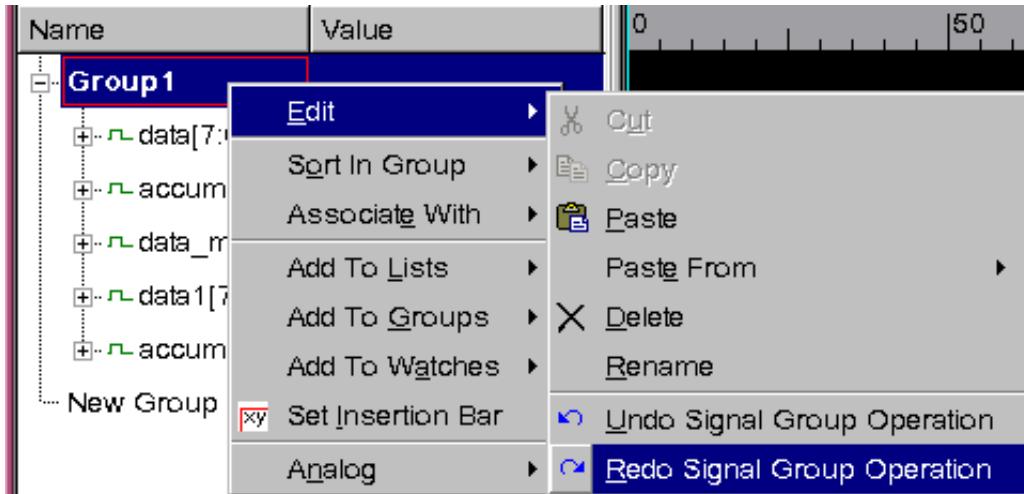
For example, rename the signal group as **Group10**, as shown in the following figure:

Name	Value
Group10	
+--> data[7:0]	8'hxx
+--> accum[7:0]	8'hxx
+--> data_m[7:0]	8'hzz
+--> data1[7:0]	8'hxx
+--> accum1[7:0]	8'hxx

- Right-click on **Group10** and select **Edit > Undo Signal Group Operation**, as shown in the following figure, to undo the renaming of signal group **Group1**.



- Right-click on **Group1** and select **Edit > Redo Signal Group Operation**, as shown in the following figure, to redo the renaming of signal group **Group1** to **Group10**.



Creating Multiple Groups when Adding Multiple Scopes

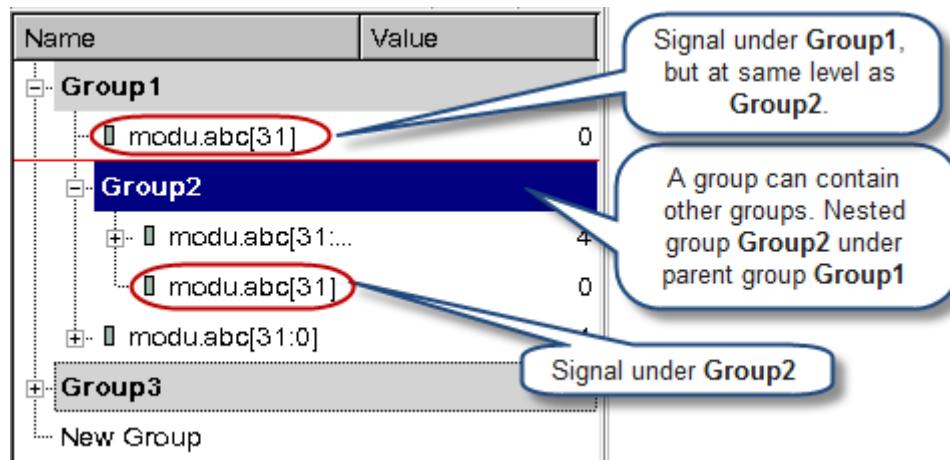
When you add the scopes to the waves, lists, or groups from the Hierarchy pane, the signal groups will be created based on their respective scopes.

If you select **Display signal group exclusively** in the Application Preference dialog box and add multiple scopes to Wave view, multiple groups will be created but only the last group will be displayed.

Creating Nested Signal Groups

DVE allows you to create nested signal groups in the Wave View and List View. That is, DVE allows a regular signal group to be part of another signal group, as shown in [Figure 5-3](#). You can create nested signal groups by dragging and dropping one signal group to another, or by using Signal Group Manager dialog box.

Figure 5-3 Nested Signal Groups



Creating Nested Signal Groups in the Wave View

Consider [Figure 5-4](#), which shows Wave View Signal Pane with signal groups *Group1* and *Group2* at the same level of hierarchy.

Figure 5-4 Signal Groups in the Wave View Signal Pane

The screenshot shows the Wave View Signal Pane. At the top, there are buttons for zooming in and out, and a toolbar with various icons. Below that is a header row with 'Name' and 'Value' columns. The main area contains two groups: 'Group1' and 'Group2'. 'Group1' contains two items: 'modu.abc[31]' with value 0 and 'modu.abc[31:0]' with value 4. 'Group2' contains two items: 'modu.abc[31:0]' with value 4 and 'modu.abc[31]' with value 0. A new group button labeled 'New Group' is visible at the bottom.

Name	Value
Group1	
modu.abc[31]	0
modu.abc[31:0]	4
Group2	
modu.abc[31:0]	4
modu.abc[31]	0
New Group	

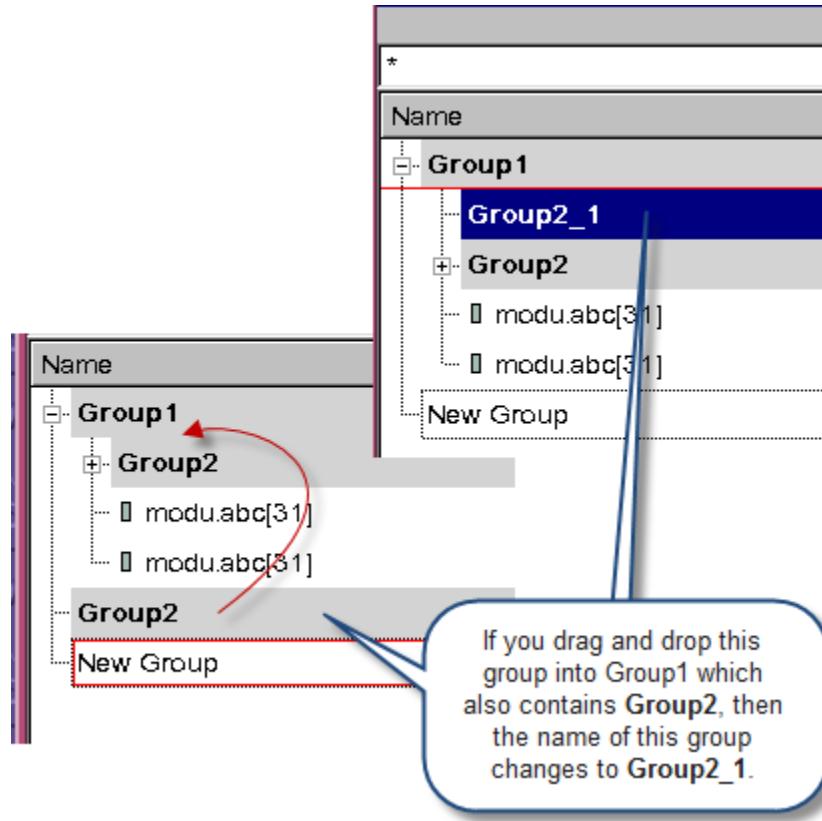
You can use one of the following methods to create nested signal groups in Signal Pane:

- Dragging and dropping one signal group into another (for example, *Group2* into *Group1*).
- Using [Signal Group Manager](#) dialog box.

Key Points to Note

- If a group (for example, *Group2*) already exists in another group (for example, *Group1*) that you want to drag and drop it into, then the name of the *Group2* will be modified to have a trailing underscore and a serial number (*Group2_<n>*), as shown in [Figure 5-5](#). This applies for both Signal Group Manager dialog box and drag and drop operation in Wave View Signal Pane.

Figure 5-5 Dragging and Dropping Groups in the Wave View Signal Pane



- You can use **Create Group** right-click option in Wave View Signal Pane to create subgroups within a group. The default group name will be `Group<index>`, where `Group` is a common base name with a serial number `<index>` appended to it, as shown in [Figure 5-6](#). This also applies to Signal Group Manager dialog box. You can use **Create Group** button in Signal Group Manager dialog box to create subgroups.
- A group can contain signals and subgroups. Signals and subgroups within a group are displayed in an arbitrary order, as shown in [Figure 5-7](#).

Figure 5-6 Creating Subgroups Within a Group in Wave View Signal Pane

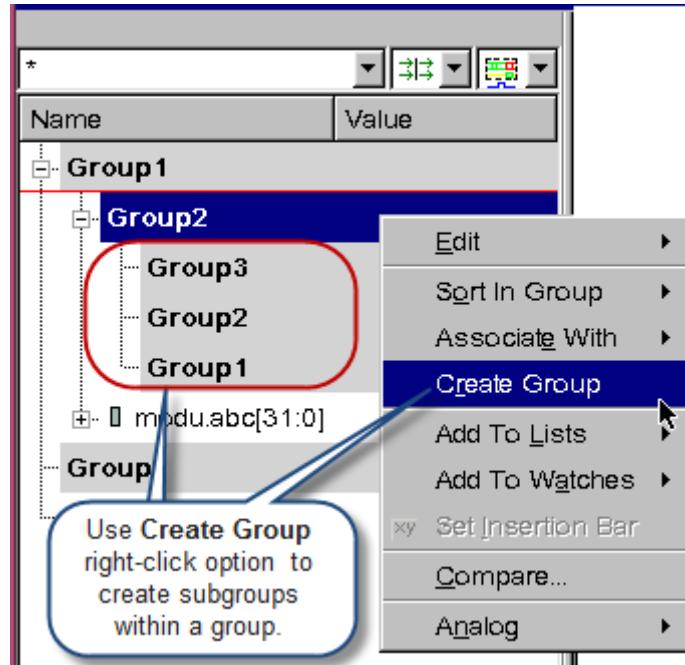


Figure 5-7 Groups Displayed in an Arbitrary Order

```

Group1          //toplevel group
---Group1_1
-----Group1_1_1
-----a
-----b
---clk
---reset
---Group1_2
-----c1
-----c2
---data
Group1_1
----request
----mem
Group3
----s1
----s2
New Group

```

The following points illustrate various drag and drop scenarios for groups and signals:

- Drag and drop *Group3* before *mem*: *Group3* will be placed as a child of *Group1_1* before *mem*.
- Drag and drop *Group3* on *Group1*: *Group3* will be placed as first child of *Group1*.
- Drag and drop *Group1_1* (with subgroup) on *New Group*: *Group1_1* will be placed as a top-level group.
- Drag and drop signal *s2* before *Group1_2*: Signal *s2* will be placed after signal *reset*.
- Drag and drop signal *s2* on *Group1_2*: Signal *s2* will be placed as first child of *Group1_2*.
- Group names does not support “|” character. This character will be automatically converted to “_”.
- The Group Filter drop-down displays all signal groups (including the nested ones) of the Wave View Signal Pane, as shown in [Figure 5-8](#). You can filter a group by unchecking the checkbox next to it.
 - If **Recursive** is selected, then checking or unchecking the checkbox of a parent group affects all of its children (subgroups) in a similar manner. For example, consider [Figure 5-8](#). If you uncheck *Group1* checkbox in the Group Filter drop-down, then *Group1* and all of its subgroups will get filtered from the Wave View Signal Pane, as shown in [Figure 5-9](#).
 - If you uncheck a parent group, but not its children (subgroups), then DVE still displays the parent group in Signal Pane, but not its signals

Figure 5-8 Viewing Groups in Group Filter

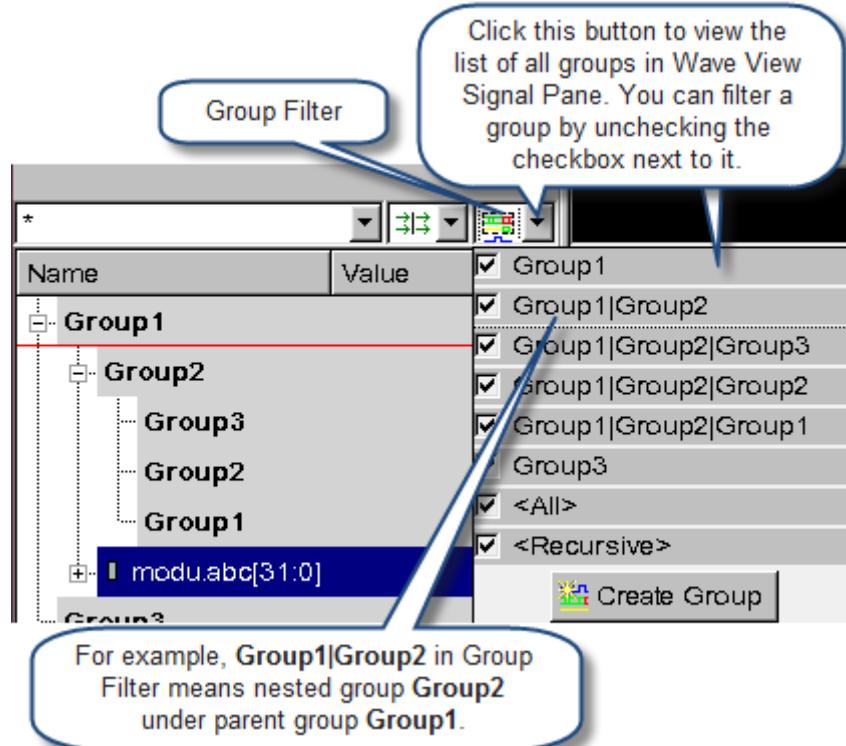
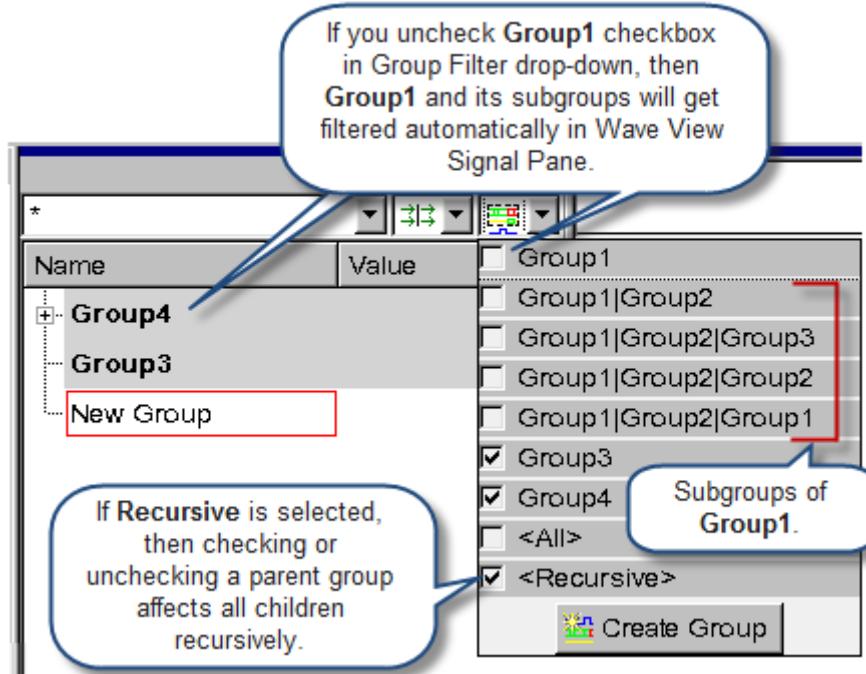


Figure 5-9 Using the Recursive Option



Creating Nested Signal Groups in the List View

List View displays nested signal groups in table format, as below. You can use **Signal Group Manager** dialog box to organize signal groups in the List View.

Figure 5-10 Viewing Nested Signal Groups in the List View

New Group	Group1 Group2	Group3	Group1
	modu.abc [29]		abc [31:0]
x ls			
0	0		0
10	0		1
20	0		2
30	0		3
40	0		4
60	0		4

Using Signal Group Manager to Create Nested Signal Groups

You can use Signal Group Manager dialog box to organize signal groups in Wave View and List View. This dialog box displays all top-level groups and nested subgroups (see [Figure 5-11](#)), and allows you to:

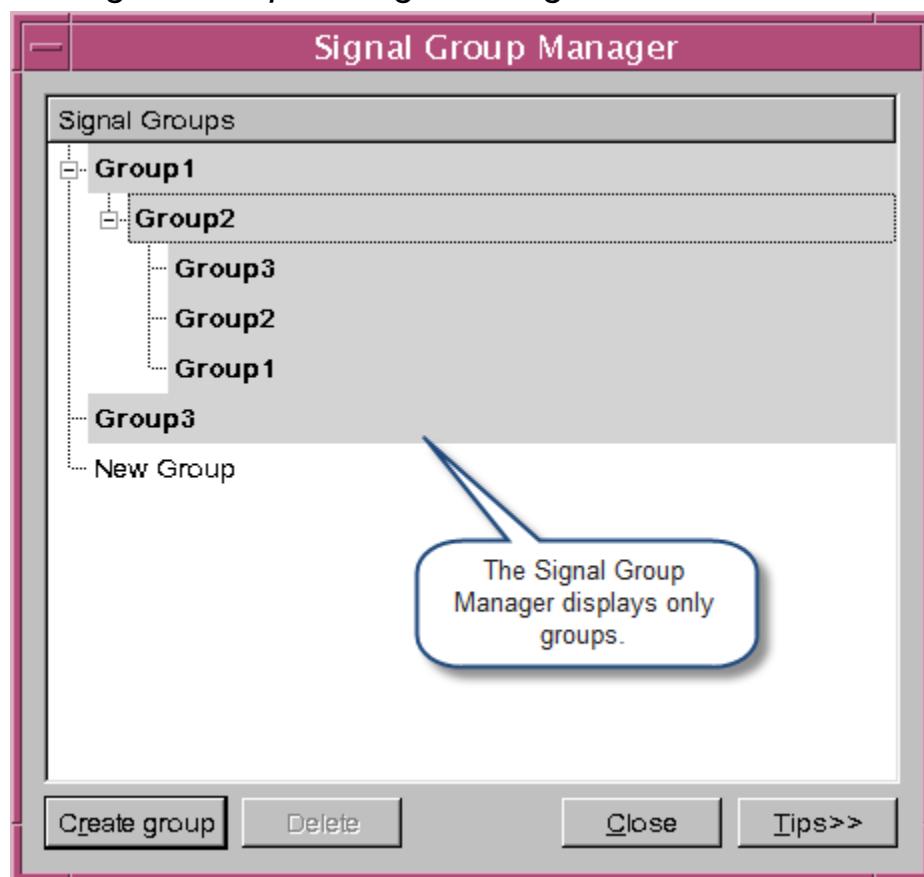
- Drag and drop one group into another
- Create new groups under a group.
- Delete a group.
- Rename a group by double-clicking on it.

Note:

Creating, deleting, moving, or renaming a group in Signal Group Manager dialog box will have similar affect in Wave View and List View, and vice versa.

To open Signal Group Manager dialog box, select **Signal > Signal Group Manager**.

Figure 5-11 Signal Group Manager Dialog Box



- You can use the **Create Group** button to create a new top-level group or a subgroup for a selected group.
- You can use the **Delete** button to delete the selected group.

Deleting Signal Group

When you delete a signal from the Wave view, it will be deleted globally. If the same signal is present in few other views, and you want to delete it, a warning message is displayed. You can either select to delete or hide the signal.

Once you select to delete the signal, it would be deleted globally from all views. If you select to hide the signal, it will be hidden in the current view.

If the signal groups are deleted, save session will not have the deleted signal groups. If you hide the signal, it will be hidden when you are saving or reloading the session.

Customizing Duplicate Signal Display

When displaying duplicate signals, you can customize the display of an instance of a signal without affecting the display of any duplicates.

To customize the signal display

1. Select a signal, then toggle **Signal > Default Properties** off.
2. Select **Signal > Properties** and make any changes to the signal scheme or color.

Changes are made to the selected signal without affecting the display of duplicate signals.

Note:

- If you do not toggle Default Properties off, the changes will become the default and duplicate signal display will also change.

- If a signal group is in two Wave views, changing a signal will change the signal in the other Wave view if it is the same instance.

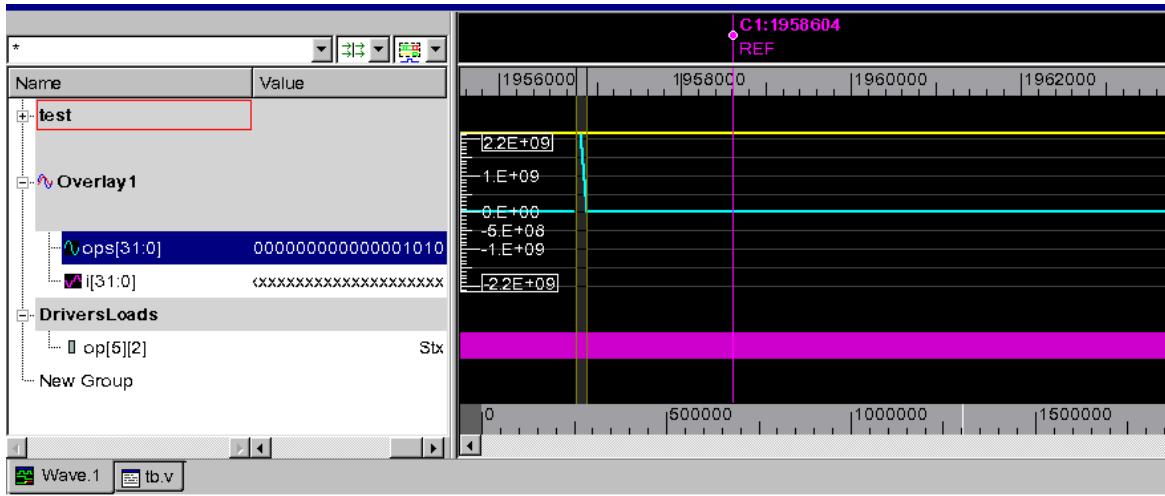
Overlapping Analog Signals

You can combine analog signal waveforms to visualize relationships between signals.

To overlap analog signals, do any of the following steps:

1. Drag and drop one or several analog signals onto an analog signal.

DVE creates a overlaying group as shown below:



2. Right-click the signal and select **Analog Overlay** from the menu.

The signals are overlapped.

3. Select a signal group with two or more analog signals, then right-click and select **Analog Overlay** from the CSM.

DVE overlay all signals in the group.

4. To restore the overlaid group back to a regular signal group, right-click the signal and select **Unoverlay** from the CSM.

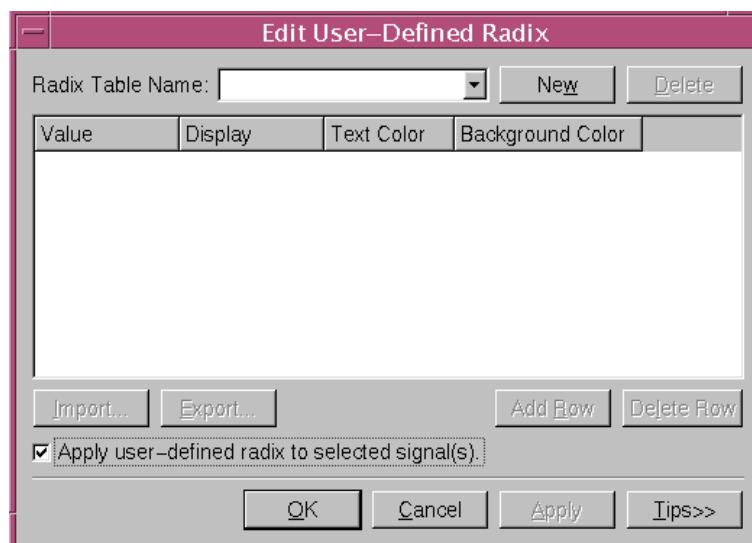
Using User-defined Radices

This section describes how to create, edit, import, and export user-defined radices. You can define a custom mnemonic mapping from values to strings for display in the Wave view.

To create, delete, import, and export a user-defined radix

1. Select **Signal > Set Radix > User Defined > Edit**.

The Edit User-defined Radix dialog box opens.



2. Click **New**, enter a radix name, then hit the **Enter** key on your keyboard.

All buttons on the Edit User-defined Radix get enabled.

3. Click **Add Row** to activate a row for the user-defined radix and perform the following steps:

- Select the text and background colors for each row entry.
- Select the radix, click a cell in the Value and Display column, then enter the values.

The radix is edited.

4. Select a row, then click **Delete Row**.

The row is deleted.

5. Select a radix from the Radix Table Name drop-down and click the **Delete** button.

The radix is deleted.

6. Click **Import**, then browse and select the desired radix.

The radix is imported.

7. Click **Export**, select the radix, then enter a radix name.

The radix is exported.

8. Select the **Apply user-defined radix to selected signal(s)** checkbox.

The user-defined radix is applied to the selected signal in the Wave view.

9. Click **OK** or **Apply** to save the user-defined radix.

Note:

Regular expression (wildcard mode) is supported with user-defined radix. For example, if you define a value 'VALUE' to the radix 2'b0* (where * is a wildcard character), all the radix whose name starts with 2'b0 such as 2'b01, 2'b00, will get a value 'VALUE'.

Currently, wildcard supports only '0' and '1'; doesn't supports 'x' and 'z'.

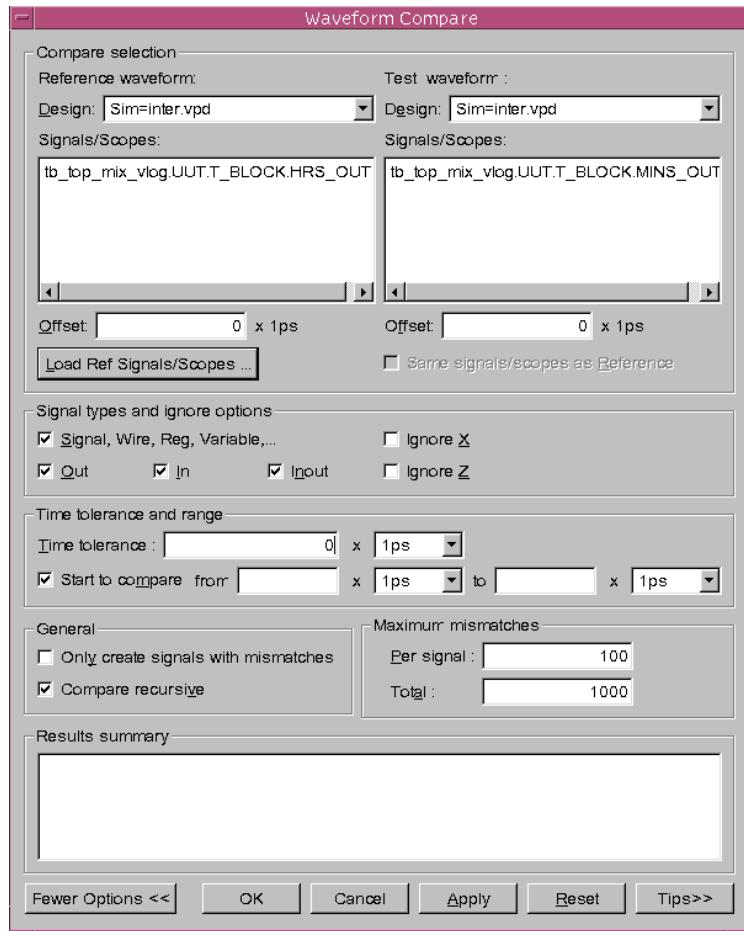
Comparing Signals, Scopes, and Groups

You can compare individual signals with the same bit numbers, scopes (for comparing variable children), buses, or groups of signals from one or two designs.

To view a comparison

1. Select one or two signals, signal groups, scopes, or buses from the Signal pane of the Wave view.
2. Right-click and select **Compare**.

The Waveform Compare dialog box opens.



3. Click **Load Reference Signals/Scopes** and select the text file with the signals and scopes to reference.

Note:

If you are comparing two designs from root, then the reference waveform region and test waveform region can be empty.

4. Click the **More Options** button.

The dialog box is expanded and additional options are displayed.

5. In the Signal types and ignore options section, select the signal types to compare and select ignore options.

For example, if you select Ignore X and if the reference signal value is X, there is always a match, whatever the values of the Test Signal.

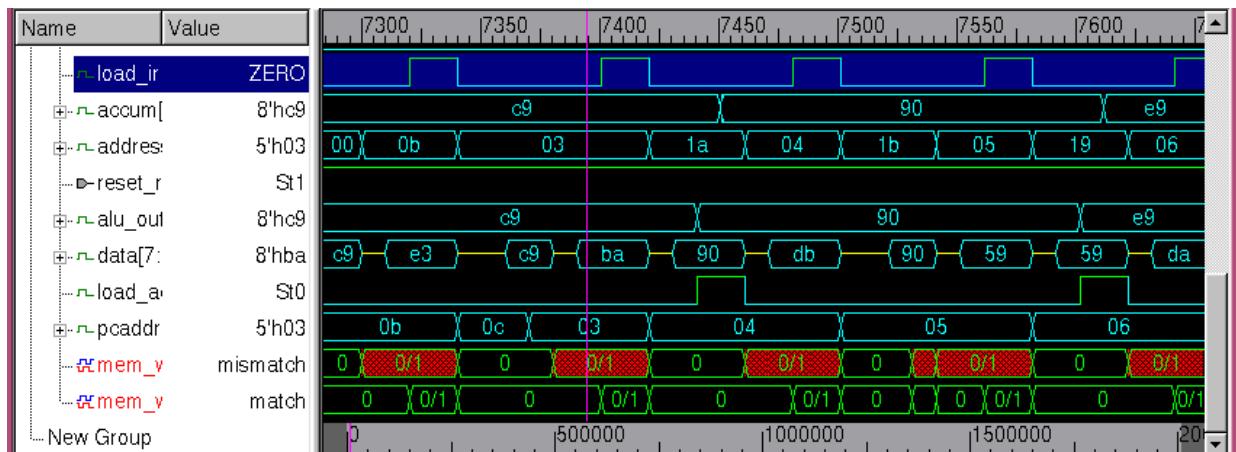
6. Enter a Time Tolerance to filter out mismatch values that have time ranges smaller than the tolerance range.
7. In the General section, select to compare recursively or to only create signals with mismatches.
8. Enter mismatch settings for maximum mismatches per signal and maximum total mismatches to report.
9. Click **Apply** to start the comparison and keep the dialog box open.

Or

Click **OK** to start the comparison and close the dialog box (you can open it at any time from the Signal pane CSM).

Results are displayed in the current Wave view.

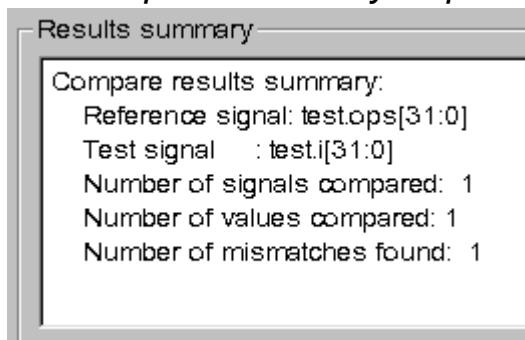
Figure 5-12 Compared Signal Groups in the Wave View



10. Select a result in the Wave view, right-click and select **Show Compare Info.**

The comparison results are displayed in the Waveform Compare dialog box.

Figure 5-13 Waveform Compare Summary Report



You can change the options, then compare them again.

Creating a Bus

You can select few signals or components and group them to create buses. You can then view the behavior of the bus in the Wave or List view. You use the Bus Builder function to create and edit buses using

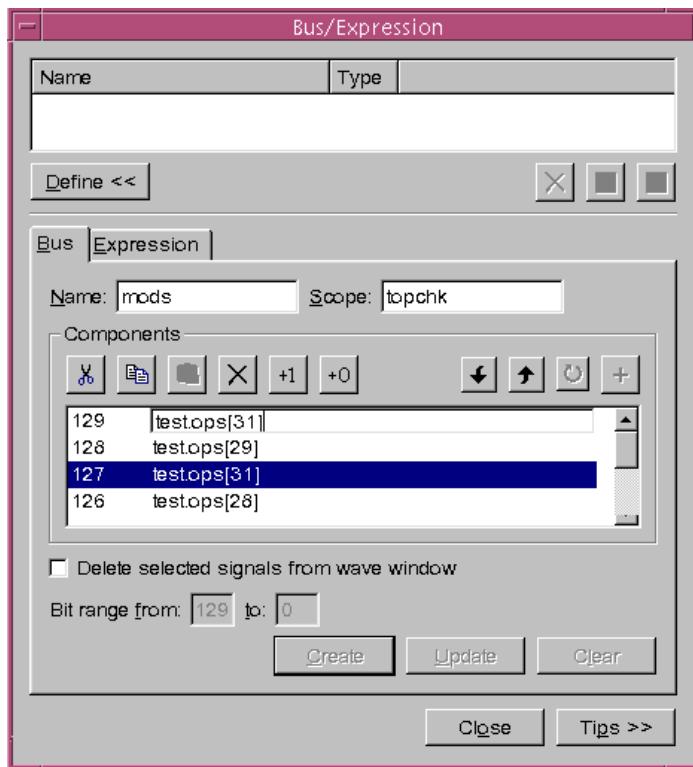
the signals from the opened designs. You can drag and drop components from the Hierarchy pane or Data pane to create buses. You can edit the bus bit range by simply changing either the MSB (most significant bit) or the LSB (least significant bit) of the bus.

After you create a bus, you can select the bus in the Bus/Expressions dialog box and add it to the Wave or List view. By default, it will reside in the highest level signal group common to its components.

To create a bus

1. Select few signals in the Signal pane.
2. Right-click and select **Set Bus**.

The Bus/Expressions dialog box opens.



3. Click the **Define** button.

The Bus/Expressions dialog box is expanded.

4. Enter a name for the new bus.

You can give any legal name to the buses for the language (for example, Verilog or VHDL).

5. Enter a scope name to create the bus under a user-specified scope.
6. (Optional) Add constant +1 or constant +0 signal to the bus using the following icons:



7. Click **Create**.

The bus is created. The bit range is shown from 0 to N, where N is the number of components in the bus. Vectors and structures are expanded to their bits. For example, if “top.risc.pc[3..0]” is added to the list, it is added as four items.

Modifying Bus Components

You can edit an existing bus or modify the components and their order in a new bus using the Bus Builder toolbar.

To modify bus components

1. Select the bus in the Wave view, then right-click and select **Set Bus**.

2. Select the components in the component list, then click the icon in the Bus Builder toolbar (see below):



The components are deleted.

3. Select one or more components in the component list, then click one of the following icons in the Bus Builder toolbar:



The components are moved up or down in the list.

4. Select two or more components from the component list, then click the following icon in the Bus Builder toolbar:



The order of components relative to each other is reversed.

5. Double-click the signal and edit the range.

For example, define a signal as $a[0:7]$, change the bit range to $a[7:6]$, and click on create/update. A bus will be created/updated with the selected 2 bits as $a[7], a[6]$.

If you enter a wrong range, a warning message is displayed and the text color of the signal becomes red. You need to enter the correct format to save the bit range.

6. Select the signal and click the "+" icon to expand.

The signal is expanded in the same order as specified in the name. For example, $a[1:3]$ signal will expand to $a[1], a[2], a[3]$.

You can perform all the toolbar operations on the expanded signals.

7. Click **Update**.

The bus is updated.

Viewing Bus Values

To view bus values in waveforms (with transitions on edges), position the cursor on the waveform.

A ToolTip is shown at a transition displaying the transition values.

Creating an Expression or a Counter

You can create expression for signals in the Wave view to view the value change when that expression occurs. You use the Expressions tab in the Bus/Expressions dialog box to create and modify expressions.

You can create counters to count the value transitions for expression or signal. Counter is treated as a special expression and you can create or update counter in the same way as you create an expression in the Bus/Expression dialog box.

The counter is supported both in post-processing mode and interactive mode. In interactive mode, with simulation going on, the count result will be updated as other signals in the Wave view.

Example - test.v

```
module top;
    reg clk, _clk;
    dut INTER(clk, _clk);
    initial begin
        clk = 0;
        forever #25 clk = ~clk;
```

```

        end
endmodule

module dut(clk, _clk);
    output _clk;
    input clk;

    not i2(_clk, clk);
endmodule

```

To compile the example code, use the following commands:

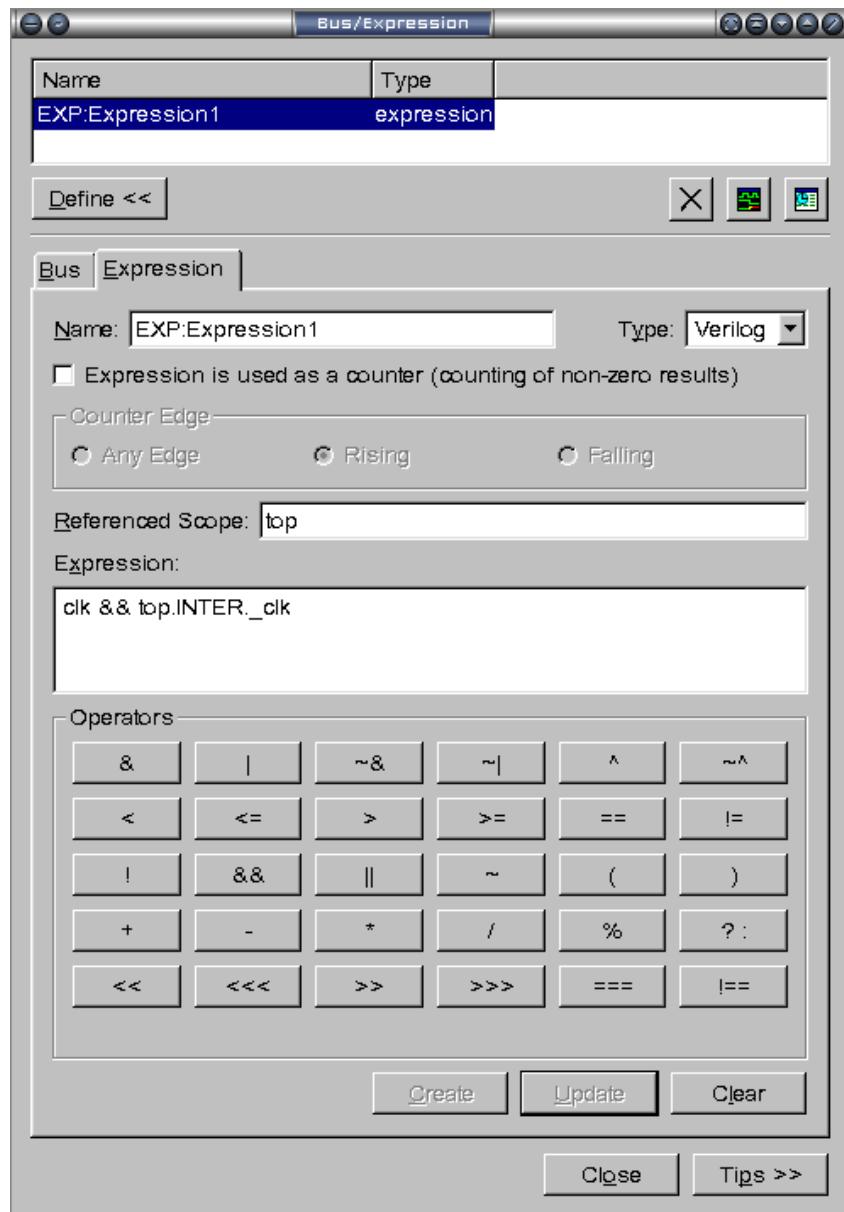
```
vcs -nc -debug_all -sverilog test.v
simv -gui &
```

To create an expression

1. Select a signal in the Wave view, right-click and select **Set Expressions**.
The Bus/Expressions dialog box opens.
2. Enter a name for the expression.
3. Select an expression type.
4. Insert the operators into the expression.
5. Type the name of a reference scope in the **Referenced Scope** field.

The **Referenced Scope** field specifies a scope from which the signal names in the Expression are referenced. If the Expression field includes full hierarchy name of the scope, the **Referenced Scope** field is ignored. When the expression does not include the full hierarchy name, DVE looks for all signals in the referenced scope.

If you select a signal in the Wave view, Source view, or Data pane and then open the Expression dialog box, the referenced scope is filled by default.



6. Click **Create**.

The expression is created.

7. Select the expression and click the **Add to Wave** or **Add to List** icons.

The expression is added in the Wave or List view. You can now run the simulation and view the values of signal when the expression occurs.

To create or update a counter

1. In the Bus/Expression dialog box, type a counter name in the Name field.

For example, EXP:Expression1.

2. Select the checkbox **Expression is used as a counter (counting for non-zero results)** to create a signal that represents the value transitions of the expression.

The Counter Edge radio buttons get enabled.

3. Select any of the following **Counter Edge** as desired.

- Any Edge - Identifies any expression/signal. Whenever the value of the expression/signal changes, counter signal counts this transaction.
- Rising - (Default) Identifies the bit type signals. Whenever the expression/signal changes from low to high, counter signal counts the transaction.
- Falling - Identifies only the bit type signals. Whenever the expression/signal changes from high to low, counter signal counts the transaction.

4. Click **Create**.

The newly created expression counter is displayed in the Bus/Expressions dialog box. You can also see the count results in the Wave view.

5. Drag and drop the counter in the Bus/Expression dialog box to update it.

For example, you can update the counter edge from “Rising” to “Any Value”.

6. Click the **Update** button.

The counter is updated. You can see the updated count results in the Wave view.

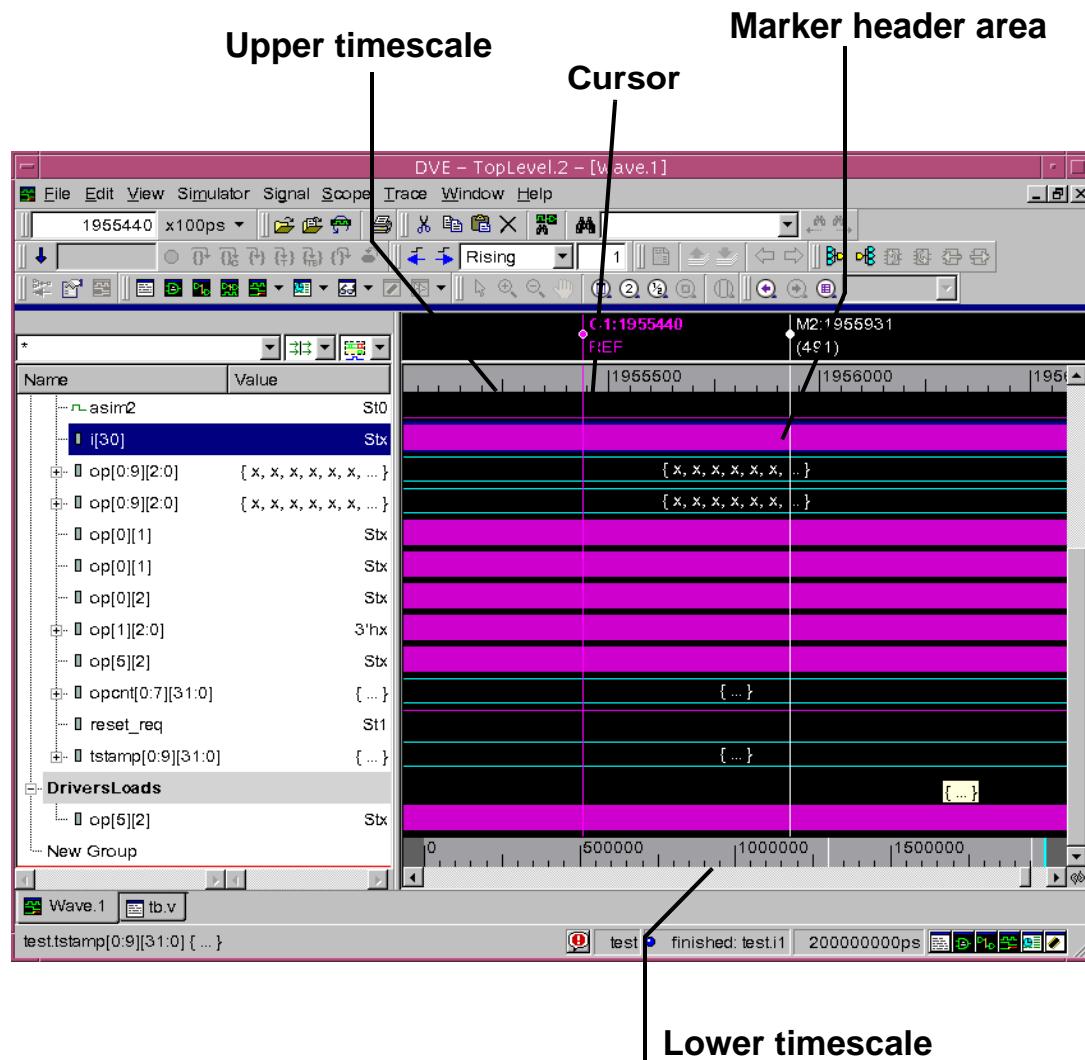
Limitations

- Mixed design is not supported.
- SystemC designs are not supported.
- Usage of macro in scope is not supported.
- Complex SystemVerilog data types are not supported.

Using the Wave View

The Wave view displays the value transitions of signals and assertions.

Figure 5-14 The Wave View



Cursors and markers are explained in “[Cursors and Markers](#)” on page 54.

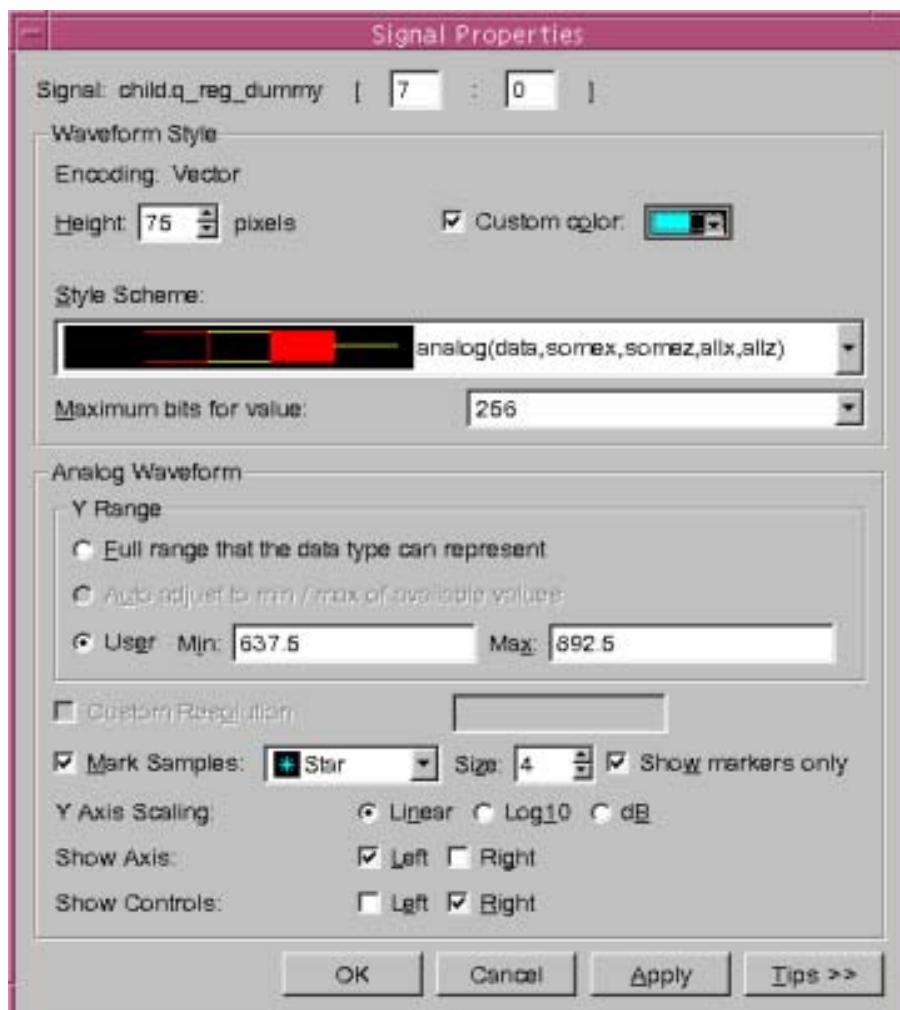
The Wave view has an upper and a lower timescale. The upper timescale displays the range of simulation times currently on display in the Wave view. The lower timescale displays the range of simulation times throughout the entire simulation.

Customizing Waveforms Display

To customize the display of waveforms

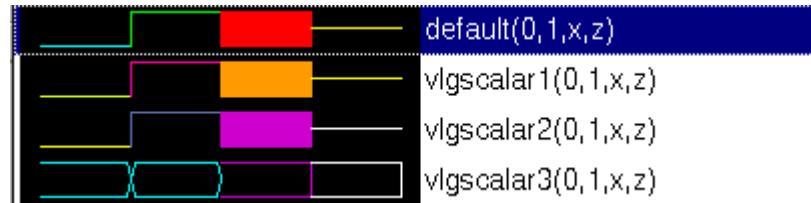
1. Select a signal in the Wave view and select **Signal > Properties**.

The Signal Properties dialog box appears:

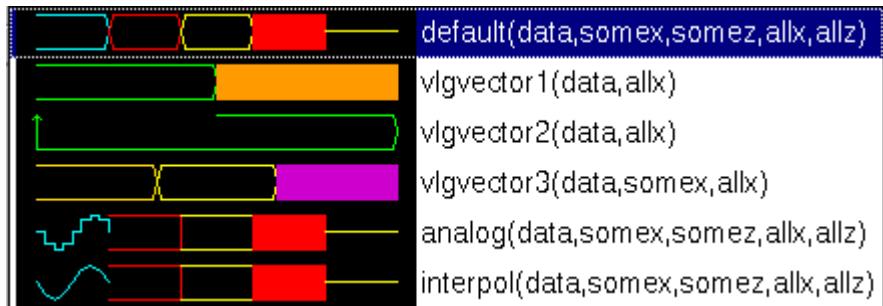


2. In the Waveform Style section, set a height and custom color for the waveform.
3. Set the Style Scheme as follows:

- For a scalar waveform, click the arrow and select from a scalar scheme as shown below:

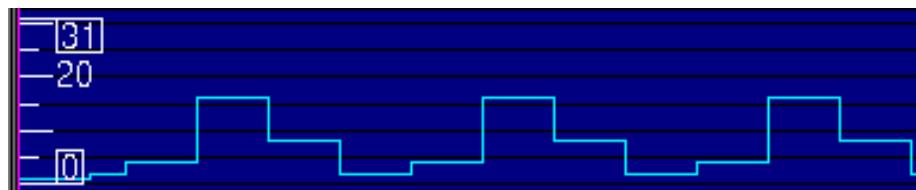


- For an analog waveform, click the arrow and select from a vector scheme, an analog scheme, or an interpolated scheme as shown below:

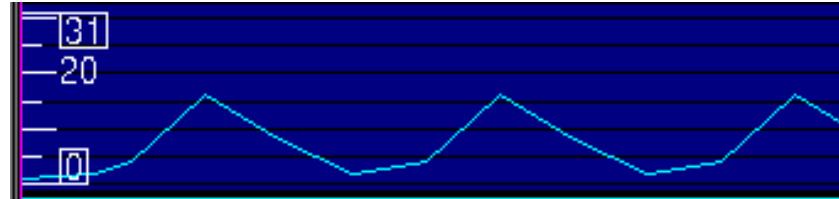


The default display is a vector scheme. Variant vector schemes alter the color and the values are displayed.

- The **analog** option displays an analog waveform as a staircase scheme, that stays at the value until the next reported value change.



- The **interp** option displays an analog waveform interpolated between each reported value change.



Note:

You cannot represent same signal in different "Drawing Style Scheme" (Digital and Analog).

4. If you are selecting the analog option in step 3, then set the following properties:
 - Set the Y range values as follows:
 - Display the full range the data type can represent.
 - Auto adjust the display to the minimum and maximum of available values.
 - Display a user-defined minimum and maximum range.
 - Select a Custom Resolution.
 - Select the Mark Samples check box and select the marker style and size.

- Select the Show Markers Only check box to draw the waveform only with markers.



- Select Y-axis scaling from **Linear**, **Log_10**, or decibel (**dB**).
 - Set the Axis display.
 - Set the Controls as left or right.
5. Click **OK** to apply the settings and close the dialog box, **Apply** to apply changes and keep the box open, or **Cancel** to close the dialog box and disregard changes.

Displaying Grid in Wave View

You can display grid (regularly spaced vertical lines) in the Wave view. The grid can be useful to measure the number of clock cycles for a signal. DVE draws grid in a grid range that you specify for a signal, and you can count and study the signal changes within the grid range.

Example

test.v

```
module top;
reg a;
wire #5 a_delay = a;
initial begin
    #20 a = 1'b0;
    #20 a = 1'b1;
    #20 $finish;
end
endmodule
```

To compile this example code, use the following commands:

```
vcs test.v -debug_all -sverilog
simv -gui &
```

Setting Grid Properties

1. Select a signal from the Wave view.
2. From the **View** menu, select **Grid Properties** or click the **Setting**

Grid Properties in the Wave view icon  on the toolbar.

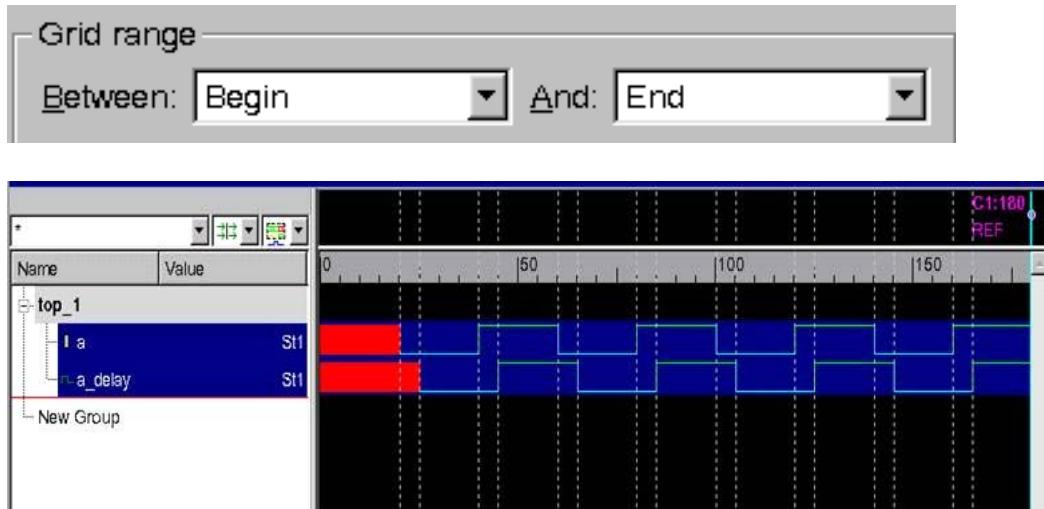
The Grid Properties dialog box appears.



3. Specify the grid properties as follows:

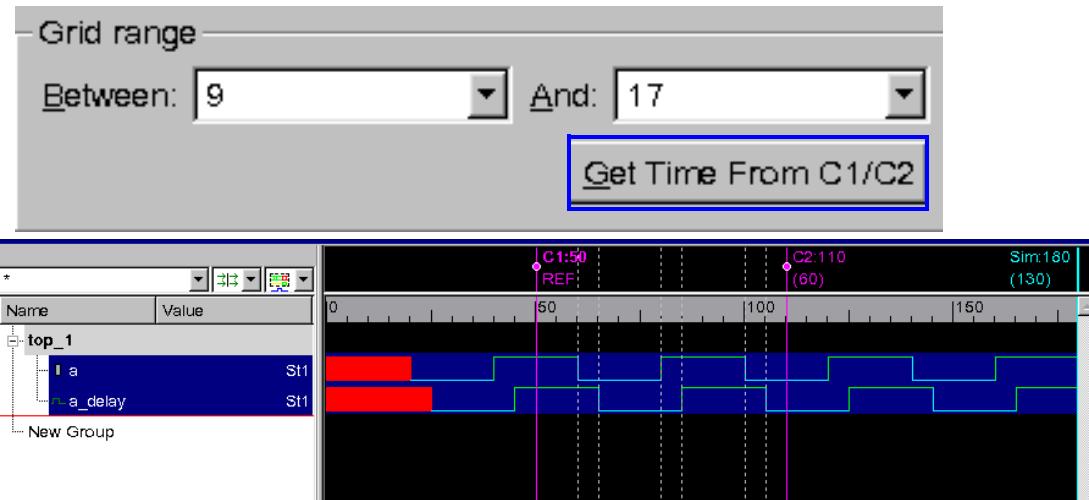
- **Grid Range** — Specifies the range for drawing the grid as explained in the following cases:

- **Case1: Default.** You can draw the grid between Begin and End, that is from the start of the simulation to the end of simulation time.

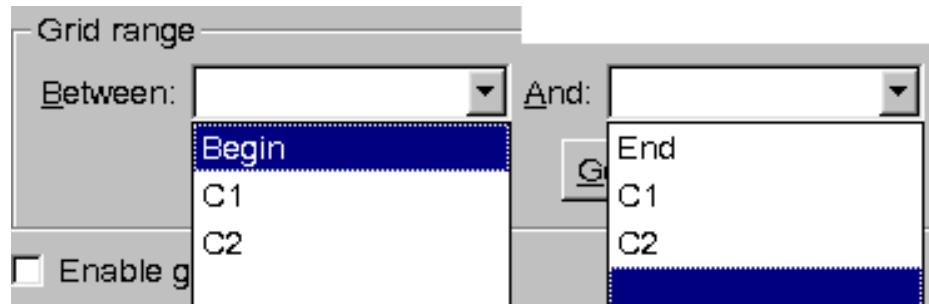


Caution!If you select the default option, for huge designs with multiple signals and huge simulation time, waveform drawing with grid can be slow.

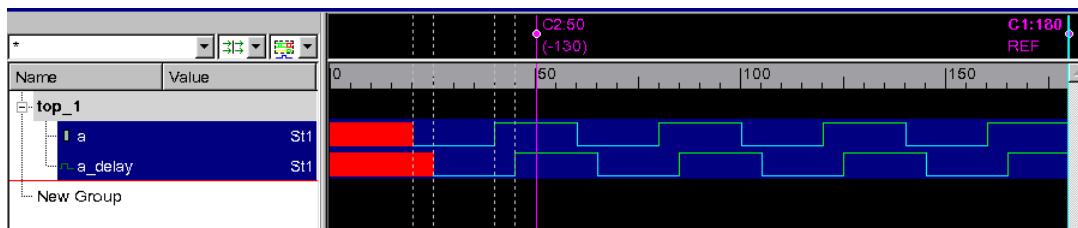
- **Case 2:** Assuming you have C2 marker, then you can click the **Get Time from C1/C2** to capture the time of C1 and C2 in the **Between** and **And** fields automatically. If there is no C2, only the time of C1 is displayed in the **Between** field and the waveform is drawn from C1 to End.



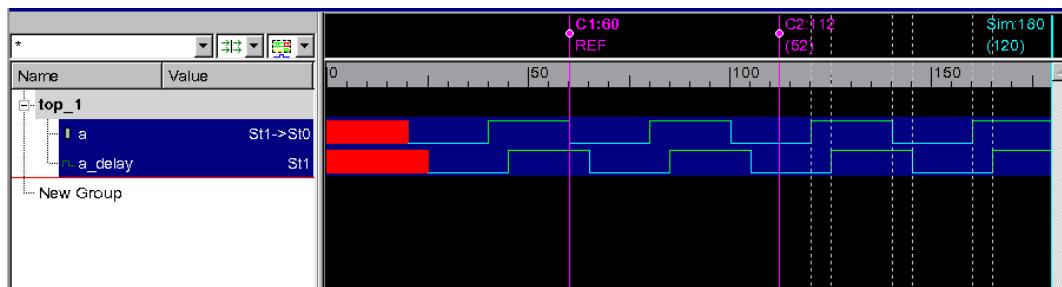
- **Case 3:** Assume you have C2 marker, then you can select any combination of range from the Grid Range drop-down menu.



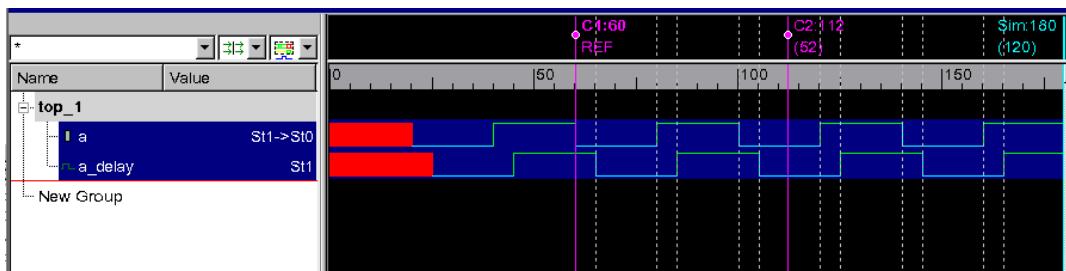
Grid between Begin and C2



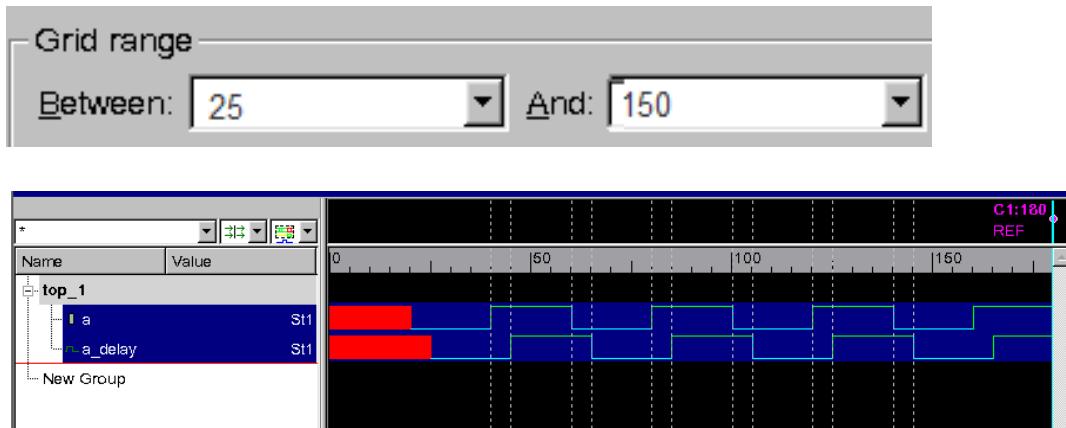
Grid between C2 and End



Grid between C1 and End



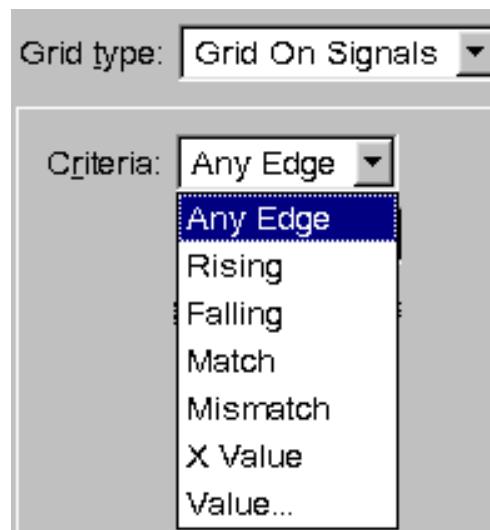
- **Case4:** You can also enter time in the **Between** and **And** fields.



Note:

You should click the **Get time from C1/C2** button only for Case2.

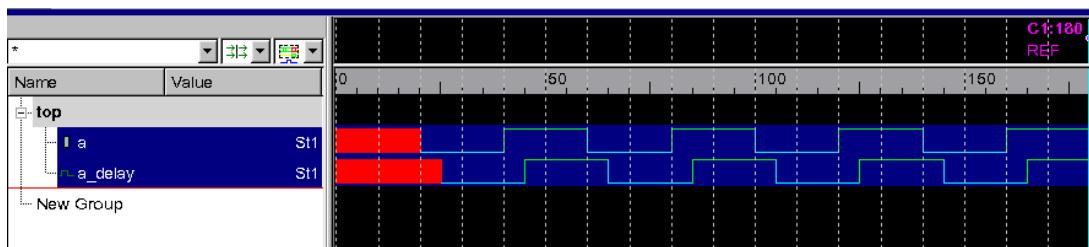
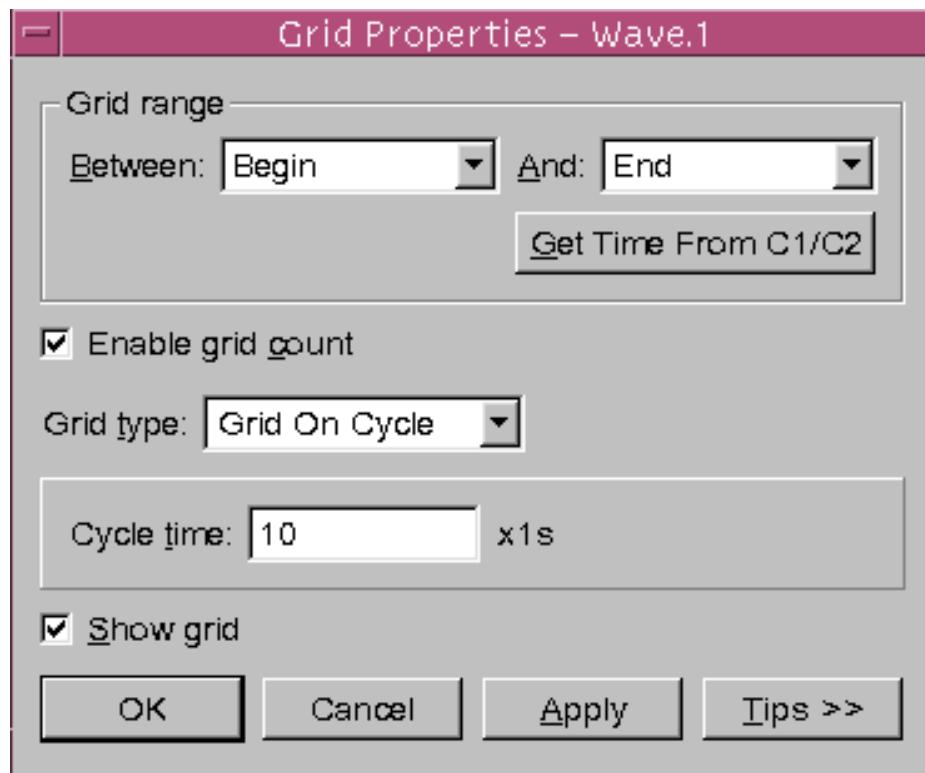
- **Grid Type**—Creates any of the following types of grid depending on the grid range that you select:
 - **Grid on Signals**—Draws the grid on the signal edges or any of the following mentioned criteria of the signals.



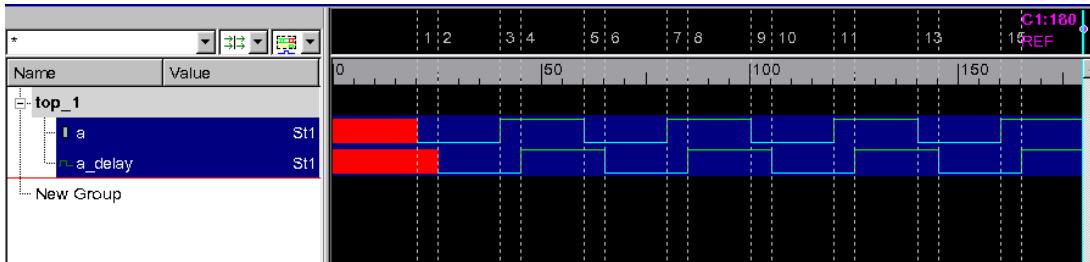
-Grid on every matches—Indicates that the grid is done for every numbered match.

-Signal List — Contains the list of selected signals in Wave view, when you are setting the grid properties. You can click the **Get Selected Signals** button to replace current signals with selected signals in the Wave view.

- Grid on Cycle** — The grid is drawn in the specified time range with the cycle as interval. Enter the cycle time in the Cycle Time field and the grid is drawn in the time interval from the range specified in the Grid Range field.



- Enable Grid Count — Adds the number of count to the grid in the Wave view when the grid is shown, if selected.



- Show grid — Enables or disabled grid drawing. When you clear the check box and click the **OK** button, the grid is removed from the Wave view.
4. Click **OK** to apply the properties setting, and close the dialog.

Click **Apply** to apply the properties settings and not close the dialog.

Click **Cancel** to abandon the changes, and close the dialog.

Click **Tips** to display the tips page on the right.

The grid is applied as per the set properties as follows:

Note:

- Grid is not supported in the Delta Cycle region.
- If you set the grid on numerous signals, the Waveform drawing will be slow.

Cursors and Markers

In the waveform display area, you can insert markers and cursors.

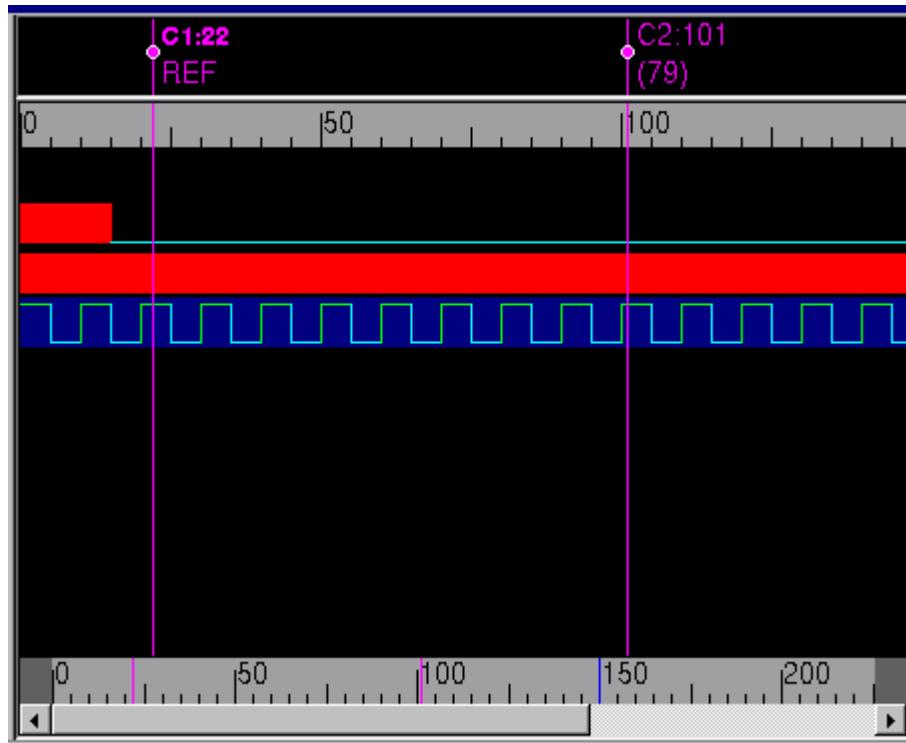
Using Cursors

To insert cursors

1. Click the mouse to place cursor C1 in the waveform display area.
The C1 cursors default position is at time 0.
2. Click somewhere in the waveform display area.
Cursor C1 moves to the new location.
3. Click the middle mouse button to place cursor C2 in the waveform display area.
4. Middle-click somewhere in the waveform display area and cursor C2 moves to this new location.
5. Place the mouse cursor on the round cursor handle in the cursor area, hold down the left mouse button, and drag the cursor to the desired location.
6. Click either the left or the middle mouse button in the waveform or cursor area to move C1 or C2, respectively.

The interval between the two cursors is always displayed in the marker header area.

Figure 5-15 Graphical Display Cursors



In the [Figure 5-7](#), the simulation time and the delta between the reference cursor (C1) and cursor C2 is shown in the marker header area.

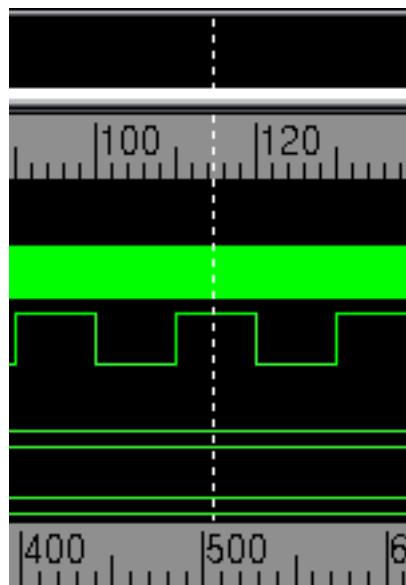
Creating Markers

Markers differ from cursors in the way you insert and move them. Like cursors, markers display the delta between the reference cursor (C1) and the marker. The Markers dialog box allows you to create, move, hide, delete markers, set the reference marker, and scroll the graphical display until it reveals a marker.

To create a marker

1. Right-click in the Wave view, and select **Create Markers** from the CSM.

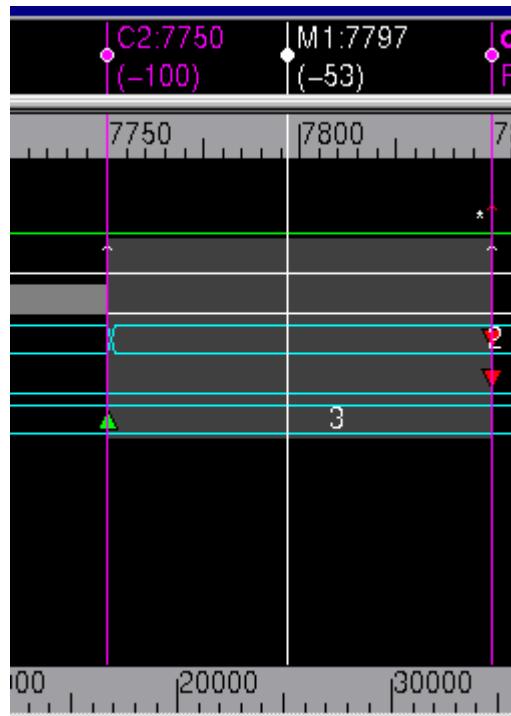
This inserts a dotted line on your mouse cursor in the graphical display.



The dotted line tracks the mouse cursor as you move the mouse in the waveform or marker header area.

2. Position the marker in the graphical display, then click to position the marker.

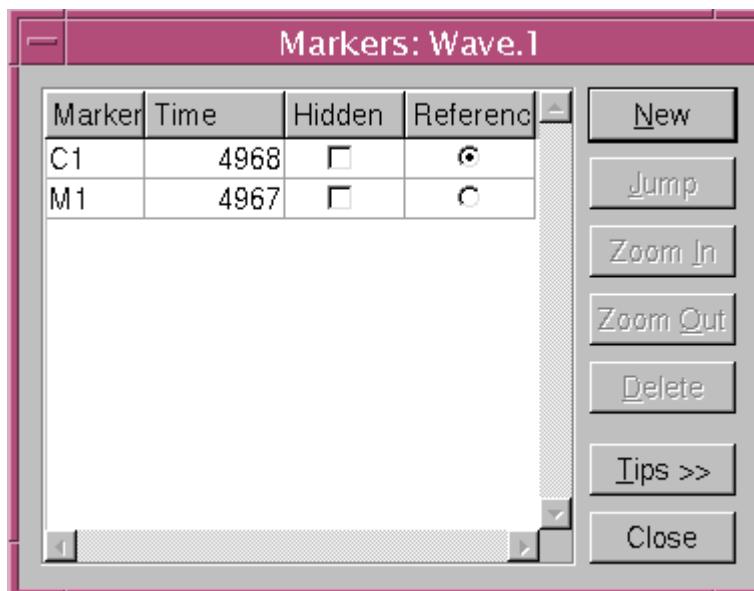
The marker annotation displays marker position and the delta between the marker and cursor



As you insert markers, DVE names them M1, M2, M3 and so forth.

3. Right-click a signal and select Markers.

The Markers dialog box appears:



4. Select the marker in the reference column.

The chosen marker is set as the reference marker.

5. Click the **Tips** button to expand the dialog box to display help for the Markers dialog box.

Extracting State Name

When you assign the value of macro/parameter/constant to a signal, you can see the state name of the macro/parameter/constant instead of the values. This is useful as the string of FSM state is easier to understand than the values.

To display the names of macro, parameter, or constant

1. Invoke DVE with your design in which you have defined parameters, constants, or macros.

2. Select the module in the Hierarchy pane.

The signals appear in the Data pane.

3. Select the signals in the Data pane, right-click and select **Add to Waves**.
4. Right-click a signal in the Wave view and select **Set Radix > State Name**.

The names of the parameters, macros, or constant are displayed in the Waveform. You can select each signal and set state radix to display the signals' names instead of values.

To export the radix

1. Select a signal, right-click and select **Set Radix > Export to Radix**.

The Edit User-Defined Radix dialog box appears. The Radix Table Name is automatically filled with the signal name (format: "STATE.<signal name>") and lists all the names of parameters, macros, or constants.

2. Modify the existing radix, save it as user-defined, or export, as desired.

The Export to Radix option is enabled only after setting the state radix and when some relevant macro, parameter, or constant is found.

Example

Following is an example of the module where parameters and constants are defined and how their names appear in the Wave view.

test.sv

```
`define AA 1
module modu;
    int abc;
    parameter bb = 2;
    const int cc = 3;
    initial begin
        #10 abc = `AA;
        #10 abc = bb;
        #10 abc = cc;
        #10 abc = 4;
    end
    initial #60 $finish;
endmodule
```

To compile this example code, use the following commands:

```
vcs test.sv -debug_all -sverilog
simv -gui &
```

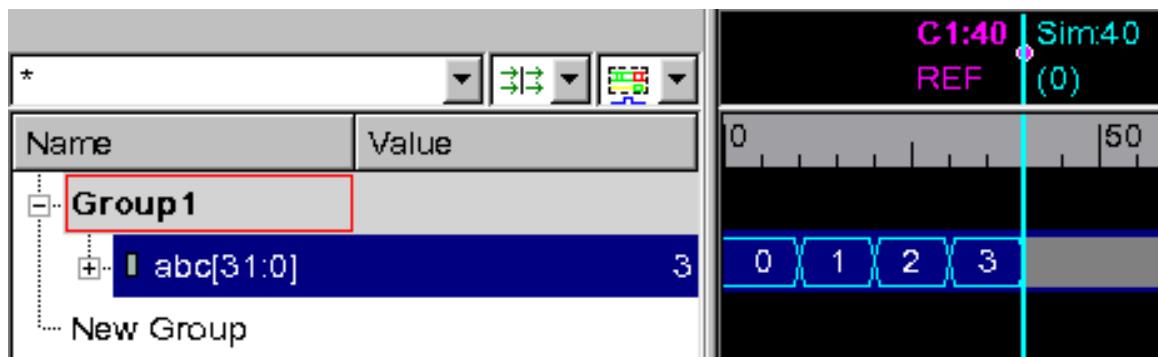
Figure 5-16 State Name

```

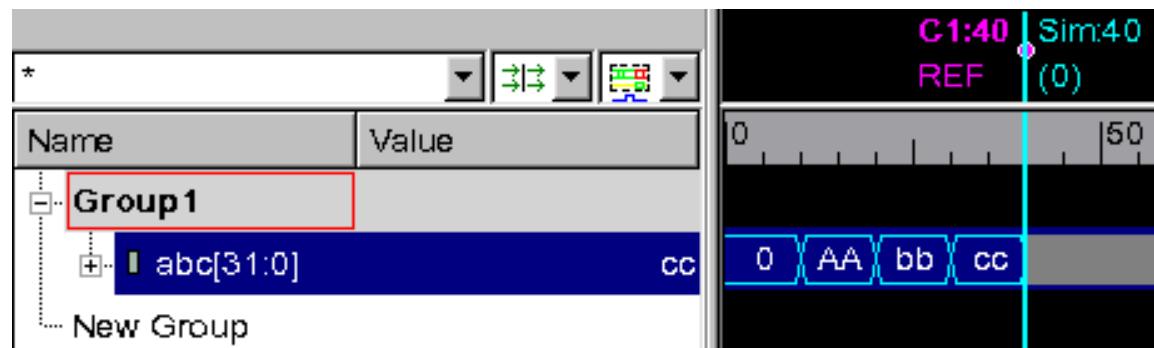
2 module modu;
3   int abc;
4   parameter bb = 2;
5   const int cc = 3;
6
7 initial begin
8   #10 abc = `AA
9
10  #10 abc = bb;

```

Value of signals seen in the Wave view



Name of signals seen in the Wave view
after selecting the State Name radix



Limitations

- In Local pane, only values are shown, and not state names. State

name information is not available for the current time step in all the views except Wave view and List view.

- In post-process debugging, state names of only macros and parameters can be seen, constants are not supported yet.
- Values are not replaced if state value is an expression containing macros/parameters/constant.

Zooming In and Out

You can zoom in to get a close-up view of your signal or zoom out to view the signal at its original size.

To choose zoom settings

1. Right-click in the Wave view and select Zoom.
2. Select the zoom options as desired.

Note:

For more information about the zoom options, see the section entitled, [View Menu](#).

Drag Zooming

You can drag the zoom to view specific transitions in a selected region of the timescale.

To drag zoom

1. Select any point in either of the timescales or in the waveform display area.

2. Hold down the left mouse button and drag the mouse to another time in the timescale.

The selected region is highlighted in blue.

3. Release the mouse button.

The signal is zoomed to the selected timescale.

Visualizing X at all Zoom Levels

You can visualize the X value at all zoom levels.

To visualize X value

1. Select a signal in the Signal pane with many value changes.
2. Zoom out the signal until the waveforms are condensed to yellow bar.
3. Right-click the signal and select **Highlight X Values** or select **Signal > Highlight X Values** from the menu.

The waveform is refreshed in the Wave view and X values are displayed as red color lines for all the zoom levels.

Expanding and Contracting Wave Signals

You can expand and contract the height of wave signals.

To expand and contract wave signals

1. Select **View > Increase Row Height**.

The signal is expanded.

2. Select **View > Decrease Row Height**.

The signal is contracted.

You can also right-click, select **Properties** and use the Signal Properties dialog box to increase or decrease row height.

Searching Value or Edge of Signal

When searching for values or edges of signals, only the values or edges of the selected signals will be searched. If no signals are selected, the values or edges of all the signals are searched.

To search for values or edge of signal

1. Select a signal in the Wave view.
2. Select the search constraint in the **Selects Search Criteria** list box.
3. The following are the search criteria:
 - Any Edge — Searches for any signal edge.
 - Rising — Searches for signals with rising edge.
 - Falling — Searches for signals with falling edge.
 - Failure — Searches for assertion failure.
 - Success — Searches for assertion success.
 - Match — Searches for match in compare results.
 - Mismatch — Searches for mismatch in compare results.
 - X Value — Searches for any value that contains X value.

- Value — Searches for specified signal value.
4. Enter the number of seeks in the **Set Number of Seek**s field.
 5. If you select "Value" as the search criteria, then Value Search dialog is displayed.
 6. Enter the value of the signal to be searched and click **OK**.
 7. Click the **Search Forward** and **Search Backwards** arrows in the toolbar.



The C1 cursor moves from its current location to the next or previous values as per the search criteria.

Shifting Signals

You shift a signal by creating a new signal based on a time shifted signal.

To shift a signal

1. Select a signal in the Wave view.
2. Select **Signal > Shift Time**.
3. The Shift Signal dialog box appears..



4. Enter the following information, as appropriate:
5. Time Offset — Specifies offset for shifting signals. A positive Time Offset shifts the signal to the right. A negative number shifts the signal to the left in the Wave view.
6. Signal Name(s) — Identifies the name of the selected signal. This field appears dimmed.
7. New Name(s) — Sets new signal name (alias) for shifted signal. This is only supported for a single signal. The signal displays with the original signal name followed by the time offset. In the previous figure, it is test1.risc.daata(7:0) ->>10.
8. Keep original signal — Keeps the original signal, if selected.

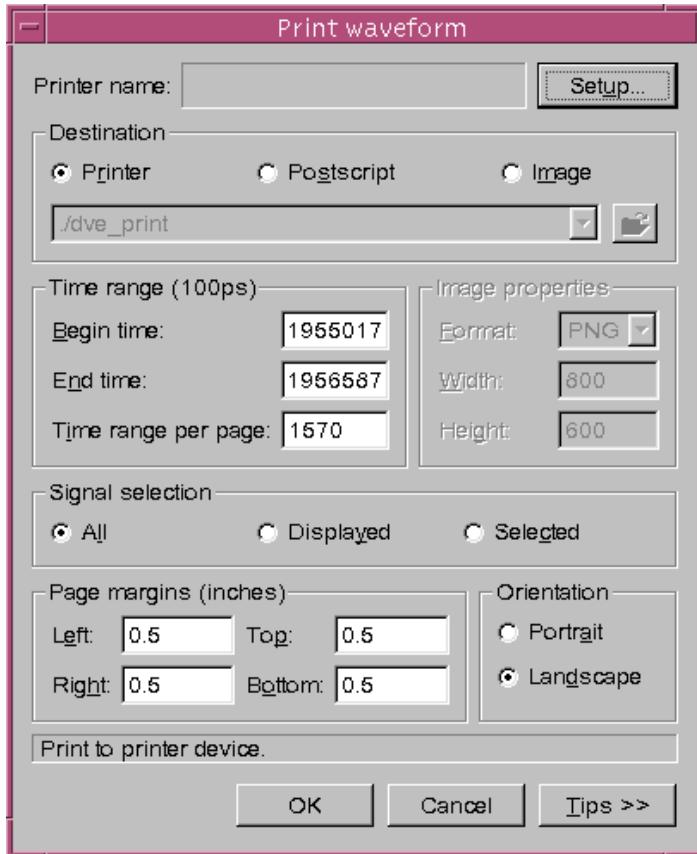
Printing Waveform

You can print waveforms to a file or printer from an active Wave view selecting time range and signals to print.

To print waveform

1. From an active Wave view, select **File > Print**.

The Print Waveform dialog box opens.



2. Click the Setup button to set printing options.

The Setup Printer dialog box appears.

3. Select the following options in the Setup Printer dialog box, as appropriate:
 - Print destination — Select Print to printer or Print to file.
 - Printer Settings — Select Print in color or grayscale.
 - Paper format — Select paper format, landscape or file, and paper size.

- Options — Select from the available options, such as range and number of copies etc.
4. Click **OK**.
- The print options appear in the Print Waveform dialog box.
5. Select the print destination.
 6. Select the Time range to save the waveform.
 7. Select the image properties if you want to save the waveform as image.
 8. Select whether to print **All**, **Displayed**, or **Selected** signals.
 9. Select the page margins.
 10. Click **OK**.

The waveform is printed. You can use the **Tips** button to view detail description of each of the options available in the Print Waveform dialog box.

Viewing PLI, UCLI, and DVE Forces in Wave View

Force and release in PLI or UCLI will be represented in DVE waveform, with special symbols. For more information, see “[Active Drivers Support for PLI, UCLI, and DVE Forces](#)” .

6

Using the List View

You can display data in the List view in the same way as you display in the Wave view. The List view displays simulation results in tabular format. For Verilog, the List view displays nets and register variables. For VHDL, it displays signals and process variables.

This chapter includes the following topics:

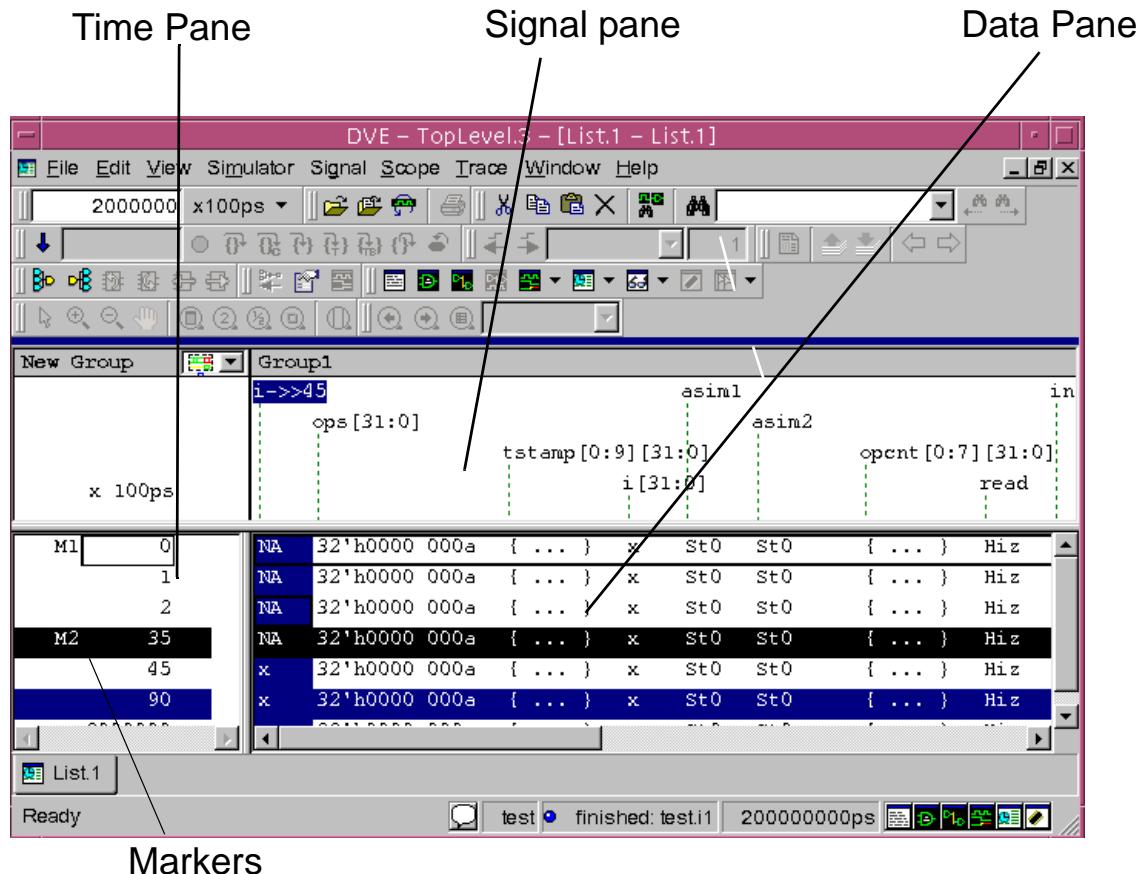
- “The List View”
- “Viewing Simulation Data”
- “Using Markers”
- “Setting Signal Properties”
- “Comparing Signals”
- “Saving a List Format”

The List View

The List view contains three panes:

- The Signal pane displays signal names as headers above the simulation data.
- The Data pane displays simulation results in tabular format.
- The Time pane displays simulation time value.

Figure 6-1 The List view



You can do the following tasks in the List view:

- View simulation data
 - Create and delete markers
 - Set signal properties
 - Associate signal with any database
 - Compare signals
 - Save signal values
-

Viewing Simulation Data

To view simulation data in the List view

1. Drag and drop scope or signal from any of the DVE panes or views.
 2. Use the bottom scroll bar to move left and right and view signals and their values.
 3. Use the right scroll bar to move up and down through simulation time.
 4. Select a signal in the signal pane to highlight the signal values.
-

Using Markers

You can create markers in the List view to speed up navigation.

To create and delete markers

1. Select a time unit in the Time pane, right-click and select **Markers**.
The Markers.List dialog box opens.

2. Click **New** to create a new marker in the list table.
3. Select the Time cell for the new marker and enter the time at which to set the marker.
4. Click **Hidden** if you don't want to display the marker in the Data pane, then click **Return**.
5. Repeat steps 2 to 4 to create more markers.
6. Select a marker in the Markers.List dialog box, then click **Jump**.
The selected marker is displayed in the Time pane.
7. Select a marker and click **Delete** in the Marker.List dialog box.
The marker is deleted from the list.

Setting Signal Properties

To customize signal display, you set signal properties for individual signals.

To set the signal properties

1. Select a signal in the Signal pane.
2. Select **Signal > Signal Properties**.
The Signal Properties dialog box opens.
3. Enter the number of characters for the selected signal value column width.
4. Select whether a signal value change triggers a new line of values in the Data pane or not.
5. Click **Apply** to make the change and keep the dialog box open.

Or

Click **OK** to apply the changes and close the dialog box.

Comparing Signals

You can compare signals in the List view similar to the way you compare signals in the Wave view.

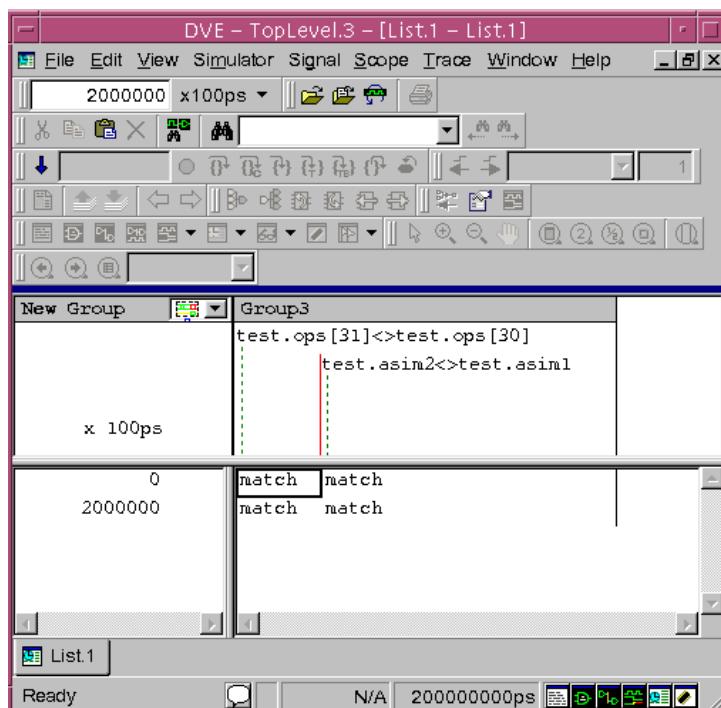
To compare signals

1. Select signals on the Signal pane.
2. Right-click and select **Compare**.

The Waveform Compare dialog box opens.

3. Select the options, as appropriate.
4. Click **OK**.

The comparison results are displayed in the current List view.



For more information about the Waveform Compare dialog box, see the topic “[Comparing Signals, Scopes, and Groups](#)” on page 28.

Saving a List Format

After you have customized the display in the List view, you can save the format for future use.

To save the list format

1. Select **File > Save Values**.

The Dump Text List dialog box opens.

2. Select the format, tabular or event based.

The event based format will save the data on the basis of simulation time value.

3. Enter a filename with a .tcl extension.
4. Select **Save**.

The list format is saved.

7

Using Schematics

This chapter includes the following topics:

- “Overview”
- “Viewing Schematic”
- “Opening a Path Schematic View”
- “Finding Signals in Schematic and Path Schematic View”
- “Back Tracing”
- “Printing Schematics”

Overview

Schematic views provide a compact, easy-to-read graphical representation of a design. You can view a design, scope, signal, or group of selected signals and select ports to expand connectivity in relevant areas. You can explore the design behavior by analyzing the annotated values for ports and nets.

There are two types of schematic views in DVE:

- Design - a design schematic shows the hierarchical contents of the design or a selected instance and allows you to traverse the hierarchy of the design.
- Path - a path schematic is a subset of the design schematic that displays where signals cross hierarchy levels. Use the path schematic to follow a signal through the hierarchy and display portal logic (signal effects at ports).

Viewing Schematic

When viewing the schematic, use the scroll bars to move up and down and left and right in the displayed graphics. You can also use toolbar and menu commands to select parts of the design to zoom in, copy, drag and drop into another DVE window, move one level up to a parent or definition. You can also add signals from the Schematic view to the Wave view, List view, or Source view.

When the Schematic view is opened, the libmdb.so file is created in the /tmp directory by default. You can specify an alternate location with the environment variable DVE_MDB_TMPDIR as follows:

```
setenv DVE_MDB_TMPDIR <some directory>
```

where, <some directory> is the location where you want to create the libmdb.so file.

To customize the schematic display

1. Set the maximum number of cells in the schematic.
2. Change the text style and size displayed on your schematic.
3. Change the visibility and colors of cells, hierarchical crossings, nets, buses, ports, pins, and rippers.

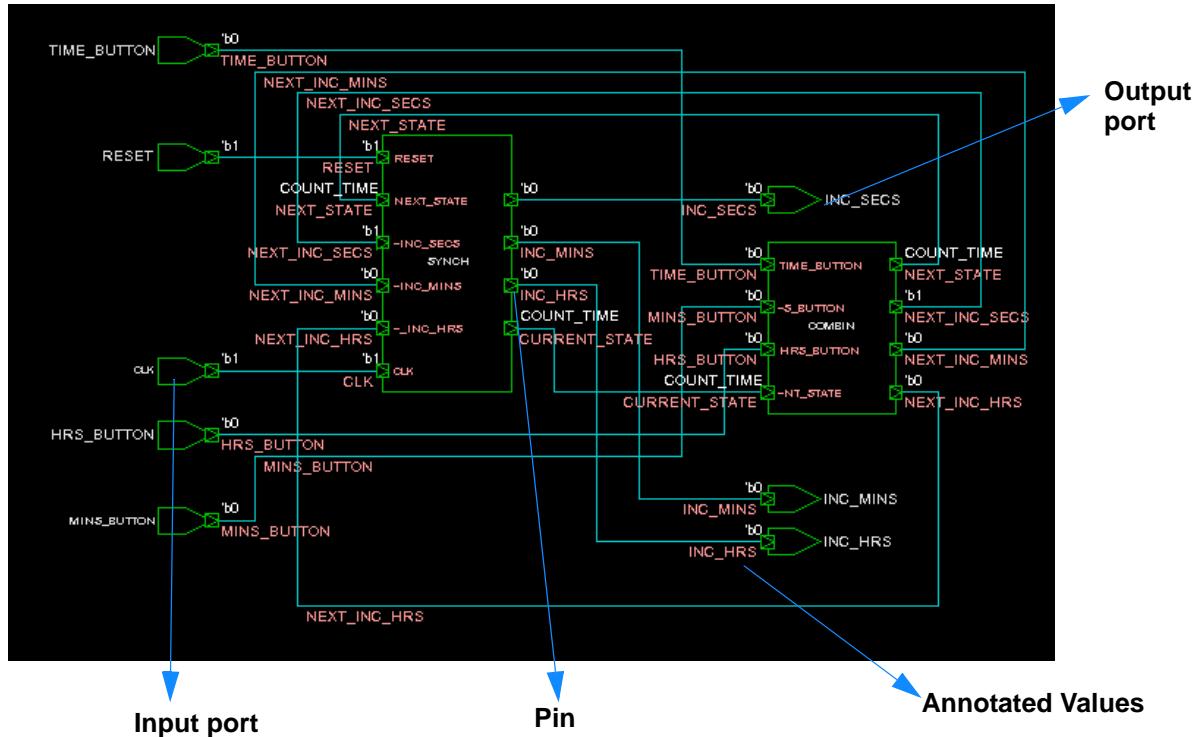
Opening a Design Schematic View

To view a schematic of a design in DVE, you must generate VPD on the same or similar platform with VCS using one of the following debug compile options: -debug_pp, -debug, or -debug_all.

To open a design schematic

1. Select an instance from the Hierarchy pane, right-click and select **Show Schematic**.

The Schematic view displays the connectivity in the selected instance.

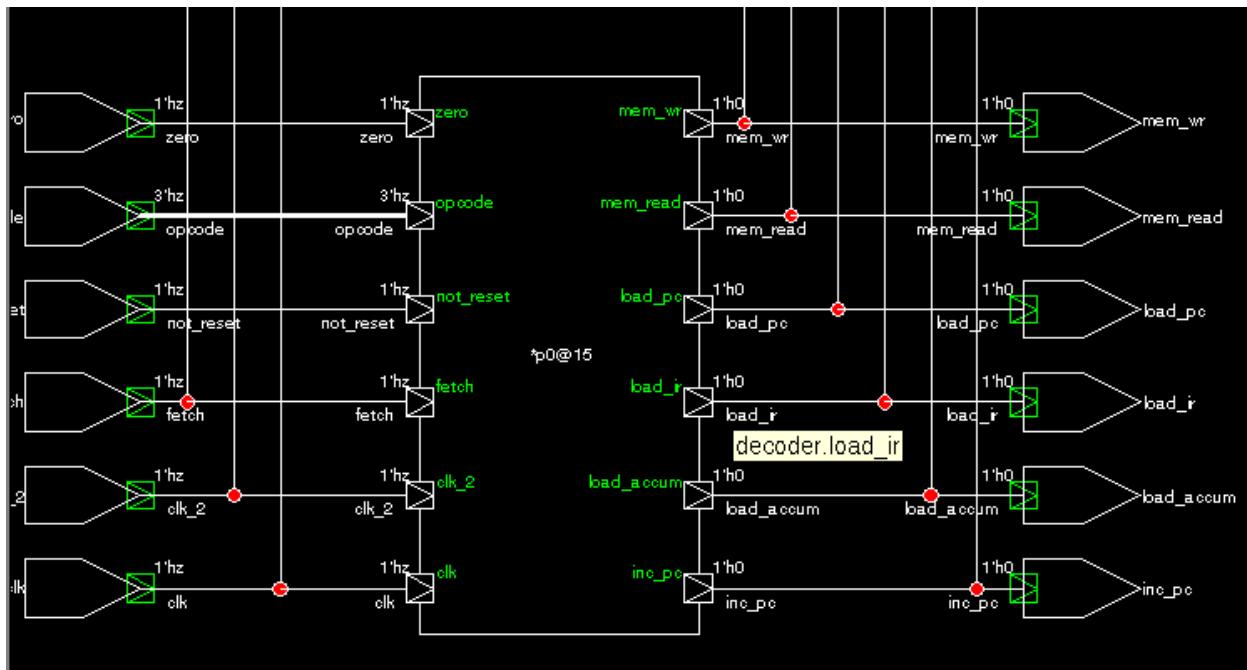


You can also open the Schematic view in the following ways:

- From the CSM of Source view, Data pane, Memory view, Wave view, and List view.
- From the menu Scope > Show Schematic.

Annotating Values

To view the annotated values of a signal, from the **Scope** menu, select **Annotate Values**.



If you hold the cursor on a signal, a ToolTip identifies the signal as shown in the diagram.

Making Modules as Black-Box

You can treat certain modules defined within the celldefined, uselib, or -y/-v modules in the design as black-box cells. When you define certain cell as black-box, you cannot view the schematic or path schematic of that module that is the schematic will be hidden.

To make modules as black-box

1. Select a module in the Hierarchy pane, right-click and select **Show Schematics**.

The Schematic view displays the connectivity in the selected module.

2. Select a module you want to treat as black-box.
3. From the **Edit** menu, select **Preferences**.

The Application Preferences dialog box appears.

4. In the Design Debug category, select the **Treat modules defined within 'cell defines as black box (library) cells** or **Treat modules defined within 'uselibs or -y/-v as blackbox (library) cells** check boxes and click **OK**.

The selected module is now defined as black-box. When this option is selected, all the cells which are declared under `celldesign are considered as blackboxes in both the schematic and path schematic windows.

The changes take effect the next time you generate the schematics. When you click the black-box modules, the schematics for the module is not shown and the control shifts to the Source view.

Mapping Symbols in Schematic

Each installation of VCS comes with a default symbol DB file, generic.sdb. The location of this file is \$VCS_HOME/gui/dve/libraries/syn/vcs.sdb.

This file contains generic symbols with all Verilog standard logic gate symbols. When DVE gets a cell-name in your design, it searches the generic.sdb file for a matching symbol definition. If the match is found, DVE retrieves symbol mapping information for this cell, or else DVE displays the cell instance as a rectangle (default representation).

Instead of using these default symbols, you can create your own symbols according to their functionality in the design and store them in the .db or .sdb file. This section explains how you can map your own symbols from the .db or .sdb files in your Schematic view.

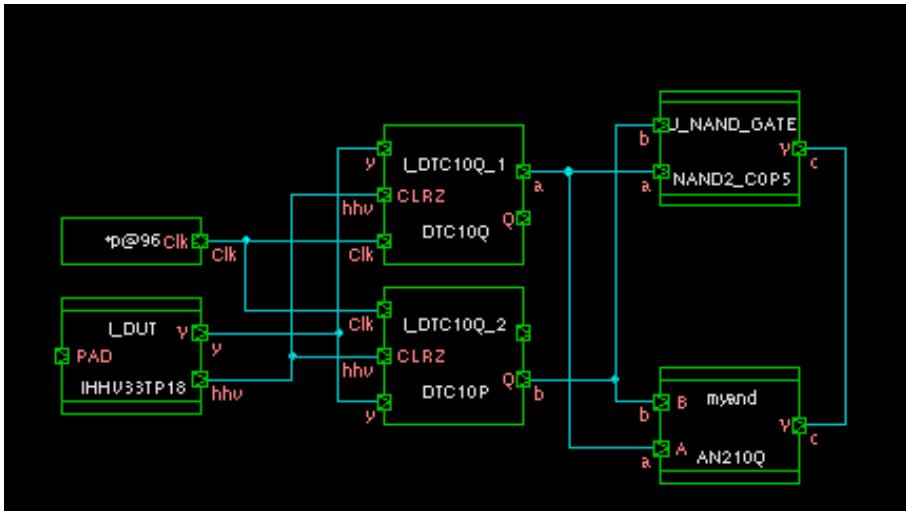
To use the symbols stored in .db or .sdb files

1. Run the design located in \$VCS_HOME/doc/examples/debug/schematic_symbol_mapping using the following commands:

```
%comp.csh  
%run.csh
```

2. Select the module in the Hierarchy pane in DVE, right-click and select **Show Schematics**.

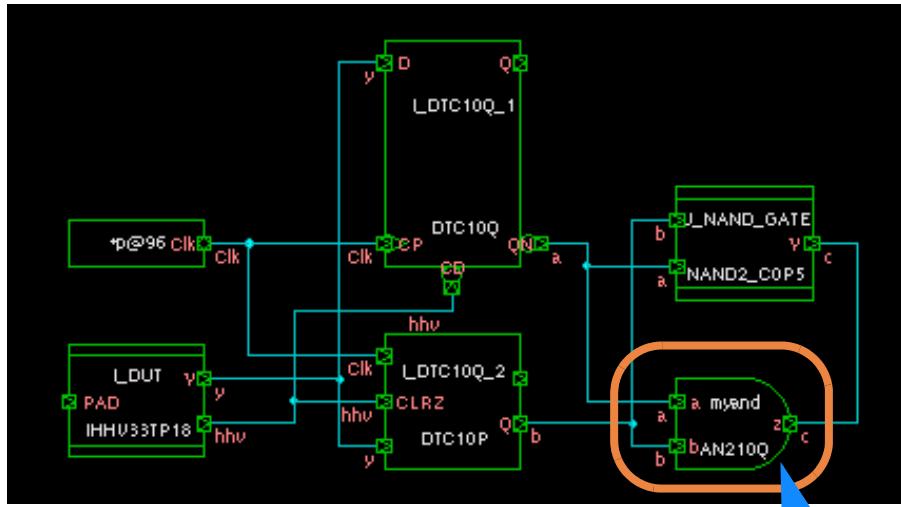
You see the following schematic.



3. Enter the following command in the DVE Console pane to map the .db or .sdb file:

```
gui_sch_set_symbol_libraries -files {*.db} -dirs {SYM}
```

The Schematic view changes as follows:



Note the orange marked symbol (NAND gate) that has changed.

Alternate Ways of Mapping the .db/.sdb Files

- Specifying files and search paths:

```
gui_sch_set_symbol_libraries -dirs { $HOME/my_dbs/  
remote/company/libs }  
gui_sch_set_symbol_libraries -files { lib1.sdb lib2.sdb  
lib3.db }
```

Here, the lib1.sdb, lib2.sdb and lib3.db files are used to display the design, if these files are found either in ".", \$HOME/my_dbs or /remote/company/libs.

In addition, the commands also accepts the same convention as the Design Compiler shell search_path and symbol_library variables:

- if the name of the symbol files already contains a path, then this name is searched only from the ".directory", the other searched paths are ignored.
- names of .db or .sdb files can contain wildcard (globbing), which means that * and ? are accepted in the name.
- Specifying multiple .sdb or .db files:

```
gui_sch_set_symbol_libraries -dirs { /depot/libs } -files  
{ *.sdb *.db }
```

- Using a library file from a given directory even if another version exists in other search path:

```
gui_sch_set_symbol_libraries -files { file1.db file2.db  
$HOME/file3.db }
```

Where, file1 and file2 is searched in search_path; file3.db is picked up from \$HOME even if it exists in the search path.

Note that two consecutive calls to
gui_sch_set_symbol_libraries -dirs or
gui_sch_set_symbol_libraries -files overrides the
setup (only the last call is used).

In order to make it permanent, you can add these
gui_sch_set_symbol_libraries command in the \$HOME/
.synopsys_dve_usersetup.tcl file (or any .tcl script used
during initialization of DVE in the user environment.

Generating .db or .sdb Files

The *.db file or *.sdb is generated by the synthesis tool, Design Compiler as follows:

```
dc_shell> read_lib my_lib.lib  
dc_shell> write_lib my_lib.sdb
```

For more information about the Synthesis Tools suite, see SolvNet at <https://solvnet.synopsys.com>.

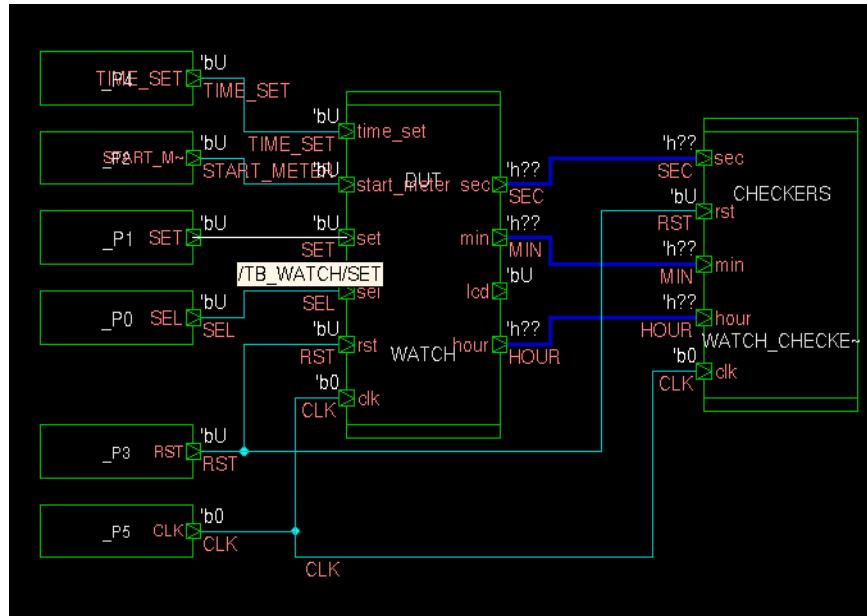
Opening a Path Schematic View

A path schematic is a subset of the design schematic displaying connections that cross hierarchical boundary.

To open a path schematic view

1. Open a design schematic view of an instance containing the hierarchical crossings of interest.
2. When you have identified the instance to display, click on the instance to select it.

The color change indicates that it is selected.



Note:

You can also drag the selection cursor over multiple objects to select multiple items.

3. Right-click and select **Show Path Schematic** or click the following icon in the toolbar to view a path schematic in a new window:



The path schematic for the selection is displayed:



Displaying Connections in a Path Schematic

With a path schematic displayed, you can add the logic fanin to, or logic fanout from, specified objects in the schematic across specified levels or the entire design.

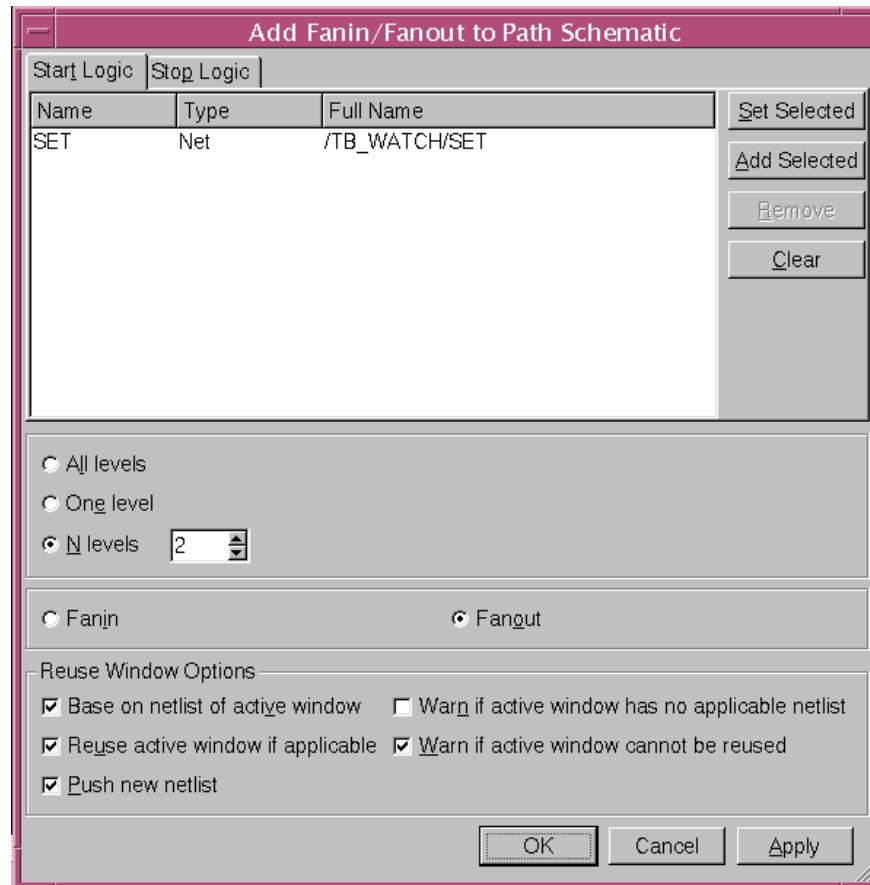
To display connections

1. Select an object in the path schematic.

The change in color confirms the selection.

2. Select **Scope > Add Fanin/Fanout**.

The Fanin/Fanout to Path Schematic dialog box opens.

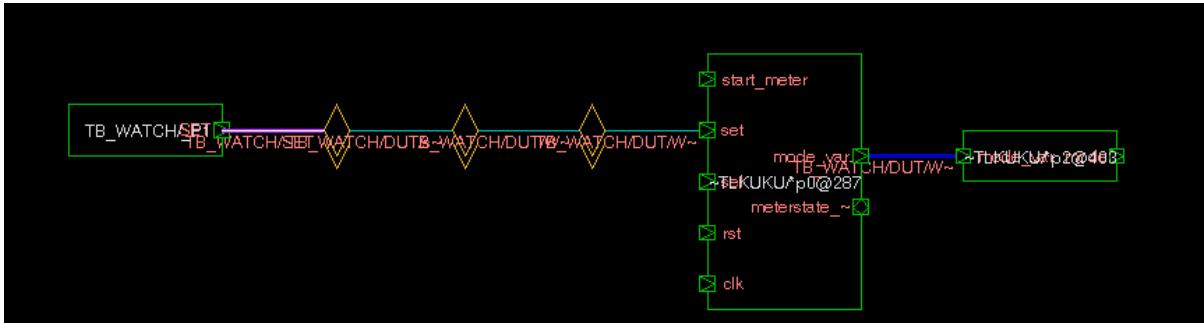


3. Click **Set Selected** to add the selected objects to the list box.

You can optionally select more objects and use the **Add Selected** button to add them to the list.

4. Set the other options, such as the number of logic levels to be added and the Reuse Windows options.
5. Click **OK**.

The schematic is updated with additional fanin or fanout logic. You can also view the signal values.



Compressing Buffer and Inverter in Schematic

If there are different kinds of buffers or inverters in the design, you can choose to compress and view them as one buffer/inverter in the path schematics.

For example, instead of having five buffers (5 instance symbols with wires between them), you can compress and have only one buffer in the schematic. The compressed buffer is shown with // symbol in the schematic. An odd number of inverters is shown with the /o/ symbol.

The buffer or inverter compression can be enabled from the Application Preferences dialog box. You can view the compressed buffers and inverters only in the next generated Schematic views.

When compression is activated, tracing a fanin/fanout on a path, will trace to the next cell, which is not a buffer or inverter. The compression considers the hierarchy crossings. If the next level of hierarchy has only one buffer or inverter, then it is also compressed. For example,

In the netlist:

```
xxx_inst: INVxxx(...)
```

In the module:

```
module INVxxx (...)  
    not instname(...)  
end module
```

--v--not--^--v--not--^--v--not--^--v--not--^--

will be compressed to:

--//--

Here, "v" is a hierarchy crossing down and "^" is a hierarchy crossing up.

This compression is independent of black-boxing. So, if the buffers or inverters are in black-box modules, they are still valid for compression. The syntax for describing hierarchical/nested cases is as follows,

- "-not-not-" means two primitive Verilog inverters are on the same hierarchy level connected directly.
- "-not- [not] -not-" means the inverter in the middle is one level down in the hierarchy.

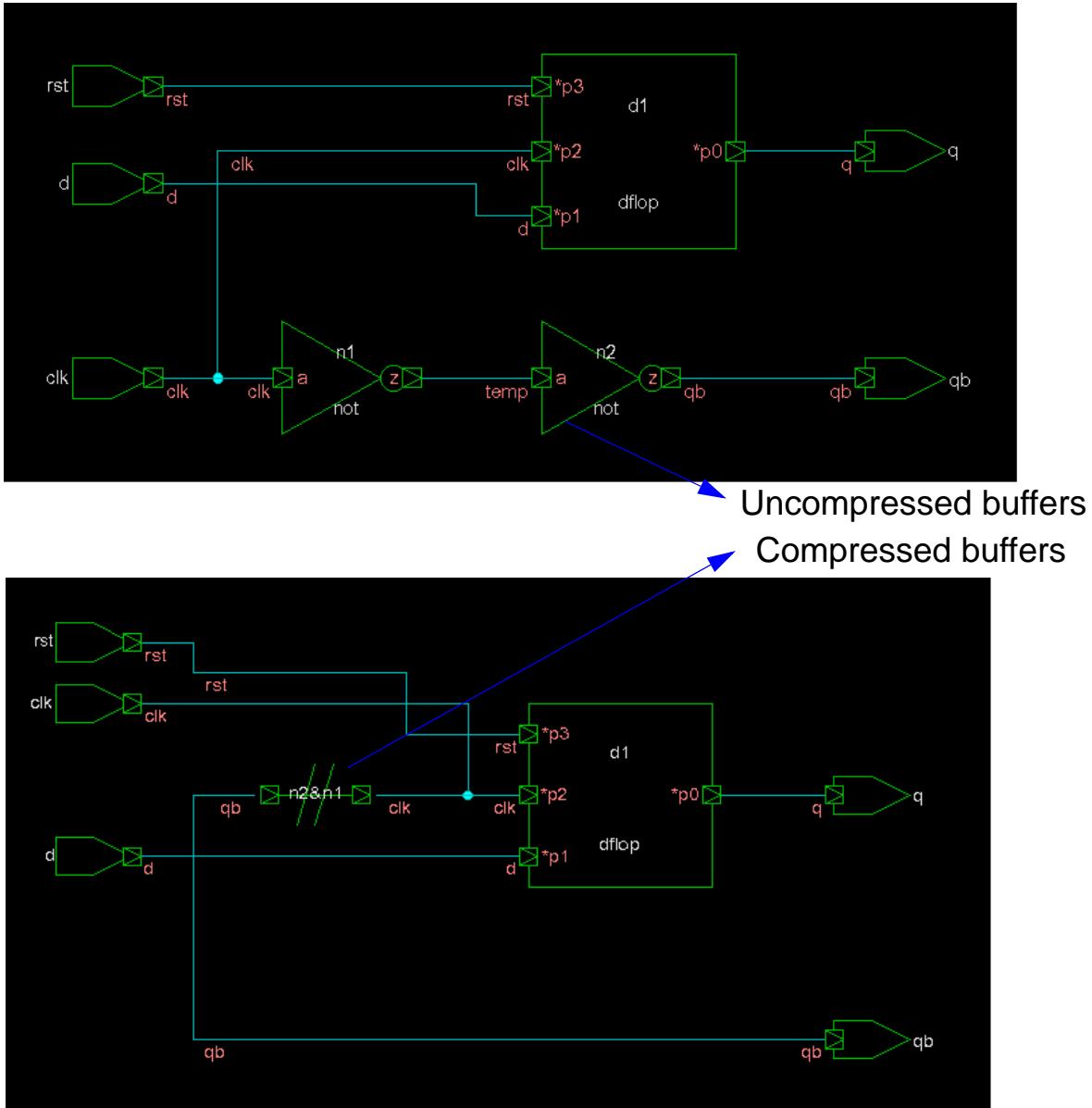
The following table shows the cases of compression.

Table 7-1 Cases of compression

Buffer/inverters	Compressed symbols
-not-not-	-//-
-not-not-not-	-/o/-
-buf-buf-	-//-
-buf-buf-buf-	-//-
-not-buf-not-	no compression
-[not]-[not]-	-//-
-not-[not]-not-	-/o/-
-not-not-[not]-not	-//-
-not-[not]-[not]-not-	-//-
-not-[not-not]-not-	-//-
-not-[not-[not]]-not-	-//-
-not-[not-and]-	-//-and-

To enable the buffer and inverter compression, select the check box **Enable buffer (buf) and inverter (not) compression under the Schematic View category in the Application Preferences dialog box.**

The following illustrations show the uncompressed and compressed schematics. The buffers n1 and n2 are compressed and the symbol // is used to represent them.



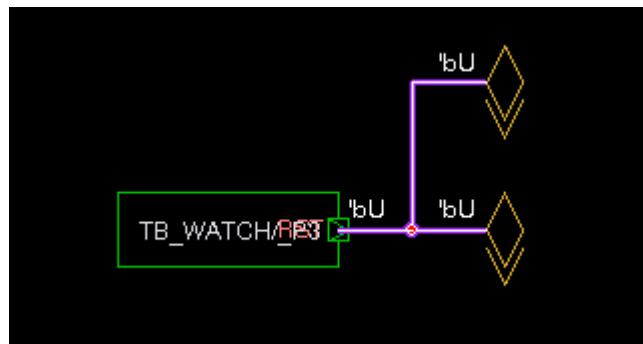
You can view the number of buffers by placing your mouse over the symbol.

Following a Signal Across Boundaries

You can select a signal and follow it across hierarchical boundaries in the Path Schematic view.

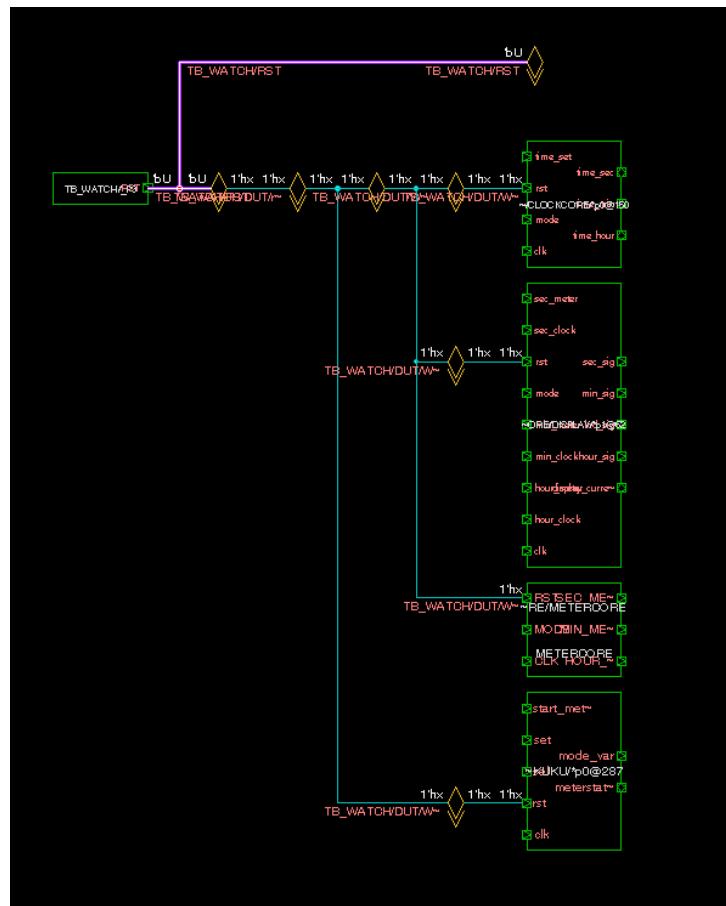
To follow a signal

1. Select a signal or signals, right-click and select **Show Path Schematic**.
2. Select a signal in the Path Schematic view.



3. Right-click and select **Expand Path** from the CSM.

The signal is highlighted in the path view.



Finding Signals in Schematic and Path Schematic View

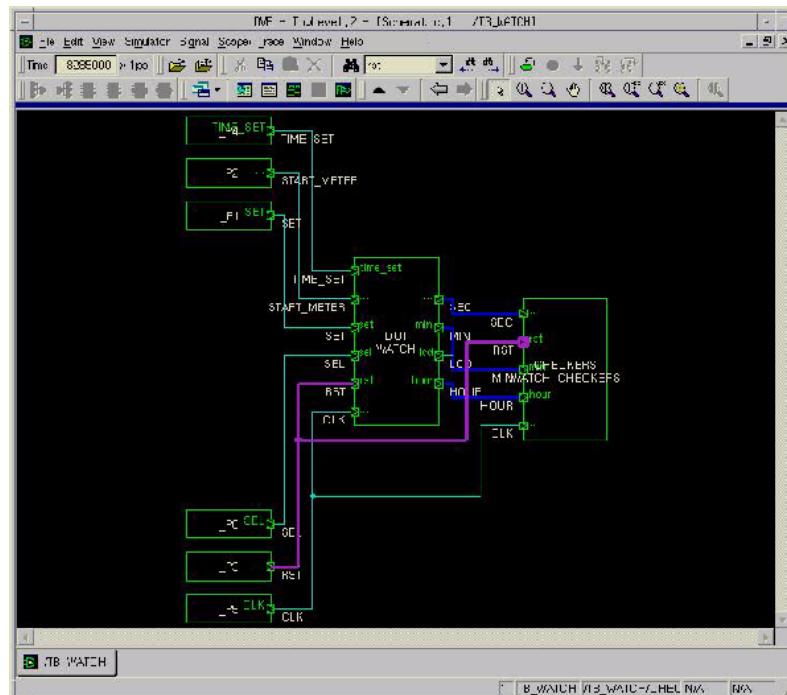
Highlighting Signals

You can select one or more signals to trace in a Schematic or Path Schematic view. With this option, the selected signals are highlighted based on the specified colors.

To highlight a signal

1. Select **Trace > Highlight**, then select **Set Current Color**.
2. Select a color for the highlight.
3. Click **OK**.

The signal is highlighted with the selected color.



Searching for Signals

You can use the Find toolbar option to search signals.

To search for signals, enter the signal name in the Find toolbar box in the Schematic view, then click the **Find Next** toolbar button.

The signal is highlighted in the schematic.

Showing Value Annotation

You can use DVE preference option **Show Value Annotation** to view the value annotation in the Schematic and Path Schematic views even after you restart DVE. You need to save the preference settings before restarting DVE to retain the value annotation.

To save the value annotation and view it upon restart

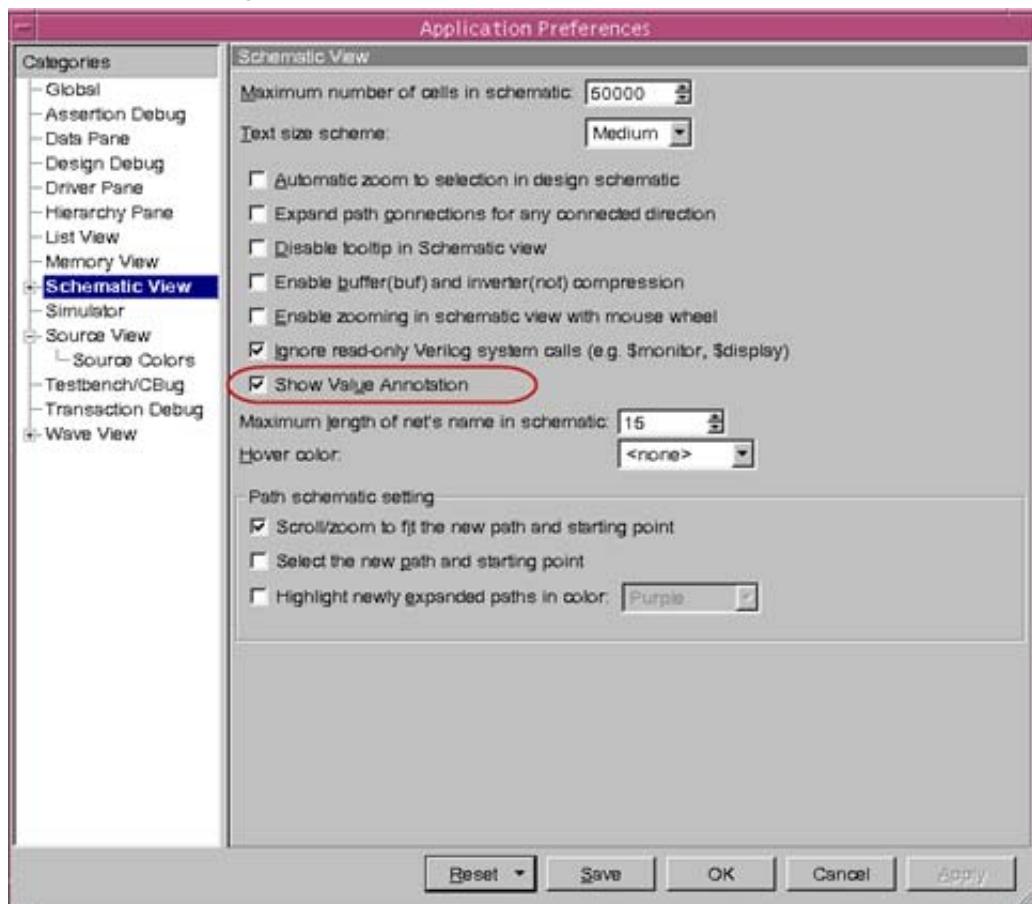
1. Select **Edit > Preferences**.

The Application Preferences dialog box opens.

2. In the Schematic View category, select **Show Value Annotation** and click **Apply**, as shown in [Figure 7-1](#).
3. Click **OK**.
4. Select a scope from the Hierarchy pane, right-click and select **Show Schematics** or **Show Path Schematics**.

The Schematic or Path Schematic view is shown with values of the signals.

Figure 7-1 Selecting 'Show Value Annotation' Preference Option



5. Exit DVE.

A message prompts you to save the preference settings.

6. Click **Yes**.

The preference settings are saved.

7. Open DVE again and view the Schematic or Path Schematic.

The annotated values are still visible.

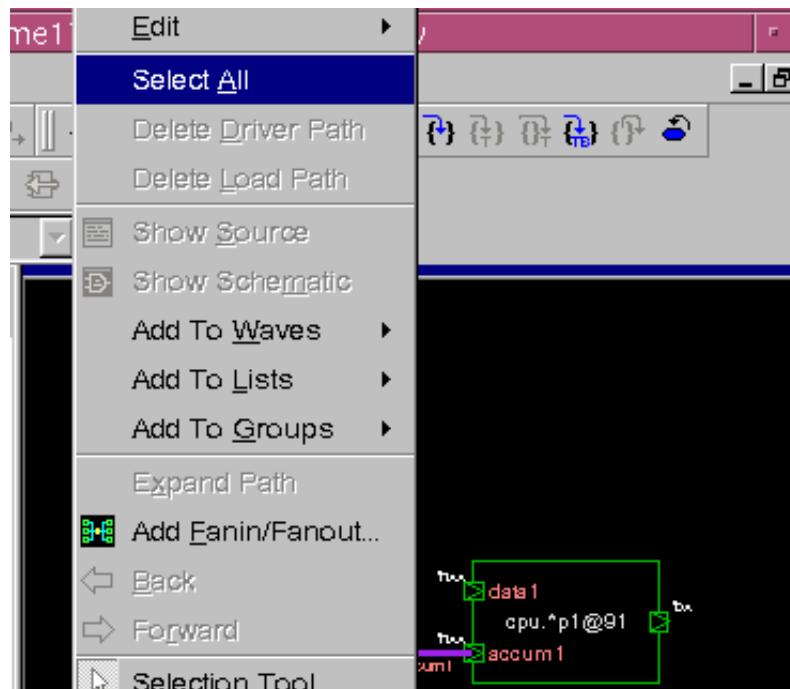
Selecting and Deleting All Objects from Path Schematic View

You can select and delete all the objects at once from the Path Schematic view without closing the view.

To select and delete all the objects from the Path Schematic view

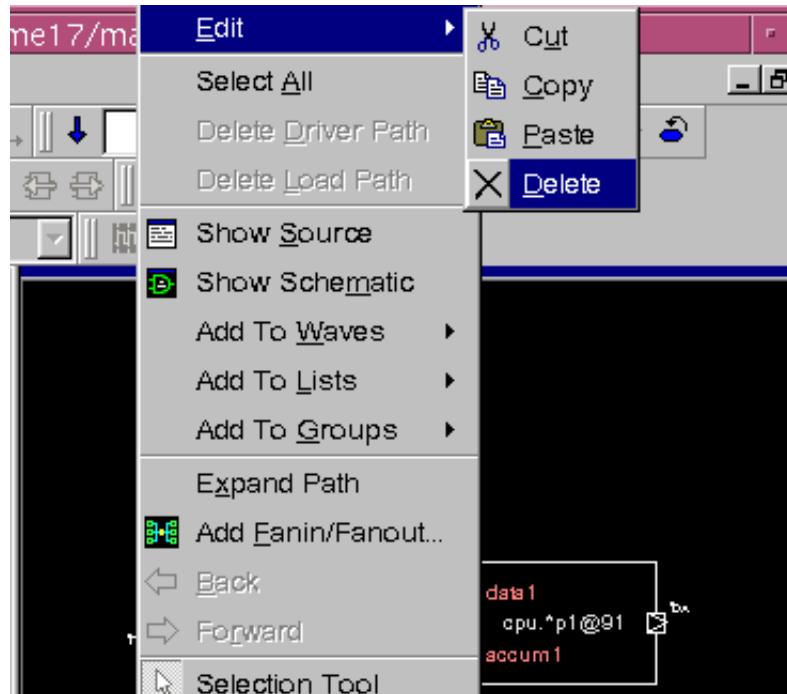
1. Right-click on the Path Schematic view and select **Select All**, as shown in the following figure.

All the objects are selected.



2. Right-click on the Path Schematic view and select **Edit > Delete**, as shown in the following figure.

All the selected objects are deleted.



Back Tracing

Back Tracing helps you debug a particular signal that has a value of X by traversing the design backwards both structurally and temporally. You can back trace an X value to its source signals, for example, across gates to identify the signal that caused the X value.

Note:

- Trace X, an older version of this feature, is now replaced with Back Tracing.
- It is recommended that you use the Back Trace feature on a gate-level design. If you use it on a non-gate level design, then it will not Back Trace sufficiently to be useful.

- Back Trace may stop tracing for various reasons (for example, multiple X's at the input of a cell). If this happens, you need to manually select the input pin to continue tracing.
- Even for gate-level designs, Back Trace may not be able to automatically trace through flip-flops. You may need to manually select the input of the flop, that is X, to continue tracing.
- Back Trace requires all values in the fanin cone of logic to be dumped to produce the correct (or complete) tracing result.

Back Tracing in DVE is performed in the Back Trace Schematic view. You can invoke the Back Trace Schematic view by selecting any signal from any of the DVE view or pane.

The Back Trace Schematic consists of two views - Wave View and Path Schematic. The Path Schematic view is the main structural view. The Wave view provides a temporal view and provides information to decide which signals need further tracing. You can close the Wave view if not needed.

Example

The Back Trace feature is explained using the following example.

test.v

```

`timescale 1ns/1ps
module top;
reg EN,ENB,hv_Input;
wire lv_Output;
test2 inst (lv_Output, EN,ENB,hv_Input);

initial begin
$vcndlplus();
EN = 0;
```

```

ENB = 1;
#4 EN = 1;
ENB = 0;
#6 hv_Input= 1;
#10 EN = 0;
#10 ENB = 1;
#1 hv_Input = 1'bx;
#6 hv_Input = 1;
#10 ENB = 0;
#10 EN = 1;
#10 ENB = 1;
hv_Input = 1'bx;
#10 ENB = 0;
hv_Input = 1;
#100 $finish;
end

endmodule

module test2 (lv_Output, EN, ENB, hv_Input);
    output lv_Output;
    input EN,ENB,hv_Input;
    wire lv_Output1;
    wire int_fwire3, int_fwire4;

    nand #1(int_fwire4, EN, ENB);
    xor(int_fwire3, lv_Output1, hv_Input);
    and (lv_Output,int_fwire3,int_fwire4);

    test1 test_inst(lv_Output1,EN,int_fwire4,hv_Input);
endmodule

module test1 (lv_Output,EN, ENB, hv_Input);
    output lv_Output;
    input EN, ENB, hv_Input;
    reg lv_Output;
    wire int_fwire_0, int_fwire_1;

    and (int_fwire_0, EN, ENB);
    and (int_fwire_1, EN, hv_Input);
    or (lv_Output, int_fwire_0, int_fwire_1);

```

```
    always @(lv_Output)
        $display("Value for lv_Output is :%b At time
%t",lv_Output,$time);
endmodule
```

Compile this example using the following command:

```
vcs -debug -sverilog test.v -R
```

You get the following simulation output:

```
Value for lv_Output is :0 At time 0
Value for lv_Output is :1 At time 4000
Value for lv_Output is :0 At time 20000
Value for lv_Output is :1 At time 57000
Value for lv_Output is :x At time 68000
Value for lv_Output is :1 At time 77000
```

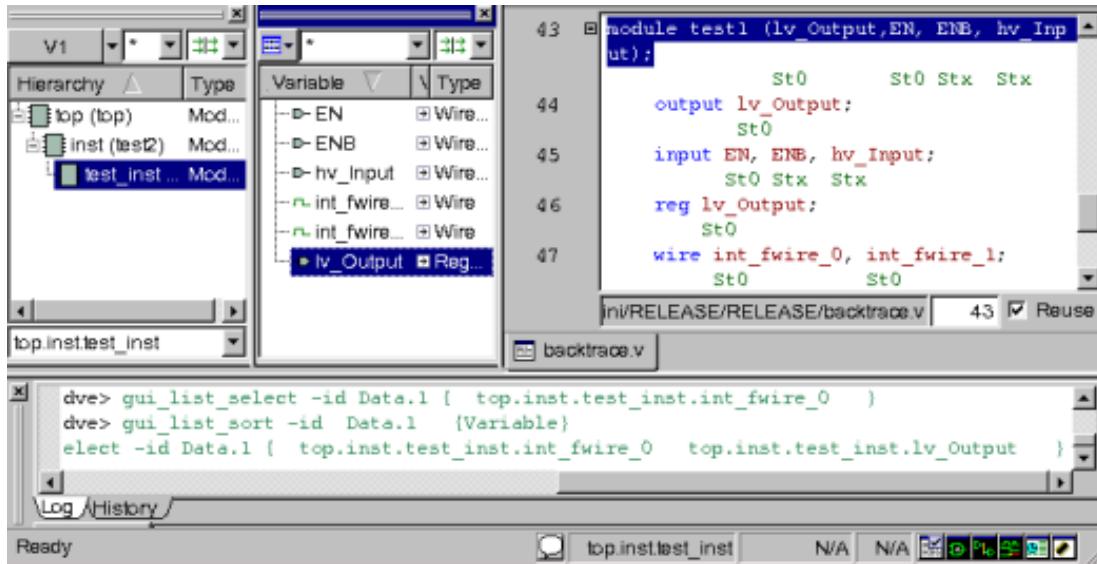
Note that the signal **1v_Output** has a value "x". Load the VPD file in DVE using the command:

```
dve -vpd vcdplus.vpd
```

DVE opens and you can see the design file test.v loaded.

To back trace the signal with x value

1. Select the module **top.inst.test_inst** from the Hierarchy pane.

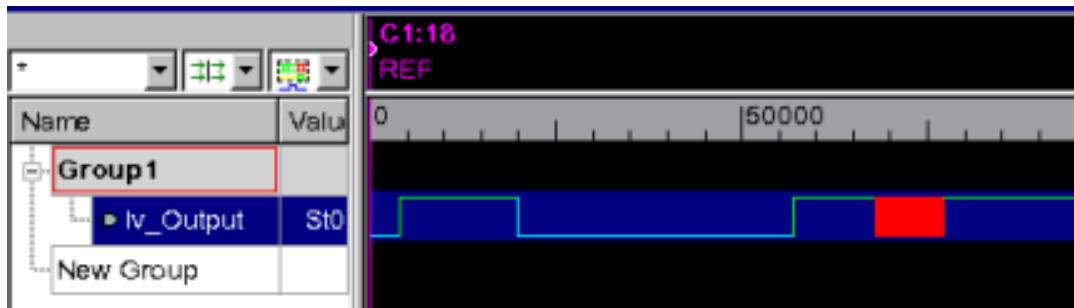


The variables are shown in the Data pane.

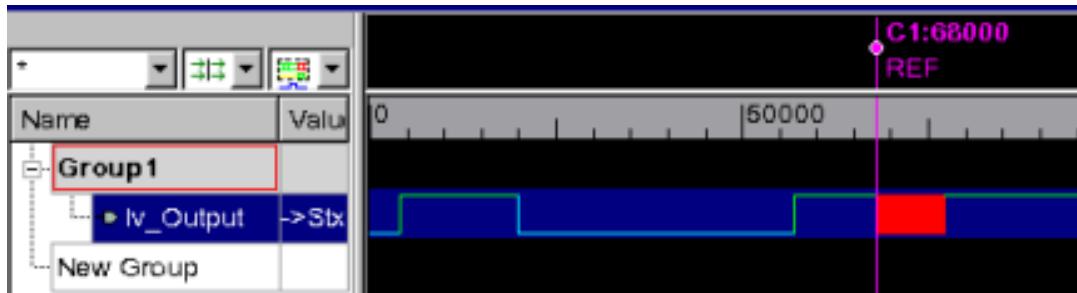
2. Select the variable **lv_Output** from the Data pane, right-click and select **Add to Waves > New Wave view**.

The signal is added to the Wave view.

3. Select the signal in the Wave view, right-click and select **Zoom > Zoom full**.

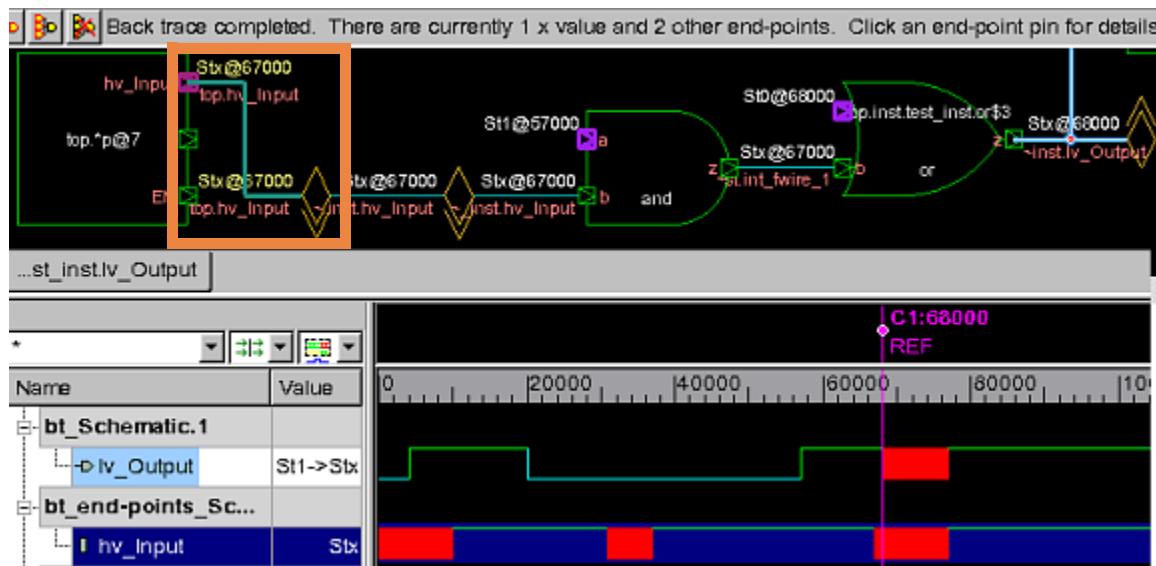


4. Move the cursor C1 to time 68000.



5. Select the signal in the Signal pane, right-click and select **Show Back Trace Schematic**.

A wave group and Waveform view of the selected signal is automatically created at the current simulation time. The schematic shows the driver **hv_input** for the x value of the signal **1v_Output**.



The current simulation time and value pair are annotated on the output pins of the driving cell. The input pins of the driving cell are annotated with the values and times of the next signal transition.

Back Trace will trace as far back as it can based upon your preferences.

6. Select the left most cell/pin in the back trace schematic to find out why tracing stopped.

A reason will be displayed in the top of the schematic.

For example, a common reason is "Trace endpoint pin 'Top.dut.a' status: Multiple X input pins on cells".

7. To continue back tracing, double-click on the input pin of the leftmost cell.

Note:

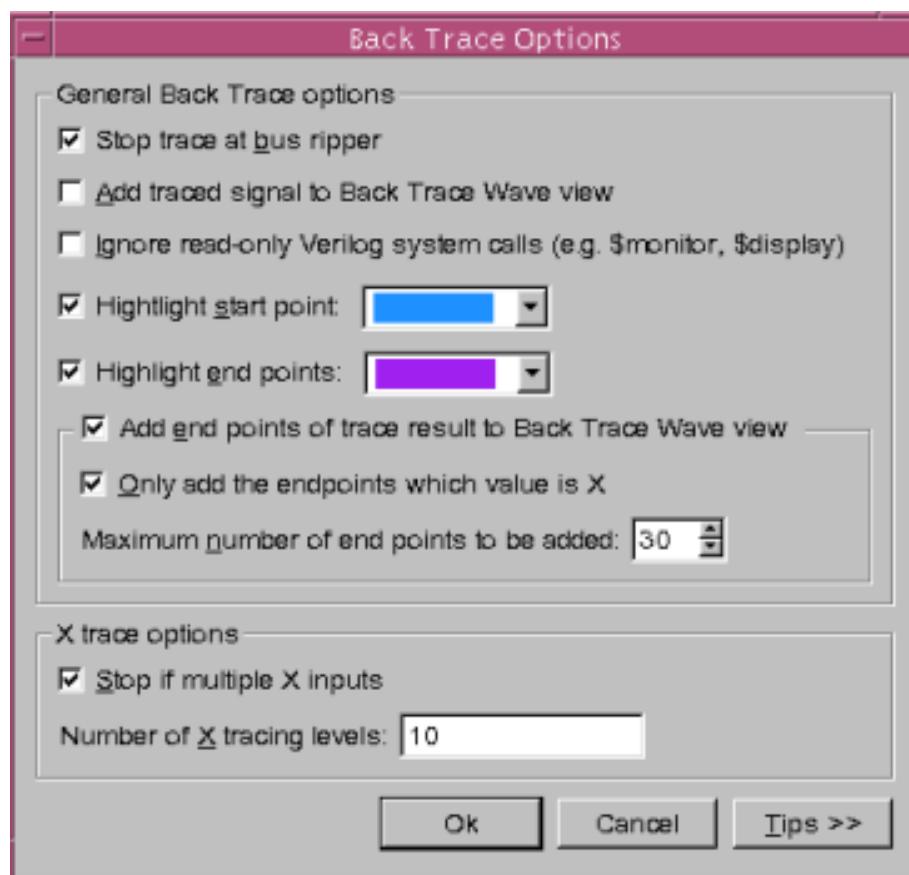
The current simulation time is moved to the earliest time over all driver inputs.

Setting the Back Trace Properties

The Back Trace Properties window is used to add multiple levels of trace, so that tracing X signals can automatically trace back multiple levels following the X value over time. Instead of expanding one level, you can draw multiple levels and add multiple signals on the traced path to the Wave view.

To set the Back Trace properties

1. Select **Show Back Trace Options** button on the toolbar.



The Back Trace Options dialog box opens that contains the following options:

- Stop trace at bus ripper - Stops tracing when a bus ripper is encountered.
- Add traced signals to Back Trace Wave view - Adds the signals on the traced path to the wave group and displayed in the Wave view.
- Ignore read-only Verilog system calls - Ignores the read-only Verilog system calls like \$monitor or \$display.

- Highlight start and end points - Highlights the start and end points of the signals in the chosen color.
- Add end signals of trace result to Back Trace Wave view - If this option is selected, when the Back Trace operation is complete, the pins on the final block(s) will be added to the Back Trace Wave view.
- Only add the endpoints which value is X - Adds end points for the signals whose value is X.
- Maximum number of end points to be added - Adds the number of end points as specified.
- Stop if multiple X inputs - Stops the trace when multiple inputs are provided.
- Number of levels to trace - Controls the maximum number of levels to search backwards when automatically searching for X values.

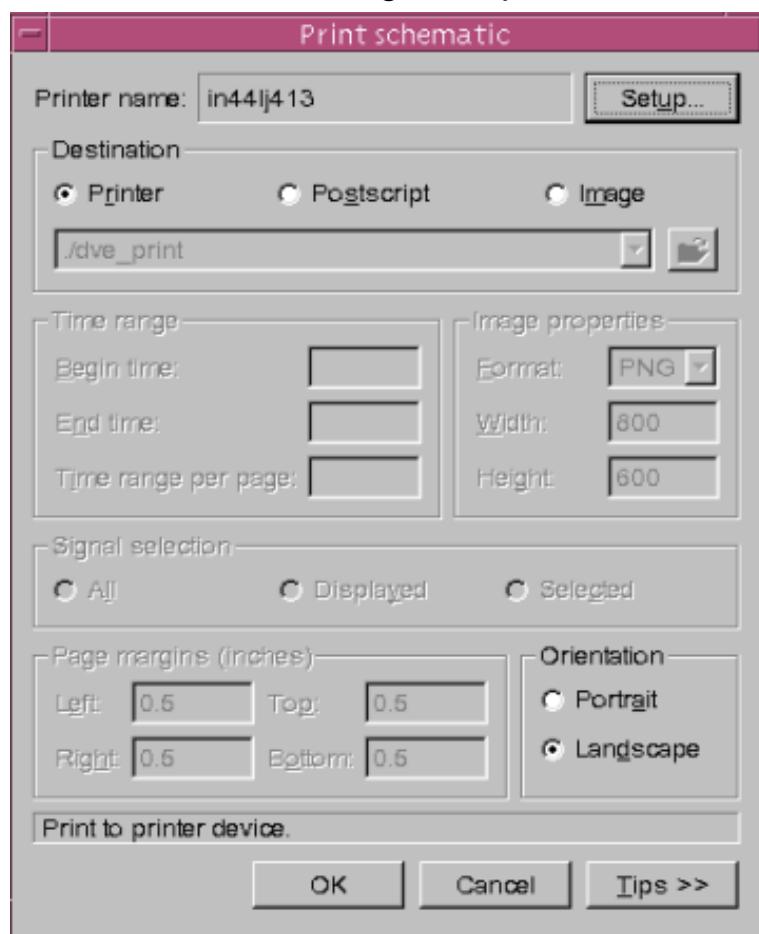
Printing Schematics

You can print schematics to a file or printer from an active Schematic or Path Schematic view selecting time range and signals to print.

To print schematics

1. From an active Schematic or Path Schematic view, select **File > Print**.

The Print Schematic dialog box opens.



2. Click the Setup button to set printing options:

- Printer or print to file
 - Print in color or grayscale
 - Print orientation and paper size
 - Print options such as range and number of copies
3. Select whether to print **All**, **Displayed**, or **Selected** signals.
 4. Select the page margins.
 5. Choose **Landscape** or **Portrait** orientation.
 6. Click **OK**. The schematic is printed.

Schematic Visualization of RTL Designs

RTL schematic view is a Register Transfer Level graphical representation of your design. In this view, a design is represented as macro blocks such as adders, multipliers, and registers. Being as close as possible to the original HDL code, this view allows you to visually check your design.

In previous versions of VCS, most design processes and instances of DVE schematic views were represented as rectangles. For example, consider the Verilog test case show in [Example 7-1](#):

Example 7-1 Schematic Visualization Design File (test.v)

```
module top;
  wire [3:0] a;
  wire q,x;
  reg b,c,d,clk,rst;
  dut U (a,q,x,b,c,d,clk,rst);
  initial
    begin
      $monitor("%t b=%b,c=%b,d=%b,a =%b", $realtime, b,c,d,a);
      b=1'b1;    c=1'b1;    d=1'b1;
      #5 b=1'b0;  #0 c=1'b1;  #0 d=1'b1;
      #5 b=1'b1;  #0 c=1'b1;  #0 d=1'b1;
      #5 b=1'bx;  #0 c=1'bx;  #0 d=1'bx;
      $finish;
    end
  endmodule

  module dut (a,q,x,b,c,d,clk,rst);

    input b,c,d,clk,rst;
    output [3:0] a;
    output reg q,x;
    wire [3:0] a;

    wire wire_with_no_input,
```

```

    simple_wire,
    inverter,
    bitwise_operator_xor,
    complex_expression_without_condition,
    complex_expression_with_condition;
wire [3:0] concatenation;

assign wire_with_no_input = 1;
assign simple_wire = b;
assign inverter = !b;
assign bitwise_operator_xor= b ^ c;
assign concatenation = {wire_with_no_input,
                        simple_wire,
                        inverter,
                        bitwise_operator_xor};

assign a = {1'b0, concatenation};
assign a= b ? c:d;
assign a= b ? !(c):d;
assign complex_expression_without_condition = (!b & !c &
d) | (!b & c & !d) | (b & !c & !d) | (d & b & c);
assign complex_expression_with_condition= b? (b? (c & d
)| (!c & !d): (!c & d) | (c & !d)) : (b? (c & d )&(!c & !d): (!c
& d)&(c & !d)) ;

always @ (d or c or b)
begin
  x=d;
end
always @ (posedge clk)
begin
  q<=d;
end
always @ (negedge clk)
begin
  q<=c;
end
always @ (posedge clk)
if (~rst)
begin
  q<=1'b0;
end

```

```
else
begin
q<=b;
end
endmodule
```

Compile the test.v file shown in [Example 7-1](#):

```
% vcs -debug_all test.v
```

Invoke the DVE GUI:

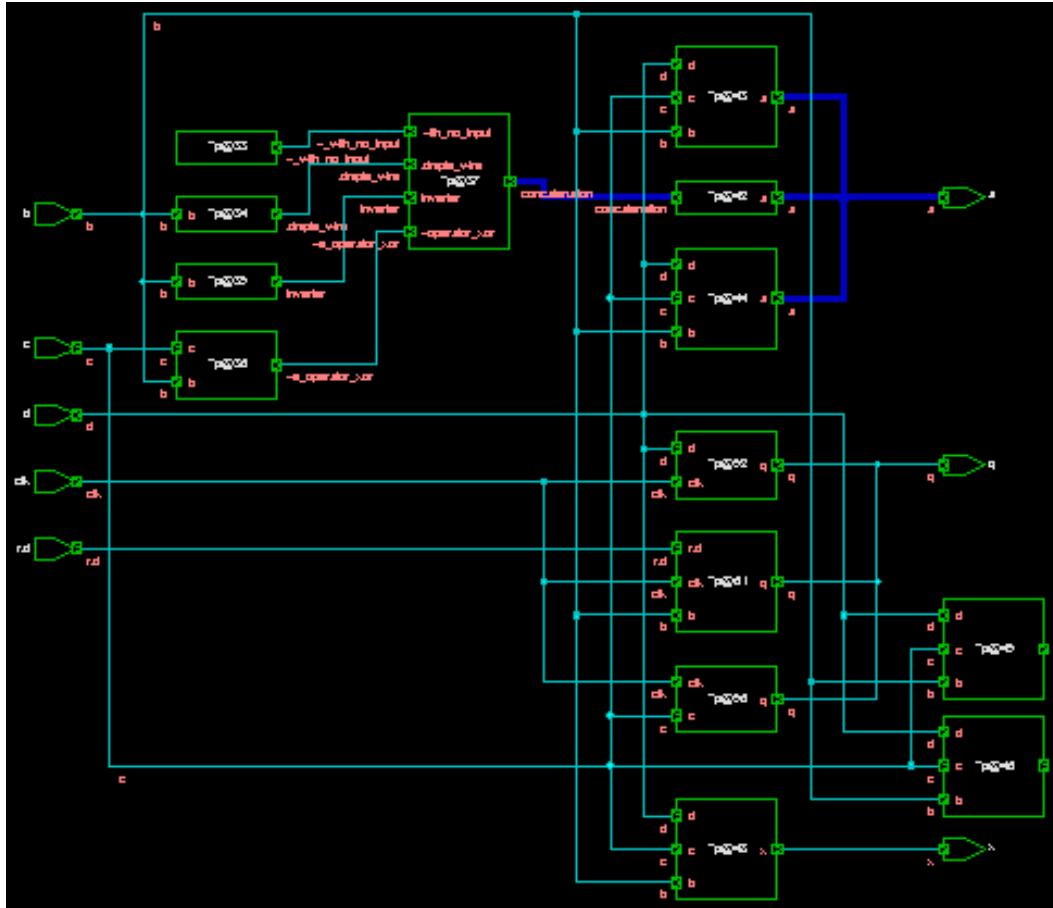
```
% ./simv -gui&
```

In previous versions of VCS, DVE generated the schematic shown in [Figure 7-2](#), where most of the design processes were represented as rectangles.

These views use the following set of symbols:

- Predefined primitives (in Verilog): corresponding logic gate
- Hierarchical element: double rectangle
- Other process: rectangle

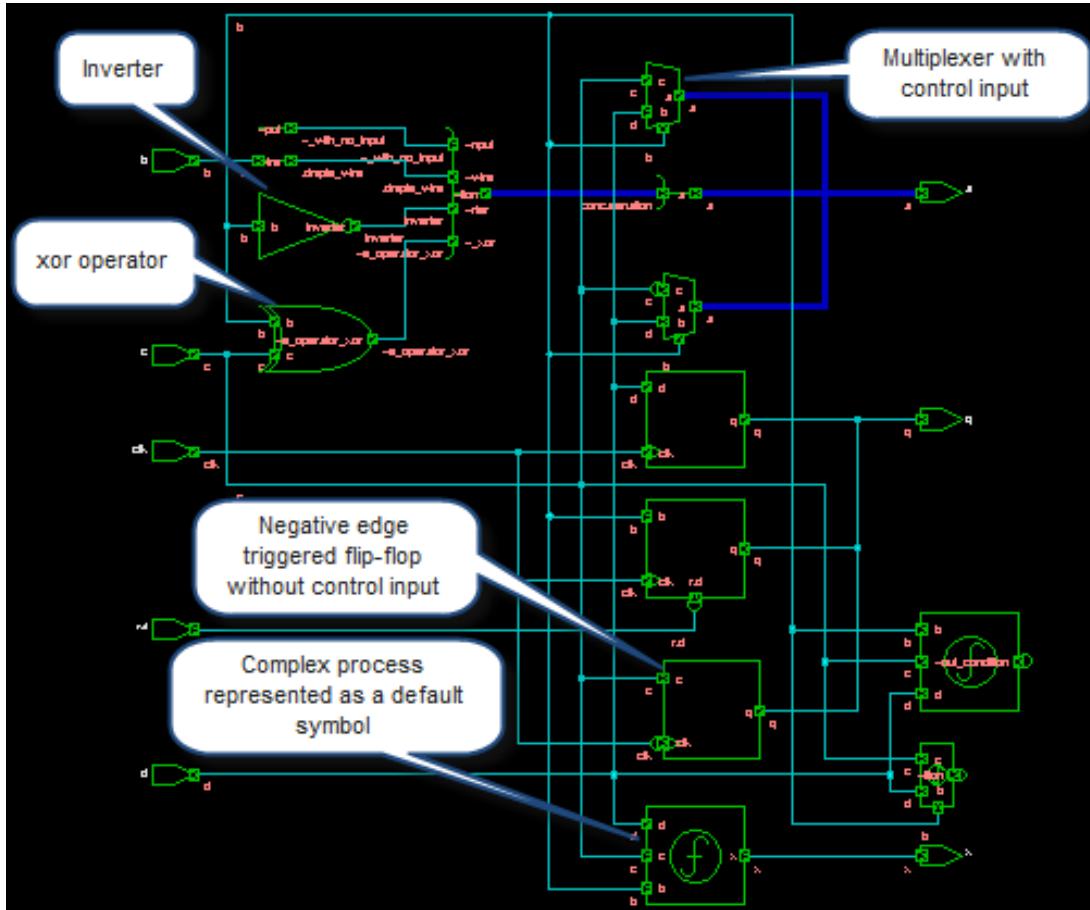
Figure 7-2 DVE Showing all Processes as Rectangles



This type of representation does not allow you to quickly analyze the functionality of each process or instance in the design.

DVE now provides more information on the RTL design process functionality by using a larger set of symbols to represent these processes. For each process, the RTL schematic view shows an appropriate symbol that describes the process characteristics (see [Figure 7-3](#)), thereby increasing the readability of the DVE design and path schematic views.

Figure 7-3 DVE Using Symbols to Represent Processes



Schematic Symbols

The symbols for nodes in the schematic represent the elements of your design. This section describes the new set of symbols that can be displayed in the schematic view.

Design Analysis for RTL Symbol Creation

DVE does not perform full synthesis of the design. It only performs a simple analysis of the RTL processes to find representative symbols.

DVE performs the following analysis to determine the RTL design functionalities:

- DVE uses the sensitivity list and the main statement (simple assignment or some conditional code) of a process to determine its nature. After this analysis, DVE determines the symbol to use for that process.
- Simple processes such as binary operations, assignments, simple muxes, and flip-flops are represented by corresponding gate-level symbols (see “[Simple Logic Schematic Symbols](#)” on [page 44](#)).

For complex processes like multiple outputs, nested if statements, and for loops, DVE uses a default symbol (see “[Default Symbol for a Process](#)” on [page 41](#)).

- DVE displays all input and inout signals of a process on the left-hand side of the symbol.
- DVE displays all output signals of a process on the right-hand side of the symbol.
- DVE displays all control inputs (input signal that is read in a condition expression of an if statement, a case statement, or a conditional assignment statement) of a process at the bottom of the symbol.
- If a signal appears on both condition and RHS of an assignment, then it is not considered as control, so it is shown on the left-hand side of the symbol.
- If an input is edge-triggered, DVE adds an arrow (clock symbol) to the corresponding pin. If the edge is negative, then DVE adds a circle to represent the falling edge.

Note:

This also includes asynchronous sets or resets. DVE adds the pin at the bottom of the symbol with an arrow that represents the edge.

- If an input signal is inverted (preceded by the “not” operator), DVE adds a circle on that pin to represent the pin inversion.
- If the main statement is an assignment statement, and if the main operator of the RHS of the assignment is a “not,” then the output pin is represented with a circle. The only exception to this is the inverted operator, since the inversion is already represented in the symbol itself.

Default Symbol for a Process

DVE uses the default symbol for a process if its expression or always block is complex. [Table 7-2](#) describes the default symbols.

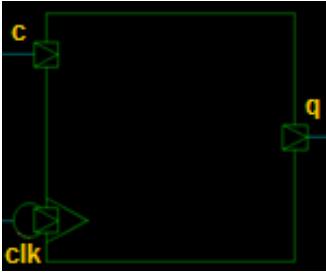
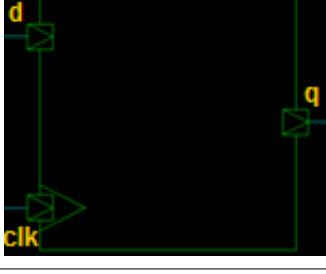
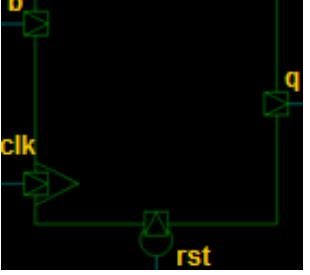
Table 7-2 Symbols in the Schematic View - Default Symbols

Symbol	Symbol Name	Example
	Default symbol for always block (or VHDL process)	always @ (d or c or b) begin x=d; end
	Default symbol for assign statement (or VHDL continuous assignment statement)	assign complex_expression_without_condition = (!b & !c & d) (!b & c & !d) (b & !c & !d) (d & b & c);

Flip-Flop Schematic Symbols

Table 7-3 describes the flip-flop schematic symbols.

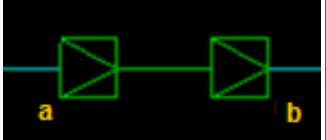
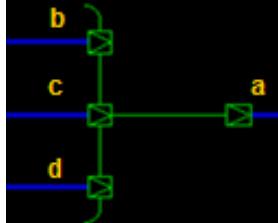
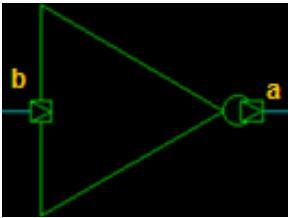
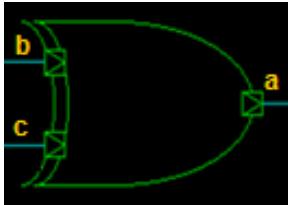
Table 7-3 Symbols in the Schematic View - Flip-Flops

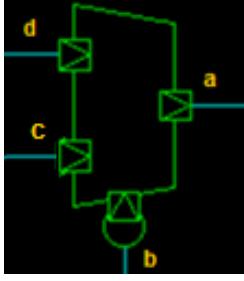
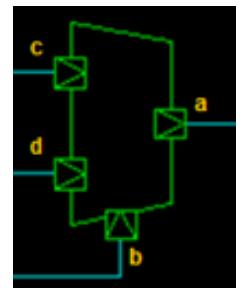
Symbol	Symbol Name	Example
	Negative edge triggered flip-flop without control input	always @ (negedge clk) begin q<=c; end
	Positive edge triggered flip-flop without control input	always @ (posedge clk) begin q<=d; end
	Positive edge triggered flip-flop with control input	always @ (posedge clk) if (~rst) begin q<=1'b0; end else begin q<=b; end

Simple Logic Schematic Symbols

Table 7-4 describes the simple logic schematic symbols.

Table 7-4 Symbols in the Schematic View - Simple Logic

Symbol	Symbol Name	Example
	Wire with no input	assign a = (constant value)
	Simple wire	assign a=b;
	Concatenation	assign a= { b,c,d };
	Inverter	assign a= ! b;
	xor operator	assign a= b ^ c

	Multiplexer with negative control input	<pre>assign a= ! (b) ? c:d;</pre>
	Multiplexer with positive control input	<pre>assign a= b ? c:d;</pre>

Using the above enhancements, DVE generates the schematic shown in [Figure 7-3](#), for the `test.v` example.

Enabling and Disabling RTL Visualization

RTL visualization is enabled by default. To disable it:

7. Select **Edit > Preferences**.

The Application Preferences dialog box appears.

8. In the Schematic View category, select the **Disable Rtl analysis for schematic symbols (draw everything as rectangle)** option.

Note:

If you modify (enable or disable) this option when a schematic view is open, you must exit and restart DVE to make the changes take effect. If you do not exit and restart DVE, then all modules already displayed in the schematic view continue to display as they appeared before you changed the preference setting.

Schematic Visualization of RTL Design Limitations

This feature does not support:

- Set or reset of flip-flops — Synchronous set or reset is shown as an input. Asynchronous set or reset is connected to an edge pin.
- Latches — Latches are represented with the default process symbol.
- FSM — No specific analysis is done for FSM, and so no specific symbol is used for it. FSM is represented with the flip-flop symbol (edge-controlled process).
- For VHDL, all processes are represented with the default process symbol. No control pin analysis is done, so all the inputs are on the left-hand side.

8

Using Smartlog

DVE Smartlog provides log analysis (diagnostic information) for each line in the log file. It takes the compile log and simulation log created by VCS and summarizes the data into reports. Smartlog provides the diagnostic information in a separate log file known as a smartlog file. Following are the main features of Smartlog:

- Hyperlink the log occurrences to the Source View
- Highlights the words, namely, Error, Warning, and so on, in different colors
- Displays the selected message within a blue rectangle

Use Model

Use the `-sml` option to enable the Smartlog feature. Following is the syntax of this option:

```
-sml -l <logfile>
```

Compile Flow

Smartlog helps you to analyze and correct the problems found during the compilation. The steps for compile flow are as follows:

1. `%vcs <options> -sml -l compile.log`
2. `%dve -viewlog compile.log`

DVE displays the compile log in the Console Pane. You can select file paths to view the source file. DVE displays the following three tabs in the Console Pane:

- **Log** – DVE command log.
- **History** – Displays the list of all commands or actions that you have taken while working in the GUI.
- **compile.log** – Displays the compile log. This tab is displayed by default in the Console Pane.

Note:

You can specify `-viewlog` multiple times on the command line. DVE opens multiple logs in separate tabs.

Post-processing Debug Flow

For this flow, the design should be compiled and simulated. You must debug the design with a vpd dump file. For example:

1. %vcs <options> -sml -l compile.log
2. %simv -sml -l simout.log
3. %dve -vpd vcdplus.vpd -viewlog simout.log

DVE displays the design for the VPD file and the simulation log file. DVE displays the following tabs in the Console Pane:

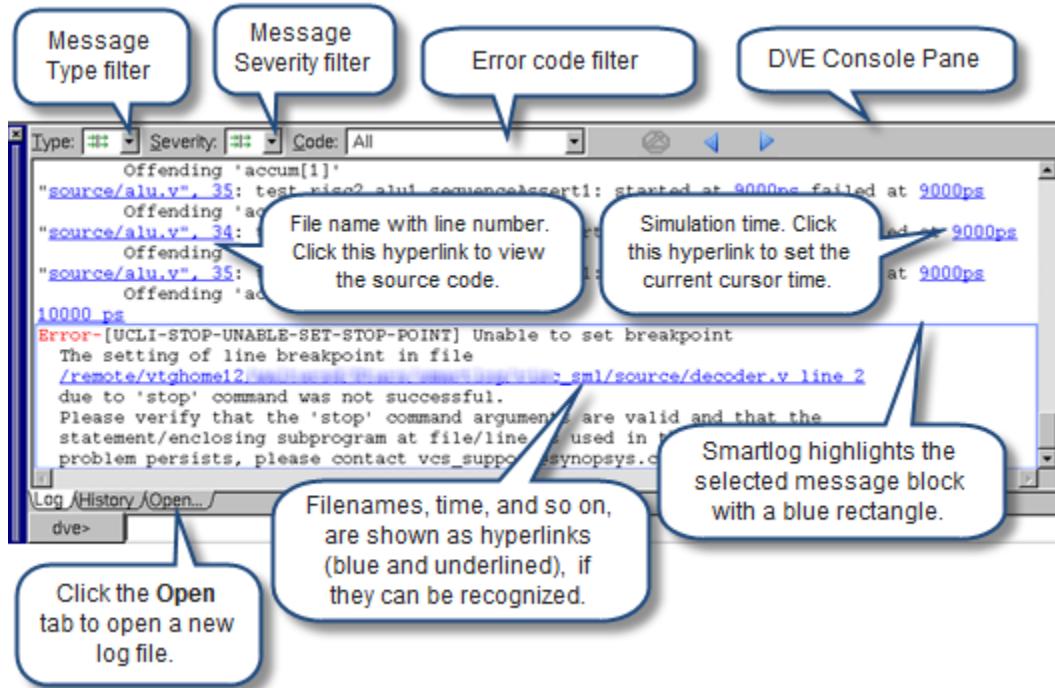
- **Log** – DVE command log.
- **History** – Displays the list of all commands or actions that you have taken while working in the GUI.
- **simout.log** – Displays the runtime log. This tab is displayed by default.

Viewing Smartlog Data in the Console Pane

DVE displays the Smartlog data in the Console Pane, as shown in [Figure 8-1](#).

DVE displays the log file as a normal text. It highlights the words “Warning”, “Error”, and so on, in different colors. When a block of text is selected, which is recognized as a message, it is highlighted with a blue rectangle.

Figure 8-1 Smartlog Data in the Console Pane



DVE highlights the file name with optional line number and the simulation time as hyperlinks. You can click a hyperlink to view its source code or set the current cursor time.

Right-click Menu Options in Smartlog

Table 8-1 describes the right-click menu options in the Smartlog view. The right-click menu displays possible actions for the selected message. The menu options are enabled only if the action is possible for the current message or sub-item.

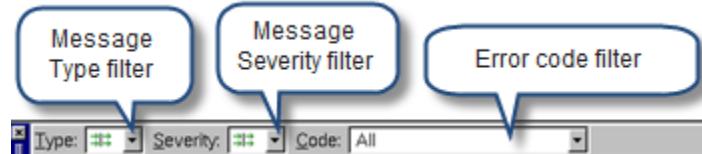
Table 8-1 Right-click Menu Options in Smartlog

Option	Description
Add to Waves, Add to Lists, Add to Watches	These options are enabled if the selected text is recognized as a scope. Note: The text is selected, but is not shown as a hyperlink.
Save Selection As, Save Contents As	These options allow you to save the selected text to a separate file.
Go To Source	This option displays the source code in the Source View. This option is enabled if there are one or more filenames recognized in the current message.
Go To Time	This option sets the current cursor time. This option is enabled if a simulation time is recognized in the current message.

Filtering Options in the Console Pane

Filtering options in the Console Pane (see [Figure 8-2](#)) allow you to configure the type of information to display in Smartlog. DVE displays only messages matching all filters in Smartlog.

Figure 8-2 Console Pane Filtering Options



Message Type Filter

This filter displays a dynamic list of the types mentioned in [Table 8-2](#), based on the types that are recognized from the log. It allows multiple selections with check boxes. The count of messages with a given type is shown in parentheses, as shown in [Figure 8-3](#).

Figure 8-3 Message Type Filter

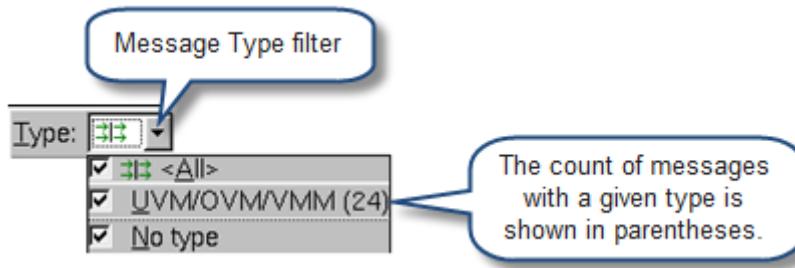


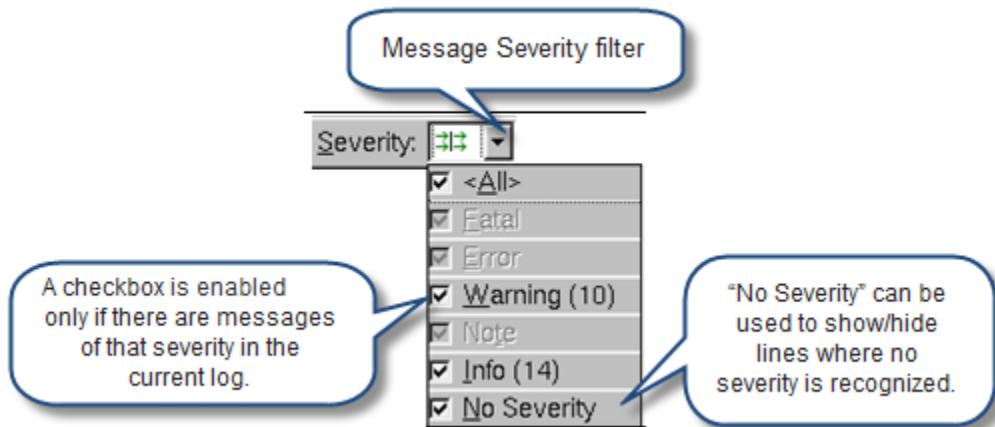
Table 8-2 List of Possible Filter Types

Filter type	Description
All	Displays all messages in the log
DVE	Displays standard DVE error messages and other messages (Tcl error, and so on)
VCS	Displays standard VCS error messages (Fatal, Error, Warning, Note, and so on)
UVM/OVM/VMM	Displays \$display from UVM/OVM/VMM packages (for example: "UVM-xxx")
Diagnostics	Displays VCS diagnostic messages. Following are the types of VCS diagnostic messages: <ul style="list-style-type: none"> • libconfig • xprop • timescale
User	Displays the output from user \$display calls during the simulation
No Type	Displays all other messages printed out by VCS/SIMV or user printf (PLI), and so on. You can use this type to view the output (in the log) that does not belong to any other filter type mentioned in this table (for example, VCS copyright and so on). This is displayed only when the other two filters, that is, Message Severity filter and Error Code filter match.

Message Severity Filter

This filter displays the fixed list of severities. For example, see [Figure 8-4](#). It allows multiple selections with check boxes. The count of messages with given severity is shown in parentheses.

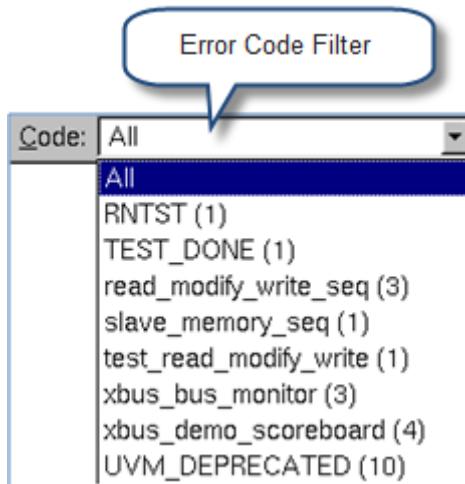
Figure 8-4 Message Severity Filter



Error Code Filter

The Error Code Filter allows you to filter messages with a specific message code. For example, see [Figure 8-5](#). It allows only single selection by clicking the **Code** drop-down. The filter displays a dynamic list, which is determined by the type/severity that has been selected. The count of messages with the code is shown in parentheses.

Figure 8-5 Error Code Filter



Opening Log File

You can use the **Open Log File** dialog box, as shown in [Figure 8-7](#), to open a new log file.

To open the **Open Log File** dialog box, click the **Open** tab in the Console Pane, as shown in [Figure 8-6](#).

Figure 8-6 Opening the Open Log File Dialog Box

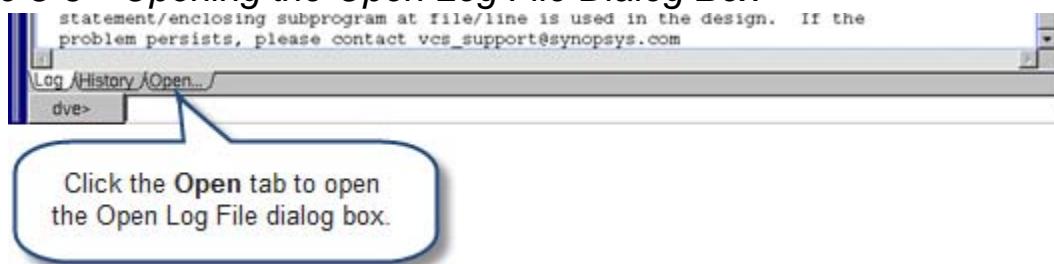
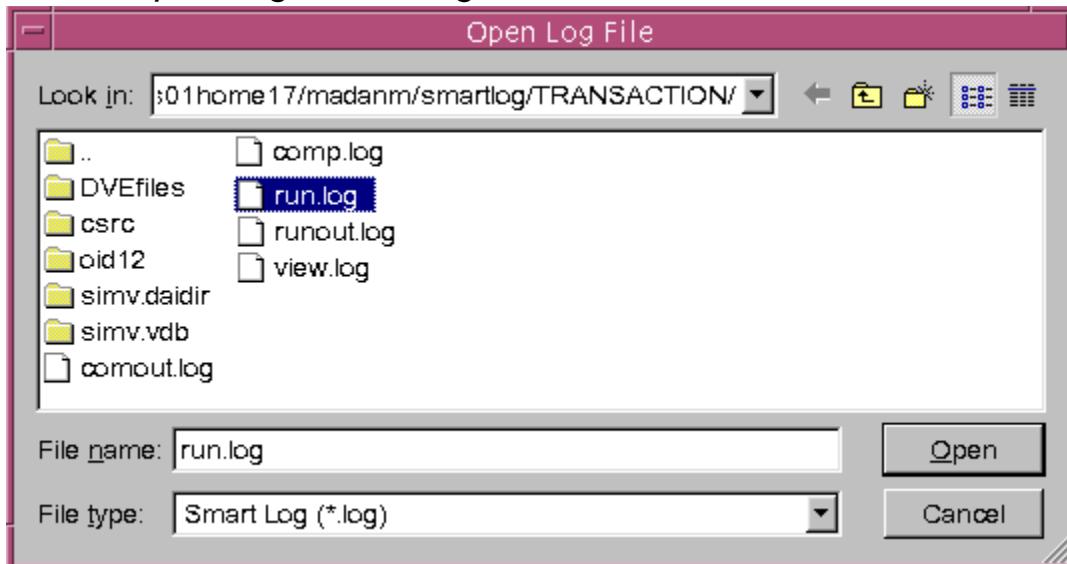


Figure 8-7 Open Log File Dialog Box



Select a log file from the list and click **Open**. You can also define the name to be used for the new tab in the log viewer. By default, the tab has the same name as that of the file name.

Usage Example

Consider the following UVM test case:

Example 8-1 Design File trans.sv

```
import uvm_pkg::*;

class trans extends uvm_sequence_item;
    typedef enum {READ,WRITE,RESET} opcode;
    rand bit [2:0] add;
    rand opcode opc;

    `uvm_object_utils_begin(trans)
        `uvm_field_int(add,UVM_ALL_ON | UVM_BIN);
        `uvm_field_enum(opcode,opc,UVM_ALL_ON)
    `uvm_object_utils_end
```

```

        function new (string name = "trans");
            super.new(name);
        endfunction

    endclass

    class trans_seq extends uvm_sequence #(trans);
        `uvm_object_utils(trans_seq)

        function new (string name="trans_seq");
            super.new(name);
        endfunction
        task body();
            if (starting_phase != null)
                starting_phase.raise_objection(this);
            repeat(10) begin
                `uvm_do(req);
            end
            if (starting_phase != null)
                starting_phase.drop_objection(this);
        endtask
    endclass

    typedef uvm_sequencer #(trans) trans_seqr;

    class driver extends uvm_driver #(trans);
        `uvm_component_utils(driver)

        function new(string name,uvm_component parent);
            super.new(name,parent);
        endfunction

        task run_phase(uvm_phase phase);
            forever begin
                seq_item_port.get_next_item(req);
                `uvm_info("DRVR",req.sprint(),UVM_MEDIUM);
            #1;
                seq_item_port.item_done();
            end
        endtask
    endclass

```

```

endclass

class agent extends uvm_agent;
  `uvm_component_utils(agent)
  trans_seqr seqr;
  driver drv;
  function new (string name,uvm_component parent);
    super.new(name,parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    seqr=trans_seqr::type_id::create("seqr",this);
    drv=driver::type_id::create("drv",this);
  endfunction

  function void connect_phase(uvm_phase phase);
    drv.seq_item_port.connect(seqr.seq_item_export);
  endfunction
endclass

class env extends uvm_env;
  `uvm_component_utils(env)
  agent age;
  function new(string name,uvm_component parent);
    super.new(name,parent);
  endfunction
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    age=agent::type_id::create("agent",this);
    uvm_config_db #(uvm_object_wrapper)::set(this,"*.seqr.main_phase","defau
    lt_sequence",trans_seq::get_type());
    $display(trans_seq::get_type());
  endfunction
endclass

class test1 extends uvm_test;
  `uvm_component_utils(test1)
  env env1;
  function new(string name,uvm_component parent=null);
    super.new("test1",parent);

```

```

    endfunction

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        env1 =env::type_id::create("env1",this);
    endfunction
endclass

module top;
initial run_test();
endmodule

```

Post-processing Mode

Perform the following steps for post-processing mode:

1. Compile the `trans.sv` code, shown in [Example 8-1](#), as follows:

```
% vcs -debug_all -sverilog -ntb_opts uvm trans.sv -sml -l comp.log
```

2. Run the `trans.sv` code, as shown below:

```
./simv -ucli -i no_delta.inc +UVM_TESTNAME=test1 -sml -l run.log
```

where, the `no_delta.inc` file contains the following run-script:

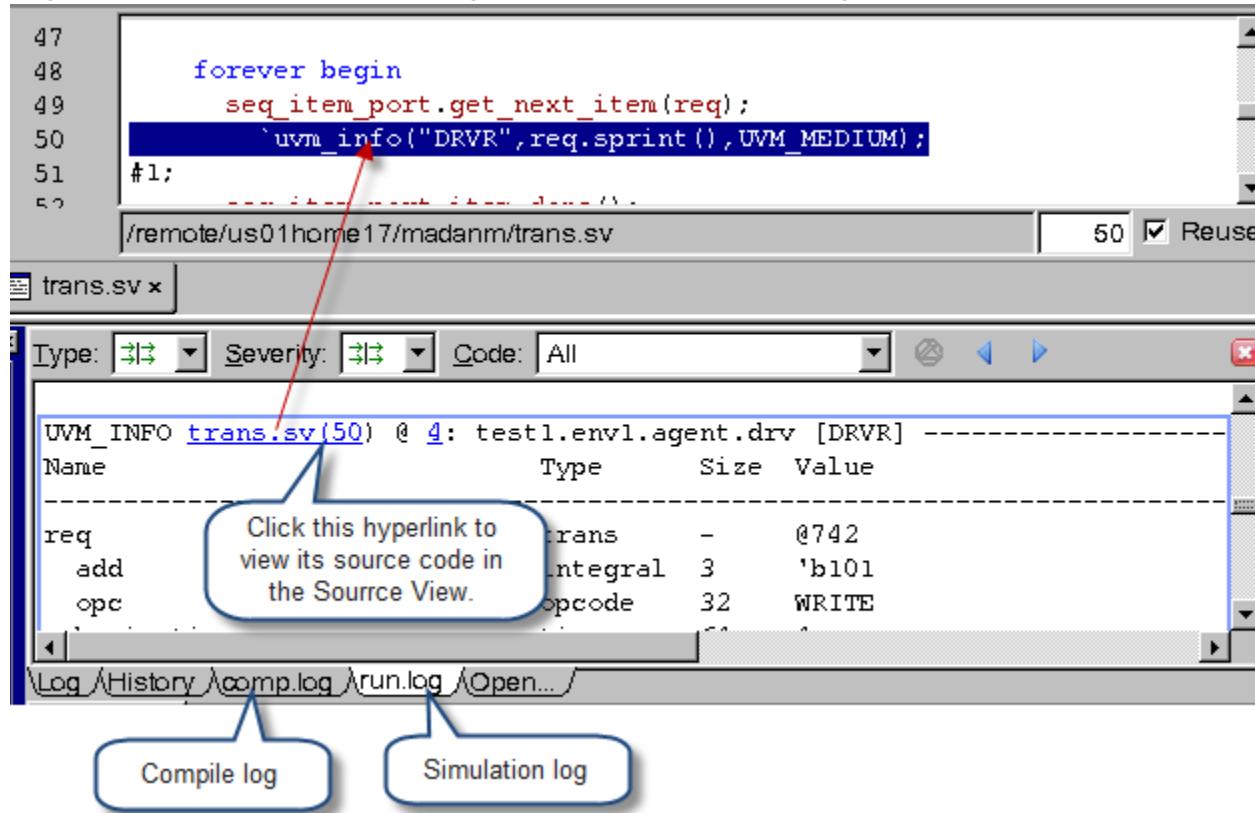
```
dump -file test.vpd
dump -deltaCycle on
dump -add / -aggregates
run
```

3. Invoke the DVE GUI, as follows:

```
% dve -viewlog comp.log -viewlog run.log -vpd test.vpd &
```

DVE displays the Source View and the simulation log (`run.log`), as shown in [Figure 8-8](#).

Figure 8-8 Post-processing Mode: Simulation Log in the Console Pane



Interactive Mode

Smartlog is enabled by default in interactive mode. Perform the following steps to use smartlog in interactive mode:

1. Compile the `trans.sv` code, shown in [Example 8-1](#), as follows:

```
% vcs -debug_all -sverilog -ntb_opts uvm trans.sv -sml -l comp.log
```

2. Invoke the DVE GUI, as follows:

```
simv -gui +UVM_TESTNAME=test1 &
```

3. Click the Run  button to run the simulation.

DVE displays the simulation log (`run.log`), as shown in [Figure 8-9](#).

Figure 8-9 Interactive Mode: Simulation Log in the Console Pane

The screenshot shows the DVE interface in Interactive Mode. The top right pane displays the simulation log (`run.log`) with the following content:

```
50 `uvm_info("DRV", "req.sprint()", UVM_MEDIUM);
51 #1;
52 seq_item_port.item
```

The bottom left pane, titled "Smartlog", shows variable values:

Name	Type	Size	Value
req	trans	-	@754
add	integral	3	'b1
opc	opcode	32	READ

A callout bubble points from the "req" entry in the Smartlog table to the "req" entry in the log table, with the text: "Click this hyperlink to view its source code in the Source View."

9

Tracing Drivers and Loads

This chapter describes how to use DVE to trace drivers and loads of signals in your design. It contains the following sections:

- “The Driver Pane”
- “Tracing Drivers and Loads”
- “Active Drivers”

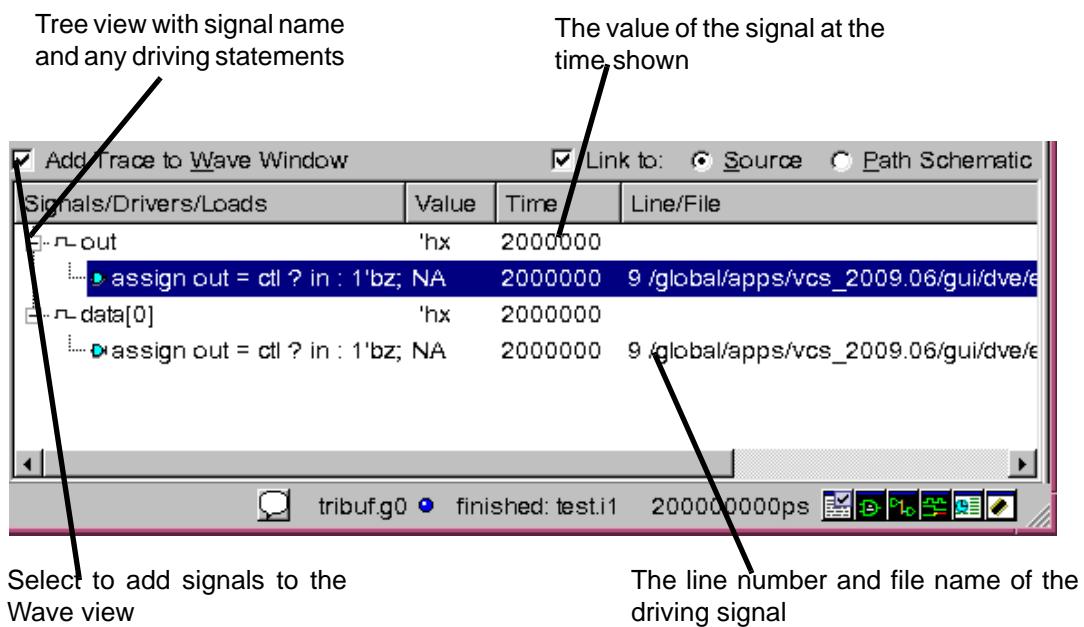
The Driver Pane

You can trace drivers and loads of a signal at any time to see the drivers and loads that caused a value change. You can see all the drivers/loads that possibly contributed to a signal value. A signal's

load(s) are the input port(s), I/O port(s), and statements that read the signal's value. Multiple driver panes are allowed as long as there are multiple top level windows to contain them.

You can perform the following tasks in the Drivers pane:

Figure 9-1 Drivers pane



- Delete the Driver pane using the X icon.
- Dock or undock the Driver pane.
- Link the Driver panes to Source view in the same top level frame and Path Schematic view. The **Link to** radio buttons, at the right top of the pane, show the current linked windows.

By linking a Source and Schematic view, when you select the object in the Drivers pane, the object will also be selected in the linked views.

- **Add signals to Wave view by selecting the Add to Waves** check box. Clearing the check box does not delete anything from the Wave view but prevents additional signals in the drivers pane from being added to the Wave view.
-

Supported Functionality

- All Verilog types, constructs, control path.
 - Verilog gate and UDPs.
 - VHDL but only down to the process statement. All drivers within a process are determined to be active.
-

Unsupported Functionality

SystemVerilog data types are not supported.

Tracing Drivers and Loads

To trace drivers and Loads

1. Select a signal in a view or pane.

For example, Data pane, Wave view, Source view, List view etc.

2. Right-click and select **Trace Drivers** or **Trace Loads**.

When a driver is traced, a new Driver pane will be created if none exists in the current top level frame. If a driver pane exists, the driver information will be added to the top of the list.

Additionally, the first driver will be highlighted in the Source view and annotated with a blue node in the gutter. In the Wave view, you can double-click on a waveform to see its drivers. For example on a transition from 0 -> 1 or 1 -> 0.

Note:

- Only one driver pane is allowed per top level frame.
- If you select multiple signals and trace their drivers or loads, the driver or load is traced only for the first signal.

Active Drivers

During debugging, you often need to find the reason for a value change of a particular signal. For this, you can:

- Perform active-driver tracing for this signal and view the active driver statement (however, for interactive mode, you must dump the signal into VPD before doing driver tracing).
- Quickly view driving signals for this statement (signals that cause the value change of the traced signal).
- Continue active-driver tracing automatically until you find the root cause of the value change.

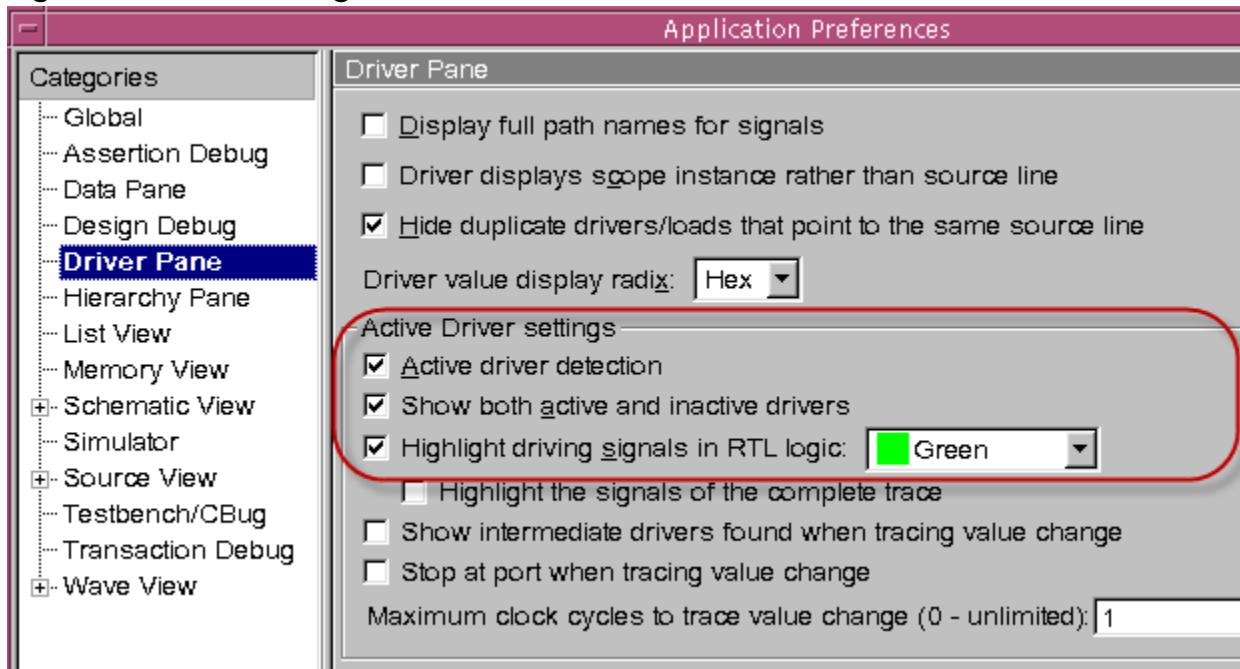
DVE now supports the tracing driver functionality by allowing you to do the following for Verilog:

- Highlight driving signals for the following active statements:
 - Continuous assignments
 - Assignments in RTL combinational logic and flip-flops
- Backtrace a signal value over RTL combinational logic.
- Incrementally continue active-driver tracing for contributors in the Driver Pane.
- View intermediate drivers in the Driver Pane.
- Use the Driver Pane to view the path between the driver and the traced signal in the path schematic view.
- Specify the maximum number of clock cycles to trace the value change.

Enabling Active Drivers

This feature is enabled by default, as shown in the following figure:

Figure 9-2 Enabling Active Drivers



Different colored icons are shown for displaying the active driver detection results as follows (see [Figure 9-3](#)):

- Hollow — Hollow icon is shown for inactive drivers.
- Yellow — Yellow icon is shown for possible active drivers. The active driver analysis stopped because of one of the following reasons:
 - Missing signal dump
 - Dynamic variables
 - Some limitations, as explained in the section "[Active Driver Limitations](#)".

- Green — Green icon is shown for active drivers.

Figure 9-3 Active Driver Detection

Signals/Drivers/Loads	Value	Time	Line/File
rst	St0->St1	75	
rst=1;	NA	75	67 example.v
#2 rst=0;	NA	75	68 example.v
#75 rst!=rst;	NA	75	77 example.v
rst	St0	20	
rst=1;	NA	20	67 example.v
#2 rst=0;	NA	20	68 example.v
#75 rst!=rst;	NA	20	77 example.v

If there are inactive drivers at the selected simulation time, you can see the inactive drivers with the active drivers using the preference option **Show both active and inactive drivers**.

Usage Example

Example 9-1 test.v File

```
module top();

    wire w1, w2, w3, w4, w5, w6;

    reg r1, r2;

    assign w1 = r1;
    assign w2 = r2;
    assign w3 = w2;
    assign w4 = w3;
    assign w5 = w4;
    assign w6 = w5;

    initial begin
        #2 r1 = 1;
        #5 r1 = 0;
        #5 r1 = 1;
        r2 = r1;
    #5 r1 = 1;
    #5 $finish();
    end
endmodule
```

Compile the test.v file shown in [Example 9-1](#):

```
% vcs -debug_all test.v
```

Invoke the DVE GUI:

```
% ./simv -gui&
```

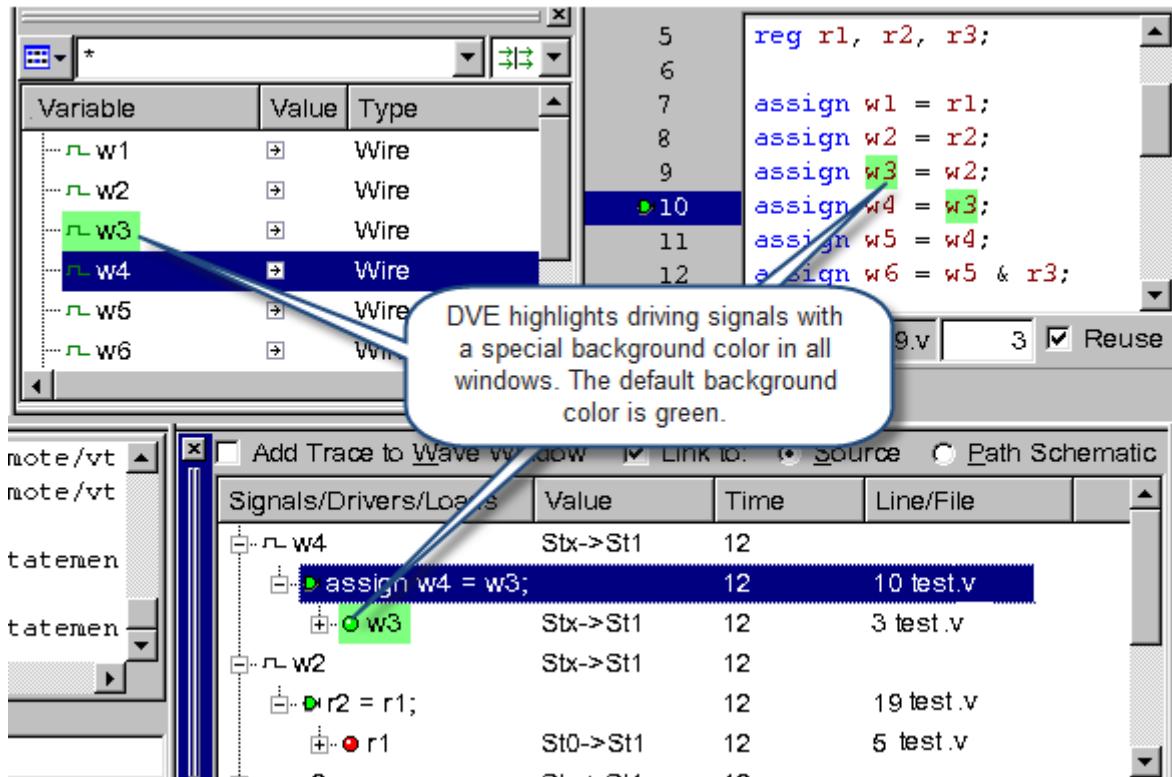
Visualizing Driving Signals

Driving signals are signals that cause value changes of the traced signal. You can use the **Highlight driving signals in RTL logic** preference option to highlight driving signals with a special background color. When this option is selected, DVE highlights the driving signals in all windows (Source View, Wave View, Path Schematic View, Data Pane, Driver Pane, and so on), as shown in [Figure 9-4](#).

This highlighting is done only for the following active statements:

- Continuous assignments
- Assignments in RTL combinational logic and flip-flops

Figure 9-4 Highlighting Driving Signals



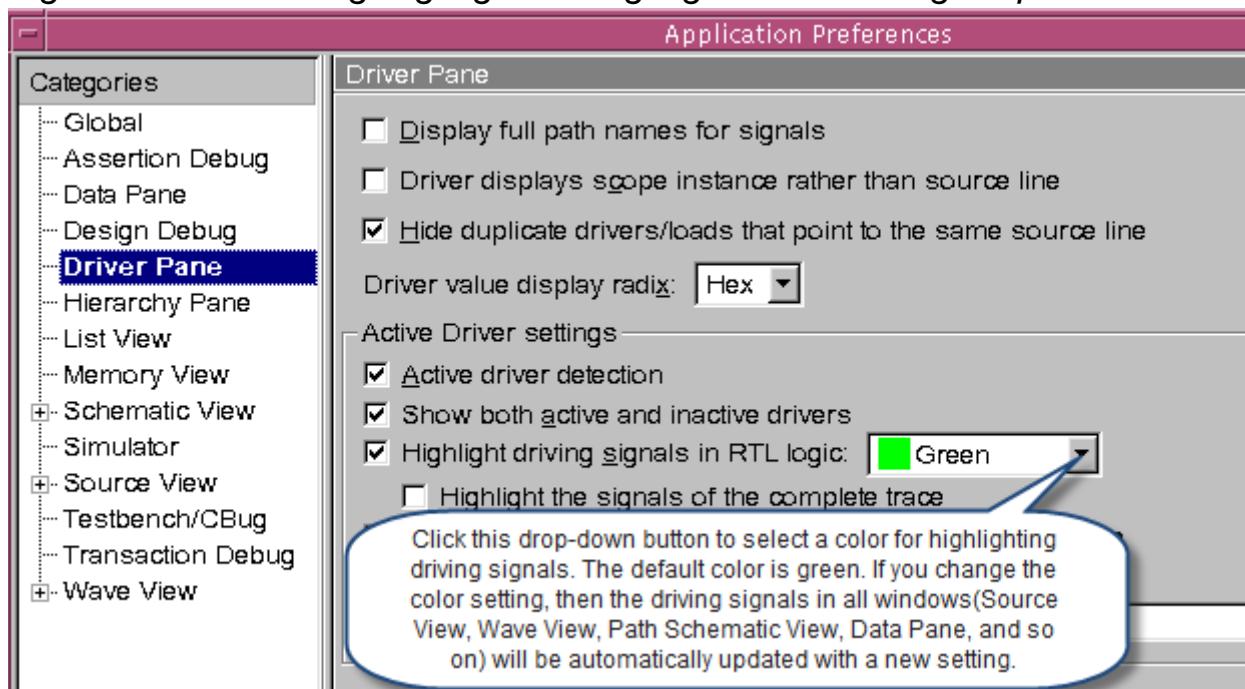
To enable the **Highlight driving signals in RTL logic** option:

1. Select **Edit > Preferences**.

The Applications Preferences dialog box appears.

2. In the Driver Pane category, select **Highlight driving signals in RTL logic** (as shown in [Figure 9-5](#)) and then click **Apply**.
3. Click **OK**.

Figure 9-5 Selecting Highlight driving signals in RTL logic Option



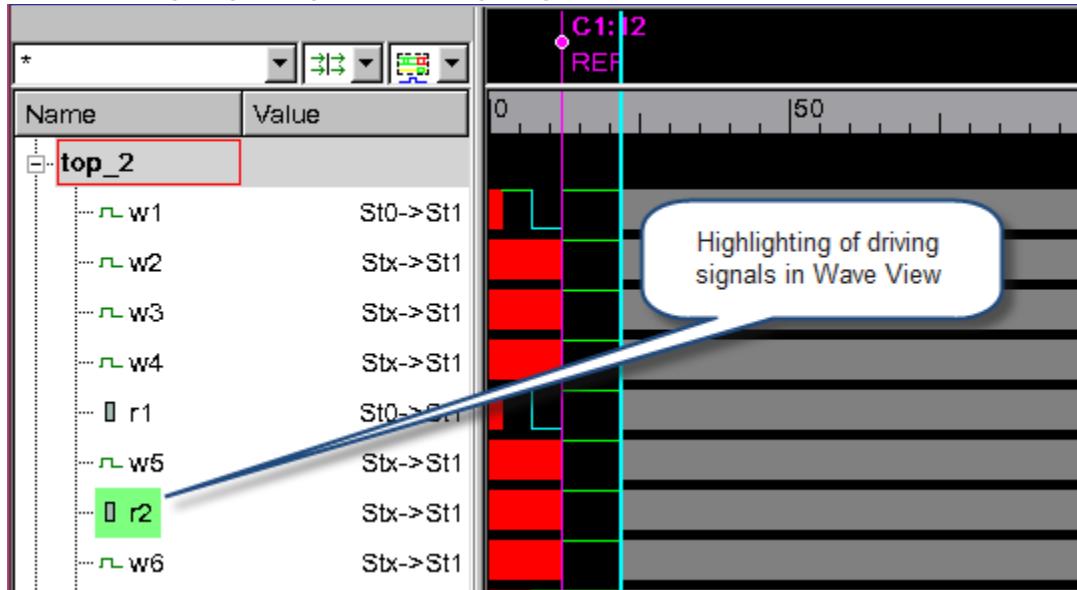
DVE highlights signals on the right-hand side (RHS) and indexes on left-hand side (LHS) of an assignment. When you trace another signal, DVE clears the highlighting from the previous driving signals.

Note:

- If you expand a traced signal, DVE automatically traces the driving signal.
- Only signals on the RHS that cause value changes on signals or variables on the LHS are highlighted.

[Figure 9-6 illustrates highlighting of driving signals in the Wave View.](#)

Figure 9-6 Highlighting of Driving Signals in the Wave View



If you double-click a driver or load in the Driver Pane, its driving signals are highlighted in the Source and Path Schematic Views.

Note:

Signal highlighting is not done if an assignment statement contains non-dumped variables, dynamic variables, or function calls.

Highlighting Driving Signals in Path Schematic View

DVE highlights the driving signal in the Path Schematic View, as shown in [Figure 9-7](#).

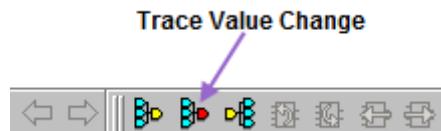
Figure 9-7 Highlighting of Driving Signals in Path Schematic View



Tracing Signal Values over Combinational Logic

You can use the **Trace Value Change** menu command or its equivalent toolbar icon to backtrace a signal value through multiple active driver statements in RTL combinational logic (see [Figure 9-8](#)). You can also execute this command by right-clicking on a signal in the Driver Pane.

Figure 9-8 Trace Value Change Toolbar Icon

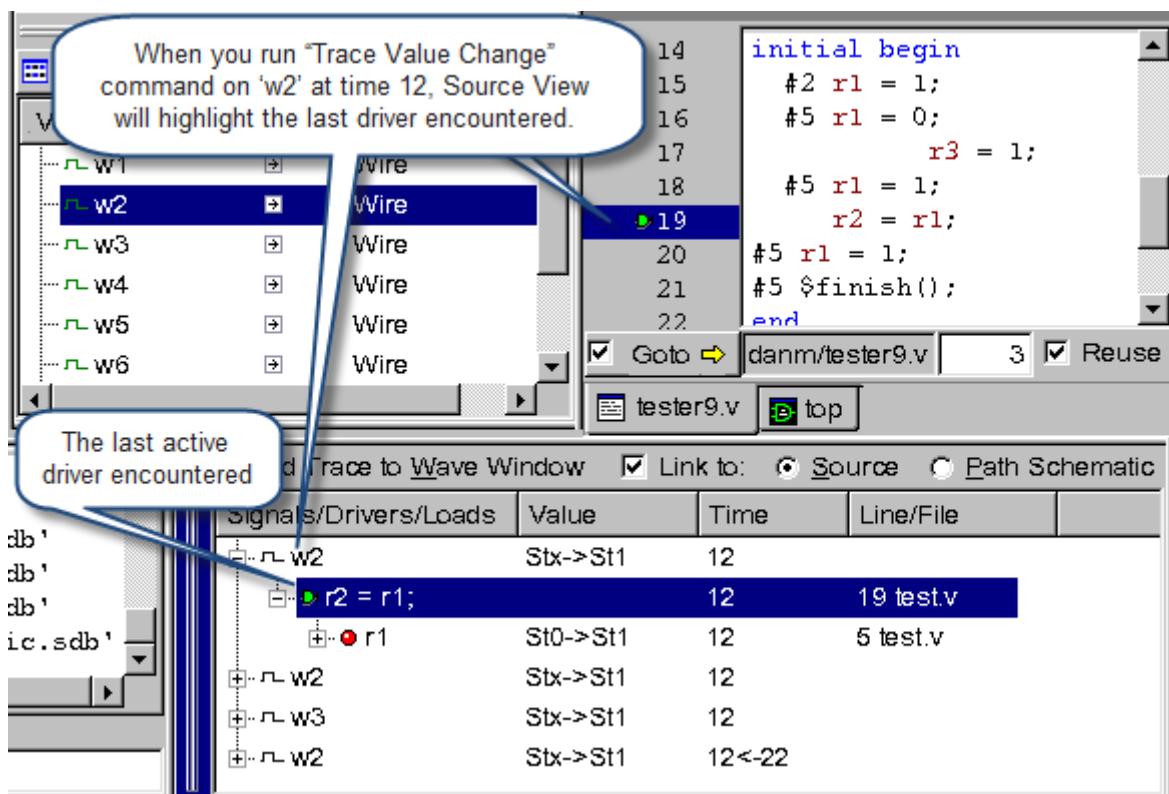


This command is enabled only when you select a signal in the Data Pane, Source View, or Driver Pane. When you execute this command, DVE takes the current application time (C1) of the selected signal and backtraces its value at that time.

DVE backtraces by finding an active statement driver for a signal, tracing its driving signal, and then repeating this on the driving signal until more than one driving signal or non-combinational logic is

encountered. The results of backtracing are shown in the Source View and Driver Pane just as driver or active drivers are shown (see Figure 9-9).

Figure 9-9 Tracing Signal Values over Combinational Logic



Note:

By default, intermediate drivers are not shown in the Driver Pane.

DVE stops backtracing in the following cases:

- Active statement driver is not a continuous assignment or an assignment in combinational logic.
- Driving signals cannot be determined due to missing dump, dynamic variables, function calls, or other limitations.

- There are no driving signals (for example, assignments to constants).
- There is more than one driving signal. In this case, these driving signals are highlighted in the source window. To continue, you can run the **Trace Value Change** command on one of them.
- There is more than one active driver; for example, due to active drivers limitations, an incomplete dump, or different bits of the traced signal driven by different drivers.

You can use the **Stop at port when tracing value change** value tracing option to stop backtracing if a driving signal on the RHS of the driver is a port. Follow these steps:

1. Select **Edit > Preferences**.

The Applications Preferences dialog box appears.

2. In the Driver Pane category, select **Stop at port when tracing value change**, and then click **Apply**.

3. Click **OK**.

[Example 9-2](#) shows an example test.v file.

Example 9-2 test.v File

```
module top();
    wire w1, w2, w3, w4, w5, w6;
    reg r1, r2;

    assign w1 = r1;
    assign w2 = r2;
    assign w3 = w2;
    assign w4 = w3;
    assign w5 = w4;
```

```
assign w6 = w5;

initial begin
    #2 r1 = 1;
    #5 r1 = 0;
    #5 r1 = 1;
    r2 = r1;
#5 r1 = 1;
#5 $finish();
end
endmodule
```

Compile the test.v file shown in [Example 9-2](#):

```
% vcs -debug_all test.v
```

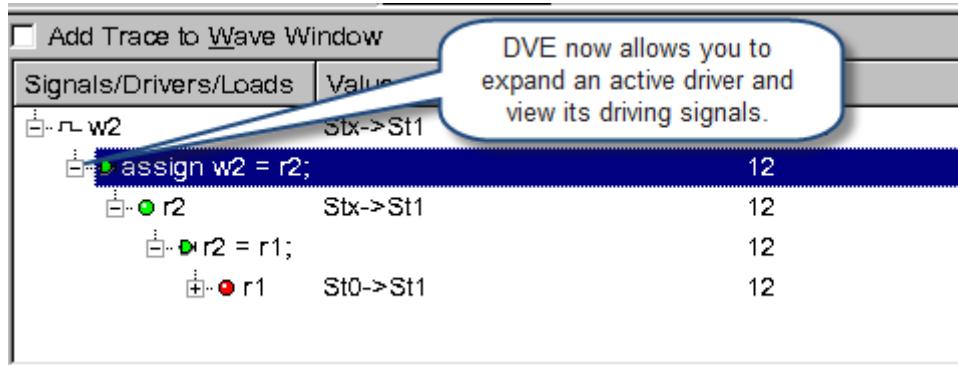
Invoke the DVE GUI:

```
% ./simv -gui&
```

Incremental Active-driver Tracing in Driver Pane

You can expand an active driver in the Driver Pane and view its driving signals, as shown in [Figure 9-10](#). This feature allows you to quickly trace multiple levels of drivers and display the hierarchical relationships between drivers.

Figure 9-10 Expanding an Active Driver in Driver Pane



If DVE does not find any driving signals for an active driver, it displays contributing signals, which can be:

- Assignment or continuous assignment signals at the RHS and in the index of the LHS.
- Primitive gates and UDP input and inout ports.
- PortOut drivers such as contributing signals for expressions connected to input, inout, or ref ports of the corresponding instance.

DVE displays contributing signals using special icons (see [Table 9-1](#)).

Table 9-1 Icons for Contributing Signals

Icon	Icon Name	Description
	Green circle	DVE displays this icon when the contributor is completely analyzed and detected as an active driving signal. This is usually the case when a driver is in combinational logic or a flip-flop, and no limitation is encountered.
	Yellow circle	DVE displays this icon when the contributor time is accurate but it is unable to determine if the signal really caused a value change of the traced signal. This is usually the case when a driver is in combinational logic, but DVE encounters some active driver limitations (for example, a function call on RHS). Also, this icon is used for contributors at time 0 in combinational logic and for the first clock transition in flip-flops.
	Red Circle	DVE displays this icon when it is not able to analyze the contributor. This happens when the driver is not in RTL code (for example, in testbench code) or when DVE encounters some active driver limitations.

Mousing over these icons displays ToolTips with relevant contributor signal information, as shown in [Figure 9-11](#). The Time column of contributing signals displays the time used to trace the parent driver item. The Value column displays signal values at this time.

You can also expand contributing signals. Expanding them triggers active drivers tracing at the time shown in the Time column for the signal.

Figure 9-11 ToolTip Displaying Contributor Signal Information

Signals/Drivers/Loads	Value	Time
↳ w2	Stx->St1	
↳ assign w2 = r2;		
↳ r2	Stx->St1	
↳ r2 = r1;		12
↳ r1	St0->St1	12
↳ (Driver) w6	Contributor was not analyzed or contributor time could not b...	

Note:

When you trace at time 0, all contributing signals are displayed with a yellow icon, and no signal highlighting is done in the Source View.

If the signal hierarchy information does not fit into the visible area of the Driver Pane, you can run the Trace Drivers command on a desired signal in that hierarchy and continue from the top level, as shown in [Figure 9-12](#).

Figure 9-12 Hierarchy Information in Data Pane

Add Trace to Wave	
Signals/Drivers/Loads	Top-level of signal w4.
↳ w4	Stx->St1
↳ assign w4 = w3;	
↳ w3	Stx->St1
↳ w6	Stx->St1
↳ assign w6 = w5 & r3;	
↳ w5	Stx->St1
↳ assign w5 = w4;	
↳ w4	Stx->St1
	12
	12
	12
	12
	12
	12

The feature is also available for drivers added using a **Trace Value Change** command. However, for such items, expanding their contributing signals triggers another **Trace Value Change**

operation. This allows you to quickly explore several alternatives (for example, when there are multiple driving signals or active statement drivers).

Viewing Intermediate Drivers

When you run the **Trace Value Change** command on a signal, it only displays the last driver encountered. It does not display intermediate drivers between a signal and the last driver encountered.

Sometimes, you may want to see all intermediate drivers (for example, if the resulting driver is unexpected).

You can use the **Show intermediate drivers** menu command or **Show intermediate drivers found when tracing value change** Preference option to view intermediate drivers found when tracing value changes. By default, this option is disabled.

You can enable this option in several different ways:

- Select **Edit > Preferences**. Then from the Application Preferences dialog, select **Show intermediate drivers found when tracing value change** option. Click **Apply** and **OK**.
- Select **Trace -> Drivers/Loads -> Show intermediate drivers**.
- Right-click on a signal in the Driver Pane.

Figure 9-13 Viewing Intermediate Drivers in Driver Pane

Signals/Drivers/Loads		Value	Time
w5		Stx->St1	12
assign w5 = w4;			
w4		Stx->St1	12
assign w4 = w3;			
w3		Stx->St1	12
assign w3 = w2;			
w2		Stx->St1	12
assign w2 = r2;			
r2		Stx->St1	12
assign r2 = r1;			
w4		Stx->St1	12
w6		Stx->St1	12
w2		Stx->St1	12
(Driver) w6			

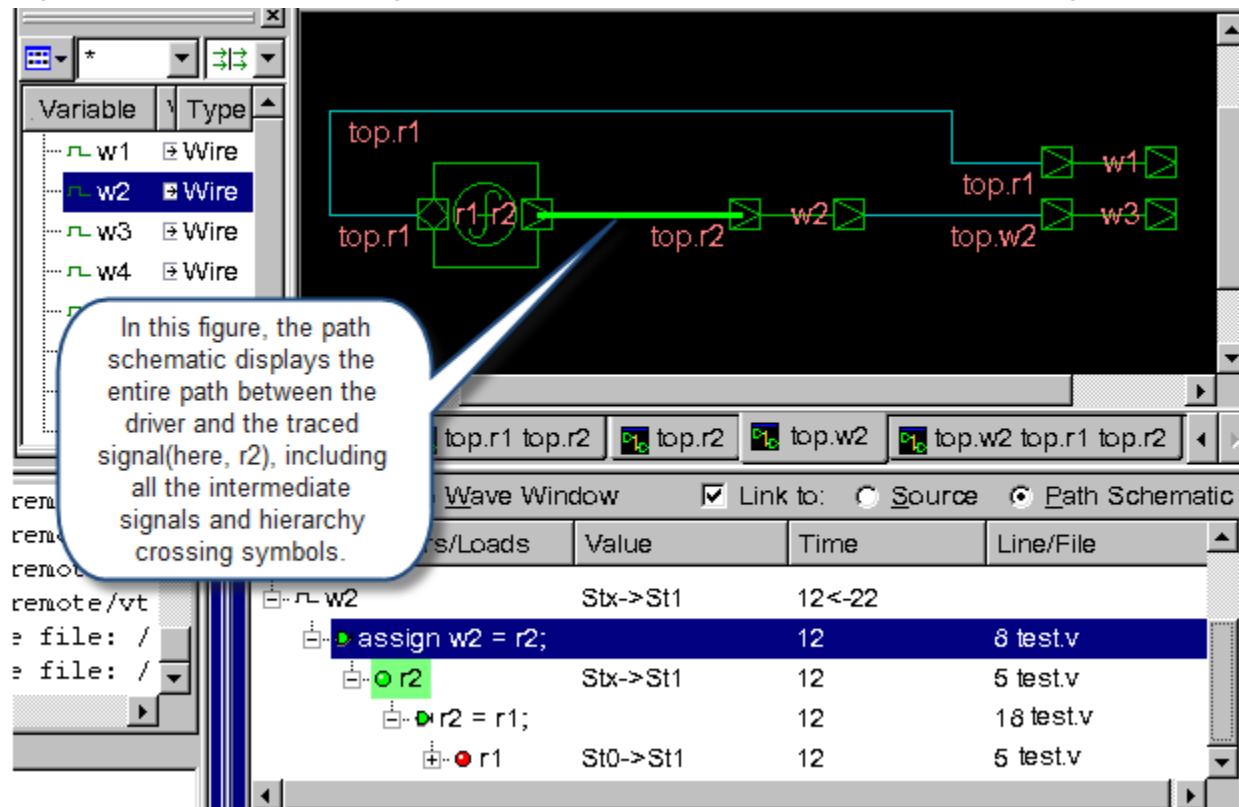
Visualizing the Path Between Driver and Traced Signal

You can use the Driver Pane to view the path between the driver and the traced signal in the Path Schematic View. Right-click on a signal and select **Show Path Schematic**. When using this feature, note the following:

- If you invoke an intermediate driver from the Driver Pane, the Path Schematic displays the entire path between the intermediate driver and the traced signal, including all the intermediate drivers between them.

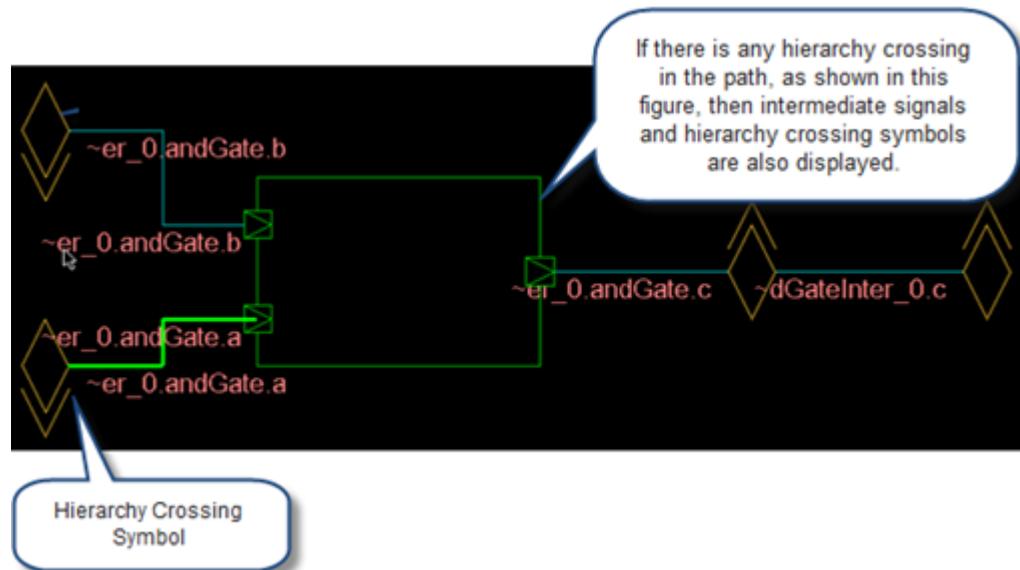
If there is a hierarchy in this path, the intermediate signals and hierarchy crossing symbols are also displayed in the path schematic, as shown in [Figure 9-14](#).

Figure 9-14 Visualizing the Path Between Driver and Traced Signal



- If there is any hierarchy crossing between the driver and the traced signal, the path schematic displays the entire path, including all the intermediate signals and hierarchy crossing symbols, as shown in [Figure 9-15](#).

Figure 9-15 Hierarchy Crossing Symbol



Multicycle Support for Value Tracing

DVE simplifies value tracing in RTL designs by recognizing certain flip-flop coding styles and finding driving signals for such flip-flops. Consider the code shown in [Example 9-3](#).

Example 9-3 test.v File

```
module top();

reg dout, enable, scanIn, din;
reg clk;

always @(negedge clk)
begin
    dout <= enable ? scanIn : din;
end

initial begin
    enable = 1;
    scanIn = 1;

```

```

    clk = 0;
#10;
enable = 0;
din = 1;
#10;
din = 0;
#10;
din = 1;
end

always
#2 clk = !clk;

endmodule

```

DVE recognizes the code in the first `always` block in [Example 9-3](#) as a flip-flop and finds the driving signal (`din`) and active time for this driving signal.

DVE highlights the driving signal `din` in all windows (Source View, Wave View, Path Schematic View, Data Pane, Driver Pane, and so on), as shown in [Figure 9-4](#) and [Figure 9-7](#).

If you declare a flip-flop cell using a ``celldefine` directive and enable the DVE preference option **Treat modules defined with ``celldefine` as black-box (library) cells**, DVE finds the flip-flop inside the cell and displays the result as an input port to the cell instance.

During the first clock cycle (for example, first negedge clock event in the simulation), DVE displays all signals from the RHS of the flip-flop using a yellow icon.

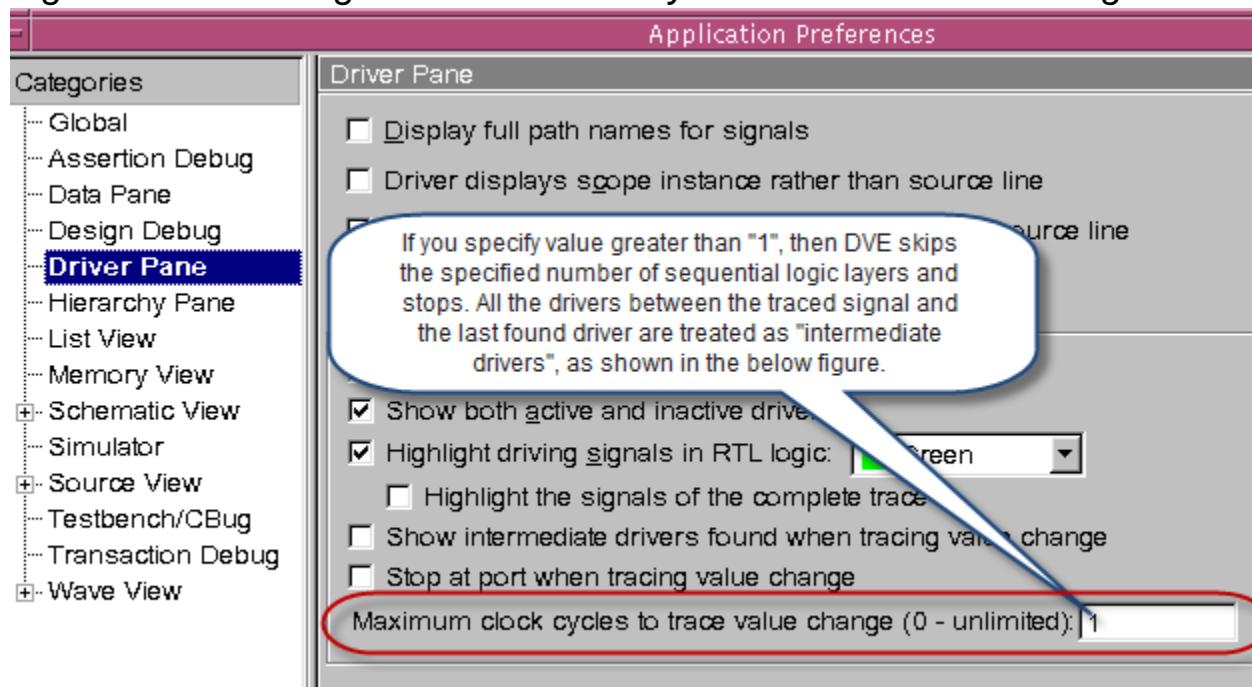
Note:

DVE recognizes only the `always` block as a flip-flop. If DVE does not recognize the construct as a flip-flop or combinational logic, it displays the driver contributors using red icons, as described in “[Incremental Active-driver Tracing in Driver Pane](#)” on page 16.

Specifying Maximum Clock Cycles to Trace Value Change

You can use the Preference option **Maximum clock cycles to trace value change** to set the maximum number of clock cycles to trace value changes in RTL logic. To use this option, select **Edit > Preferences**. The Application Preferences dialog box appears. For information on setting the maximum number of clock cycles to trace value changes, see [Figure 9-16](#).

Figure 9-16 Setting Maximum Clock Cycles to Trace Value Change



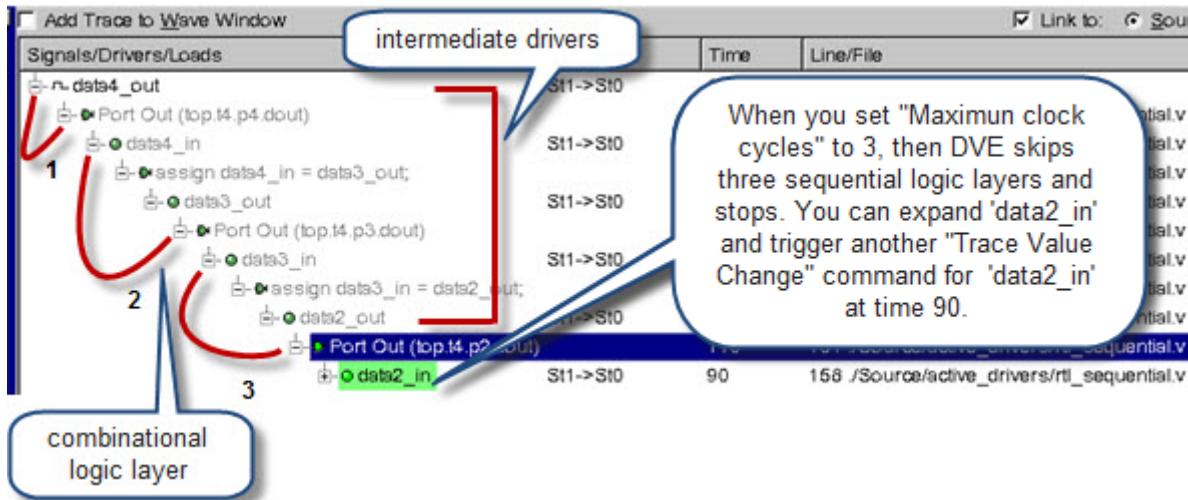
[Table 9-2](#) lists information about value settings in the **Maximum clock cycles to trace value change** Preference setting.

Table 9-2 Maximum Clock Cycles to Trace Value Change.

Value	Description
0	Unlimited tracing. This means DVE traces value changes until the root cause of the signal value change is found or some other condition, as described in “Tracing Signal Values over Combinational Logic” on page 13 , is encountered (for example, there is more than one active driver or more than one contributing signal for certain driver). You can also cancel backtracing in this case using the <ESC> key.
1	Default value. If you use this value, DVE traces value changes over combinational logic only, and stops as soon as it encounters a flip-flop or some other non-combinational code. The behavior in this case is the same as described in “Tracing Signal Values over Combinational Logic” on page 13 .
Value greater than the default value(1)	DVE skips the specified number of sequential logic layers and stops. All drivers between the traced signal and the last found driver are treated as intermediate drivers, as described in “Viewing Intermediate Drivers” on page 20 .

[Figure 9-17 on page 27](#) shows an example for Trace Value Change. In this example, the **Maximum clock cycles to trace value change** option is set to 3 and the **Show intermediate drivers** command is enabled.

Figure 9-17 Tracing Value Change over Multiple Clock Cycles



Active Drivers Support for PLI, UCLI, and DVE Forces

Active drivers can now detect signal forces applied from PLI, DVE, and UCLI. You can force a value from PLI, DVE, and UCLI, as shown below:

- PLI — Using the `vpi_put_value` function call.
- UCLI — Using the `force` command.
- DVE — Using the **Simulator > Force Value** menu option.

DVE now displays the forced active drivers for a signal that has a PLI, UCLI, or DVE force at a particular trace time, with a special notation `<forced driver>`, as shown in [Figure 9-18](#).

Figure 9-18 Forced Driver

The screenshot shows a software interface for tracing drivers and loads. At the top, there's a menu bar with 'Add Trace to Wave Window', 'Link to:', 'Source' (which is checked), and 'Path Schematic'. Below the menu is a table titled 'Signals/Drivers/Loads' with four columns: 'Signals/Drivers/Loads', 'Value', 'Time', and 'Line/File'. The table contains one row for a signal named 'n~b'. The 'Value' column shows '0->^1'. The 'Time' column shows '5<-6'. The 'Line/File' column shows 'N/A'. A blue bar highlights the entire row for 'n~b'.

Signals/Drivers/Loads	Value	Time	Line/File
n~b	0->^1	5<-6	N/A

The forced value is displayed with a special prefix \wedge in the "Value" column of all views. For example, Figure 9-18 shows $0 \rightarrow \wedge 1$ transition at time 5, which means that the signal is forced to value 1 at time 5.

For the traced signal, DVE displays the 'driver change time' (signal value change time) and 'start trace time' (time at which tracing is performed) information in "Time" column. For example, in the above figure, this information is shown in the "Time" column as $5 < -6$, where 5 is the 'driver change time' and 6 is the 'start trace time'.

If both 'driver change time' and 'start trace time' are same, then DVE displays only 'driver change time' instead of 'driver change time<- start trace time'.

If the PLI, UCLI, or DVE force on a driver is released, but its value is not changed by some design code, then DVE displays the released driver with a special notation `<force released>`, as shown in Figure 9-19.

Figure 9-19 Force Released

Add Trace to Wave Window			
Signals/Drivers/Loads	Value	Time	Line/File
⊕-n~b	$^1\rightarrow 1$	10<-15	
└ <force released>		10	N/A

In the waveform, DVE shows the forced and released drivers in the form of two yellow triangle symbols, as shown in Figure 9-20.

Figure 9-20 Forced and released drivers in the form of two yellow triangle symbols

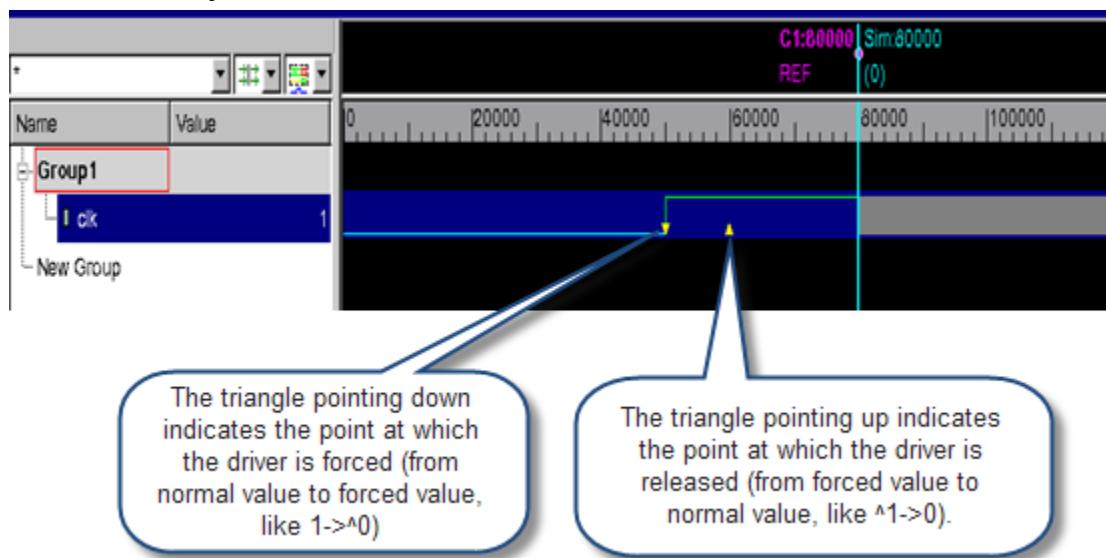
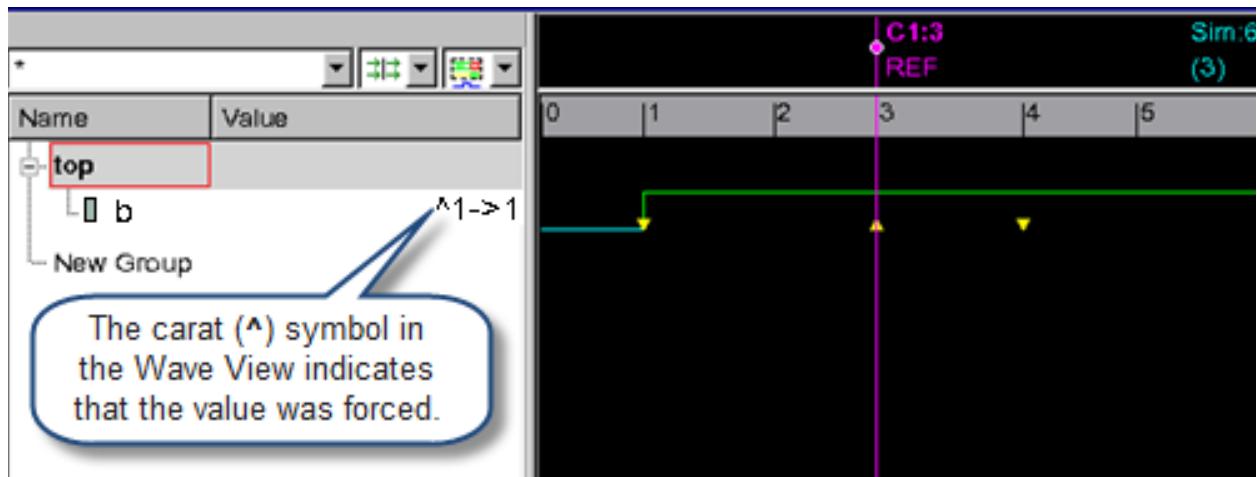


Figure 9-21 Carat symbol in the Wave View



Usage Example

Consider the following example testcase `testcase.sv`:

```
module top;
    bit b;
    initial
        begin
            #0 b<=1'b1;
            #1 b<=1'b0;
            #5 b<=1'b1;
            #10 $finish;
        end
    endmodule
```

To view the forced/released signal values:

1. Compile the above example code

```
% vcs -debug_all -sverilog test.sv
```

2. Open the DVE GUI

```
% ./simv -gui
```

3. Perform the following commands in the DVE GUI:

```
Dve%dump -add /
```

```
Dve%run 5
```

```
Dve%force top.b 1'b1 //force value at time "5"
```

```
Dve%run 5
```

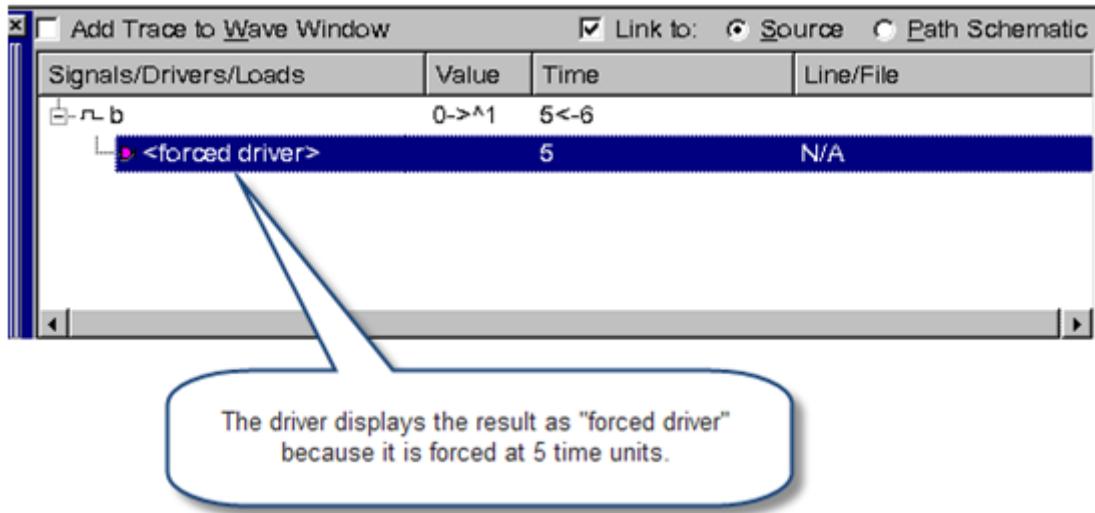
```
Dve%release top.b //release value at time "10"
```

```
Dve%run
```

4. Right-click on the `top` module and select **Add to Waves > New Wave view**.
5. In the wave view, select signal 'b' from the signal pane.
6. Move the cursor to 6 time units.
7. From the **Trace** menu, click **Trace Drivers**.

The forced driver appears in the Signals/Drivers/Loads pane, as shown in [Figure 9-22](#).

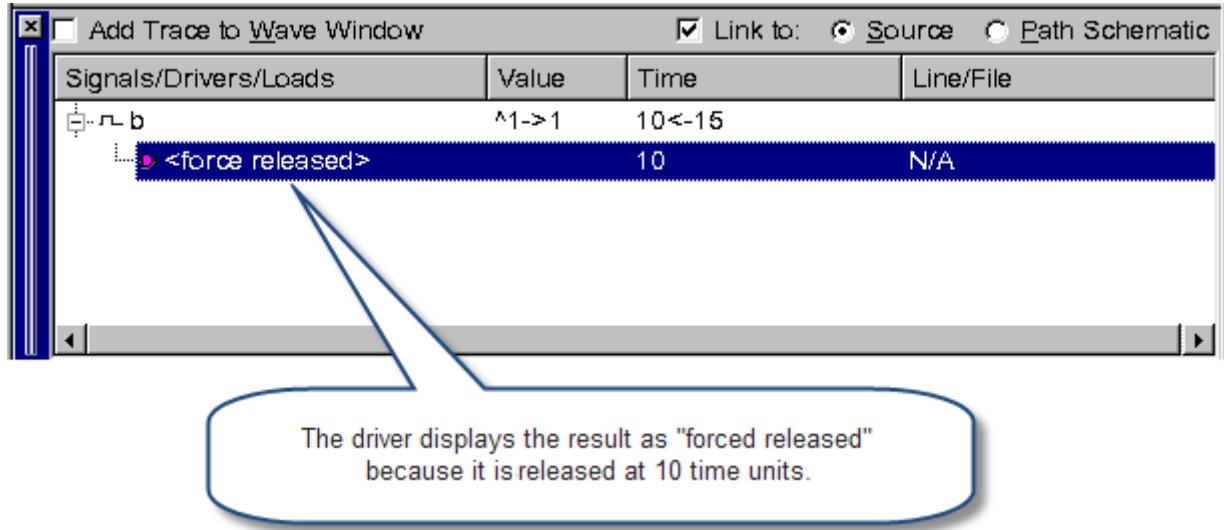
Figure 9-22 Forced driver in the Signals/Drivers/Loads pane



8. Move the cursor to 15 time units.
9. From the **Trace** menu, click **Trace Drivers**.

The driver displays the result as “force released” in the Signals/Drivers/Loads pane, as shown in [Figure 9-23](#).

Figure 9-23 Driver displaying result as “force released”



Driver Tracing Support for Virtual Interfaces and Clocking Blocks

DVE extends the driver tracing functionality by allowing you to view all assignments using virtual interface or clocking block, which caused the value change. DVE displays these assignments as a list of drivers in the Driver Pane.

Driver Tracing Support for Virtual Interfaces

Consider the following test case where interface is instantiated in top-level module, and members of this interface are changed from test bench code using virtual interface declared in a class task.

Example 9-4 Driver Tracing Test File (vitest.v)

```
interface interface1;
    logic x;
endinterface
class class1;
    virtual interface1 vc;

    function new(virtual interface1 i);
        vc = i;
    endfunction
    task test1;
        #1;
        vc.x = 0;
    endtask
    task test2(logic x);
        #1;
        vc.x = x;
    endtask
endclass
module top;
    interface1 i1();
    //... DUT connected to 'i1' members...
    class1 c1;
```

```
initial begin
    $vcpluson();
    c1 = new(i1);
    #10;
    c1.test1();
    #10;
    c1.test2(1);
    #10;
    $finish();
end
endmodule
```

Compile the `vitest.v` file shown in [Example 9-4](#):

```
% vcs -sverilog -nc -debug_all vitest.v
```

Invoke the DVE GUI, as follows:

```
% ./simv -gui&
```

Run the `vitest.v` file and trace drivers for the signal `top.i1.x`. All assignments using virtual interface of matching type are displayed in the Driver Pane, as shown in [Figure 9-24](#).

Note:

Since dynamic variables (virtual interfaces) are involved on LHS of assignments, no active driver analysis will be done in this case. Therefore, DVE displays all assignments using virtual interface as static drivers.

Figure 9-24 Driver Pane Showing all Assignments Using Virtual Interface

<input type="checkbox"/> Add Trace to Wave Window		<input checked="" type="checkbox"/> Link to: <input checked="" type="radio"/> Source <input type="radio"/> Path Schematic	
Signals/Drivers/Loads	Value	Time	Line/File
↳ x	1'b0->1'b1	22<-32	
↳ vc.x = 0;		22	11 vitest.v
↳ vc.x = x;		22	15 vitest.v

DVE also supports this feature in the following cases:

- Driving interface members over modport.
- Driving interface members over combination of modport and clocking block with virtual interfaces.

Driver Tracing Support for Clocking Blocks

Consider the following test case where interface is instantiated in top-level module, and members of this interface are changed using a clocking block declared in it. The clocking block member is changed from testbench using virtual interface.

Example 9-5 Driver Tracing Test File (clkbltest.v)

```
interface interface1;
    logic x;
    logic clock = 0;
    always #10 clock = ~clock;
    clocking cb @(posedge clock);
        output x;
    endclocking
endinterface
class class1;
    virtual interface1 vc;
    function new(virtual interface1 i);
        vc = i;
    endfunction
endclass
```

```

endfunction
task test1;
#1;
vc.cb.x <= 0;
endtask
task test2(logic x);
#1;
vc.cb.x <= x;
endtask
endclass
module top;
interface1 i1();
//... DUT connected to 'i1' members...
class1 c1;
initial begin
$vcddpluson();
c1 = new(i1);
#10;
c1.test1();
#10;
c1.test2(1);
#10;
$finish();
end
endmodule

```

Compile the `clkbltest.v` file shown in [Example 9-5](#):

```
% vcs -sverilog -nc -debug_all clkbltest.v
```

Invoke the DVE GUI, as follows:

```
% ./simv -gui&
```

DVE displays the clocking block member, which causes the value change, as a contributor icon in the Driver Pane. In the above case, DVE displays clocking block member as a contributor for “Clocking Out” driver, as shown in [Figure 9-25](#).

Figure 9-25 Clocking Out Driver with Contributor

DVE displays clocking block member as a contributor for "Clocking Out" driver. You can expand this contributor signal to view its drivers.

Signals/Drivers/Loads	Value	Time	Line/File
↳ x	1'b->1'b1	30<-32	
↳ Clocking Out (top.i1.cb.x)		30	6 clkbltest.v
↳ x	1'b1	30	6 clkbltest.v
↳ vc.cb.x <= 0;		22	16 clkbltest.v
↳ vc.cb.x <= x;		22	20 clkbltest.v

Similarly, you can also do active driver analysis for "Clocking In" driver, which is returned when tracing "input" clocking block member. The contributor will be interface member in this case.

Active Driver Analysis for Clocking InOut Driver

In case of "Clocking InOut" driver, you cannot do active driver analysis and analyze its contributor, when you trace interface member. However, DVE displays input drivers in this case.

When you trace clocking block member, DVE displays both input driver and output driver of the "Clocking InOut" driver in the Driver Pane.

For example, change the clocking block code in [Example 9-5](#), as follows:

```
clocking cb @ (posedge clock);
  inout x;
endclocking
```

If you trace interface member `top.i1.x`, then DVE displays the "Clocking InOut" driver, as shown in [Figure 9-26](#).

Figure 9-26 Viewing the “Clocking InOut” Driver

Add Trace to Wave Window			
Signals/Drivers/Loads	Value	Time	Line/File
..< x	1'bx->1'b1	30<-32	
Clocking InOut (top.i1.cb.x)		30	6 clkbltest1.v
x	1'bx	30	6 clkbltest1.v

You cannot do active driver analysis for this input driver and analyze its contributor.

DVE displays this icon when it cannot analyze the contributor.

If you trace clocking block member top.i1.cb.x, then DVE displays both input and output drivers of the “Clocking InOut” driver, as shown in [Figure 9-27](#).

Figure 9-27 Viewing Input and Output Drivers of the “Clocking InOut” Driver

Signals/Drivers/Loads	Value	Time	Line/File
..< x	1'bx	0<-32	
vc.cb.x <= 0;	0	0	16 clkbltest1.v
vc.cb.x <= x;	0	0	20 clkbltest1.v
Clocking InOut (top.i1.cb.x)	0	0	6 clkbltest1.v

DVE displays both input and output drivers when you trace drivers for inout clocking block member.

Limitations

DVE does not support active driver analysis for “Clocking InOut” drivers.

Active Driver Limitations

- If there is ‘timing specification’ for primitive gates and UDPs, then you cannot do Active Driver analysis for these drivers.
 - Driving signal detection is not done when the RHS or LHS index contains one of the following:
 - Function call (including system function calls).
 - Indexed part select (example: `a [2+ : 4]`).
 - Stream operator (example: `{ >> { a, b, c } }`).
- In this case, DVE displays all signals with transitions.
- You cannot perform active driver analysis for SystemVerilog and Verilog code in the following cases:
 - No driving signal analysis will be done for sequential UDPs, and “Trace Value Change” will always stop at them (that is, multi-cycle tracing is not possible in this case).
 - If the traced signal is an output of timing check system call (notifier, delayed clock, or data).
 - Task/function calls (including system calls) in the RHS of a driver statement or in control logic preceding the driver under analysis.
 - Loops (for example, for/while) with non-zero delays preceding the driver under analysis. Driver under analysis inside a loop.
 - Sequence match or event used in control logic preceding the driver under analysis (even if sequences are dumped with `- assert dump_sequences`).
 - Constructs not currently supported in VPI.

- When design contains the `$sdf_annotate()` call.
- Traced signal or driver is located in the module with specify blocks.
- Currently in Driver Pane, DVE shows a statement as an active statement driver if it is driving the last value change of the signal under query.

Consider the following code:

```
always @(posedge clk or posedge rst)
if (rst)
    q <= 1'b0;
else
    q <= d;
```

For example, if the `if` block is active at time 5, then statement driving `q` is `q<=1'b0`; And at time 20, `else` block gets active, now the statement driving `q` is `q<=d`.

However, if the value of `d` is `1'b0` at time 20, then active statement driver for `q` will still be shown as `q <= 1'b0` at time 5, as there was no value change on `q` at time 20.

Limitations of Active Drivers Support for PLI, UCLI, and DVE Forces

- If you apply force deposit on some elements of MDA, then complete MDA is shown as forced. However, if you expand MDA, then individual elements are shown as forced.
- This feature does not support variables of type ‘real’.

10

Using the Assertion Pane

The Assertion pane displays SVA and OVA assertion and cover properties results. This chapter includes the following topics:

- “Compiling SystemVerilog Assertions”
- “Displaying Assertions”
- “Displaying Cover Properties”
- “Debugging SystemVerilog Immediate and Concurrent Assertions”

Compiling SystemVerilog Assertions

Use the `-assert dve` flag on the VCS command line when compiling SystemVerilog assertions (SVA) for debugging with DVE. You need to use the `-debug` option to enable SVA tracing in DVE.

Note:

The link step can take a long time if you use a Solaris linker prior to version 5.8.

To avoid linking delays when using DVE to debug designs compiled on Solaris, perform either of the following:

- Make sure your Solaris C compiler is version 5.8 or above. To check your compiler version, enter the following on the command line:

```
ld -v
```

The system returns your linker version, for example:

```
ld: Software Generation Utilities - Solaris Link Editors:  
5.8-1.283
```

- Use the gcc C compiler when compiling your design. For example:

```
vcs -assert dve -debug_pp -sverilog a.v -ld gcc
```

Displaying Assertions

DVE displays assertion results in the Assertion pane by instance, start and end times of assertion events, the delta, the assertion failures, total failures, real and vacuous successes, incomplete and attempted assertions. Successful assertions are displayed in green, vacuous successes in brown, and failed assertions in red.

Figure 10-1 Assertion Results

The screenshot shows the DVE Assertion pane with the following data:

Assertions		all	at time	current	to end	<input checked="" type="radio"/> failures	<input type="radio"/> incompletes	<input type="radio"/> successes	<input type="radio"/> unattempted	<input type="radio"/> all
Name	Instance	Start	End	Delta	Reason	Failures	Successes	Incompletes	Attempts	
operator_preced_and_or_delay1	testbench.d0	100	100	0	(c == 1'b0)	2	2	1	5	
Failure1		100	100	0	(c == 1'b0)					
Failure2		140	140	0	(c == 1'b0)					
Incomplete1		120	155	35						
Success1		110	110	0						
Success2		130	130	0						

Below the table are filters: Name filter: *, Instance filter: *, #Attempts - failures: 10, successes: 10, incompletes: 10.

When you open a design that contains assertions, DVE displays the Assertion pane even if all the assertions pass. The default is to display failed assertions.

To display assertion

1. Run the design containing assertions and open DVE in interactive mode.

The scope containing assertion is loaded in the Hierarchy pane.

2. Expand the scope and click the assertion to display the variables in the Data pane.
3. Double-click the assertion variable to it in the Source view.
4. Run the simulation to view the assertion results in the Assertion pane.

The Assertion pane is not displayed by default. You need to change the preference setting to automatically display the Assertion pane.

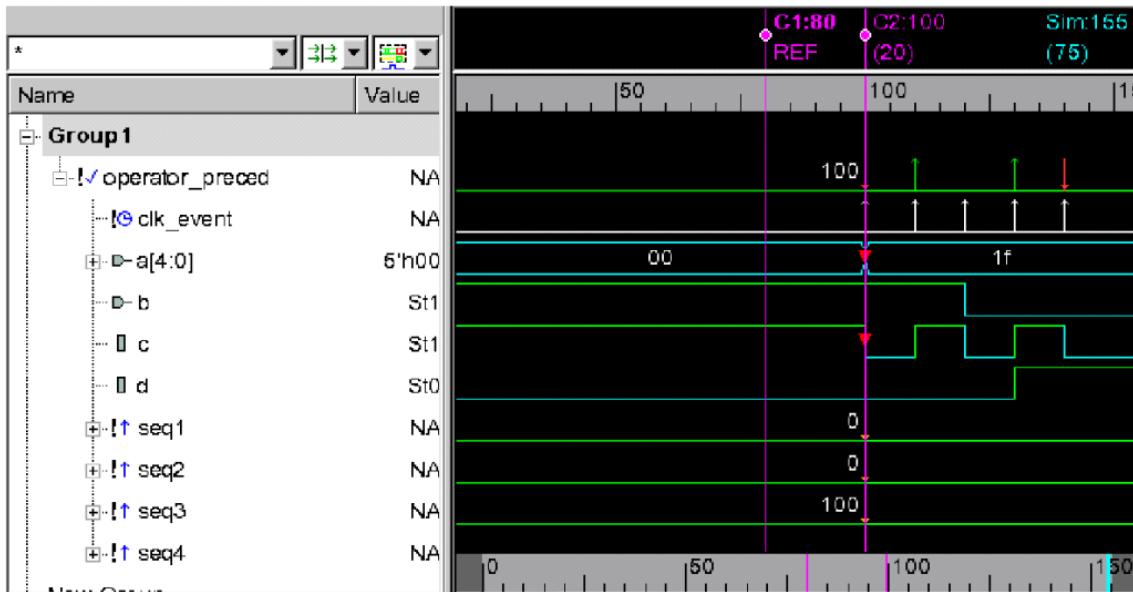
Viewing Assertion in the Wave View

To view assertion in the Wave view, double-click the assertion attempts in the Assertion pane. You can also select the assertion attempt, right-click and select **Trace Assertions** to trace the assertion in the Wave view.

The Wave view displays the assertions as follows:

- The cursors in the Wave view mark the start and end time of the selected assertion with the area between the cursors grayed.
- A green circle indicates a signal value at a specific time that contributed to a successful sub-expression in the assertion.
- A red circle indicates a signal value at the time which caused a sub-expression to fail. A sub-expression failing may result in the overall assertion failing.
- To display the first 10 failures and successes, click the "+" next to an assertion of interest. [Figure 10-2](#) shows an assertion with no delta between the start and end time.

Figure 10-2 Assertion in the Wave view



- In Signal Group 1, the assertion `operator_preced` is listed first in the tree view. This is the assertion result signal. The waveform consists of red, green, and white arrows. Green arrows indicate where the assertion was determined to be a success, and red arrows indicate where it failed; the red arrow illustrates the first failure. A white arrow indicates assertion clock events.
- `operator_preced` is expanded into the following components:
 - The first component is `clk_event`. Each clock event shows you when the assertion fired and the clock ticks that happen for sequences.
 - The rest of the signals are those that contributed to the success or failure.
 - The green dots on the waveform indicate that the value of the signal is as expected at that clock tick. The red dots indicate that the signal contributed to the failure of the assertion at that clock tick.

- Hold the mouse cursor over the assertion to view the tooltip that displays details about the assertion failure or success. For each success or failure attempt, the tooltip contains start time, result, and reason.

Displaying Cover Properties

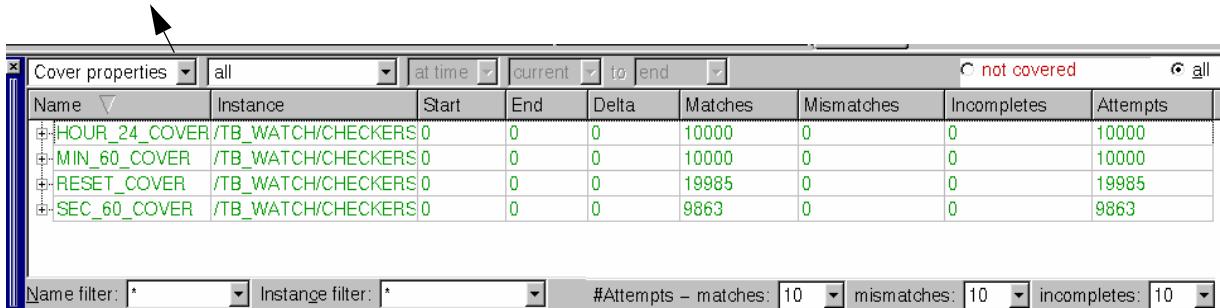
DVE displays Cover properties in the Assertion pane by instance, start and end time, the delta, total number of matches, mismatches, incomplete and attempted cover properties. You can view the cover properties in other views, such as Wave view and List view and also create groups for the cover properties in the Signal pane.

To view cover properties

1. Run the design containing cover properties and open DVE in interactive mode.
The scope containing cover properties is loaded in the Hierarchy pane.
2. Select the scope to display the cover properties in the Data pane.
3. Double-click the cover property variable to view it in the Source view.
4. Run the simulation.

The Assertion pane opens. The Assertion pane is not displayed by default. You need to change the preference setting to automatically display the Assertion pane.

Select **Cover properties** in this drop-down list



A screenshot of the Assertion pane in VCS. The title bar says "Assertion". The pane contains a table with the following columns: Name, Instance, Start, End, Delta, Matches, Mismatches, Incompletes, and Attempts. There are four rows in the table:

Name	Instance	Start	End	Delta	Matches	Mismatches	Incompletes	Attempts
HOUR_24_COVER	/TB_WATCH/CHECKERS 0	0	0	10000	0	0	0	10000
MIN_60_COVER	/TB_WATCH/CHECKERS 0	0	0	10000	0	0	0	10000
RESET_COVER	/TB_WATCH/CHECKERS 0	0	0	19985	0	0	0	19985
SEC_60_COVER	/TB_WATCH/CHECKERS 0	0	0	9863	0	0	0	9863

Below the table are filters: Name filter: * and Instance filter: *. At the bottom are buttons for #Attempts – matches: 10, mismatches: 10, incompletes: 10.

5. Select the drop-down at top left of the Assertions pane, and select Cover properties.

The cover properties are displayed.

Debugging SystemVerilog Immediate and Concurrent Assertions

VCS supports debugging SystemVerilog (SV) immediate and concurrent assertions using the `+vpi` compile-time option, as described below.

With this option, you can,

- get handles to assertions inside a scope using the VPI routines `vpi_iterate()` or `vpi_handle_by_name()`.
- control individual assertions by switching them ON and OFF using the VPI routine `vpi_control()`.

- register callbacks on assertions using the VPI routine `vpi_register_assertion_cb`.

For more information about the VPI routines, see the *IEEE SystemVerilog LRM*.

Usage Model

Compile the design with the `+vpi` option to access all assertions in the design and dynamically trace the assertions through assertion callbacks using a custom PLI application. You can also control assertions individually by turning them on and off at desired times during the simulation.

For more information on VPI options supported by VCS, refer to `+vpi`, `+vpi+1`, and `+vpi+1+assertion` compile-time options in *VCS User Guide*.

PLI use model

```
% vcs +vpi -P <pli>.tab [compile_options]
% simv [simv_options]
```

UCLI use model

```
% vcs -debug_pp +vpi [compile_options]
```

or

```
% vcs -debug +vpi [compile_options]
% simv -ucli
```

DVE use model

You can dump assertions with the `-assert dve` option and also access the same in the post-process mode. This helps to identify the contributing signals to immediate and concurrent assertions.

```
% vcs -debug_pp +vpi -assert dve \
[compile_options]

% simv -gui //interactive mode
% dve //post-process mode
```

Example

The following example shows how assertions are defined in the design file and how you can view the assertions in the DVE Data pane.

example.v

```
module top;
reg a, b; reg clk=1;
always #1 clk = ~clk;

function reg fn(reg a );
    return fn_a(a);
endfunction
function reg fn_a(reg a );
    return (a);
endfunction

always @(a or b) begin: BLOCK1
A1: assert final($changed(fn(a),@(posedge
clk))&&$changed(fn(b),@(posedge clk)))
    $display($time, " %m: Pass");
    else      $display($time, " %m Fail");
end

always_comb
A2: assert final($changed(fn(a),@(posedge
clk))&&$changed(fn(b),@(posedge clk)))
    $display($time, " %m: Pass");
    else      $display($time, " %m Fail");

always_latch
A3: assert final($changed(fn(a),@(posedge
clk))&&$changed(fn(b),@(posedge clk)))
```

```

                $display($time, " %m: Pass");
        else      $display($time, " %m Fail");

always_ff @(posedge clk) begin: BLOCK2 begin: BLOCK3
    A4: assert final($changed(fn(a))&&$changed(fn(b)))
                    $display($time, " %m: Pass");
        else      $display($time, " %m Fail");
end end

initial
begin
    #1 a=1; b=1;
    #5 b=0; #0 a=0;
    #1 b=1; #0 a=1;
    #10 $finish;
end

always @(a or b) begin
    unique case({a,b})
        2'b11 : $display({a,b});
        2'b11 : $display({a,b});
        2'b10 : $display({a,b});
    endcase
end

endmodule
module test;
    reg rst;
    initial begin
        rst = 0;
        #1 rst= 1;
        #4 rst = 0;
        #2 rst = 1;
    end

    always@ (rst) begin
        if(rst == 0) begin
            $uniq_prior_checkoff(0,t1);
            $display( $time,, " RT OFF ");
        end
        else begin
            $uniq_prior_checkon(0,t1);
            $display( $time,, " RT ON ");
        end
    end

```

Using the Assertion Pane

```

    end

top t1();
  A5: assert property(@(posedge t1.clk)
($changed(t1.fn(t1.a))&&$changed(t1.fn(t1.b)))) ;
  A6: cover property(@(posedge t1.clk)
($changed(t1.fn(t1.a))&&$changed(t1.fn(t1.b)))) ;
  top t2();
endmodule

```

Steps to compile the example

```

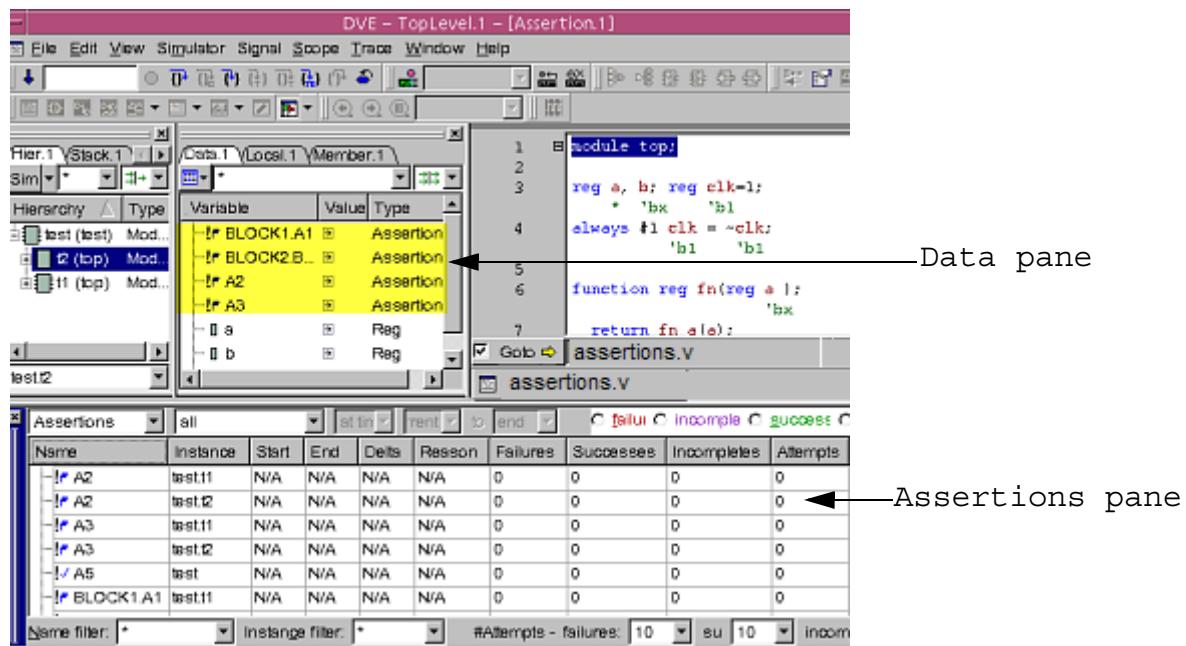
% vcs -sverilog -debug_pp +vpi example.v \
-assert dve -assert enable_hier -x1rm uniq_prior_final \
-assert svaext

% dve -script sva_vpi+lassertions.tcl

```

The script file `sva_vpi+lassertions.tcl` is available in the `$VCS_HOME` directory.

Figure 10-3 Assertions seen in DVE Data pane and Assertions pane



11

Using the Testbench Debugger

This chapter describes the DVE (Discovery Visual Environment) Testbench Debugger. It includes the following topics:

- “Overview”
- “Enabling Testbench for Debugging”
- “Invoking the Testbench Debugger GUI”
- “Testbench Debugger Panes”
- “Testbench Debug”
- “Debugging Threads”
- “Debugging UVM Testbench Designs”

Overview

The DVE integrated testbench graphical debugger provides a common interface for debugging HDL and Testbench code simultaneously and is seamlessly integrated with the current DVE HDL debug windows.

In interactive mode, the Testbench Debugger provides you visibility into the testbench-related dynamic constructs and their values during simulation. This is done by using the proven visualization of the Testbench GUI's stack pane, local pane, and watch pane combined with DVE's Source view and its intuitive look and feel.

Using the salient features of the new testbench debugger, you can analyze, understand, and debug the behavior of your complicated verification environment faster. You will be able to perform a comprehensive analysis using the seamless design and verification environment.

The testbench debugging interface enables you to perform the following:

- Navigate HDL or Testbench source code in a single DVE Source view
- View HDL and Testbench scopes in DVE's Hierarchy pane
- Analyze HDL and TB signals together in the Watch pane
- Run HDL and Testbench-related UCLI commands all from a single application

Enabling Testbench for Debugging

To enable the debugging capabilities for the testbench, you must specify the `-debug_all` switch along with your compilation command.

Note:

If you separately compile your design and testbench (NTB-OV separate compile flow), ensure that you use the `-debug_all` switch when compiling both your design and the testbench.

Invoking the Testbench Debugger GUI

You can start the Testbench Debugger from the command line and then run your simulation from the GUI.

- From the command prompt, enter the following:

```
%> simv -gui
```

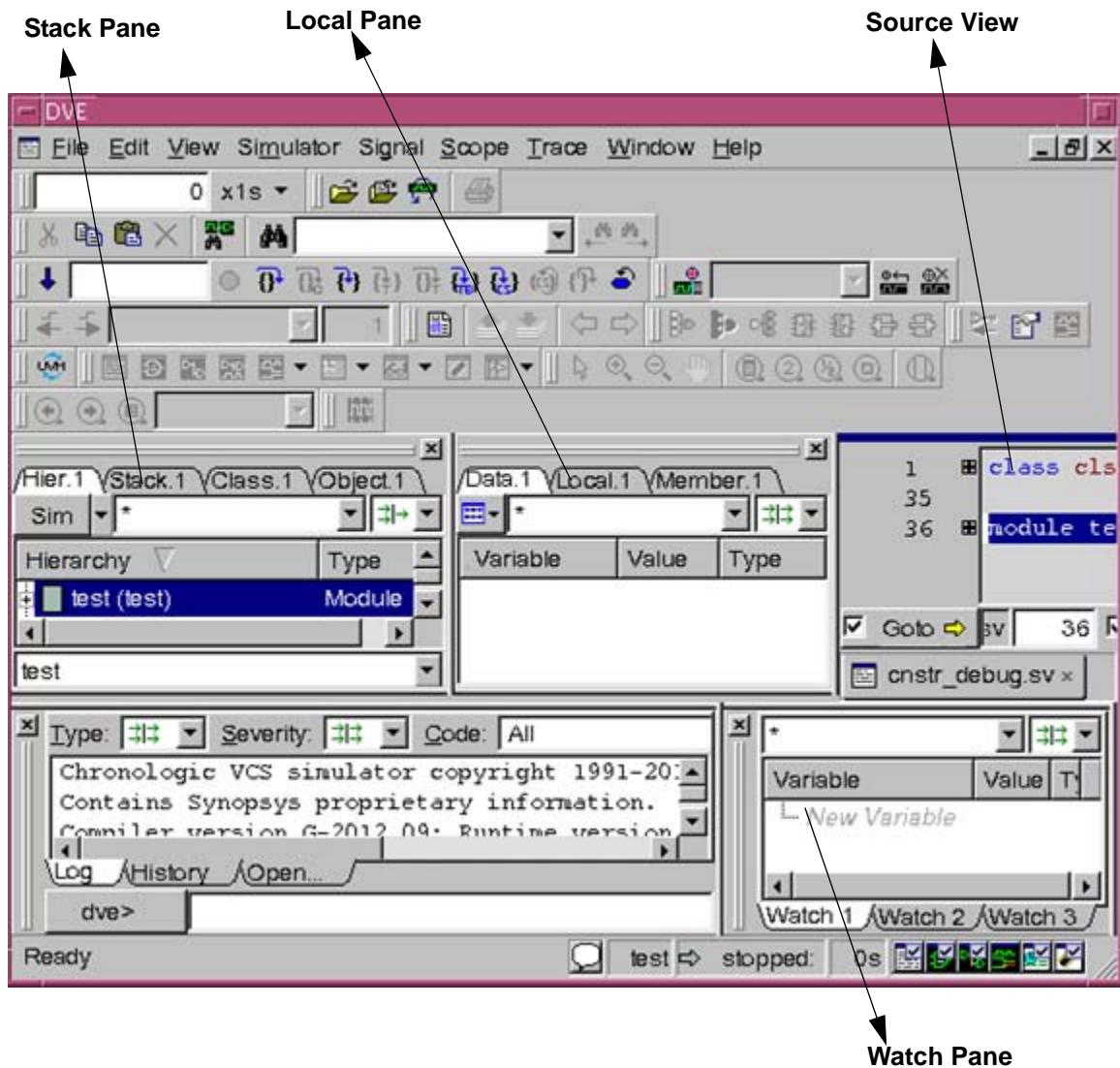
In this example, `simv` is the executable.

Note:

If you use the `-debug_all` switch when compiling your design that contains testbench code, the DVE automatically provides debugging options for your testbench program. However, you can disable these settings in the Preferences dialog box by clearing the option "Enable testbench debugging for interactive design" in the Edit menu. Select a transition in the Wave view to display the driver.

Testbench Debugger Panes

The top-level window of the Testbench Debugger GUI contains three additional panes. The three additional panes are – Stack pane, Local pane, and Watch pane as illustrated in the following diagram:



Note:

The Watch pane appears only when you add variables or signals to it for monitoring purposes.

- Stack Pane - Displays the testbench dynamic hierarchy tree along with all the testbench threads and their status. This pane is highlighted when the Local tab is selected.
- Local Pane - Displays all the testbench variables and dynamic objects with their current values based on the currently selected scope within the call stack. The testbench variables and dynamic objects will change when you select different testbench scopes in the Stack.
- Watch Pane - Enables you to monitor the status of your variables during simulation.

Stack Pane

This pane shares a tabbed view with the hierarchy pane. Select the Stack tab to display the Stack pane and view the status of various threads. This view is cross-linked with the Source and Local panes. Double-clicking on objects in this pane synchronizes the display in the source and local panes. The hierarchy tab displays only the static objects in your design, whereas the Stack tab displays the dynamic threads created during runtime.

Note:

The Stack pane appears empty when you invoke the Testbench Debugger at time 0. The dynamic objects are displayed as and when they are created in the testbench during simulation.

The following figure illustrates the Stack tab and the Hierarchy tab:

Hier.1	Stack.1		
?	Scope	File : Line	Thread
✓	└ mgmt_if	mem.v : 58	8
→	└ \$root.top.t1	test_00_debug.sv : 19	0
→	└ unnamed\$\$_1	test_00_debug.sv : 31	4
→	└ env::test	env.sv : 90	4
→	└ apb_master::reset	apb_master.sv : 85	4
✗	└ env::pre_test	env.sv : 78	4
✗	└ unnamed\$\$_4	env.sv : 81	4
✓	└ unnamed\$\$_3	env.sv : 84	7
✗	└ unnamed\$\$_2	env.sv : 83	6
✗	└ unnamed\$\$_1	env.sv : 82	5

The above illustration shows the current active threads and their status. The status of a thread could be – Ready, Running, Stopped, or Suspended. The following table illustrates the conventions denoted by these icons:

-  Ready to execute.
-  Thread is executing.
-  Thread is stopped.
-  Thread is suspended.

The thread column displays the unique id of the thread. It can be the same if function calls in the stack belong to the same thread.

Using the Stack Pane Context Sensitive Menu

The Stack pane context sensitive menu (CSM) provides various options. You can quickly access and start using these options through the context menu. To invoke the CSM, right-click from the Stack pane. The following menu options appear:

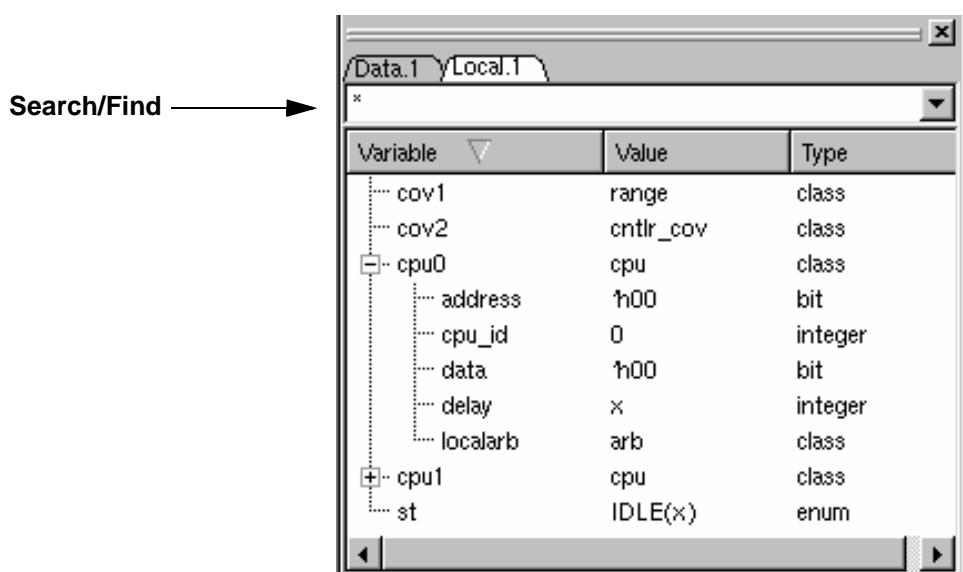


The following table explains the menu options:

Option	Description
Copy	Takes copy of the object.
Show Source	Displays source code of your testbench program.
Add To Watches	Adds signals to monitor in the Watch pane.
Expand All	Expands the tree.
Collapse All	Collapses the tree.
Select All	Takes copy of all the objects.

Local Pane

The local pane shares a tabbed view with the ‘Data’ tab. The local pane displays variables in a selected scope in the stack pane. This view is tied to the stack pane and the default view shows variables of the current active thread. This pane also has a Filter feature that you can use to search or find variables.



A screenshot of the Local pane interface. The title bar says 'Data.1 \ Local.1'. Below it is a search bar with the placeholder 'x'. The main area is a table with three columns: 'Variable', 'Value', and 'Type'. The data is organized into a hierarchical tree:

Variable	Value	Type
cov1	range	class
cov2	cntlr_cov	class
cpu0	cpu	class
address	1h00	bit
cpu_id	0	integer
data	1h00	bit
delay	x	integer
localarb	arb	class
cpu1	cpu	class
st	IDLE(x)	enum

Note:

The Local pane displays the variables when you select an object in the Stack pane.

Watch Pane

Occasionally, you may need to monitor the status of testbench and HDL variables throughout the simulation regardless of the active thread. You can select all the variables and objects to watch their behavior in this pane. The Watch pane displays the selected item, its value and the type for tracking, regardless of the active context.

By default, the Watch pane contains three tabs labeled Watch 1, Watch 2, and Watch 3. You can add as many tabs as you want. Use the Watch panes to monitor values of variables regardless of the current context. You can add variables from the Source or Local window or by performing a drag-and-drop.

To add a Watch tab, go to the menu **View > Watch > Add New Page**. You can also delete the watch tabs.

The following figure illustrates a typical Watch pane:

Variable	Value	Type	Scope
APB_DATA_...	'h00000020	Parameter	<input checked="" type="checkbox"/> \$root
APB_DATA...	'h0	Scalar	<input checked="" type="checkbox"/> \$root
APB DATA...	'h0	Scalar	<input checked="" type="checkbox"/> \$root

Watch 1 Watch 2 Watch 3



This figure illustrates the variable, its value, type, and the scope. Using the check box in the Scope column, you can tie the variable to a given thread throughout simulation or tie the variable to the currently selected thread in the call stack. This feature is available for all object types, including the design signals.

For example, add a variable called ‘x’ in the Watch pane and select the check box to tie it to a given thread. This variable is displayed throughout the simulation from the same dynamic instance of the scope (active thread), irrespective of the thread being alive or not. By default, this check box is selected.

Clearing the check box evaluates the variable in the currently active thread in the call stack. For example, add a variable ‘x’ from the active thread, ‘main’, during the beginning of simulation. Assume the active thread changes to some other thread at a later point of time.

The variable ‘x’ in the Watch pane now refers to the same variable in the dynamic instance of the scope (active thread), but not from the active thread, ‘main’.

Class Browser

You can now view the definitions and methods of classes in DVE. You use the Class Browser to browse, navigate, or visualize the classes defined in the design.

The Class Browser consists of the following panes:

- Class pane — displays all the classes defined in the design in a hierarchical view.
- Member pane — displays the content or methods of the selected class.

Usage Model

Example

In this example, there's hierarchy of base class and derived class.

class_browser.sv

```
// calling base class pre and post randomize methods inside
// the derived
// class method with the help of "super" keyword
`include "vmm.sv"
program p;

class base;
    static reg aa;
    protected static reg bb;
    protected logic mem [1:0];
```

```

rand logic [1:0] a,c;
randc logic [2:0] b;
mailbox mbox;

function void reset();
    a = 1; b = 1;
endfunction

function void pre_randomize();
    $display("Hello World");
endfunction

function void post_randomize();
if (!(a+c <= b))
begin
    $display("a == ", a, " c == ", c, " b == ", b);
    $display("Post Randomization Failed");
end
endfunction

endclass

class derived extends base;

rand byte d;

function void myfunc_pre_call();
    super.pre_randomize();
    d = 5;
endfunction

function void myfunc_post_call();
    super.post_randomize();
    d = b;
endfunction

endclass

derived c1 = new;

```

```

int ret;

initial begin

    repeat (3)
        begin
            c1.reset();
            ret = c1.randomize();
            if(ret == 0)

                $display("Randomization Failed");
            else
                begin
                    c1.myfunc_pre_call();
                    c1.myfunc_post_call();
                end
            end
        end
    endprogram

```

To compile this example code, use the following commands:

```
vcs class_browser.sv -ntb_opts rvm -debug_all -sverilog
simv -gui &
```

To open the Class Browser

The procedure to view the Class/Member pane is similar to viewing the Stack/Local pane in DVE. You can view the Class/Member panes in interactive mode when your design contains testbench. Also, in post-process mode when the variables are dumped in the VPD file by the \$vcplustblog task. For more information about \$vcplustblog task, see the LCA category in the VCS Online Documentation.

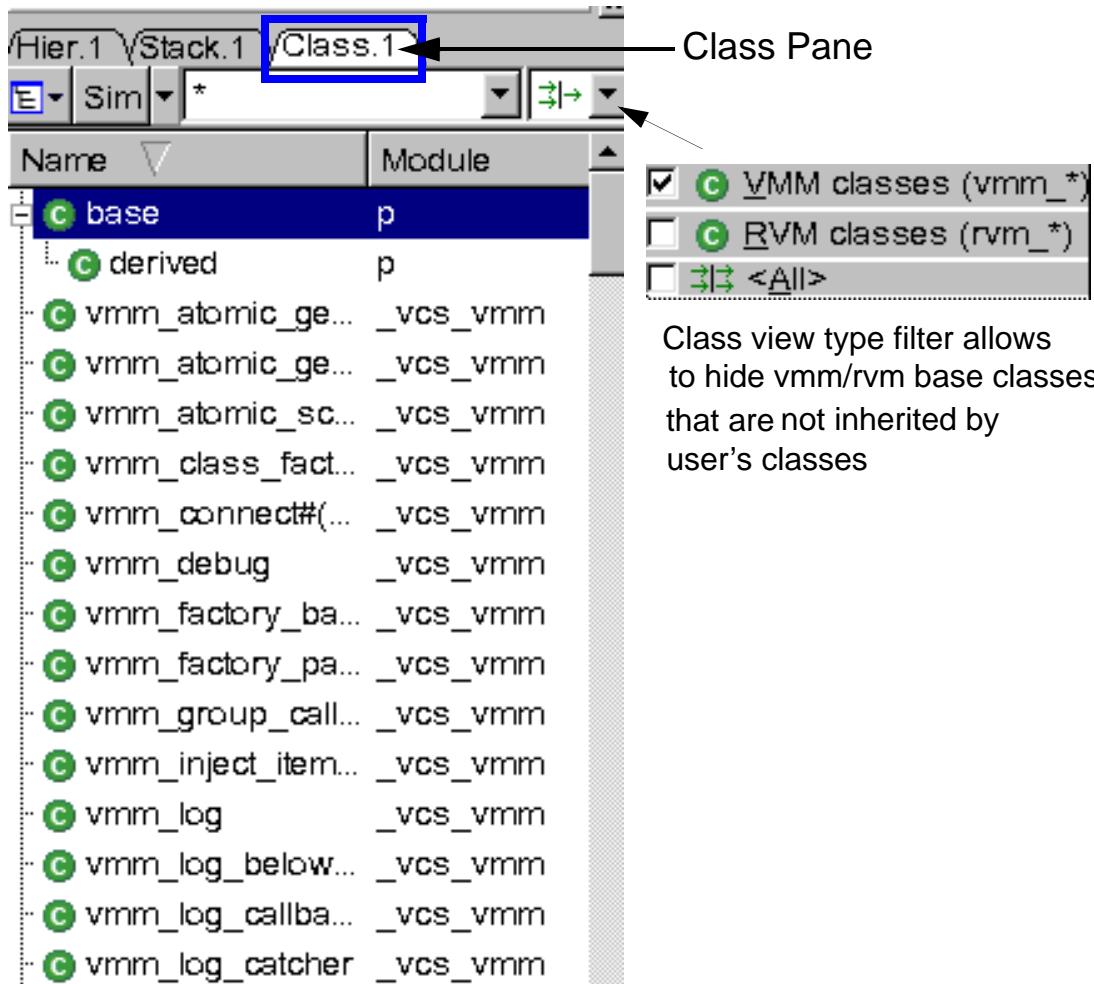
You can also open the Class/Member pane from the menu **Window > Panes**.

1. Run the design where you have defined classes and invoke DVE.

The scopes with classes appear in the Hierarchy pane.

2. Select a class, right-click and select **Show in Class Browser**.

The classes are displayed in the Class pane.



3. Select a class to see the methods and member variables in the Member pane.

Member Pane

The screenshot shows the Member pane interface. At the top, there are tabs for 'Data.1', 'Local.1', and 'Member.1', with 'Member.1' highlighted by a blue box. To the right of the tabs is a 'Type' drop-down menu button, which is also highlighted by a blue box. A callout arrow points from the text 'Type drop-down menu' to this button. Below the tabs is a table with columns: Name, Type, Attribute, and a checkbox column for filtering. The table is divided into sections: 'Methods' and 'Variables'. The 'Methods' section contains four entries: 'new' (Function, Public), 'post_randomize' (Function, Public), 'pre_randomize' (Function, Public), and 'reset' (Function, Public). The 'Variables' section contains seven entries: 'a' (Logic, Public, Rand), 'aa' (Reg, Static, Public), 'b' (Logic, Public, RandC), 'bb' (Reg, Static, Protected), 'c' (Logic, Public, Rand), 'c3' (Class base, Public), 'mbox' (Mailbox, Public), and 'mem' (MDA Logic, Protected). To the right of the table is a vertical scrollable list of filter options. The filters listed are: Base members, Internal members, Static members (which is checked), Automatic members, Local members (which is checked), Protected members, Public members, Virtual members, Rand members (which is checked), and <All>. The '<All>' option is at the bottom of the list. A callout arrow points from the text 'Member view type filter allows filter of base class variables, task, functions, class objects, non class objects' to the '<All>' option.

Name	Type	Attribute	
Methods			
new	Function	Public	<input type="checkbox"/>
post_randomize	Function	Public	<input type="checkbox"/>
pre_randomize	Function	Public	<input type="checkbox"/>
reset	Function	Public	<input type="checkbox"/>
Variables			
a	Logic	Public, Rand	<input type="checkbox"/>
aa	Reg	Static, Public	<input type="checkbox"/>
b	Logic	Public, RandC	<input type="checkbox"/>
bb	Reg	Static, Protected	<input type="checkbox"/>
c	Logic	Public, Rand	<input type="checkbox"/>
c3	Class base	Public	<input type="checkbox"/>
mbox	Mailbox	Public	<input type="checkbox"/>
mem	MDA Logic	Protected	<input type="checkbox"/>

Type drop-down menu

Member view type filter
allows filter of base
class variables, task,
functions, class
objects, non class objects

You can click the **Type** drop-down menu to select the filters based on which you can sort the member variables.

[Table 11-1](#) describes the filters of Member Pane type filter.

Table 11-1 Filters of Member Pane type filter

Filter	Description
 Base members	Select this filter type to view the members of base class. By default, DVE hides these members.
 Internal members	Select this filter type to view the internal members. By default, DVE hides these members.
 Automatic members	Filters the non-static members.
 Local members	Filters the members with “Local” visibility.
 Protected members	Filters the members with “Protected” visibility.
 Public members	Filters the members with “Public” visibility.
 Virtual members	Filters virtual members.
 Rand members	Filters random members.

4. Select a class and double-click to view the class definition in the Source view.

Or

Select a class, right-click and select **Show Source**.

The class definition is displayed in the Source view as follows:

```
4  `include "vmm.sv"
5  program p;
6
7  class base;
8      static reg aa;
9      protected static reg bb;
10     protected logic mem [1:0];
11
12     rand logic [1:0] a,c;
13     randc logic [2:0] b;
14     mailbox mbox;
15     base c3 = new;
16
17     function void reset();
18         a = 1; b = 1;
19         endfunction
20
21     function void pre_randomiz
22         e();
23             $display("Hello Wo
rld");
24         endfunction
```

Note:

Constraints and Structure/Union properties are not visible in the Member pane.

Dynamic Object Browser

DVE allows you to view and browse all existing class objects and member values using the Object Browser feature. This feature consists of three major pieces of functionality:

- Object Hierarchy Browser which displays current dynamic objects and its values.
- Using Class Browser to view object instances.
- Using Local Pane filter to search for dynamic objects.

This feature helps you to:

- Locate an object without having to set a breakpoint.
- View the object along with its values/attributes.
- View aggregate paths that point to an object.
- View source lines related to the object.

Object Browser Example

Consider the following UVM test case:

Example 11-1 Design File (test.sv)

```
program top;  
  
`include "uvm_macros.svh"  
import uvm_pkg::*;  
  
class test extends uvm_test;  
  
`uvm_component_utils(test)
```

```

function new(string name, uvm_component parent = null);
    super.new(name, parent);
endfunction

virtual function void report();
    $write("** UVM TEST PASSED **\n");
endfunction
endclass

initial
begin
    run_test();
end

endprogram

```

Compile the `test.sv` code, shown in [Example 11-1](#), as follows:

```
% vcs -sverilog -ntb_opts uvm -debug_all test.sv
```

Invoke the DVE GUI, as follows:

```
% simv +UVM_TESTNAME=test -gui
```

Object Hierarchy Browser

The **Object Hierarchy Browser** displays hierarchical structures which include top-level modules, programs, and packages within the current browsing scope, as shown in [Figure 11-1](#), in the Objects Pane. You can expand structures in the Objects Pane to view the following:

- Reference variables in modules, programs, or packages.

Note:

A reference variable is a variable that can point to an instance of a class (that is, an object instance). For example, consider the following code:

```
class MyClass;
    SomeOtherClass object;
    function new();
        object = new;
    endfunction
endclass

module top;
    MyClass p;
    initial begin
        p = new;
    end
endmodule
```

In the above code, `p` is a reference variable which is defined in the static module `top`, so `top.p` is always a valid path. Also, `object` is a reference variable which is not defined in a static scope. It is a class member variable.

- Static reference variables in tasks, functions, or classes.
- Active or suspended static task or functions which contain reference variables.
 - Task or function can be expanded to view the variables.
 - A task or function name contains thread ID. For example:

```
foo (thread 3)
```

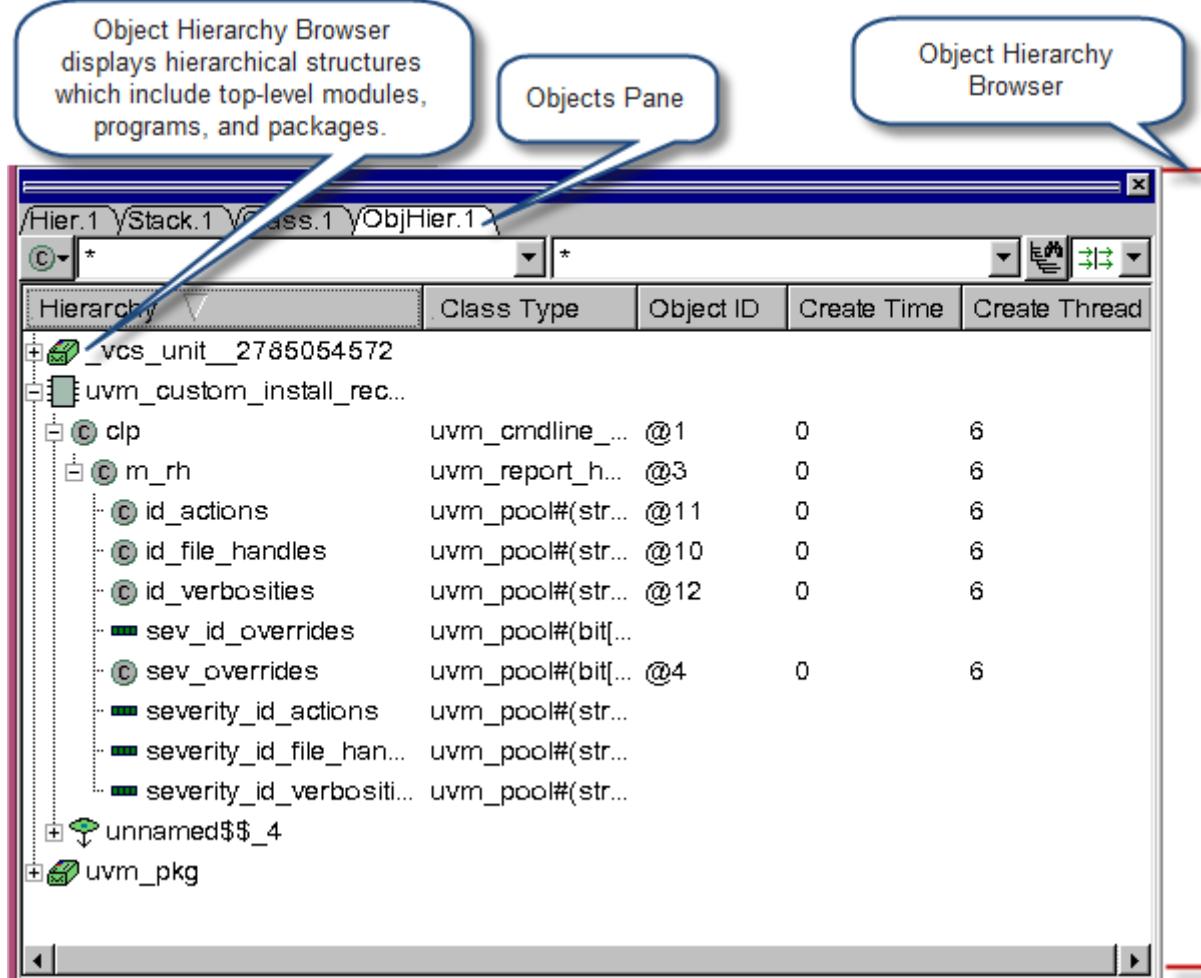
The **Object Hierarchy Browser** displays scopes, reference variables defined within a static scope, and object instances. It will display only reference variable members, if an object instance is expanded.

The Object Hierarchy Browser contains the following columns:

Table 11-2 Object Hierarchy Browser Columns

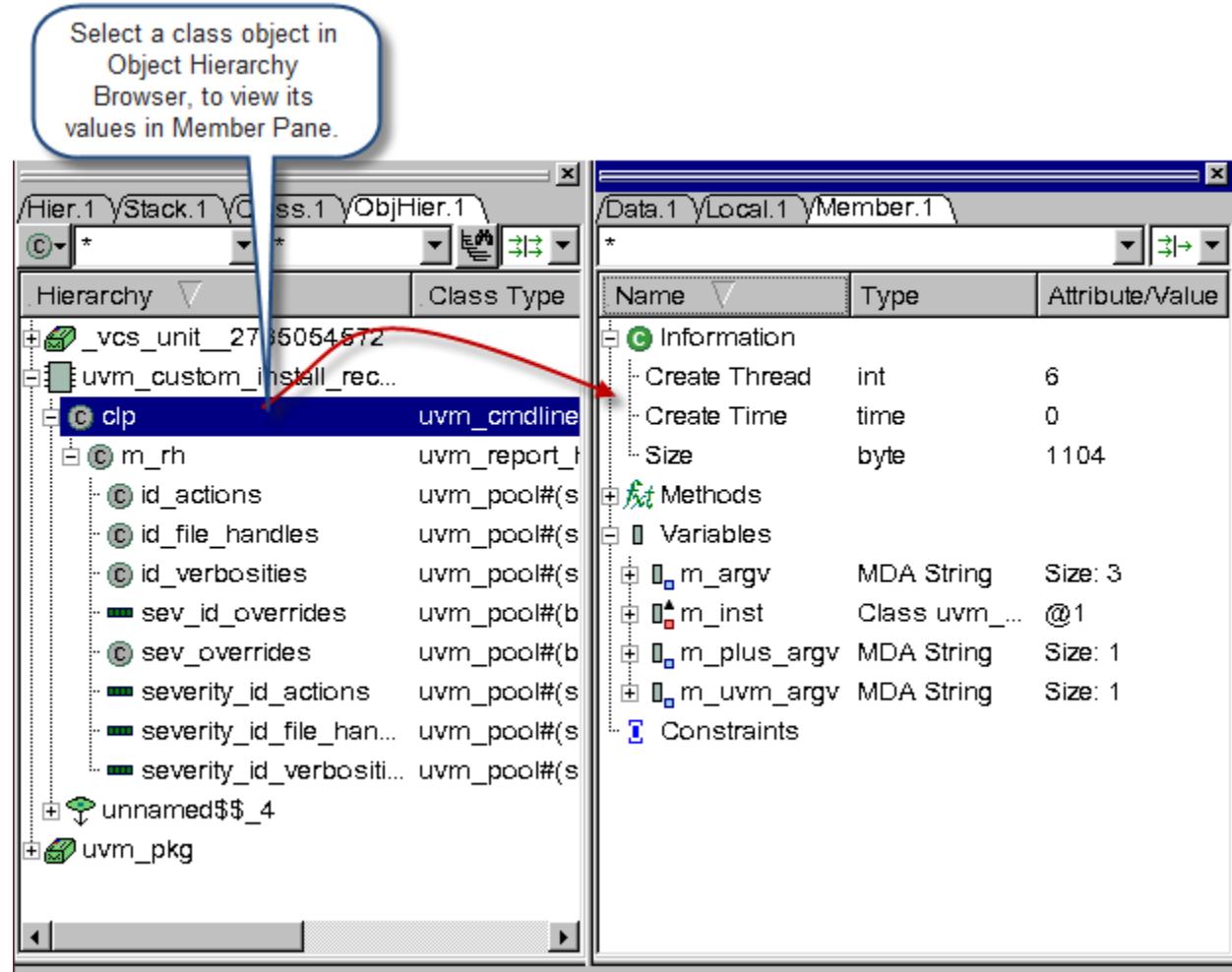
Column Name	Description
Hierarchy	Displays hierarchical structures which include top-level modules, programs, and packages
Class Type	Displays the name of object's class
Object ID	Displays object ID of an object
Create Time	Displays the time the object is created
Create Thread	Displays thread ID

Figure 11-1 Object Hierarchy Browser



The Member Pane displays members and values of a class object selected in the Object Hierarchy Browser, as shown in [Figure 11-2](#).

Figure 11-2 Viewing Values of a Class Object in the Member Pane



Right-click Menu Options in the Object Hierarchy Browser

Table 11-3 describes the right-click menu options available in **Object Hierarchy Browser**.

Table 11-3 Right-click Menu Options in the Object Hierarchy Browser

Option	Description
Show Source	Displays the definition of the selected object in the Source View.
Show In Class Browser	Displays class of the selected object in Class Pane.
Show Create Location	Displays the location where the selected object is created (that is, location of statement with 'new' call) in the Source View.
View References	Displays the reference paths of the selected object instance in the References dialog box.
Add Object To Watches	Adds the selected object to the Watch Pane.
Show Total Memory	Displays Total Memory column in the Object Hierarchy Browser. This column displays the memory size of objects in KB.

Viewing Memory Size of Objects in Object Hierarchy Browser

DVE allows you to view information about the memory consumed by the objects of a design in the **Object Hierarchy Browser**.

To view information about the memory consumed by objects of a design:

Right-click an object in the Object Hierarchy Browser and select **Show Total Memory**, as shown in [Figure 11-3](#).

Or

Click **View > Show Total Memory**

DVE displays **Total Memory** column in the Object Hierarchy Browser. This column displays the memory size of objects in KB, as shown in [Figure 11-4](#). If an object is present multiple times in the object tree, then only its first occurrence is counted. You can hide this column by clicking the **Show Total Memory** option again.

Figure 11-3 Viewing Memory Size of Objects in Object Hierarchy Browser

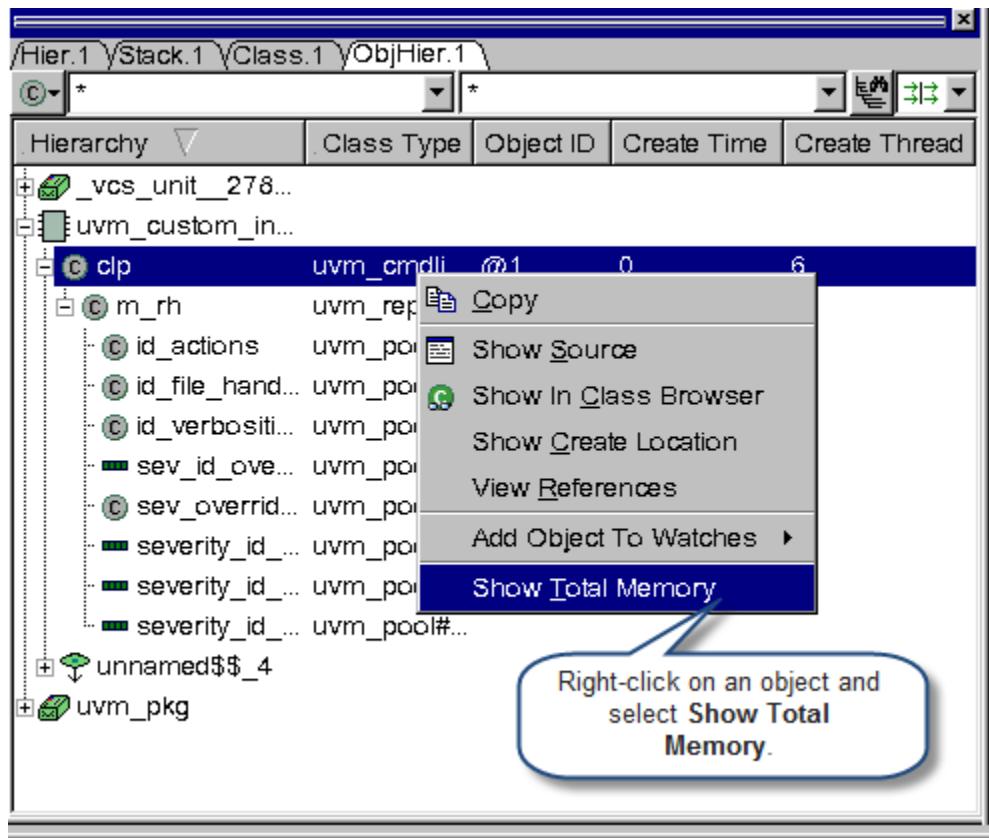


Figure 11-4 Viewing Memory Consumed by Objects in a Design

Hierarchy	Total Memory	Class Type	Object ID	Create Tid
+ vcs_unit_278...	0 KB			
- uvm_custom_in...	1 KB			
clp	3 KB	uvm_cmndl... @1	0	
m_rh	2 KB	uvm_compor... @3	0	
id_actions	1 KB			
id_file_hand...	1 KB			
id_verbosity...	1 KB			
sev_id_over...	0 KB			
sev_overrid...	1 KB	uvm_pool#... @4	0	
severity_id_...	0 KB			
severity_id_...	0 KB			
severity_id_...	0 KB			
unnamed\$\$_4	0 KB			
uvm_pkg	202 KB			

The object memory size will be displayed in the following format:

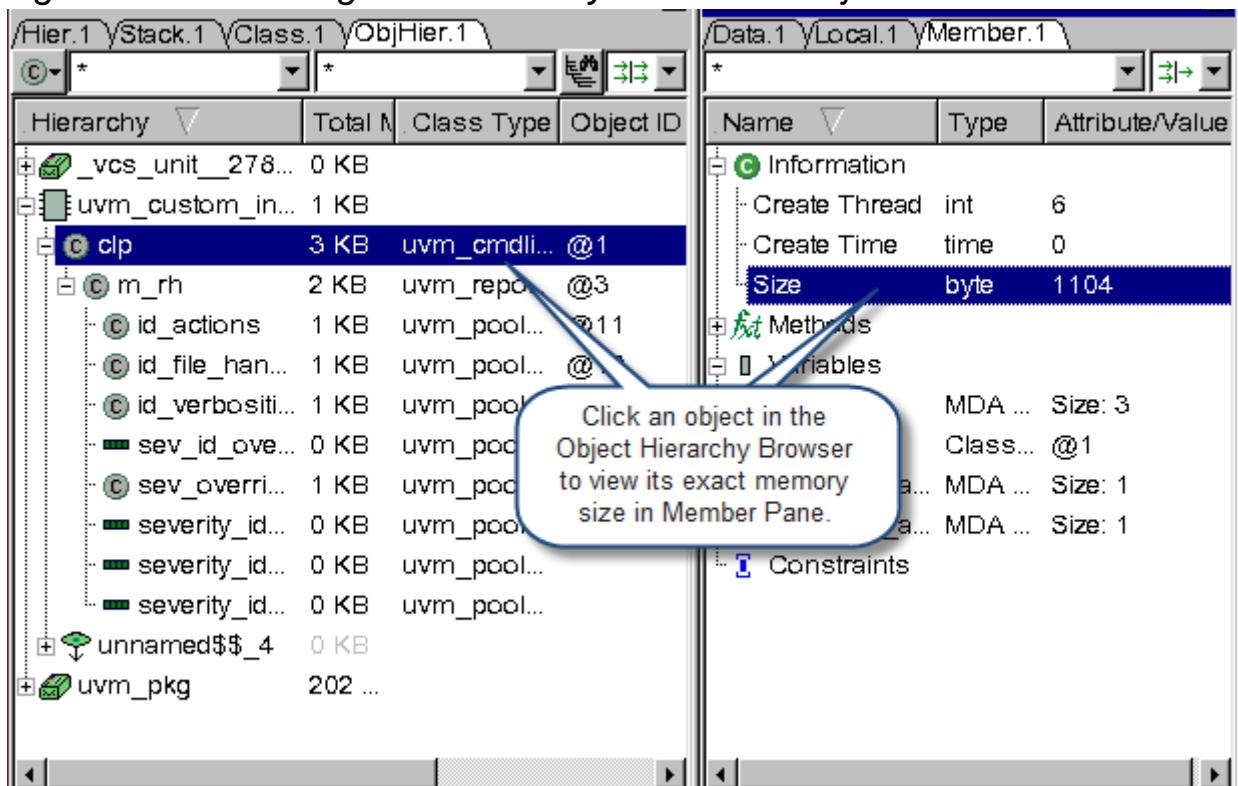
- DVE displays the memory size of all objects in KB.
- DVE will not display fractional parts of memory size, it rounds the memory size to the nearest whole number or integer value. For example, if memory size is less than 1 KB, then DVE will display “1 KB”. You can select the desired object in Object Hierarchy Browser to view its exact memory size in Member Pane, as shown in [Figure 11-5](#).

Note:

Object hierarchy browser displays the size of an object, including its children. But, Member Pane displays only the size of the selected object, not including its children.

- DVE uses “,” after three digits. For example, DVE displays 1MB as 1,024 KB, 1GB as 1,048,576 KB.

Figure 11-5 Viewing Exact Memory Size of an Object in the Member Pane



The numbers in **Total Memory** column are updated automatically during the simulation run. You can sort the **Total Memory** column. The memory size of each object is counted only once by its parent object or scope. The memory size of an object will be displayed in gray, as shown in [Figure 11-6](#), if it is not counted by the current parent object.

Figure 11-6 Memory Size Displayed in Gray Color

Hierarchy	Total Memory	Class Type
+ _vcs_unit_2785054572	0 KB	
- uvm_custom_install_recording	1 KB	
+ clp	3 KB	uvm_cmdline_
+ unnamed\$\$_4	0 KB	
+ uvm_pkg	202 KB	

Memory size displayed
in gray color.

Using Object Hierarchy Browser Filters

Object Hierarchy Browser allows you to view the desired objects using the **Object Browser Mode** drop-down button, and allows you to filter objects by hierarchy or class type using the **Filter by Hierarchy** and **Filter by Class Type** text filters.

Figure 11-7 Object Hierarchy Browser Filters

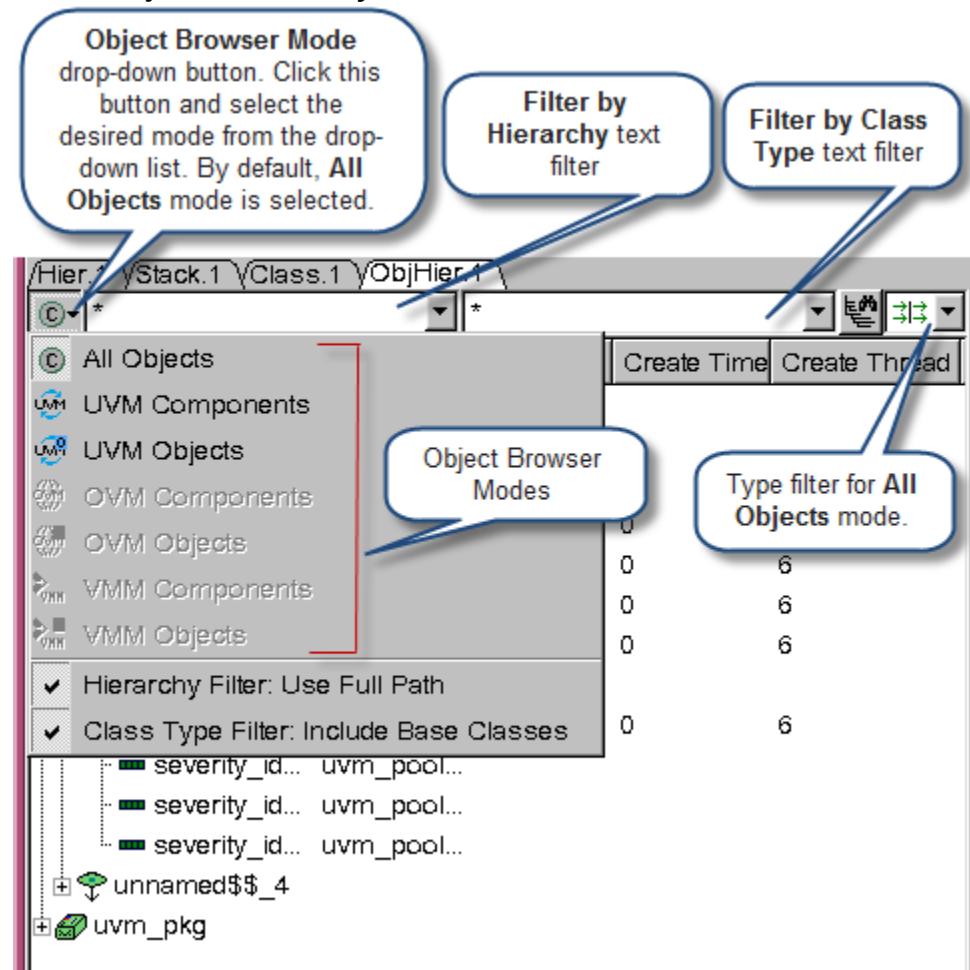


Table 11-1 describes the Object Browser modes.

Table 11-4 Object Browser Modes

Mode	Description
 All Objects	Default mode. Displays all class objects (including UVM, OVM, and VMM) in the Object Hierarchy Browser.
 UVM Components	Displays UVM components in the Object Hierarchy Browser.
 UVM Objects	Displays UVM objects in the Object Hierarchy Browser.
 VMM Components	Displays VMM components in the Object Hierarchy Browser.
 VMM Objects	Displays VMM objects in the Object Hierarchy Browser.
 OVM Components	Displays OVM components in the Object Hierarchy Browser.
 OVM Objects	Displays OVM objects in the Object Hierarchy Browser.

Filtering Objects in the Object Hierarchy Browser

The **Filter by Hierarchy** and **Filter by Class Type** text filters allow you to configure the type of information to display for Hierarchy column in the Object Hierarchy Browser. [Table 11-5](#) describes the text filters in Object Hierarchy Browser.

These filters allow you to specify the text to filter, and stores the previously specified filter strings. By default, these filters use '*' wildcard character as the filter string.

Table 11-5 Text Filters in Object Hierarchy Browser

Filter Name	Description
Filter by Hierarchy	Allows you to filter objects by hierarchy name in the Object Hierarchy Browser. DVE will use full hierarchical path name of an item to check if it matches to the specified string.
Filter by Class Type	Allows you to filter objects by class name in the Object Hierarchy Browser. This filter matches the current type of the object and base classes of the object (“Include Base Classes”). For example, filtering for <code>*uvm_object*</code> will also show objects with class <code>uvm_component</code> , since class <code>uvm_component</code> is derived from <code>uvm_object</code> .

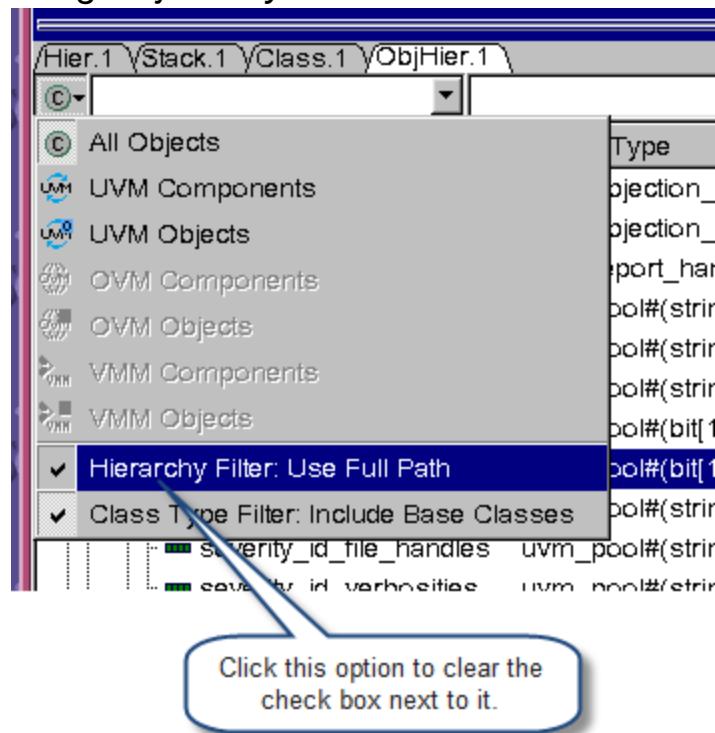
Filtering Objects by Name in the Object Hierarchy Browser Using ‘Filter by Hierarchy’ Text Filter

You can also use the **Filter by Hierarchy** text filter to filter objects by simple name (for example, class variable name or object member name) in the Object Hierarchy Browser.

To filter objects by name in the Object Hierarchy Browser:

1. Click the **Object Browser** Mode drop-down button and clear the check box next to **Hierarchy Filter: Use Full Path** option, as shown in [Figure 11-8](#), to disable it. By default, this option is enabled.
2. Type the string to search in the **Filter by Hierarchy** text filter.

Figure 11-8 Filtering Objects by Name



Filtering Objects by Class Name in the Object Hierarchy Browser Using the 'Filter by Class Type' Text Filter

You can also use the **Filter by Class Type** text filter to filter objects by simple class name in the Object Hierarchy Browser.

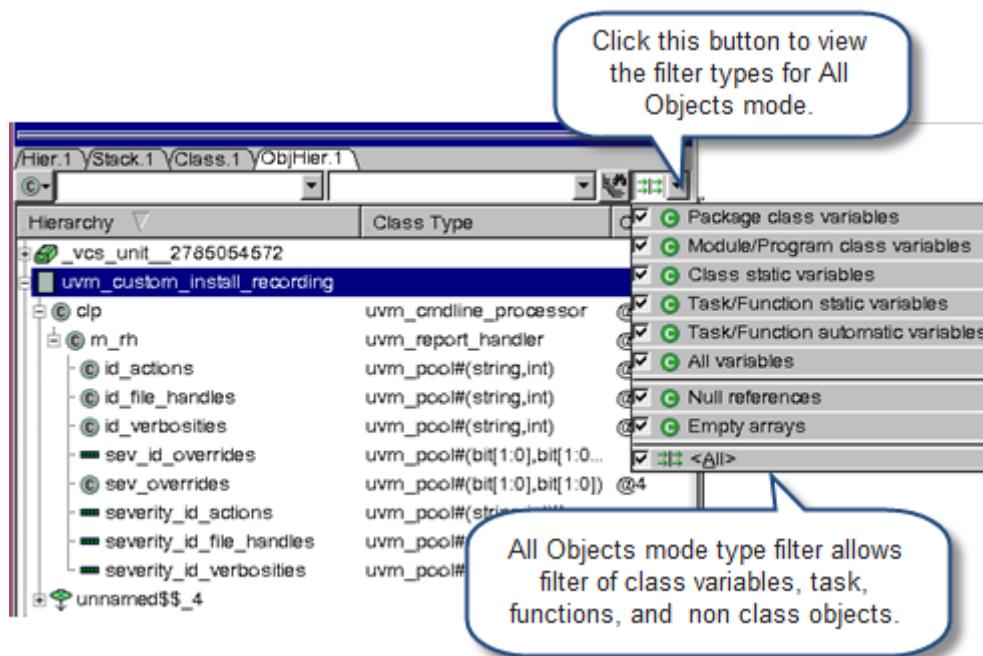
To filter objects by name in the Object Hierarchy Browser:

1. Click the **Object Browser** Mode drop-down button and clear the check box next to **Class Type Filter: Include Base Classes** option, as shown in [Figure 11-8](#), to disable it. By default, this option is enabled.
2. Type the string to search in the **Filter by Class Type** text filter.

Type Filter for “All objects” Mode

You can click the Type Filter drop-down button, as shown in [Figure 11-9](#), to select the filters based on which you can filter the objects in the Object Hierarchy Browser.

Figure 11-9 Filter Types for All Objects Mode



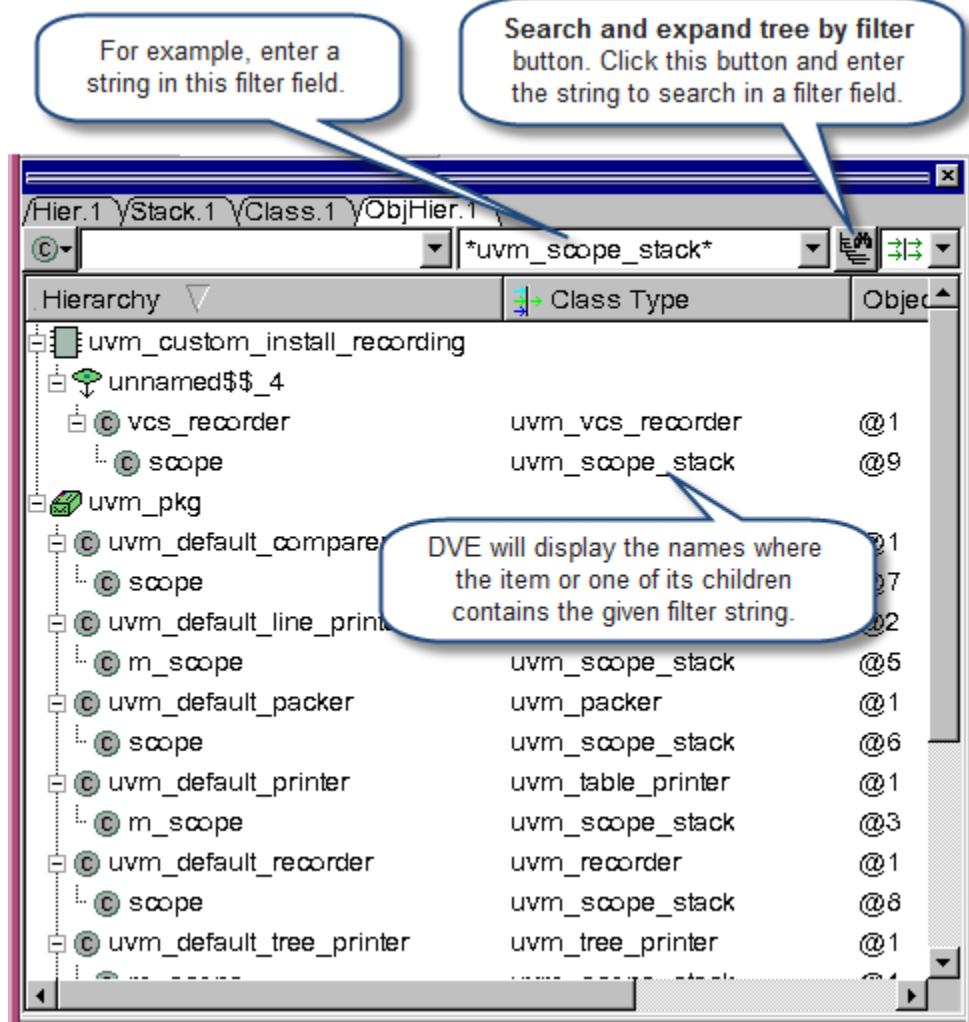
Note:

- DVE displays a scope (package, module, program, class, task, or function), only if it contains a variable of the selected type.
- DVE displays an object if the variable of the selected type is present in its object path. This is required to expand matching objects after a filter is applied.

Searching Objects in the Object Hierarchy Browser

You can use **Search and expand tree by filter** button , as shown in [Figure 11-10](#), to search for the filter string in the entire tree of objects (both expanded and collapsed folders), and view the names where the item or one of its children contains the given filter string.

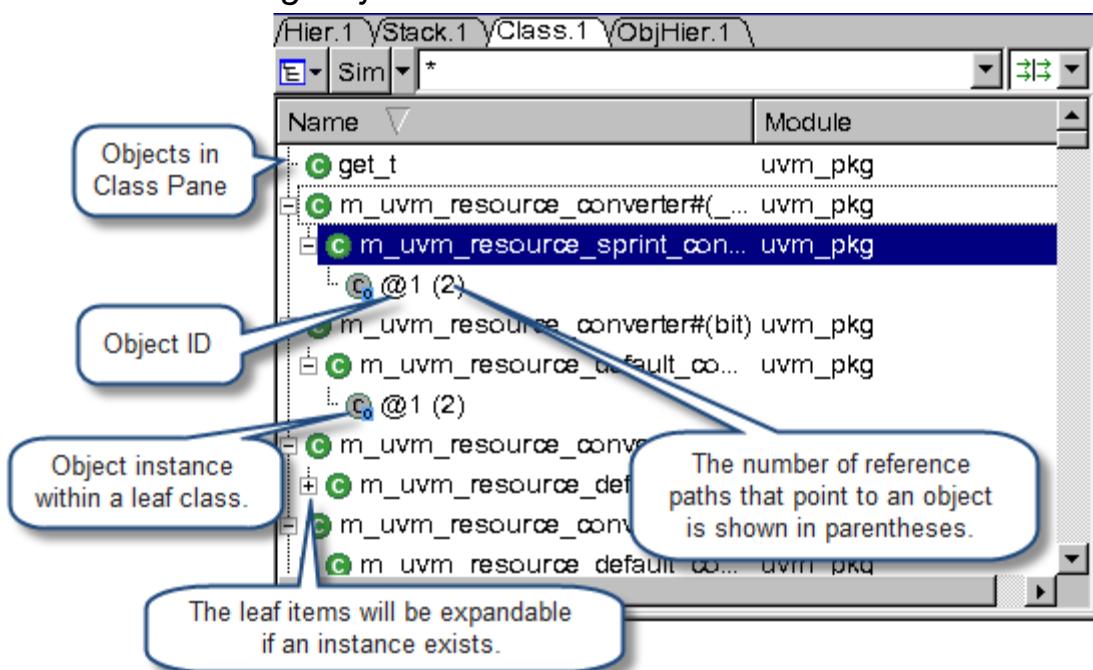
Figure 11-10 Searching Objects in the Object Hierarchy Browser



Viewing Objects in the Class Pane

DVE allows you to view objects in Class Pane. If a leaf class contains object instances, you can view them by expanding the leaf class in the Class Pane. When expanded, the **Name** column lists all object IDs that exist for that class, as shown in [Figure 11-11](#). The number of reference paths that point to an object is shown in parentheses.

Figure 11-11 Viewing Objects in the Class Pane

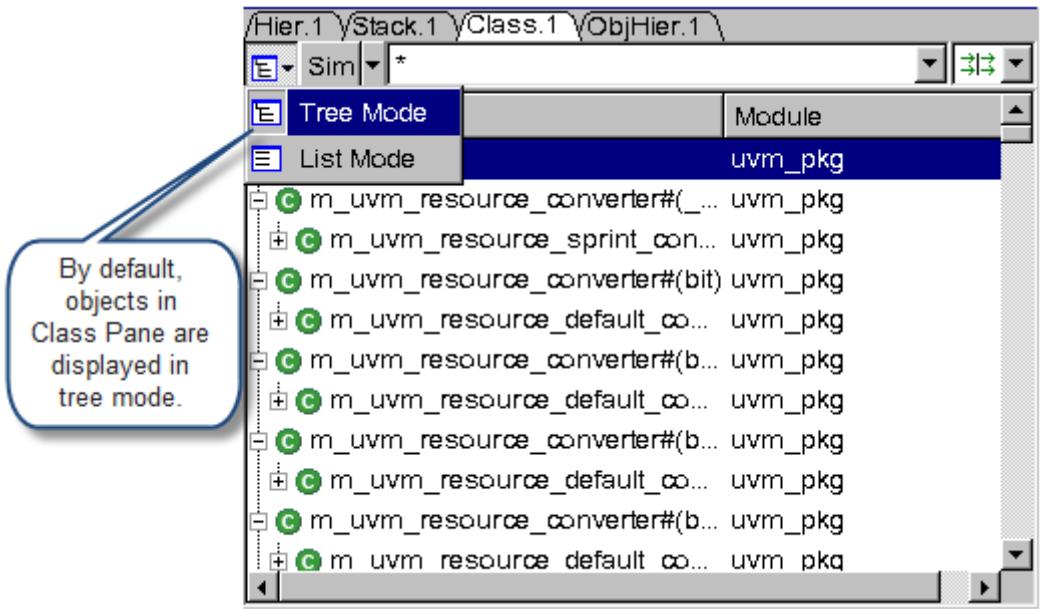


DVE allows you to view objects in tree mode (default mode) or list mode, as shown in [Figure 11-12](#) and [Figure 11-13](#).

To view objects in tree mode

In the Class Pane, click the **Views** button and select **Tree Mode** from the drop-down list, as shown in [Figure 11-12](#). This is the default mode.

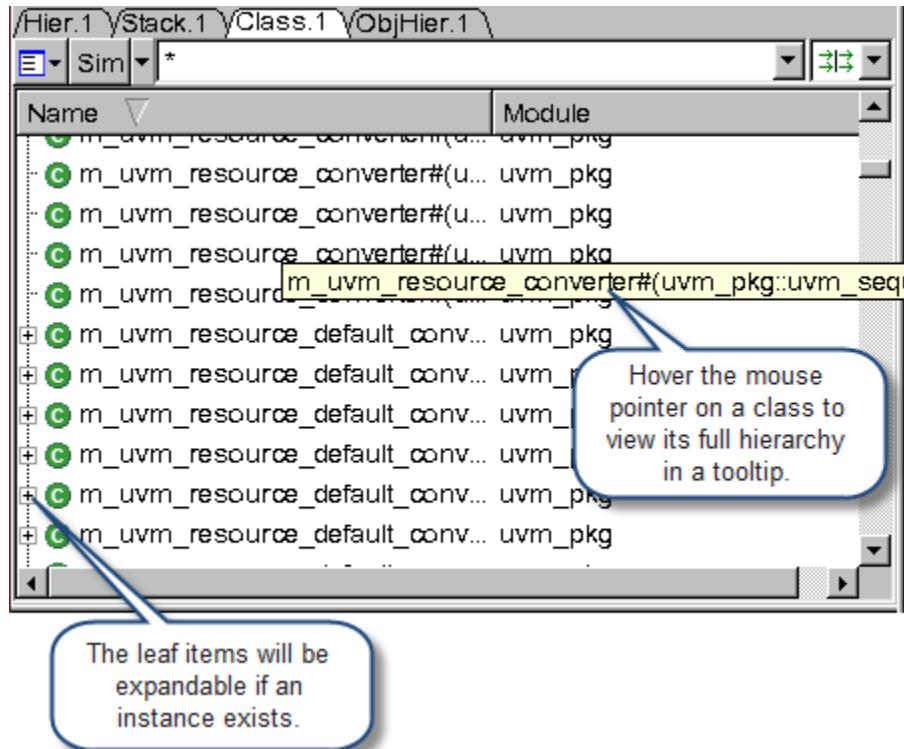
Figure 11-12 Viewing Objects in the Tree Mode



To view objects in list mode

In the Class Pane, click the **Views** button and select **List Mode** from the drop-down list, as shown in [Figure 11-13](#). This mode displays only the leaf classes.

Figure 11-13 Viewing Objects in the List Mode



Right-click Menu Options in the Class Pane

Table 11-6 describes the new right-click menu options in the Class Pane.

Table 11-6 Right-click Menu Options in the Class Pane

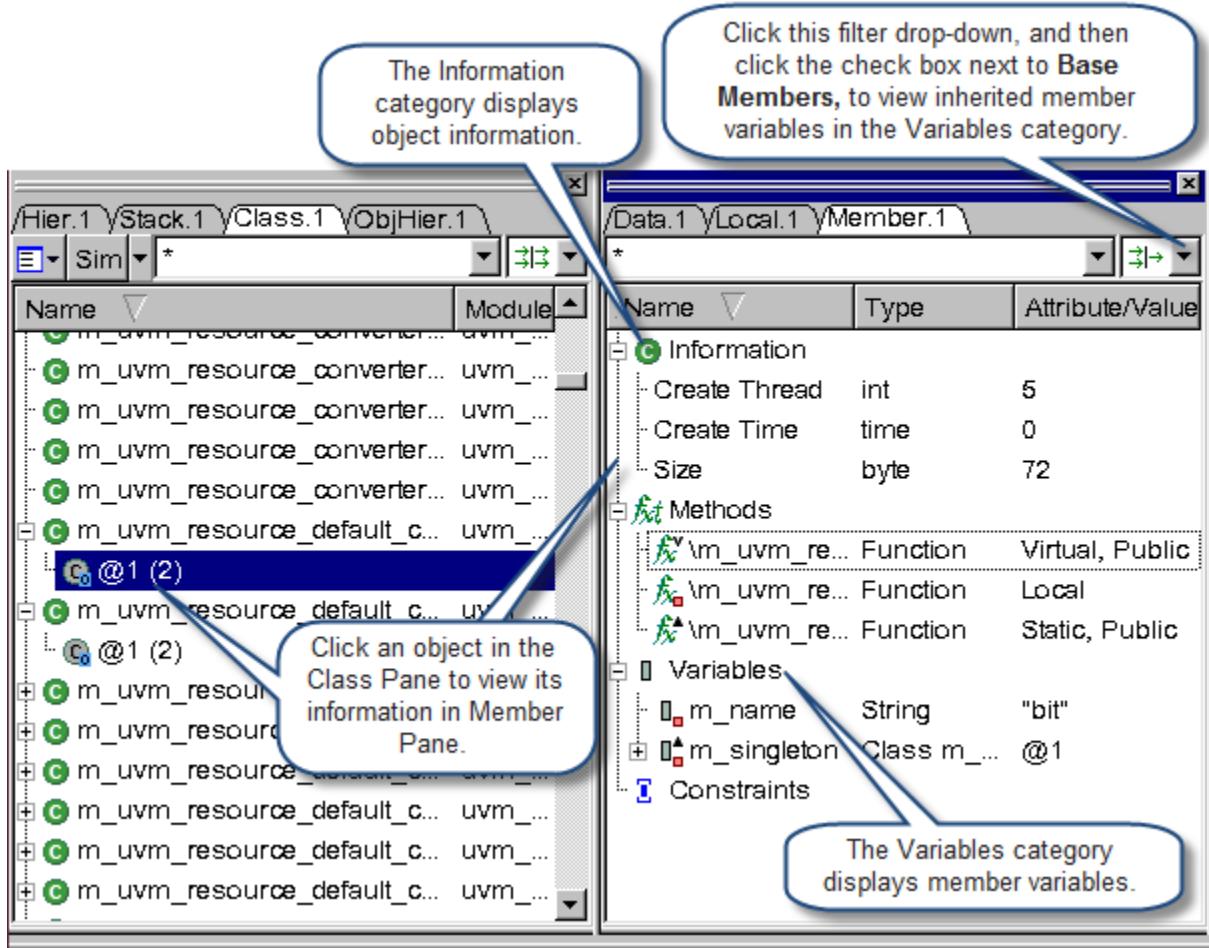
Option	Description
Show Source	Displays the source line of the selected class object in the Source View.
Show Create Location	Displays the location where the selected object instance is created (that is, location of statement with ‘new’ call) in the Source View.
UVM/VMM Reference	Displays UVM/VMM documentation in a web browser, for the selected item.
Add Object To Watches	Adds the selected object to the Watch Pane.
View References	Displays the reference paths of the selected object instance in the References dialog box.

Viewing Object Instance Information in the Member Pane

Select an object instance in the Class Pane to view its information in the Member Pane, as shown in [Figure 11-14](#). Member Pane allows you to do the following:

- Double-click a member variable or right-click on a member variable and select **Show Source** to view its source line in the Source View.
- Right-click on a member variable and select **Add Object To Watches** option to add it to the Watch Pane.
- Use the **Show In Class Browser** right-click menu option to view the selected object in the Class Pane, and its member variables in the Member Pane.

Figure 11-14 Viewing Class Object Information in the Member Pane



Viewing Reference Path of an Object Instance

To view the reference paths that point to an object, right-click an object instance in Class Pane, Local Pane, or Object Hierarchy Browser, and select **View References**, as shown in Figure 11-15. DVE displays the reference paths of the selected object instance in the References dialog box, as shown in Figure 11-16.

Figure 11-15 Viewing References of an Object Instance

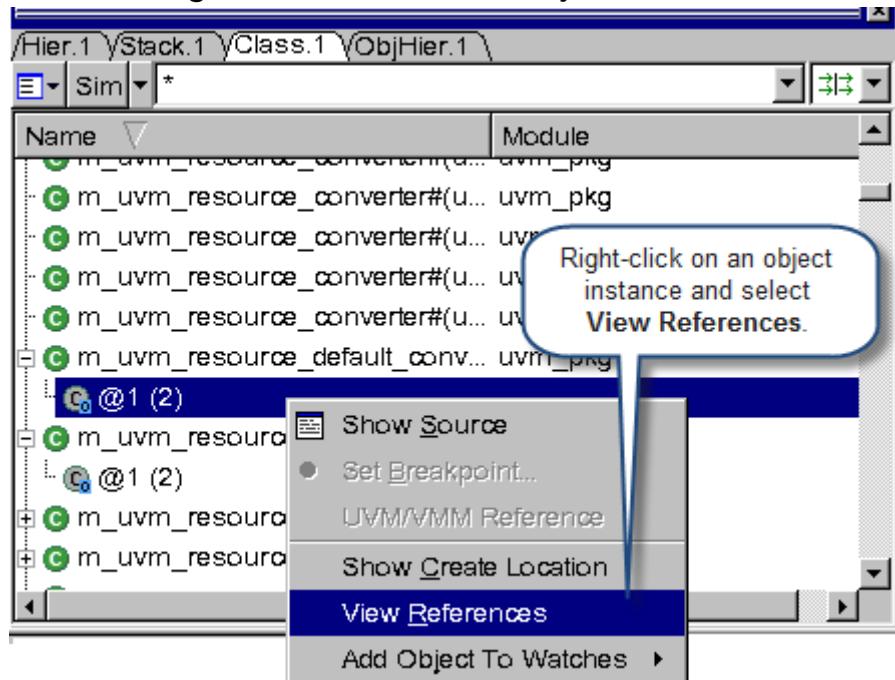


Figure 11-16 References Dialog Box

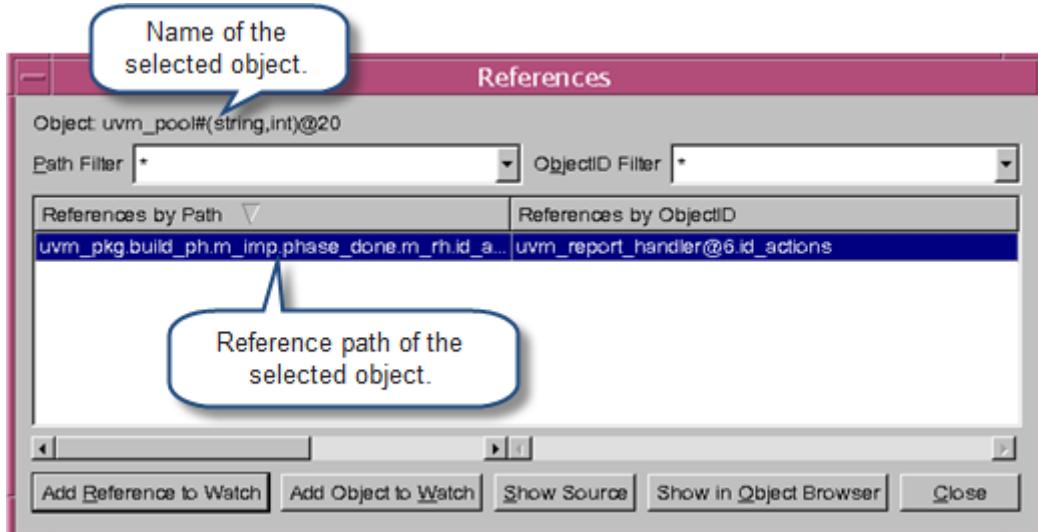
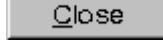


Table 11-7 describes the buttons of the References dialog box:

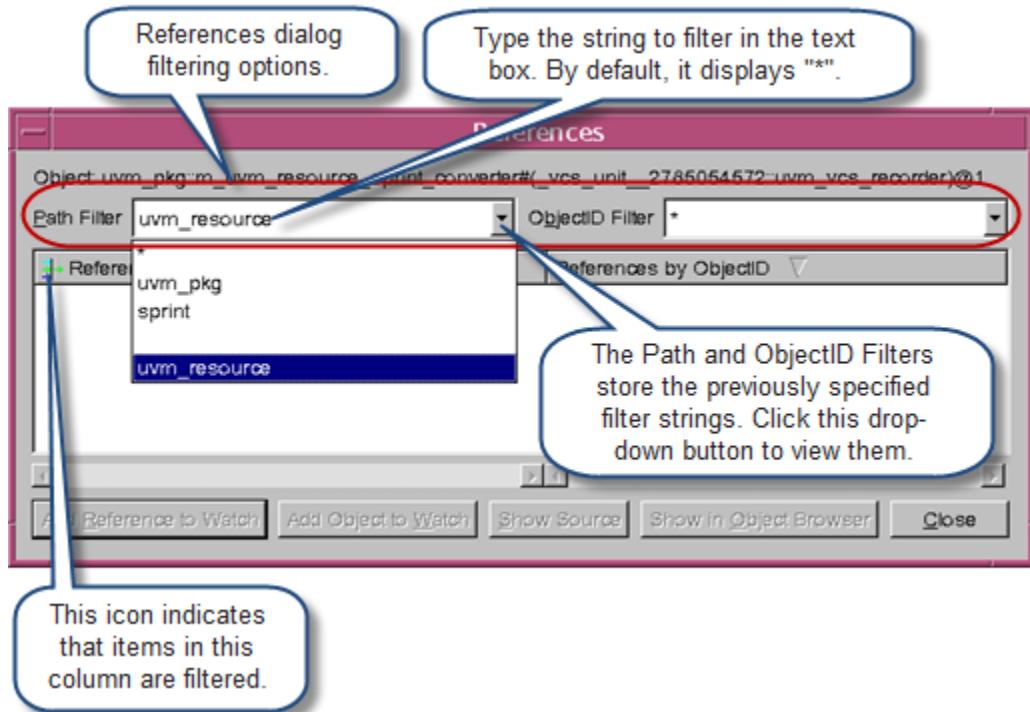
Table 11-7 References Dialog Box Buttons

Button	Description
	Includes the selected reference path in the Watch Pane.
	Includes the selected reference object ID in the Watch Pane.
	Displays the source code of the selected reference path in the Source View.
	Displays the selected reference path in the Object Hierarchy Browser.
	Exits the References dialog box.

Filtering Options in the References Dialog Box

The References dialog box filtering options (see [Figure 11-17](#)) allow you to configure the type of information to display for **References by Path** and **References by ObjectId** columns. [Table 11-8](#) describes the filtering options in the References dialog box.

Figure 11-17 References Dialog Box Filtering Options



These filters allow you to specify the text to filter, and stores the previously specified filter strings. By default, these filters use '*' wildcard character as the filter string.

Table 11-8 Filtering Options in the References Dialog Box

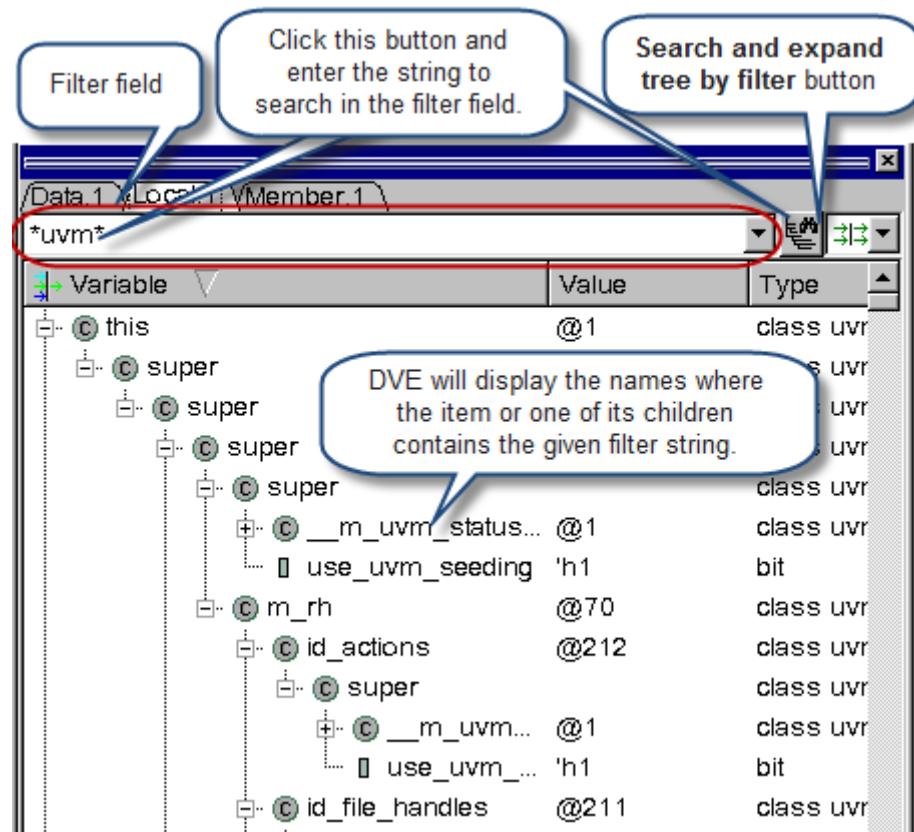
Filter Name	Description
Path Filter	Displays the desired reference paths in the References by Path column.
ObjectID Filter	Displays the desired reference object ID in the References by ObjectID column.

Searching for Dynamic Objects in the Local Pane

DVE allows you to search for dynamic objects in both currently expanded and collapsed folders present in Local Pane. In previous versions, the filter field in the Local Pane supports viewing the filter

string only in the currently expanded contents. From this release onwards, you can use **Search and expand tree by filter button** , as shown in [Figure 11-18](#), to search for the filter string in the entire tree of objects (both expanded and collapsed folders), and view the names where the item or one of its children contains the given filter string.

Figure 11-18 Object Search in the Local Pane

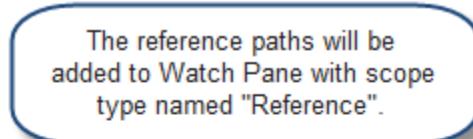


Adding Reference Paths to the Watch Pane

You can use the **Add Object To Watch** button in the Reference dialog box (see section [“Viewing Reference Path of an Object Instance”](#)), to include the selected reference object ID in the Watch Pane.

The reference paths will be added to the Watch Pane with the scope type named as **Reference**, as shown in [Figure 11-19](#), which will have a top-level scope.

Figure 11-19 Reference Paths in the Watch Pane



Variable	Value	Type	Scope
+ C uvm_pkg.build...	@20 class uvm_pool#(...	Reference (uvm_pkg)	
+ C uvm_pkg::uvm...	@7 class uvm_pool#(...	Object (uvm_pkg::uvm_poo...	
... New Variable			

Watch 1 \ Watch 2 \ Watch 3 /

Note:

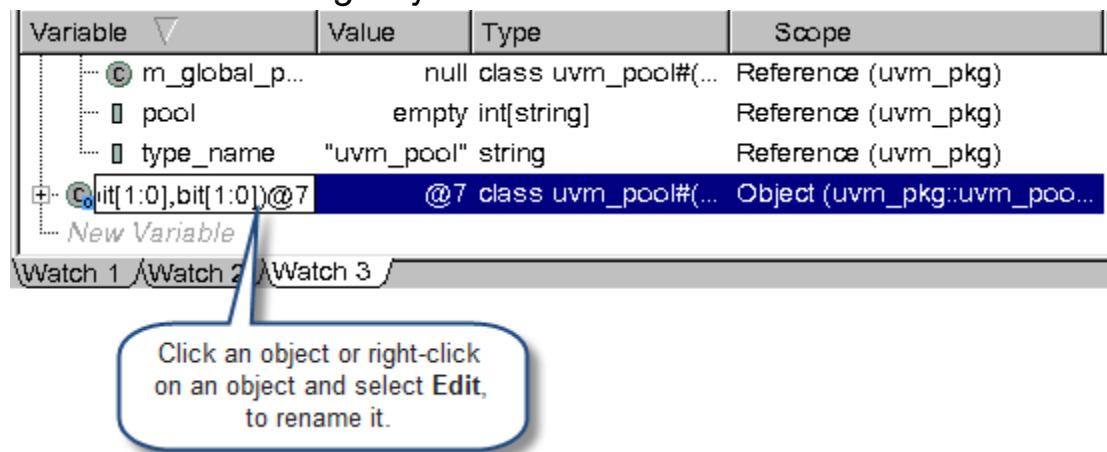
You can add a reference path to the Watch Pane by specifying the path name as a new variable. But these items will be identified as “Local” even though they are present at the top-level.

Renaming Object Name in the Watch Pane

You can rename an object in the Watch Pane by performing one of the following:

- Click the object you wish to rename, as shown in [Figure 11-20](#).
- Right-click an object you wish to rename and select **Edit**.

Figure 11-20 Renaming Object Names in the Watch Pane



Viewing Virtual Interface Object in DVE

You can view the actual value of the Virtual Interface objects in the Local pane, Watch pane, and Source view in DVE. The value of the Virtual Interface object is shown only if it is initialized or else the value “null” is shown. The Type column shows the type as “Virtual Interface” in the Local pane.

The following illustration show the Virtual Interface type with its actual value.

Variable	Value	Type
- c		class C
busIF	mod.trkIF	virtual interface
busmpIF	null	virtual interface
i	0	int
- clk	'hx	reg
modbusIF	mod.trkIF	virtual interface
modportIF2	null	virtual interface

Testbench Debug

Using the following testbench debugger features, you can more efficiently analyze, understand, and debug the behavior of your verification environment.

- Use DVE or UCLI to view object identifier values
- Use DVE or UCLI to create object identifier breakpoints
- Create a breakpoint at the end of a method
- View the parameter type for each parameterized class name in the Hierarchy Pane
- Avoid stepping into VMM, UVM, or OVM code
- Change dynamic variable values in the Local Pane and Watch Pane
- Filter variables in the Local Pane
- Filter objects in the Stack Pane
- View the class in the Class Browser from the Source View and Member Pane
- View the VMM or UVM online help documentation in an external web browser from the Help menu, Class Pane, or Member Pane
- View struct variables in the Local Pane
- View the .size of Dynamic Arrays in Local Pane

This section contains the following topics:

- [“Viewing Object Identifier Values”](#)

- “Creating Object Identifier Breakpoints”
- “Parameterized Class Support”
- “Avoiding Stepping into VMM/UVM/OVM Code”
- “Changing Dynamic Variable Values in DVE”
- “Filtering Variables in Local Pane”
- “Filtering Objects in Stack Pane”
- “Viewing the Class in Class Browser from Source View and Member Pane”
- “Viewing VMM/UVM Documentation”
- “Viewing Struct Variables in the Local Pane”
- “Viewing the .size of Dynamic Arrays in Local Pane”

Viewing Object Identifier Values

You can use DVE or UCLI to view values for a specific instance of an object in your code.

Viewing Object Identifier Values in DVE

DVE displays (in the Local Pane and Watch Pane) a unique object ID for every class instance in the following format:

`<classname>@<instance number>`

Where,

- `<classname>` is the name of a class

- *<instance number>* is the instance number of *<classname>*

For example, if there is a class named `MyClass`, and if you want to refer to the third instance of that class, then DVE displays the ID of the object as `MyClass@3`.

Key points to note

- If *<classname>* is a parameterized class, then the object ID class name is based on the VCS internal name for the class, and not the display name (which includes the types). For example, for the following class:

```
class stack #(type T=int);
function new;
//...
endfunction
endclass

stack #(longint) s1;
stack #(byte) s2;
stack #(shortreal) s3;
```

The VCS internal names are `stack`, `stack_0` and `stack_1`. Therefore, the object ID for the third instance of the `stack #(byte)` class is `stack_0@3`.

- If a class is defined within a package, the object ID consists of the package name, a dot, and then the normal object ID, as shown in the following example:

`MyNamespace.MyClass@3`

- SystemVerilog also supports nested classes (classes defined within other classes). For example, consider the following code:

```
class Env;
    class Other; // Nested class
```

```

        string var1;
    endclass
endclass

class Test;
    class Other; // Nested class
        int count;
    endclass
endclass

```

In this example, class `Other` is declared as an inner class. To remain unique, the real name is: `Env::Other` or `Test::Other`. The object ID is represented with a unique name such as `Env::Other@3`.

Viewing Object Identifier Values Using UCLI Commands

You can use the `show -object` command to view an object ID. This command displays the object ID in the following format:

`{<classname>@<instance number>}`

The object ID is only shown for class objects. For other objects, the value is shown as `{ }`.

Viewing Object Identifier Values in Local Pane

The Local Pane displays the object instance number in the value field, as shown in [Figure 11-21](#).

Figure 11-21 Viewing Object Identifier Values in Local Pane

Variable	Value	Type
outer	@1	class Ext_Outer
super		class Outer
inner	@1	class Outer::Inner (\Ou...
a	10	integer

No instance number is shown for super class

First instance of class Ext_Outer

Viewing Object Identifier Values in Watch Pane

The Watch Pane displays the object instance number in the value field, as shown in [Figure 11-22](#).

Figure 11-22 Viewing Object Identifier Values in Watch Pane

The screenshot shows a Watch pane with the following data:

Variable	Value	Type	Scope
outer	@1 class Ext_...	Object (Ext_Outer@1)	
super	class Outer	Object (Ext_Outer@1)	
inner	@1 class Out...	Object (Ext_Outer@1)	
a	10 integer	Object (Ext_Outer@1)	
outer	@1 class Ext_...	Object (Ext_Outer@1)	

Three callout boxes provide additional information:

- The Value column displays the instance number of an object. This may be different than the object ID shown in the Scope column, if the variable is a member object of another class.
- Click this spin button to switch between the possible values depending on the context for variable values, as per the following table.
- The Scope column displays the object ID for dynamic objects, or the static scope for static variables.

Variable Type	Root of Variable	Default when added to Watch Pane
Class Object	Object, Dynamic, Local	Object
Local/Automatic Variables	Dynamic, Local	Dynamic
Static Design Elements	Static, Local	Static

Figure 11-23 Viewing the Scope of Dynamic and Local Scope Types

Variable	Value	Type	Scope
- outer	@1 class Ext_...	Dynamic	
super	class Outer	Dynamic	
inner	@1 class Out...	Dynamic	\$prog
a	10 integer	Dynamic	Dynamic
- outer	@1 class Ext_...	Dynamic	Object (Ext_Outer@1)

Watch 1 / Watch 2 / Watch 3

Hover the mouse pointer over the Dynamic or Local scope type to view its scope information in the tooltip.

The Scope Column displays the Scope type, that is, Dynamic, Object, Local, or Static.

Figure 11-24 Viewing the Scope Type Information in a ToolTip

Value	Type	Scope
@1 class Ext_...	Dynamic	
class Outer	Dynamic	
@1 class Out...	Dynamic	Select root context for variable:
Object:	Object	Object: Object ID that points to the class instance.
Dynamic:	Dynamic	Dynamic: Context is based on the scope that existed when the variable was added to the Watch pane.
Local:	Local	Local: Context is based on the active scope.
@1 class Ext_...	Dynamic	

Watch 3 /

Viewing Object Identifier Example

[Example 11-2](#) shows a test file (test.v) used to illustrate viewing object identifiers.

Example 11-2 Viewing Object Identifier Design File (test.v)

```
class Outer;
    class Inner;
        integer a;
        function new(integer a);
            this.a = a;
        endfunction
    endclass

    Inner inner;
endclass

class Ext_Outer extends Outer;
    function new(integer a);
        super.inner = new(a);
    endfunction
    task print;
        $display("%m.a = %0d",super.inner.a);
    endtask
endclass

Ext_Outer outer;
initial begin
    outer = new(10);
    outer.print;
end
endprogram
```

Compile the test.v file shown in [Example 11-2](#).

```
% vcs -sverilog -debug_all test.v
```

Invoke the DVE GUI

```
% simv -gui&
```

Creating Object Identifier Breakpoints

You can use DVE or UCLI to set a breakpoint for a specific instance of an object in your code.

Creating Object Breakpoints Using UCLI Commands

You can use the UCLI commands shown in [Table 11-9](#) to create object breakpoints:

Table 11-9 UCLI Commands to Create Object Breakpoints

UCLI Command	Description
<code>stop -file <file> -line <lineno> -object_id <objectID></code>	Stops the execution at a specified line if the object is available.
<code>stop -in <classname method name> -object_id <objectID></code>	Stops the execution at a specified function or task if the object is available.

Note:

- You can specify an object instance using `-object_id <classname@instance_number>` (even before the given class or instance exists).
- If you do not specify either `-object` or `-object_id`, this command sets the breakpoint on all instances of the class.

- If you restart the simulation, the breakpoint is restored using object_id.

Creating Conditional Breakpoints

You can use the `-condition <expression>` option to specify a conditional expression.

Creating Object Breakpoints Using DVE Breakpoints Dialog

You can also use the DVE Breakpoints dialog box to create object breakpoints. Select **Simulator > Breakpoints**. In the Breakpoints dialog box, click **Define>>** to display the breakpoint creation tabs, as shown in [Figure 11-25](#).

**Figure 11-25 Creating Object Breakpoints using Breakpoints Dialog Box
Breakpoints**

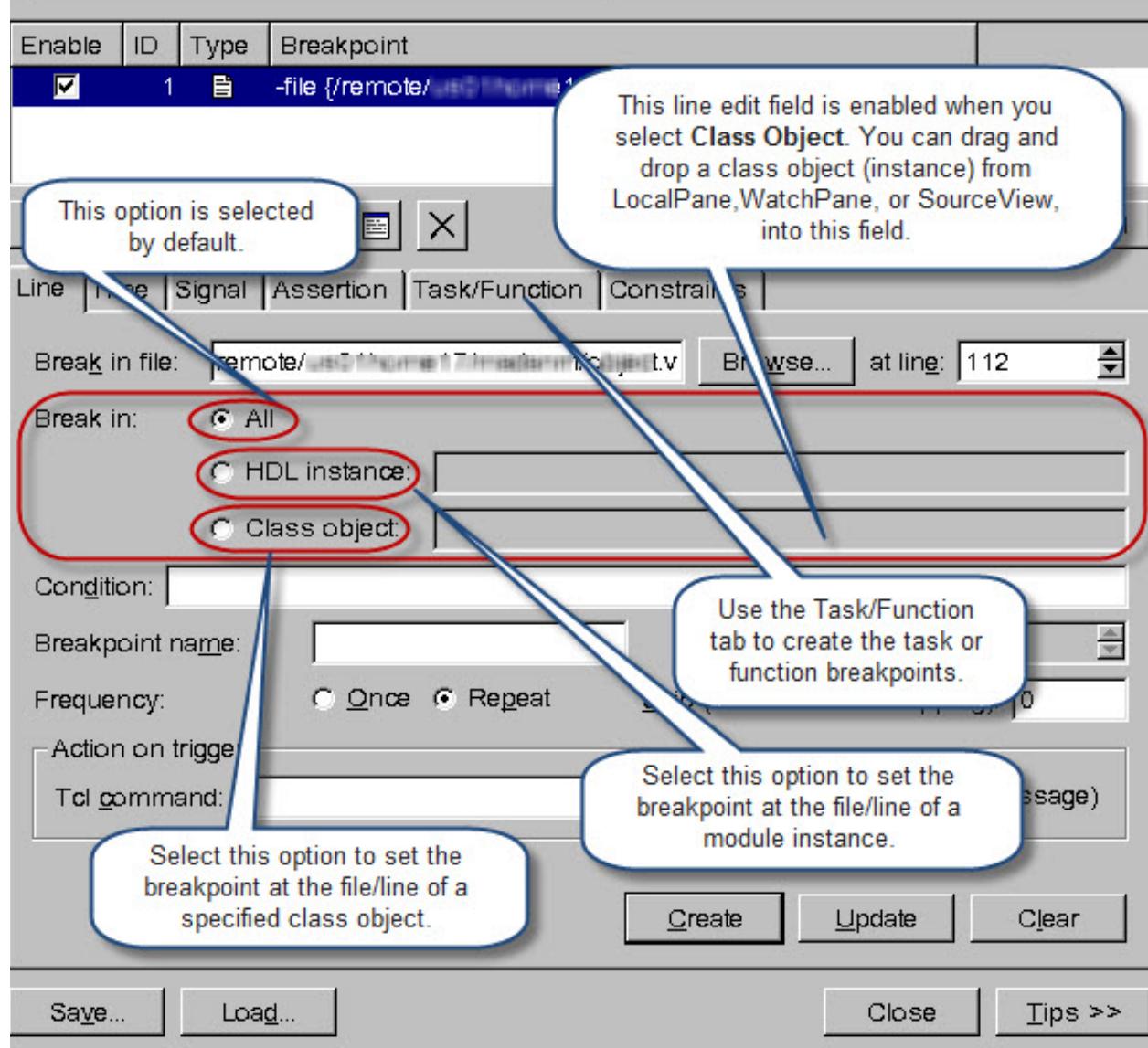


Figure 11-26 Dropping Objects into the Line Edit Fields

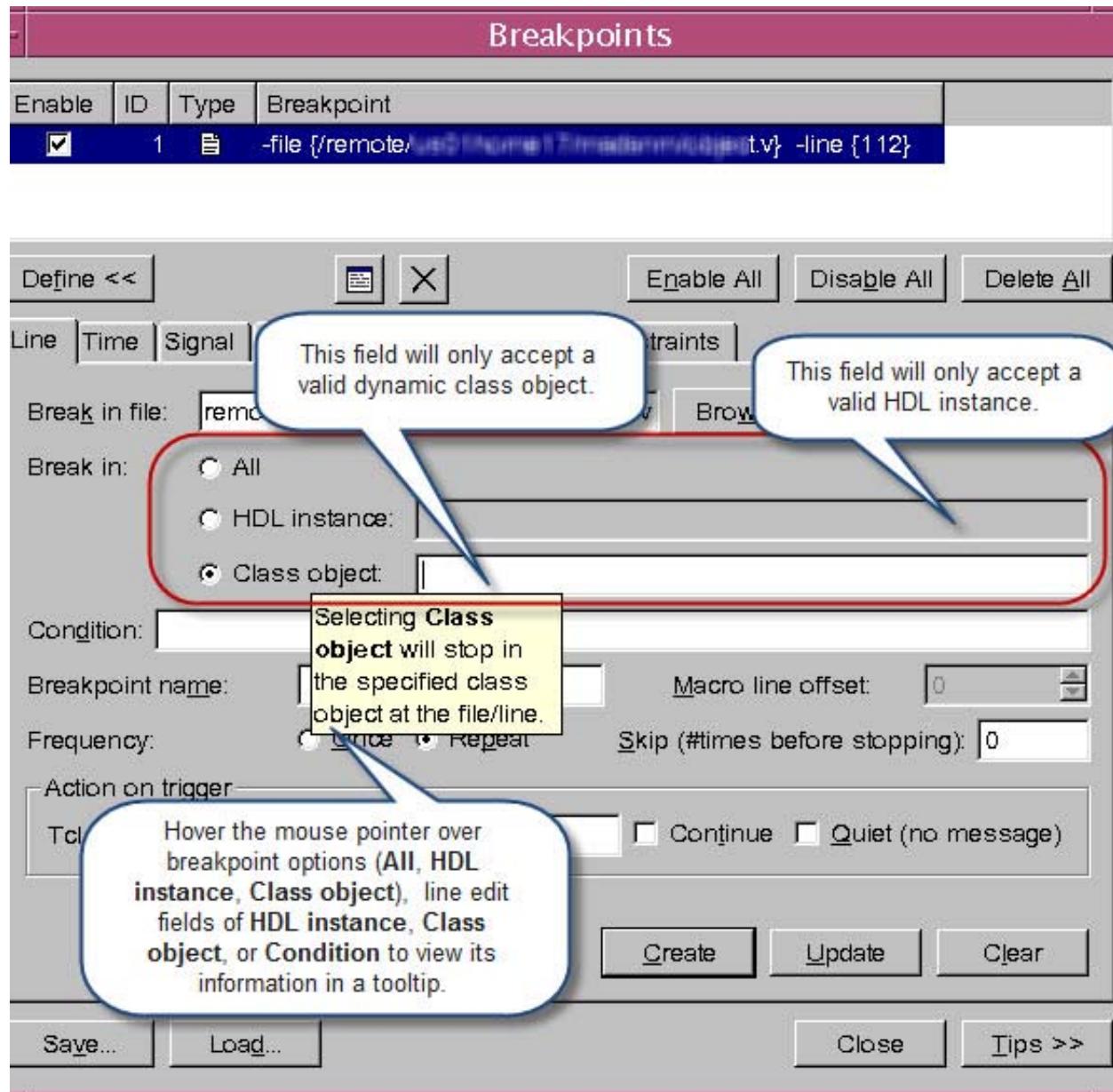
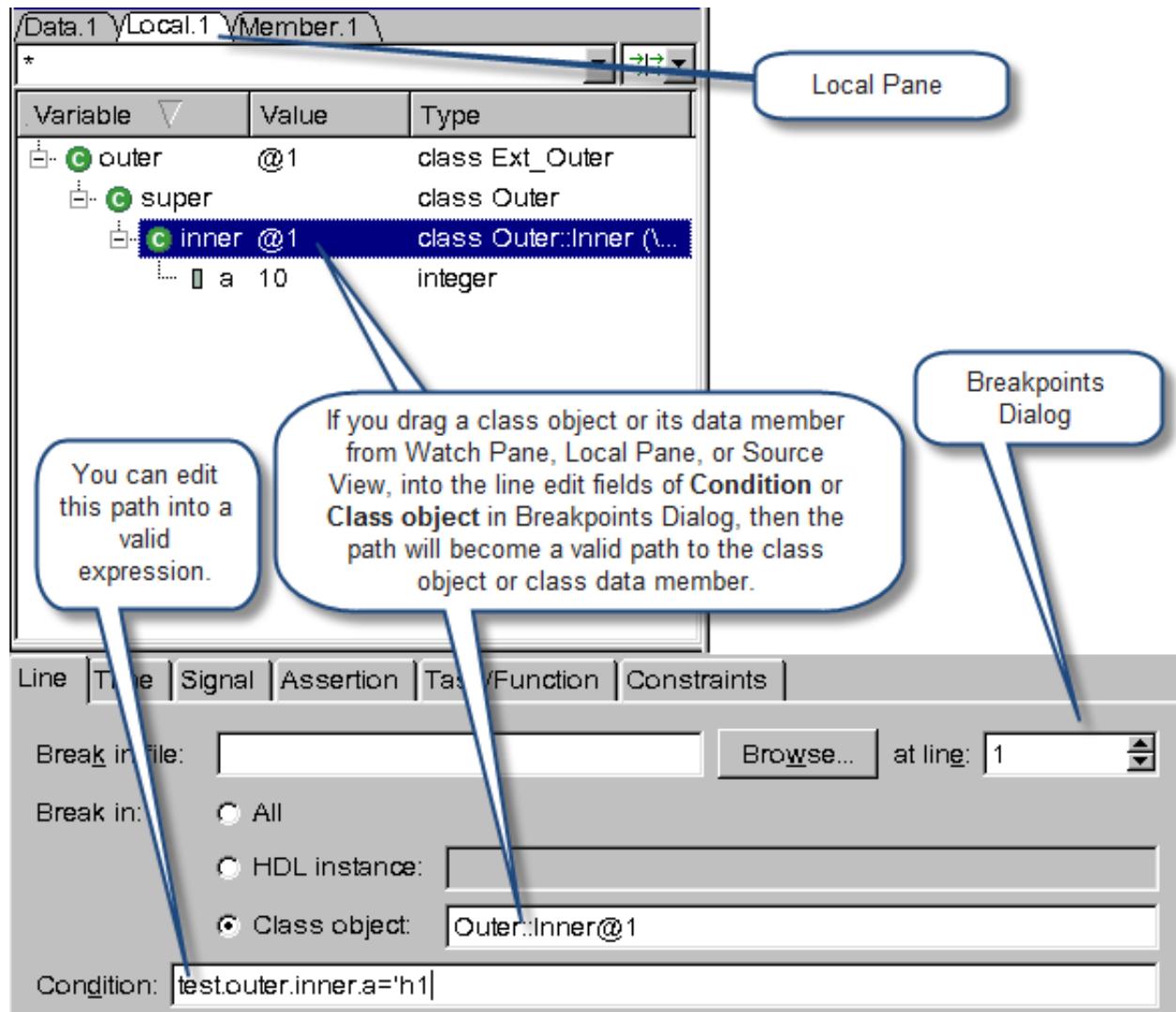


Figure 11-27 Dragging Objects into Condition and Class Object Fields



Creating Breakpoints at the End of a Method

You can use the UCLI command shown in [Table 11-10](#) or the DVE Breakpoints dialog box (see [Figure 11-28](#)) to create a breakpoint at the end of a method:

Table 11-10 Command to Create Breakpoint at the end of a Function or Task

UCLI Command	Description
<code>stop -in <function/task name> -end</code>	Stops the execution at the line of <code>endfunction</code> or <code>endtask</code> . When a function has multiple return statements, the breakpoint hits at the <code>endfunction</code> line after one of the return statements is executed.

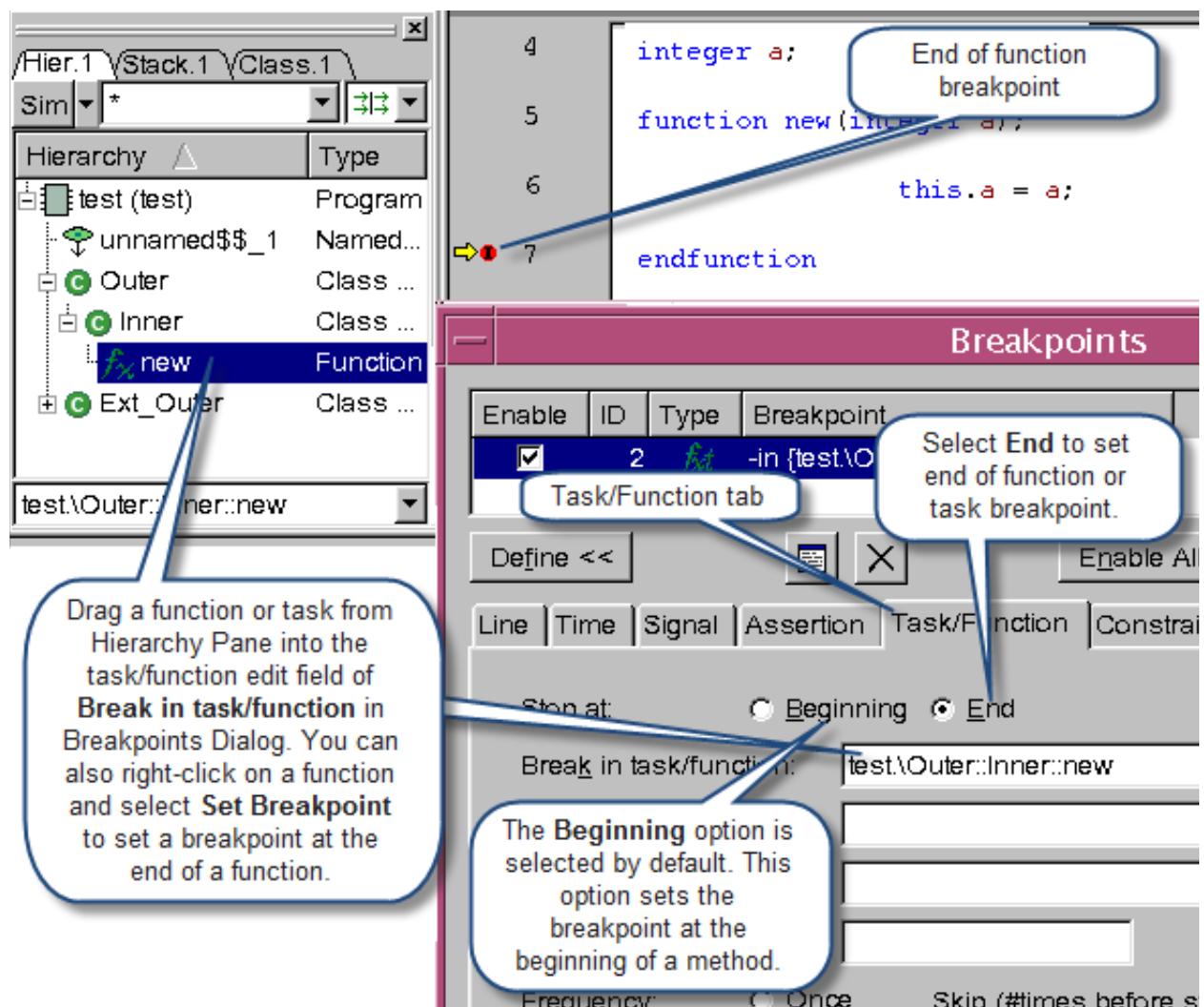
Task

You can use the task/function tab in the Breakpoints dialog box to create a breakpoint at the end of a function or task.

To create a breakpoint at the end of a method using DVE:

1. Drag a function or task from the Hierarchy Pane into the line edit field of **Break in task/function**, as shown in [Figure 11-28](#).
2. Select **End** in the **Task/Function** tab.
3. Click **Create** in the Breakpoints dialog box to view the breakpoint in the Source View, as shown in [Figure 11-28](#).

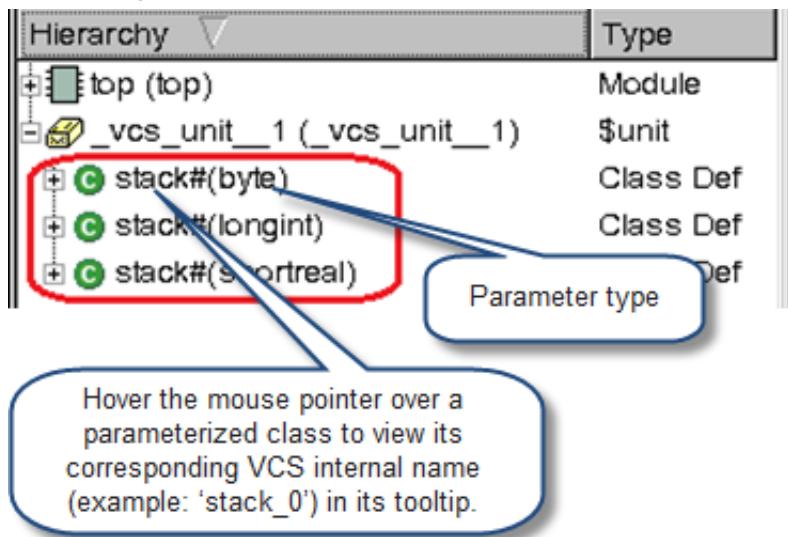
Figure 11-28 Creating a Breakpoint at the end of a Function or Task in DVE



Parameterized Class Support

DVE displays the parameter type for each parameterized class name in the Hierarchy Pane, as shown in Figure 11-29.

Figure 11-29 Viewing Parameterized Class Types



Avoiding Stepping into VMM/UVM/OVM Code

You can use the DVE Preference option **Avoid stepping into UVM/VMM code for ‘next’ and ‘step’ commands** or the `stepintotblib` UCLI configuration variable to avoid stepping into VMM, UVM, or OVM code.

Note:

The value of the `stepintotblib` variable is synchronized with the above-mentioned DVE preference option. Enabling this variable (`config stepintotblib on`) in DVE enables the above-mentioned DVE preference option and vice versa.

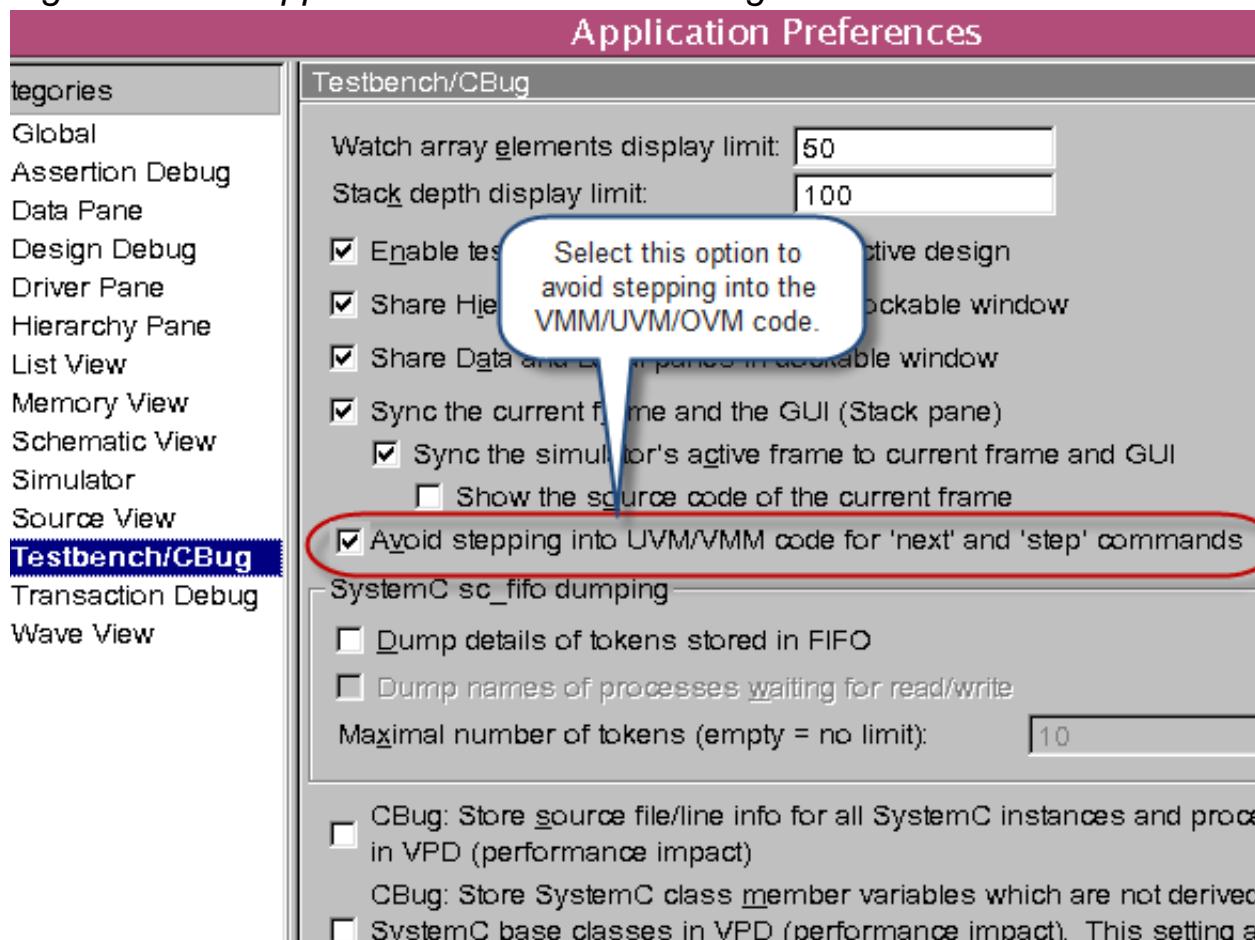
To select the **Avoid stepping into UVM/VMM code for ‘next’ and ‘step’ commands** option:

1. Select **Edit > Preferences**.

The Application Preferences dialog box appears.

2. In the Testbench/CBug category, select **Avoid stepping into UVM/VMM code for ‘next’ and ‘step’ commands**, as shown in Figure 11-30.

Figure 11-30 Application Preferences Dialog Box



Changing Dynamic Variable Values in DVE

You can use the **Change Value** right-click menu option to change the values of dynamic variables in the Local Pane and Watch Pane.

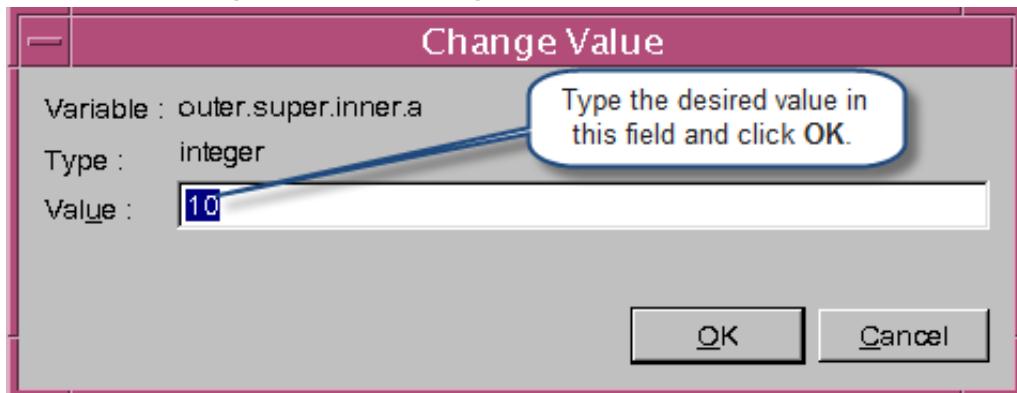
This menu item is enabled only if:

- The right-click selection in the Local Pane or Watch Pane is a single item.
- DVE is in interactive mode.
- The selected item corresponds to a testbench dynamic variable at the current simulation time (for example, not a \$vcdblustblog recorded variable).

To change dynamic variable values in DVE:

1. Select a variable in the Local Pane or Watch Pane, right-click, and select **Change Value**. Or select **Simulator > Change Value**. The Change Value dialog box appears, as shown in [Figure 11-31](#).

Figure 11-31 Change Value Dialog Box

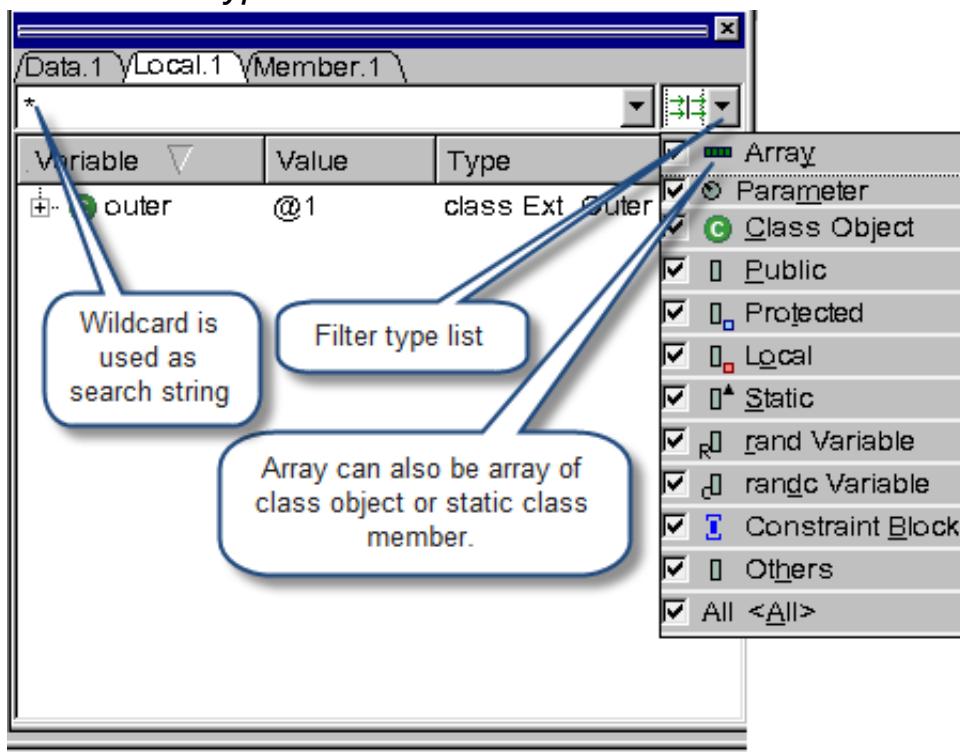


2. To change the value of selected variable, enter a value and click **OK**.

Filtering Variables in Local Pane

You can filter variables based on their types in the Local Pane. To filter signals based on their types, click the Filter type list and select or clear the desired type. [Figure 11-32](#) shows the available filter types.

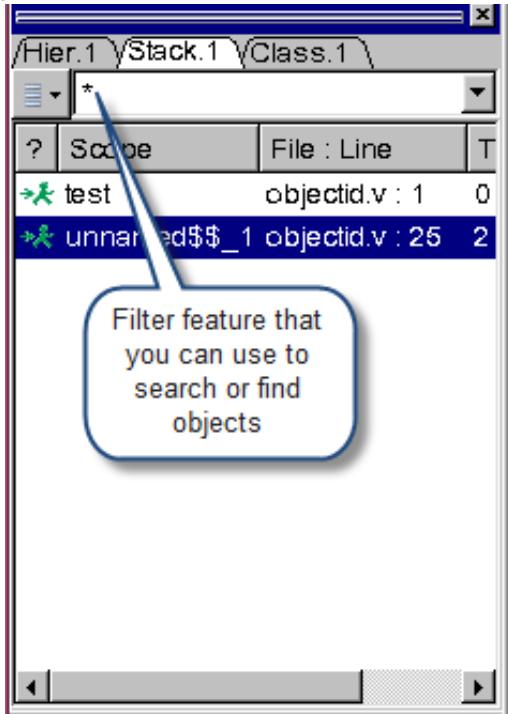
Figure 11-32 Filter Types in Local Pane



Filtering Objects in Stack Pane

You can filter objects based on the text string in the Stack Pane. For example, type the search string using regular expressions or wildcards (*) in the text box to filter the objects (see [Figure 11-33](#)).

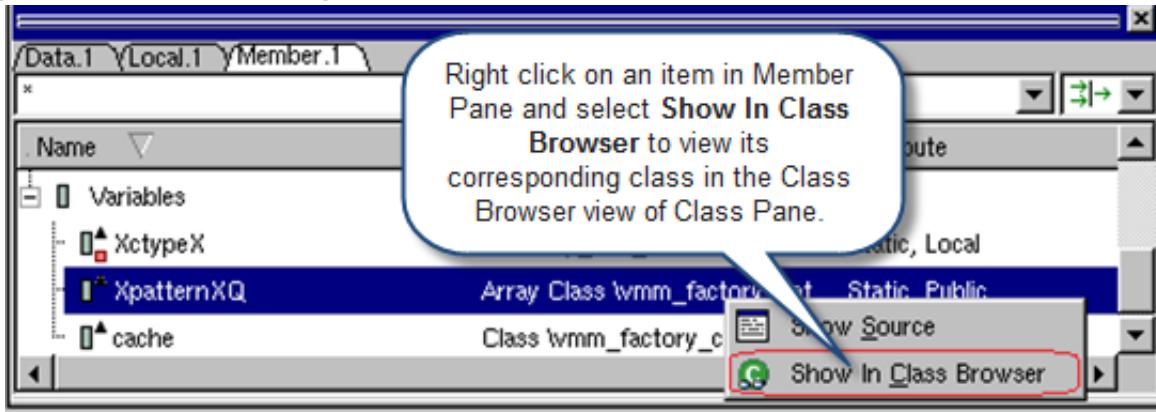
Figure 11-33 Filtering Objects in Stack Pane



Viewing the Class in Class Browser from Source View and Member Pane

To view the class in the class browser from the Member Pane, select a class in the Member Pane, right-click, and select **Show in Class Browser**, as shown in [Figure 11-34](#). Or from the Member View, drag and drop an item into the Class Pane to view its corresponding class in the Class Browser.

Figure 11-34 Viewing Class in Class Browser from Member Pane



The **Show in Class Browser** menu option is enabled only if:

- A single item is selected.
- The text in the Type column indicates a class name. This text is in the format of `Class <class_name>` or `Array Class <class_name>`. The class can be user-defined or from libraries such as VMM or UVM. [Figure 11-35](#) shows an example for VMM classes.

Figure 11-35 VMM Classes

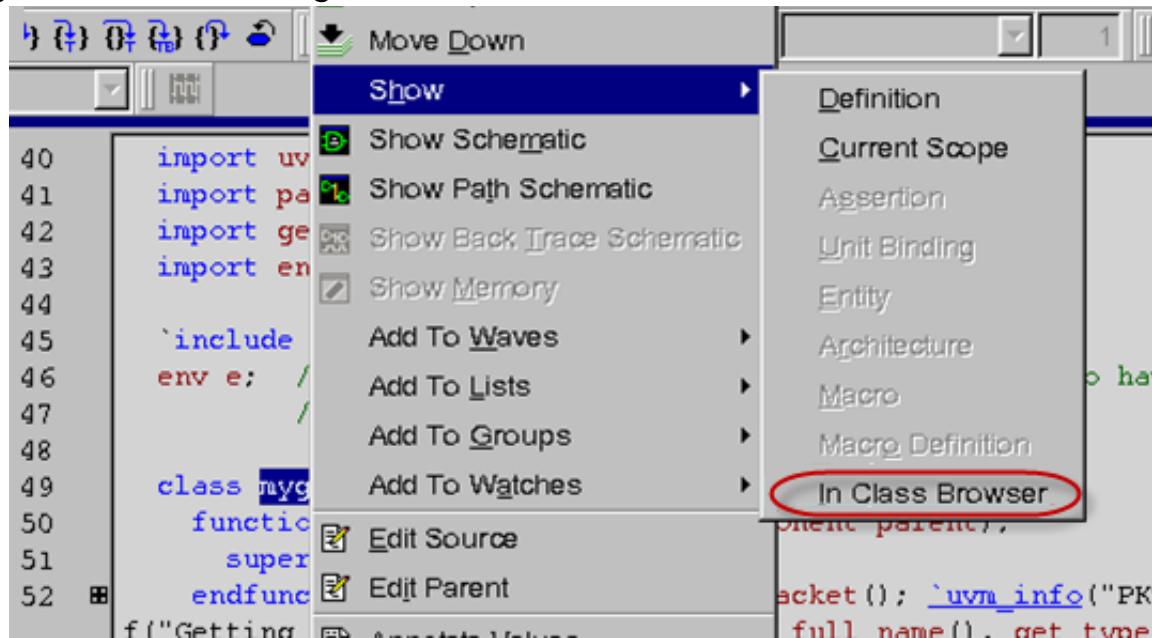
/Data.1 \Local.1 \Member.1			
Name	Type	Attribut	
Variables			
XcbsX	Array Class vmm_ral_field_c...	Public	
access	Enum	Local	
cover_on	Int	Local	
desired	Bit	Local	
fname	String	Local	
individually_accessi...	Bit	Local	
lineno	Int	Local	
log	Class vmm_log	Static, F	
lsb	Int	Local	
mirrored	Bit	Local	
name	String	Local	
parent	Class vmm_ral_reg	Local	

To view the class in the class browser from the Source View, select a class name in the Source View, right-click, and select **Show > in Class Browser** (see [Figure 11-36](#)).

The **Show > in Class Browser** option is enabled only if:

- The selection is in a scope that is currently active.
- The selected text is a class name.

Figure 11-36 Viewing Class in Class Browser from Source View



Viewing VMM/UVM Documentation

DVE allows you to view the VMM or UVM online help documentation in an external web browser from the Help menu, Class Pane, or Member Pane.

To view the VMM or UVM online help documentation from the Help menu, select **Help > UVM/VMM Reference**.

To view the VMM or UVM online help documentation from the Class Pane or Member Pane, select a UVM or VMM class in the Class Pane or Member Pane. Right-click and select the **UVM/VMM Reference** option to view its corresponding documentation in the VMM or UVM online help. If you select multiple items, the VMM or UVM online help documentation opens for the first selected item.

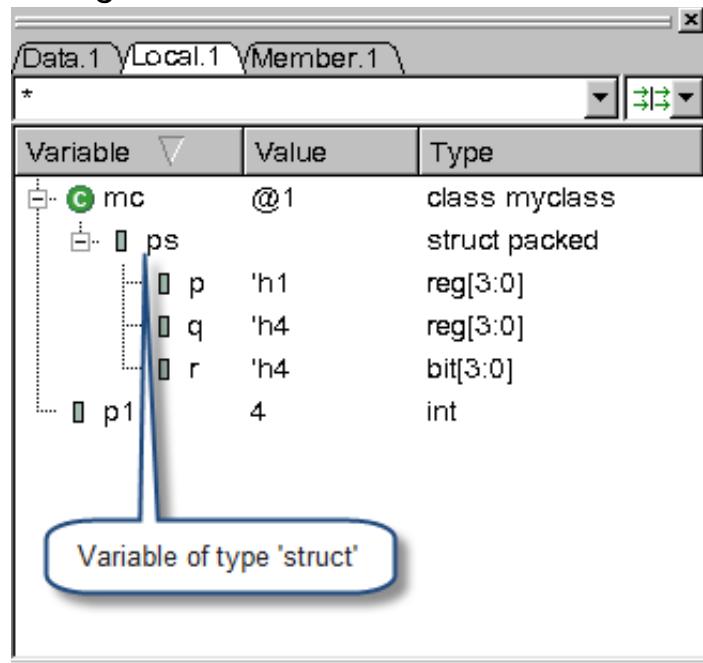
Note:

The **UVM/VMM Reference** option is enabled only if the selected class (or the parent class of the selected method or variable in Member Pane) is a UVM or VMM class.

Viewing Struct Variables in the Local Pane

DVE displays struct variables in the Local Pane, as shown in [Figure 11-37](#).

Figure 11-37 Viewing Struct Variables in the Local Pane



The screenshot shows the DVE Local pane with the path `/Data.1\Local.1\Member.1`. The pane lists variables with their values and types. A tooltip at the bottom indicates that the selected variable is of type 'struct'. The data is as follows:

Variable	Value	Type
mc	@1	class myclass
ps		struct packed
p	'h1	reg[3:0]
q	'h4	reg[3:0]
r	'h4	bit[3:0]
p1	4	int

You can also use the `show` and `get` UCLI commands to view struct variables and their values.

Struct Variables Example

[Example 11-3](#) shows the contents of the test.sv file.

Example 11-3 Struct Variables Design File (test.sv)

```
program test;
int p1=4;

class myclass;

struct packed {
reg [3:0] p;
logic [3:0] q;
bit [3:0] r;
} ps;

endclass

myclass mc = new;
initial begin
mc.ps= {p:1'b1,default:p1};
// initialization with variable:value
#2 $display("%b %b",mc.ps.p,mc.ps.q,mc.ps.r);
end
endprogram
```

Compile the test.v file shown in [Example 11-3](#).

```
% vcs -sverilog -debug_all test.sv
```

Invoke the DVE GUI:

```
% simv -gui&
```

Viewing the `.size` of Dynamic Arrays in Local Pane

DVE displays the `.size()` of dynamic arrays, associate arrays, and queues in the following format:

```
size: <current_level_size> (<accumulative_size>)
```

where,

- `<current_level_size>` is the size of the current dimension
- `<accumulative_size>` is the total size of all dimensions

For example: `size: 2 (20)`

That is, DVE displays both the cumulative and descendants for `.size()` in the Local Pane, as shown in [Figure 11-38](#). Mouse over the desired value to view this information in the ToolTip.

Figure 11-38 Viewing the .size of Dynamic Arrays in Local Pane

Variable	Value	Type
A	size: 2 (7)	int[][0:2][]
	[1] size: 3 (7)	int[0:2][]
	[2] size: 7	int[]
B	size: 3 (0)	int[2:0][][]
C	empty	int[][0:2][]
D	size: 3 (6)	int[0:2][][]
	[2] size: 2 (6)	int[][]
	[0] size: 6	int[]

Dynamic Arrays Example

[Example 11-4](#) shows the contents of the test.sv design file.

Example 11-4 Dynamic Arrays Design File (test.sv)

```
module top;
int A [] [3] [] ;
int B [2:0] [] [] ;
int C [] [3] [] ;
int D [3] [] [] ;

initial begin
A = new [2];
A[1][2] = new [7];
D[2] = new [2];
D[2][0] = new [6];

$display("Size of A: %d", A.size());
```

```
$display("Size of A[1] [2]: %d", A[1][2].size());
$display("Size of D[2]: %d", D[2].size());
#1 $finish;
end
endmodule
```

Compile the test.sv file shown in [Example 11-4](#):

```
% vcs -sverilog -debug_all test.v
```

Invoke the DVE GUI:

```
% simv -gui&
```

Following is the console output from the simv run:

```
-----
Size of A: 2
Size of A[1] [2]: 7
Size of D[2]: 2
-----
```

Debugging Threads

DVE allows you to do the following:

- View all the threads in your design and the status of the selected thread in the Stack Pane.
- Filter the named and unnamed scopes which are not active call stacks.
- Provide better names for unnamed scopes (of type initial, always or fork) from active call stacks.
- Search for a thread in the Stack Pane
- Set thread-specific breakpoints in the Stack Pane
- Double-click a thread ID in Console Pane to view it in the Stack Pane
- Set different background color for stack frame from user code and UVM (VMM, OVM) library code.

Thread Debugging Example

Consider the following test case:

Example 11-5 A Design File with Thread Debug (test.sv)

```
program top;

`include "uvm_macros.svh"
import uvm_pkg::*;

class test extends uvm_test;
    `uvm_component_utils(test)
```

```

function new(string name, uvm_component parent = null);
    super.new(name, parent);
endfunction

virtual function void report();
    $write("** UVM TEST PASSED **\n");
endfunction

endclass

initial
begin
    run_test();
end

endprogram

```

Compile the `test.sv` code shown in [Example 11-5](#), as follows:

```
% vcs -sverilog -ntb_opts uvm -debug_all test.sv
```

Invoke the DVE GUI, as follows:

```
% simv +UVM_TESTNAME=test -gui
```

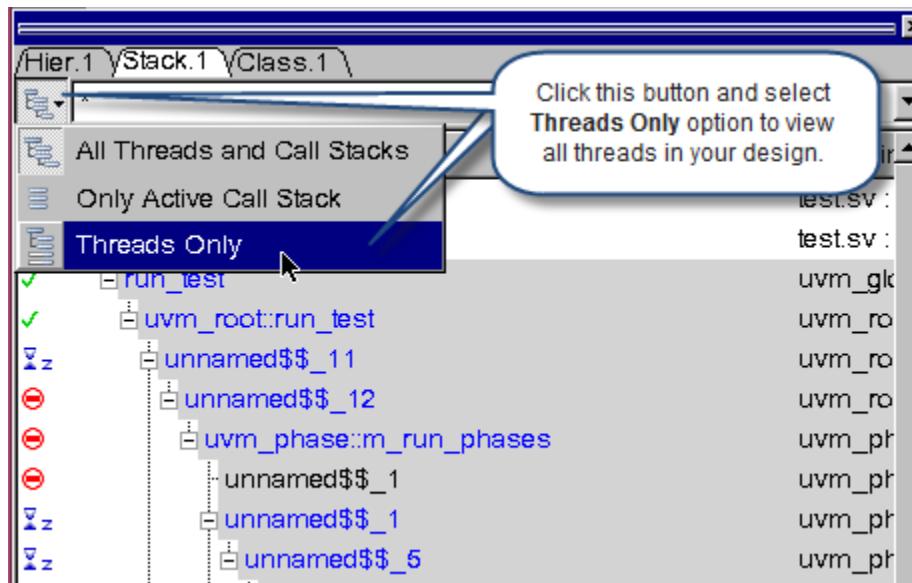
Viewing Status of a Thread in the Stack Pane

DVE allows you to view all the threads in your design and the status of the selected thread in the Stack Pane. This helps you to view the thread-related information and top active call stack for the selected thread.

To view only threads in your design and status of the selected thread, perform the following steps:

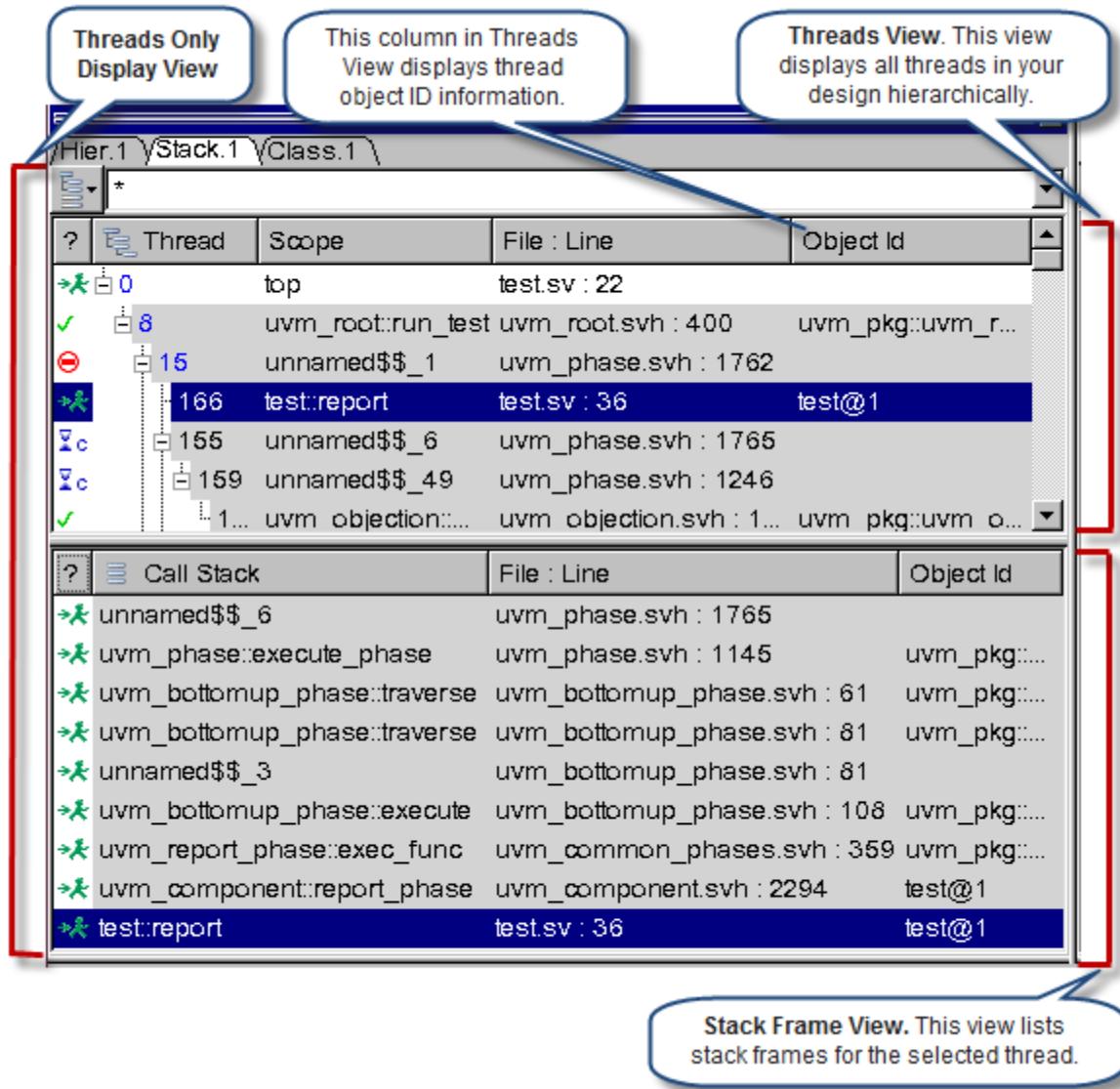
1. In the Stack Pane, click the Stack Mode button  and select **Threads Only** from the drop-down list, as shown in [Figure 11-39](#), to view the Threads Only display view (see [Figure 11-40](#)).

Figure 11-39 Viewing All Threads in Your Design



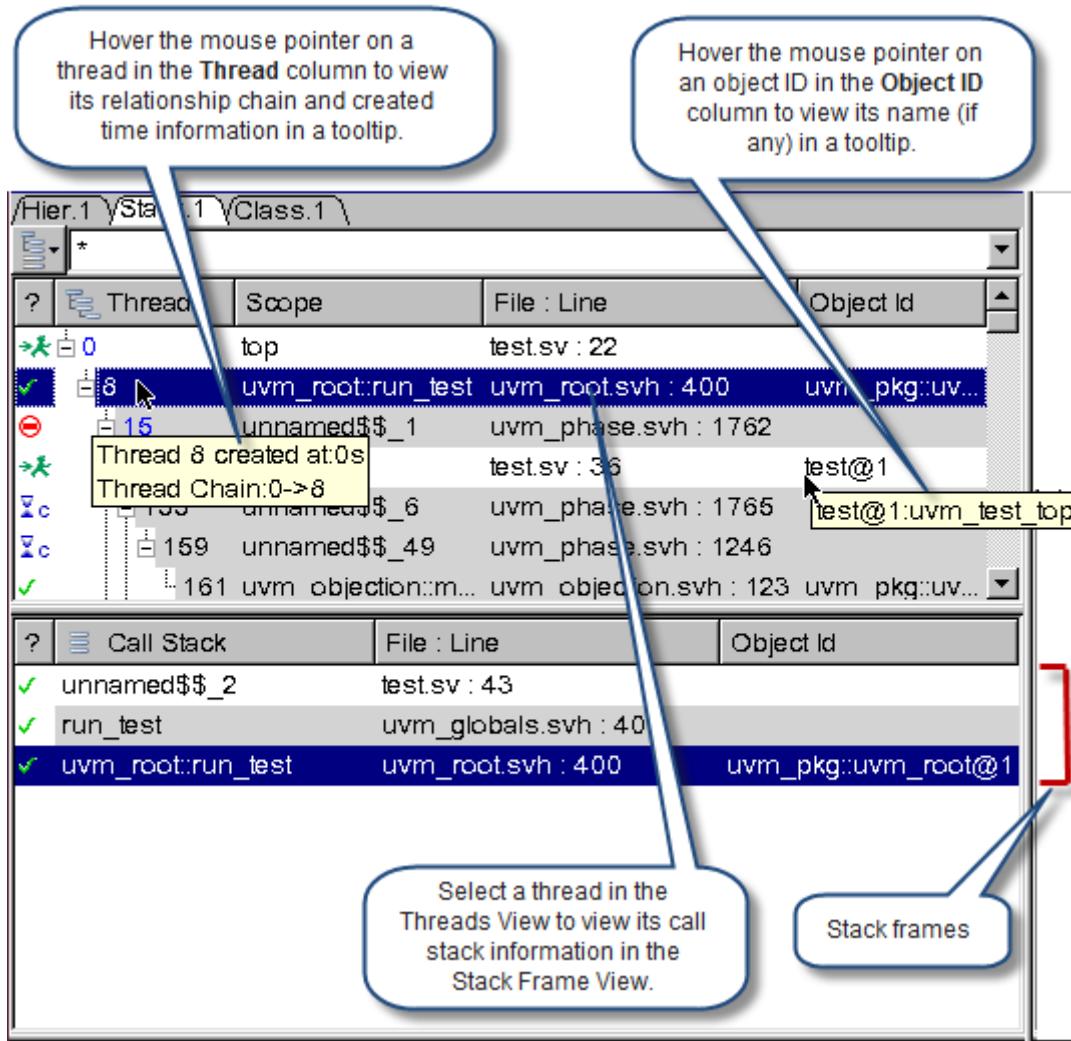
The Threads Only display consists of the threads view and stack frame view, as shown in [Figure 11-40](#). The threads view displays all threads in your design hierarchically. The stack frame view displays stack frames for the selected thread.

Figure 11-40 The Threads Only View



2. Select a thread in the threads view to view its stack trace information in the stack frame view, as shown in [Figure 11-41](#).

Figure 11-41 Viewing the Call Stack Information of a Thread



Place the cursor over a thread in the Thread column to bring up a tool tip that displays its relationship chain and created time information, as shown in [Figure 11-41](#).

Searching a Thread in the Stack Pane

You can use the Find dialog box, as shown in [Figure 11-42](#), to search for a thread in the Stack Pane using the following steps:

1. In the Stack Pane, click the Stack Mode button  and select the desired stack mode:



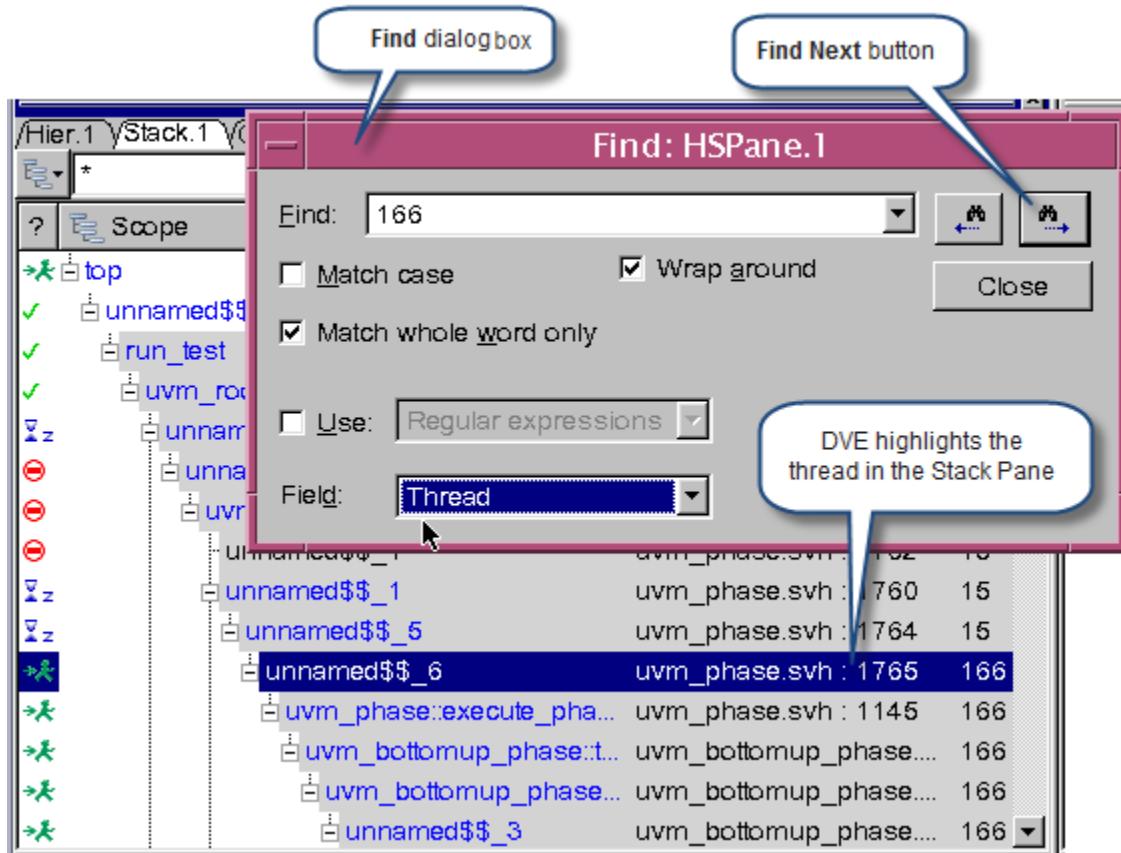
2. From the **Edit** menu, select **Find**. The Find dialog box appears.
3. From the **Field** drop-down list, select **Thread**.
4. Enter the string that you want to find in the **Find** field.
5. You can refine your search by selecting the **Match Whole Word Only** or **Match Case** check boxes.

Table 11-11 Search Options to Customize a Search

Option	Description
Match Whole Word Only	Match a complete word.
Match Case	Search for a word or text string with specific capitalization.

6. Click the **Find Next** button in the dialog box. DVE highlights the thread in the Stack Pane, as shown in [Figure 11-42](#).
7. Click **Close** to close the Find dialog box.

Figure 11-42 Searching a Thread in the Stack Pane



Using Object ID Column in the Threads Only Display View

You can drag and drop the desired object ID from the Object ID column into:

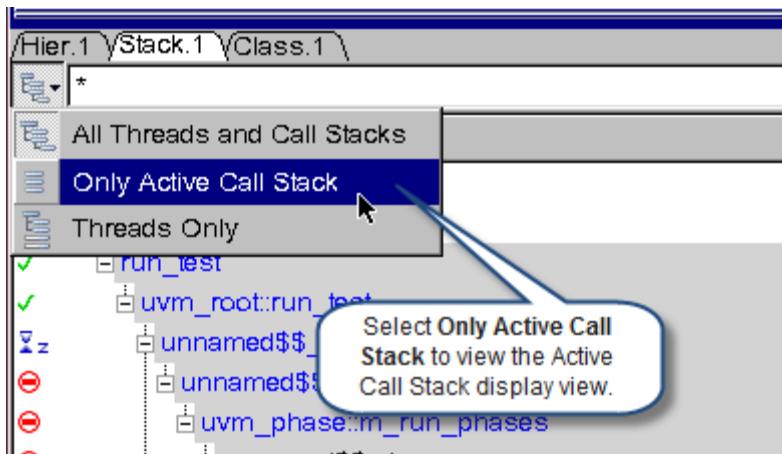
- the source view to view the source code for the object ID
- the Watch Pane
- the **Breakpoint name** field of the Breakpoints dialog box as a condition

Filtering Unnamed Scopes in the Active Call Stack View

The active call stack view allows you to filter the named and unnamed scopes (call stacks of type fork, initial, and always) which are not active call stacks.

To display the active call stack view, click the Stack Mode button  and select **Only Active Call Stack** from the drop-down list, as shown in [Figure 11-43](#).

Figure 11-43 Viewing The Active Call Stack Display



The active call stack view displays call stacks, as shown in [Figure 11-44](#). Click the **Filter** button to filter out the named and unnamed scopes that are not active call stacks. The scope that is located at the top of a new thread is named after the scope above it (for example, the scope below fork, initial or always, is named Fork Thread, Initial Thread, or Always Thread), and is the start of a new thread, as shown in [Figure 11-45](#).

Figure 11-44 The Active Call Stack Display View

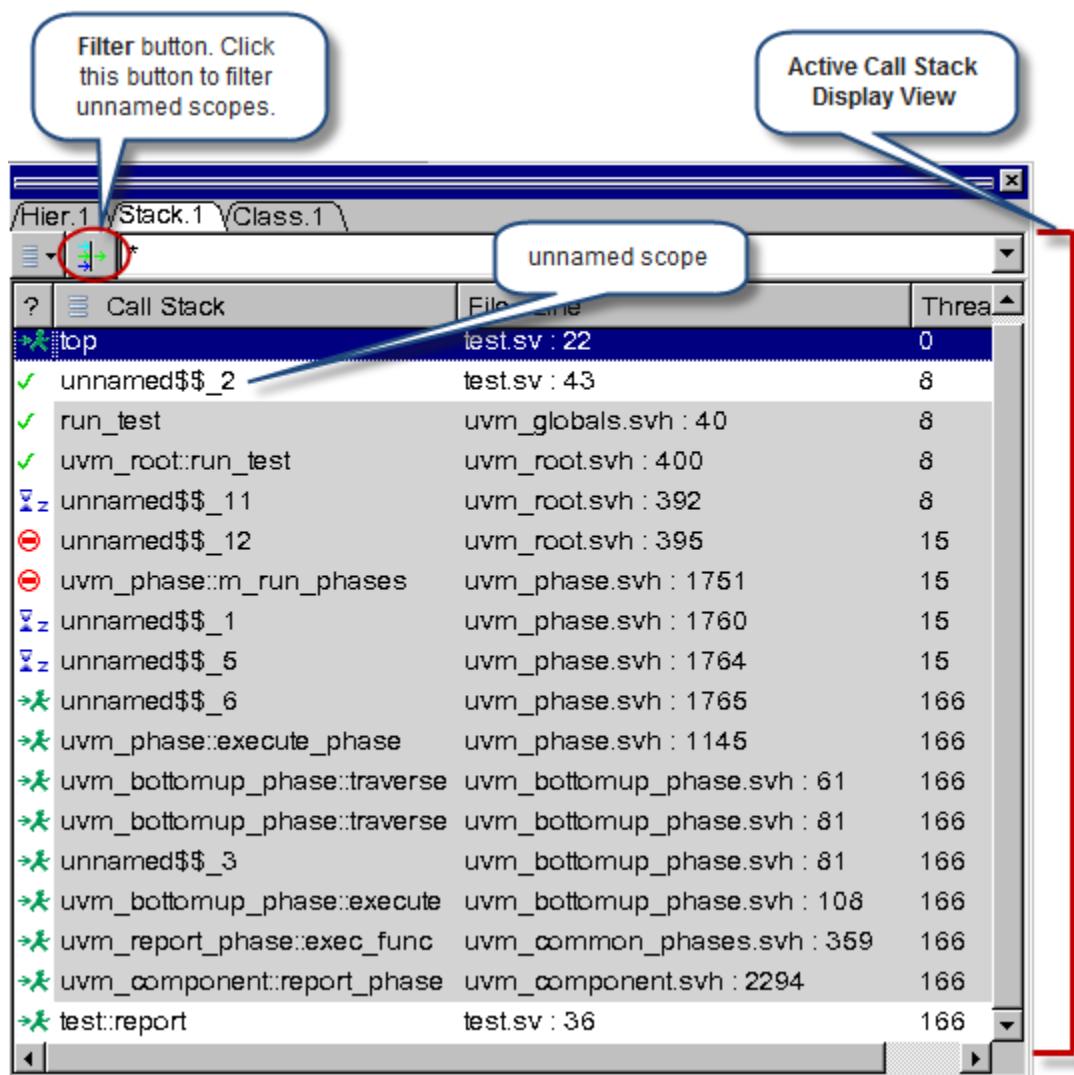


Figure 11-45 Filtering Unnamed Scopes

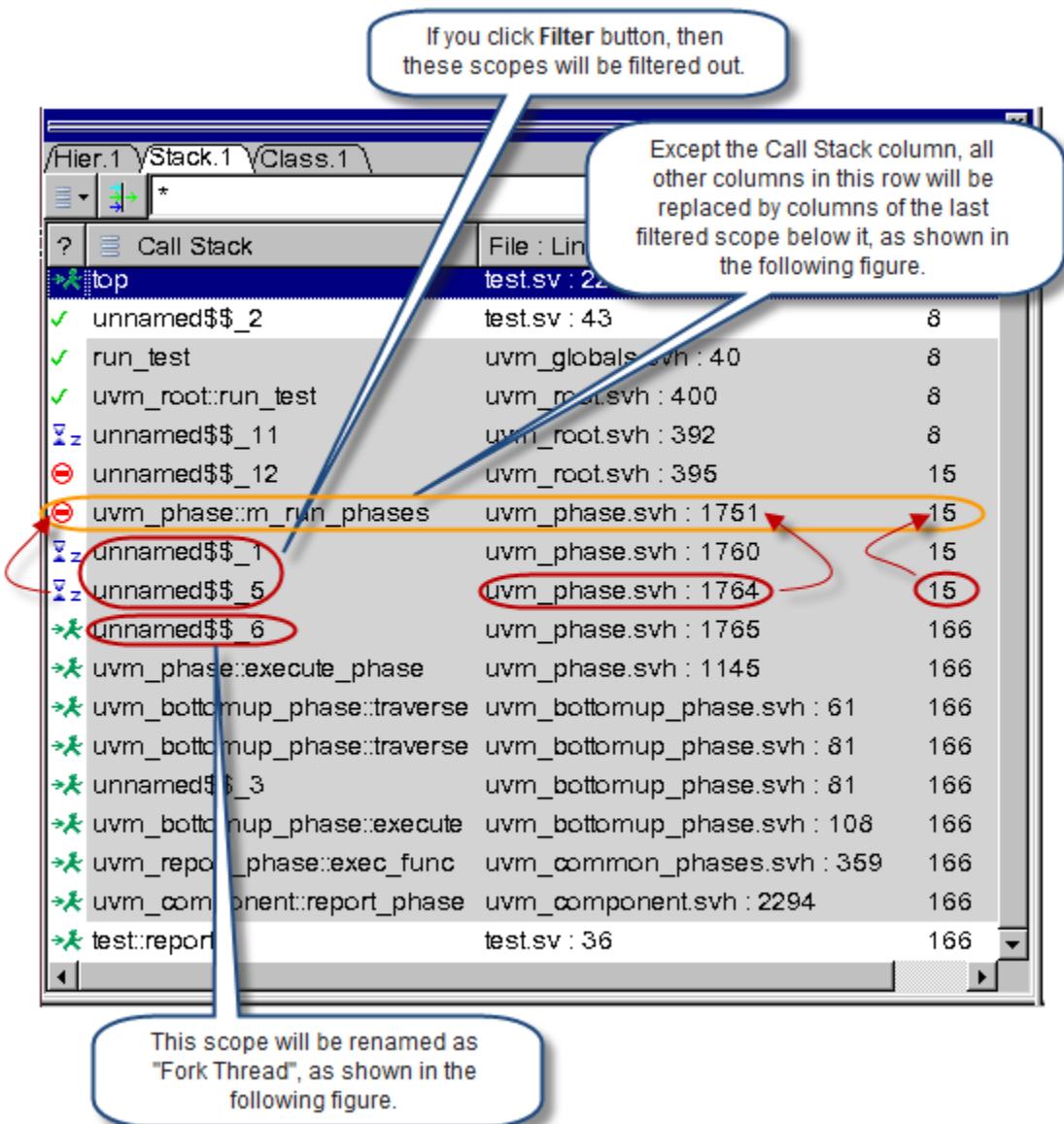


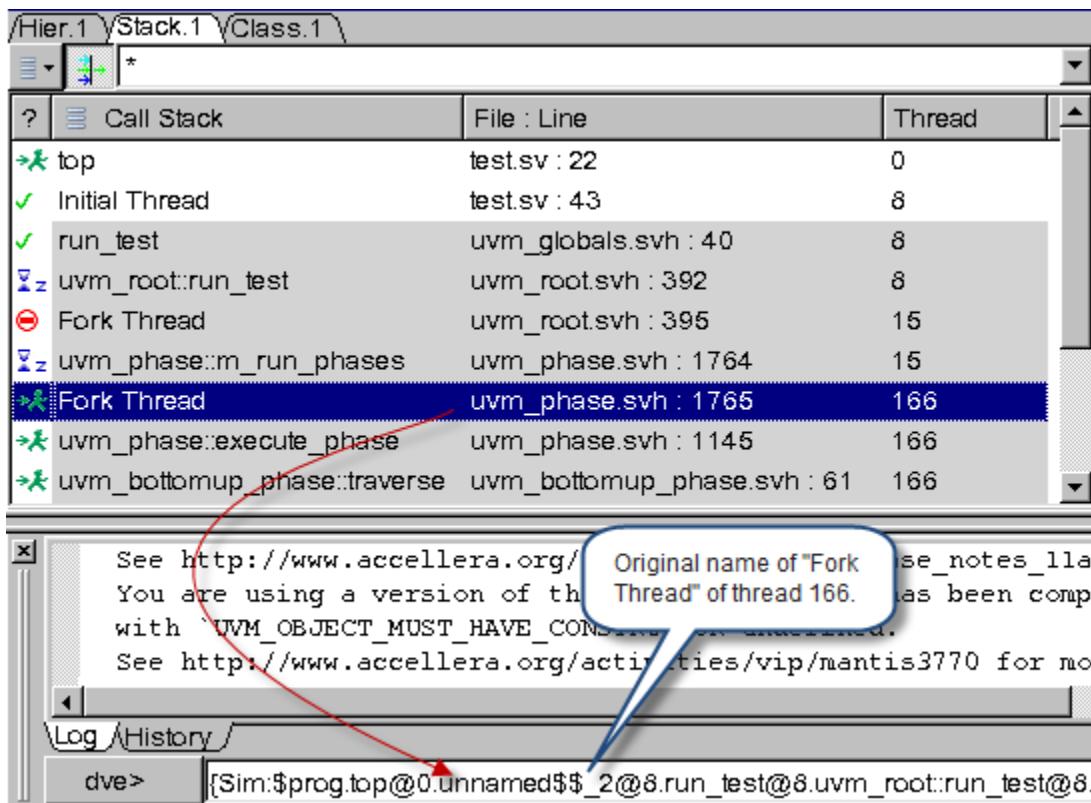
Figure 11-46 shows the active call stack view after filtering.

Figure 11-46 The Active Call Stack View After Filtering

Hier.1 \ Stack.1 \ Class.1			
	Call Stack	File : Line	
		Thread	
→*	top	test.sv : 22	0
✓	Initial Thread	test.sv : 43	8
✓	run_test	uvm_globals.svh : 40	8
✗	uvm_root:run_test	uvm_root.svh : 392	8
✗	Fork Thread	uvm_root.svh : 395	15
✗	uvm_phase::m_run_phases	uvm_phase.svh : 1764	15
→*	Fork Thread	uvm_phase.svh : 1765	166
→*	uvm_phase::execute_phase	uvm_phase.svh : 1145	166
→*	uvm_bottomup_phase::traverse	uvm_bottomup_phase.svh : 61	166
→*	uvm_bottomup_phase::traverse	uvm_bottomup_phase.svh : 81	166
→*	uvm_bottomup_phase::execute	uvm_bottomup_phase.svh : 108	166
→*	uvm_report_phase::exec_func	uvm_common_phases.svh : 359	166
→*	uvm_component::report_phase	uvm_component.svh : 2294	166
→*	test::report	test.sv : 36	166

The Fork Thread, Initial Thread, or Always Thread scope retains its original name if you drag and drop it into other view. For example, [Figure 11-47](#) shows the original name of thread 166 that was dragged and dropped into the DVE command-line field, it is named Fork Thread in the active call stack view (see [Figure 11-46](#)).

Figure 11-47 Dragging and Dropping Scopes Into Other Views



Support for Thread-Specific Breakpoints in the Stack Pane

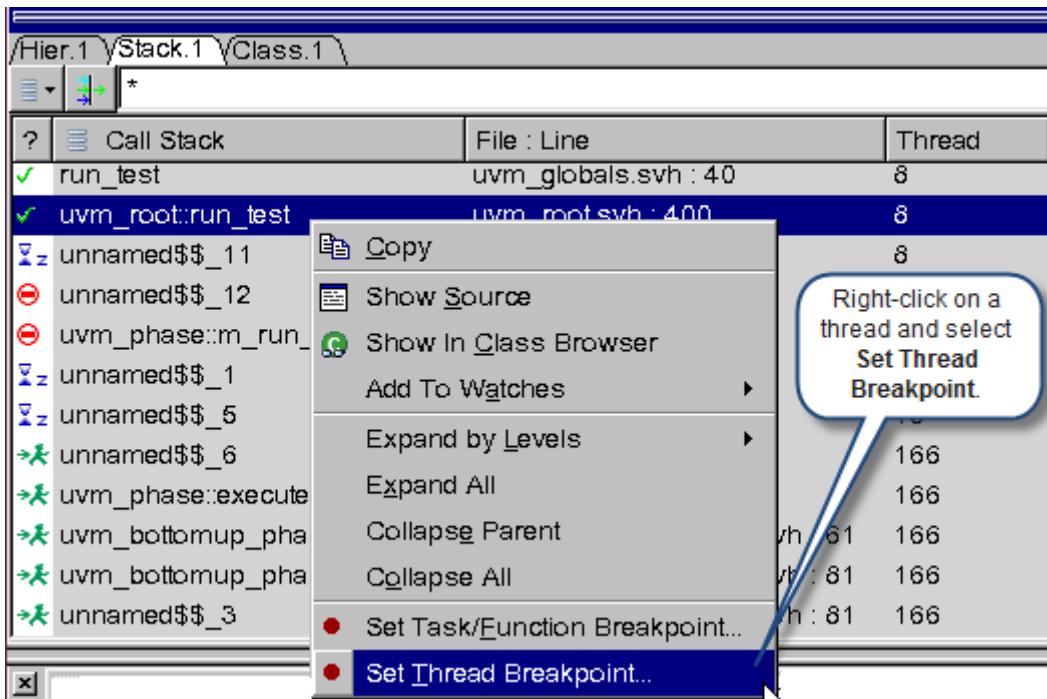
You can use the **Set Thread Breakpoint** right-click option to set breakpoint on the desired thread in the Stack Pane.

To set a breakpoint on a thread, do the following:

1. In the Stack Pane, right-click on a thread and select the **Set Thread Breakpoint** pull-down menu command, as shown in Figure 11-48.

This brings up the Breakpoints dialog box.

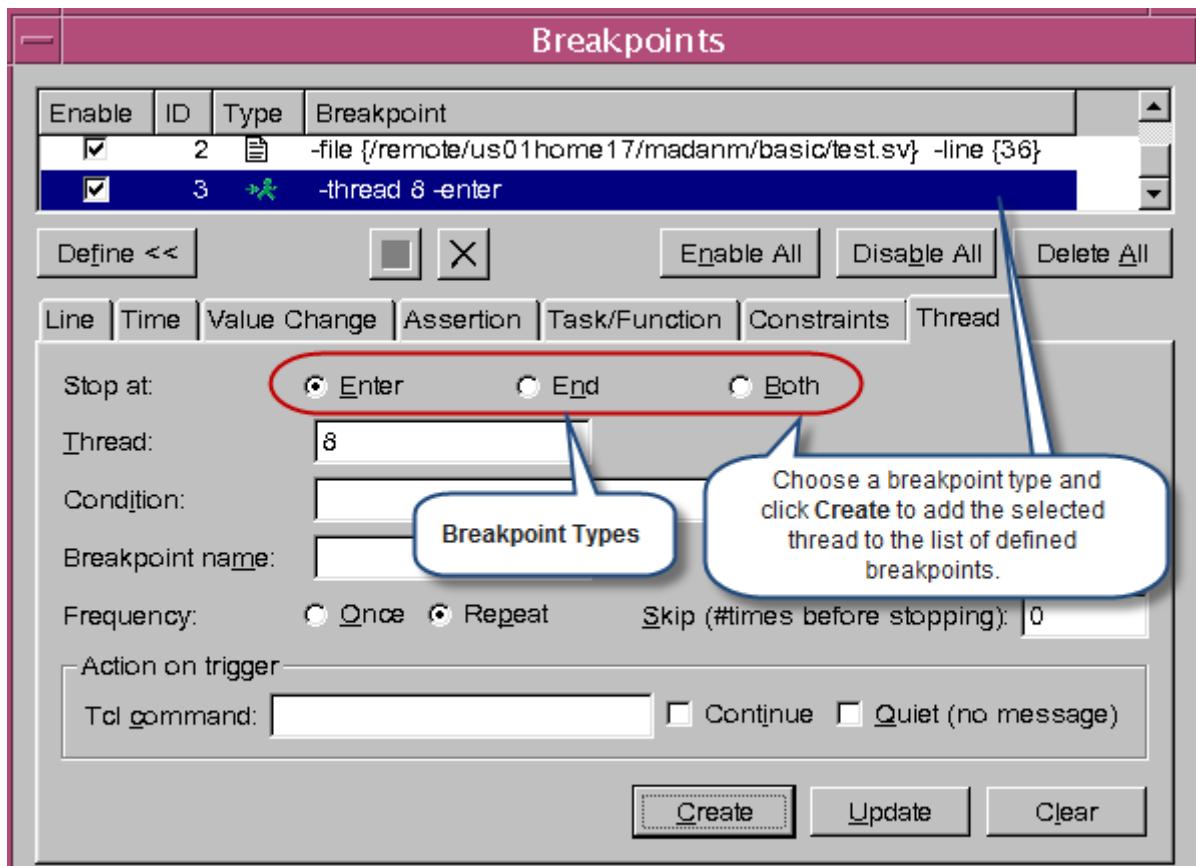
Figure 11-48 Setting a Breakpoint on a Thread in the Stack Pane



2. In the Thread tab in the Breakpoints dialog box, choose the breakpoint type (**Enter**, **End**, or **Both**) that you want to set. [Table 11-12](#) lists the breakpoint types available in the Thread tab.

4. Click **Close**.

Figure 11-49 Selecting Breakpoint Type in the Breakpoints Dialog Box



Viewing the Console Pane Thread in the Stack Pane

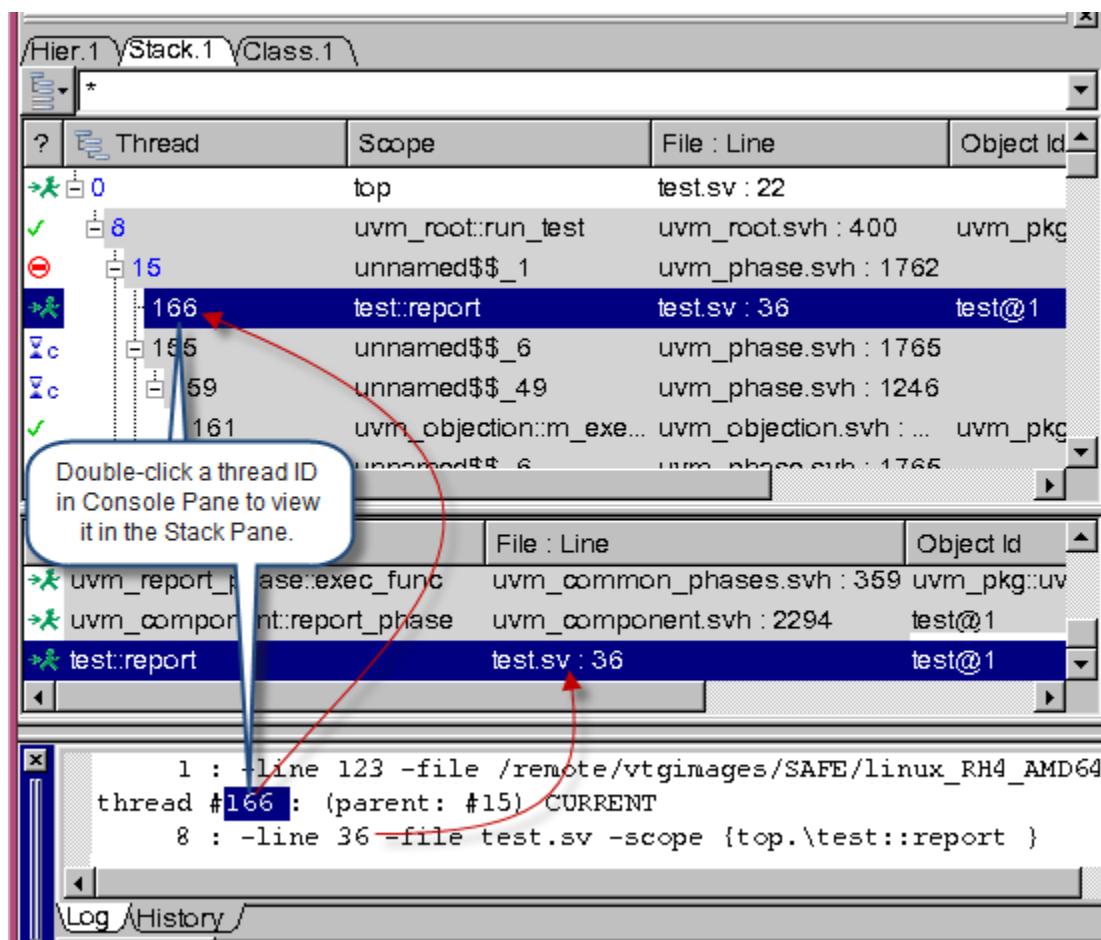
You can double-click a thread ID in the Console Pane to view it in the Threads Only view of the Stack Pane.

To view the Console Pane thread in the Stack Pane, do the following:

1. Use the UCLI `thread` command in the DVE command-line to view the list all active threads in the Console Pane.
2. Click a thread ID in the Console Pane.

DVE highlights the thread in the Stack Pane, as shown in [Figure 11-50](#).

Figure 11-50 Viewing the Console Pane Thread in the Stack Pane



Configuring the Background Color of a Stack Frame in the Stack Pane and Class Pane

You can set different background colors for the stack frame from user and library code. [Table 11-13](#) lists the default background colors of the Stack Pane from user and library code.

Table 11-13 The Default Background Colors of the Stack Frame

Code Type	Default background color of stack frame
User Code	White
UVM (VMM, OVM) library Code	Gray

To change background color of the stack frame from user code and library code:

1. Select **Edit > Preferences**.

The Applications Preferences dialog box appears.

2. In the **Testbench/CBug** category, **Background color for stack frame item** region, select one of the following:

- The drop-down button for **Stack frame from user code**
- The drop-down button for **Stack frame from library code**

Either choice brings up a color palette for the background color as shown in [Figure 11-51](#).

3. Select a color from the color palette and click **Apply**.
4. Click **OK**.

For example, if you want to set green color for the stack frame from library code, click the drop-down button, as shown in [Figure 11-51](#), and select green color from the color palette. The background color, of the stack frame from library code, changes to green in the Stack Pane and Class Pane, as shown in [Figure 11-52](#) and [Figure 11-53](#).

Figure 11-51 Setting the Background Color for the Stack Frame in the Stack Pane

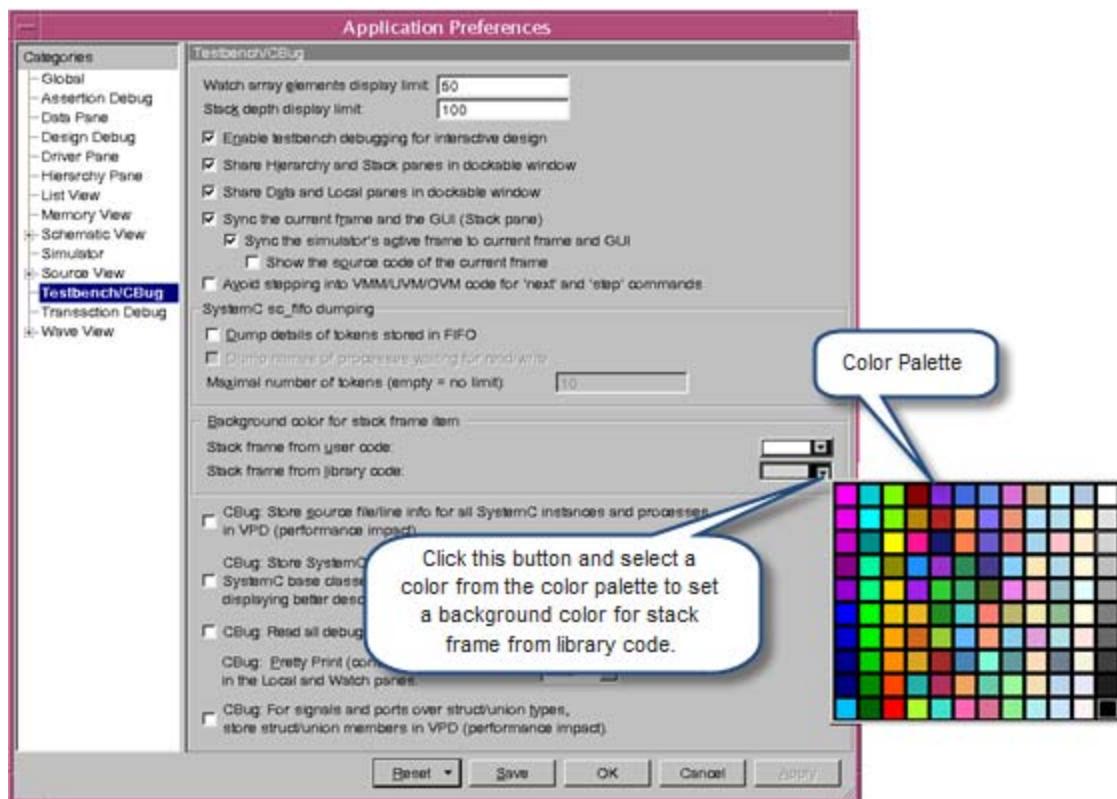


Figure 11-52 The Background Color of the Stack Frame in the Stack Pane

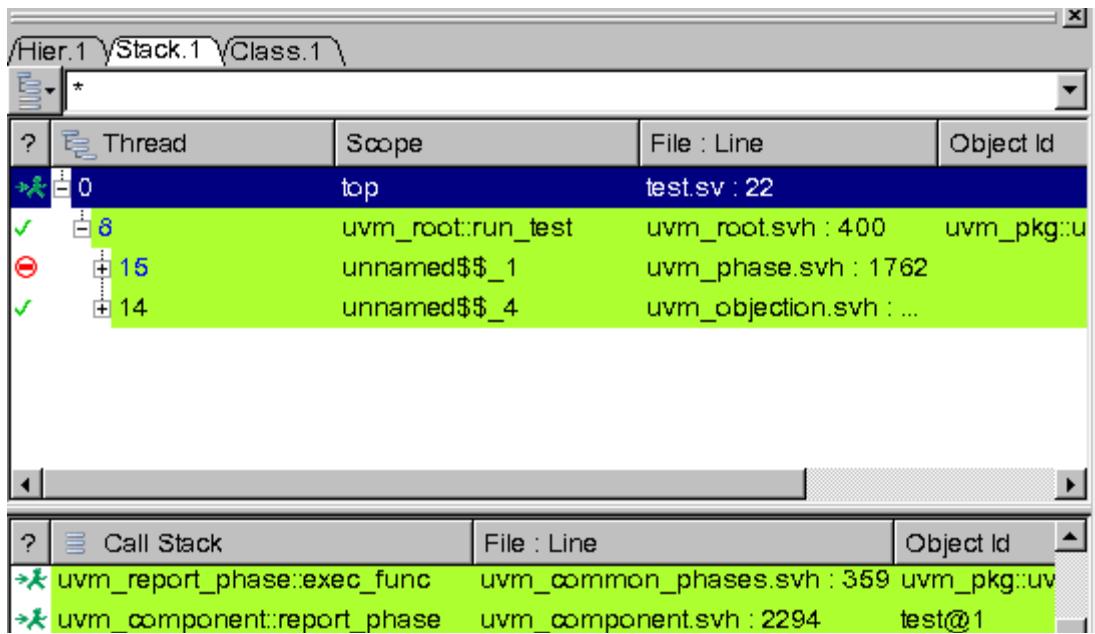
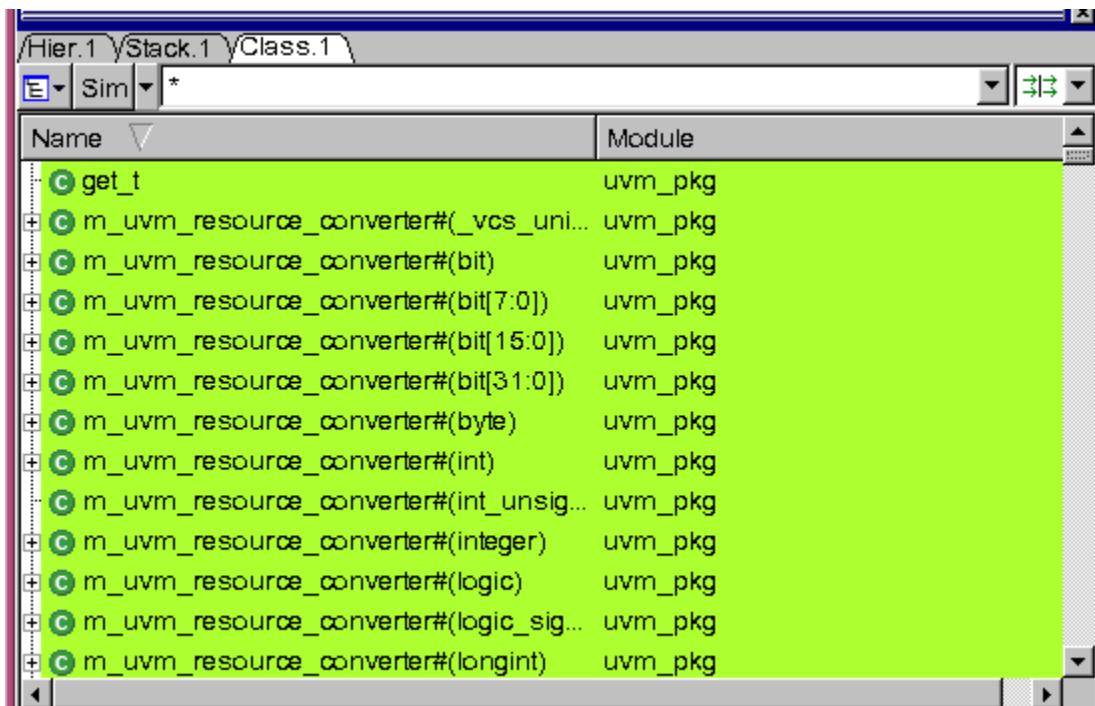


Figure 11-53 The Background Color of the Stack Frame in the Class Pane



Debugging UVM Testbench Designs

The Universal Verification Methodology (UVM) is a methodology for functional verification using the SystemVerilog base class library. It allows you to construct class-based verification environments using verification component objects. It also allows you to create stimulus using sequence objects.

Since UVM defines class-based testbenches, you need a debug environment that allows you to view the entire UVM class structure, including class inheritance relationships and the instance trees. This provides you a complete information of your verification environment, which helps you to understand the UVM architecture and to make the debug of UVM designs easier.

DVE supports the debugging of UVM testbench designs and allows you to do the following:

- View all available configurations in your design
- View the set/get history of a configuration item
- View all the predefined phases of common and UVM domain
- Set breakpoints on the important phases or on the phase methods of `uvm_component`.
- View runtime arguments using the Simulation Arguments dialog box
- Filter UVM object items in the Watch Pane

UVM Testbench Design Debug Example

Consider the following test case:

Example 11-6 Design File With Thread Debug (test.sv)

```
program top;

`include "uvm_macros.svh"
import uvm_pkg::*;

class test extends uvm_test;
    `uvm_component_utils(test)

    function new(string name, uvm_component parent = null);
        super.new(name, parent);
    endfunction

    virtual function void report();
        $write("## UVM TEST PASSED ##\n");
    endfunction
endclass

initial
begin
    run_test();
end

endprogram
```

Compile the `test.sv` code shown in [Example 11-6](#) as follows:

```
% vcs -sverilog -ntb_opts uvm -debug_all test.sv
```

Invoke the DVE GUI using the following command:

```
% simv +UVM_TESTNAME=test -gui
```

UVM Resource Browser

The UVM Resource Browser is a configuration interface which displays all available configurations/resources in your design. It allows you to exchange information across different components to configure topology, mode of operation, and runtime parameters.

The classes derived from `uvm_component` can use the `uvm_resource_db` methods or the `set_config_int`, `set_config_string`, or `set_config_object` methods to store this information. Other components can use `uvm_resource_db` methods or the `get_config_int`, `get_config_string`, or `get_config_object` methods to get this information.

Viewing the UVM Resource Browser

You can view the UVM Resource Browser in the **Resource** tab of the UVM Debug Pane. It contains Resource View and Resource History View, as shown in [Figure 11-55](#). This view updates when the simulation stops.

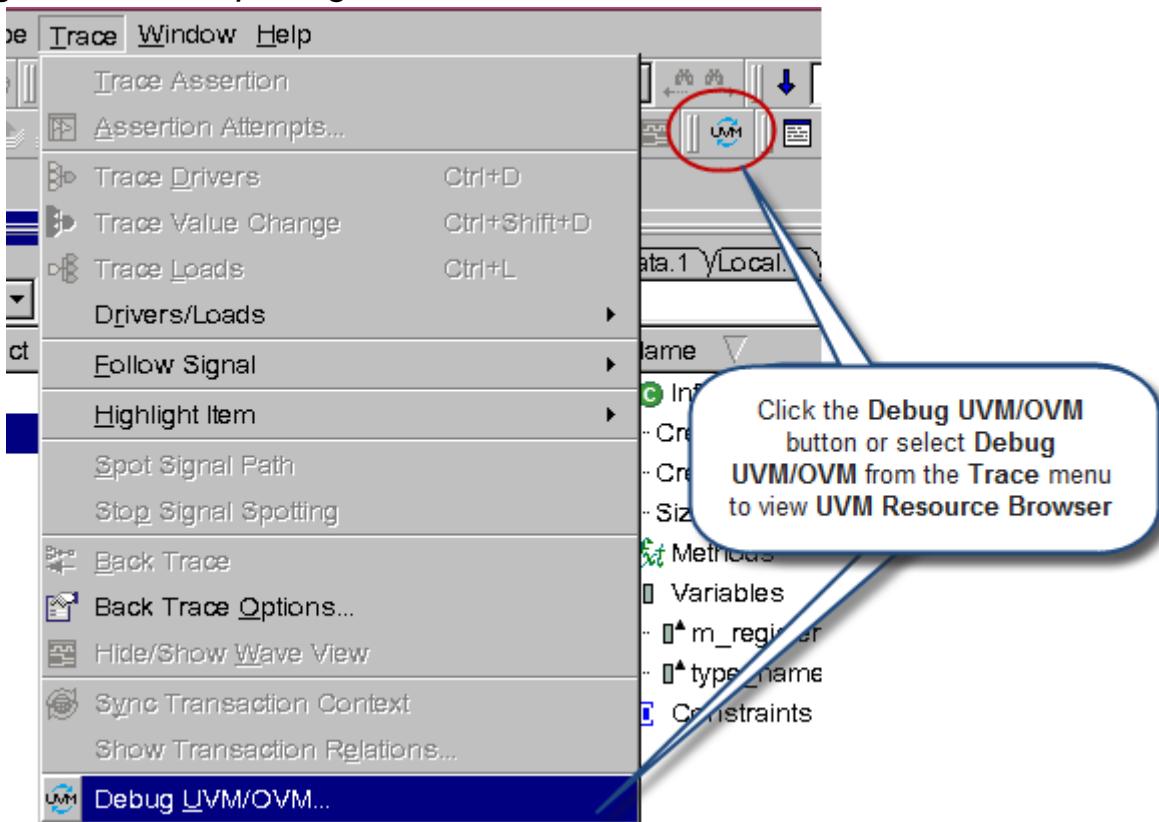
To view the UVM Resource Browser, perform the following steps:

Click the **Debug UVM/OVM**  button.

Or

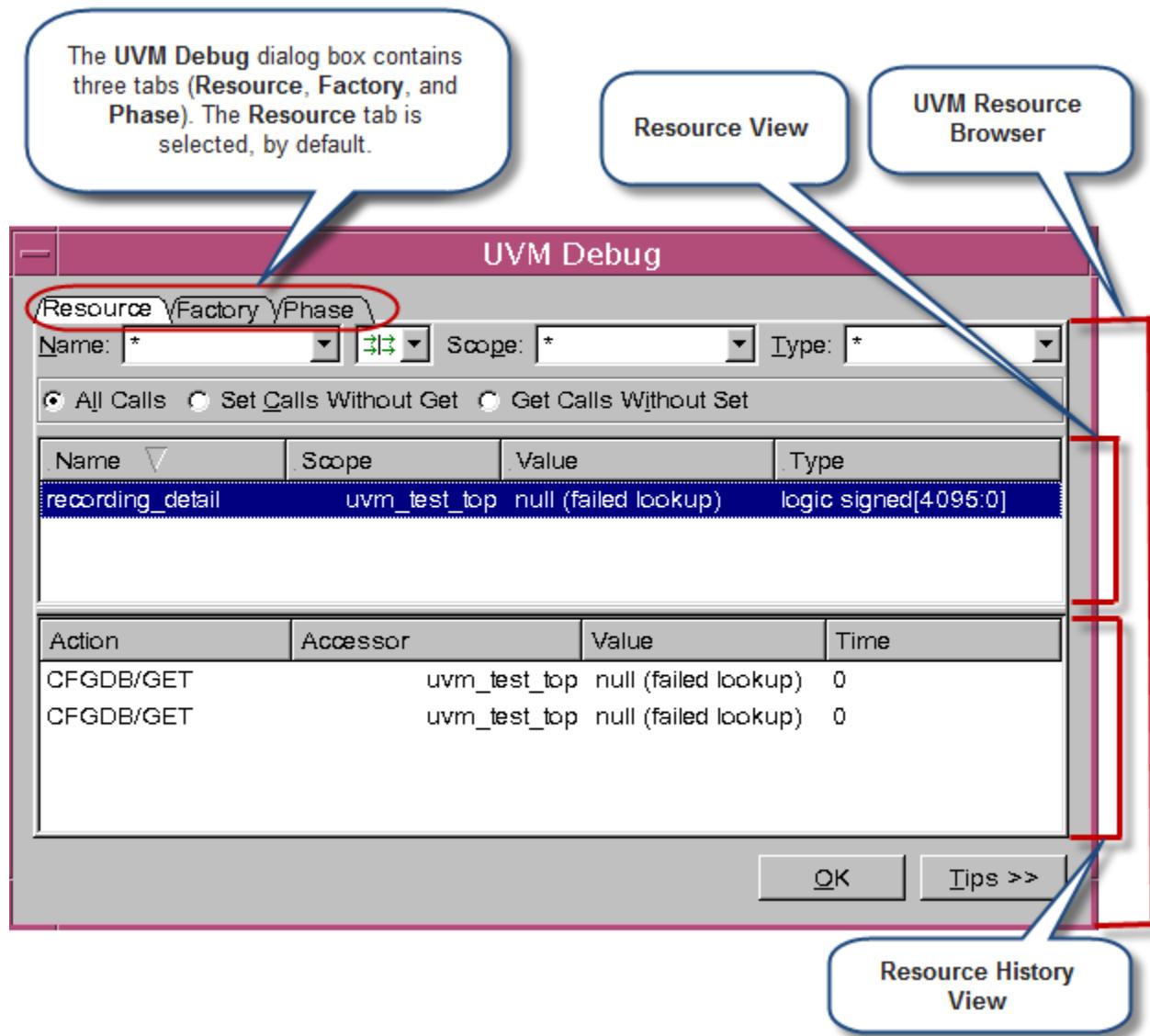
Select **Debug UVM/OVM** from the **Trace** menu, as shown in [Figure 11-54](#).

Figure 11-54 Opening the UVM Resource Browser



The UVM Debug dialog box appears, as shown in Figure 11-55. This dialog box displays the **Resource** tab which contains the **UVM Resource Browser**.

Figure 11-55 Viewing the UVM Resource Browser



Using the Resource View

The **Resource View** displays all available configurations in your design. This view contains four columns, namely, Name, Scope, Value, and Type, as shown in [Figure 11-56](#), and displays names and values of function arguments defined in your design. This view

allows you to sort the data in each column. [Table 11-14](#) describes the columns of the **Resource View**. You can select only one item at a time.

Figure 11-56 Resource View

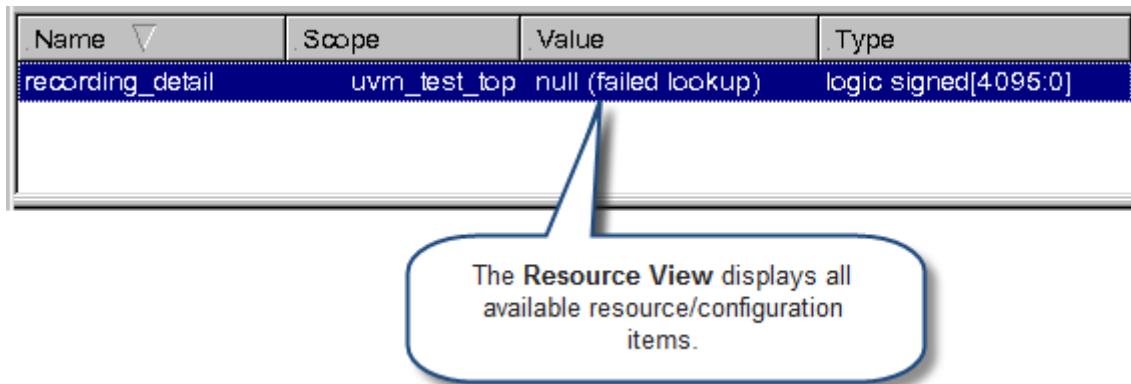


Table 11-14 Resource View Columns

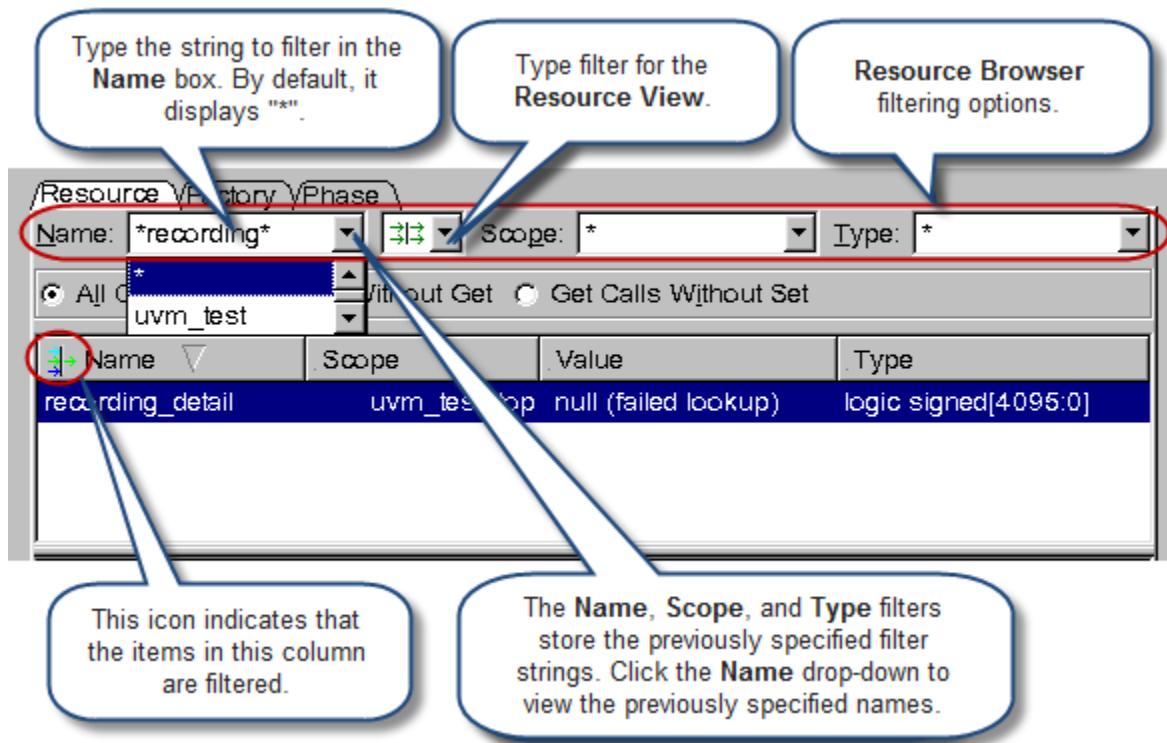
Column name	Description
Name	Displays the following function arguments: <ul style="list-style-type: none"> • The <code>name</code> argument of the <code>set</code> function of <code>uvm_resource_db</code> • “The <code>field_name</code> argument of <code>set_config_*</code> and <code>set_resource_*</code> functions of <code>uvm_component</code>
Scope	Displays the <code>scope</code> argument of a function
Value	Displays the <code>value</code> or <code>val</code> argument of a function.
Type	Displays the argument type. The type can be string, integer, bit, <code>bit[0:0]</code> , <code>bit[1:1]</code> , enum, object, virtual interface, or array. The string of scope is taken from the member of <code>uvm_resource_db</code> . It is a regular expression without the leading <code>/^</code> and ending <code>\$/</code> .

Filtering Configurations in the Resource View

Filtering options in the **UVM Resource Browser** (see [Figure 11-57](#)) allow you to configure the type of information to display for **Name**, **Scope**, and **Type** columns in the **Resource View**. [Table 11-15](#)

describes the filtering options in the **UVM Resource Browser**. You can use one or more filters to filter configurations in the **Resource View**.

Figure 11-57 Resource Browser Filtering Options



These filters allow you to specify the text to filter and to store the previously specified filter strings. By default, these filters use '*' wildcard character as the filter string. You can change this default setting to a simple string or regular expression using the **Syntax** drop-down in the **Global** category of the Application Preferences dialog box.

Table 11-15 Filtering Options in the UVM Resource Browser

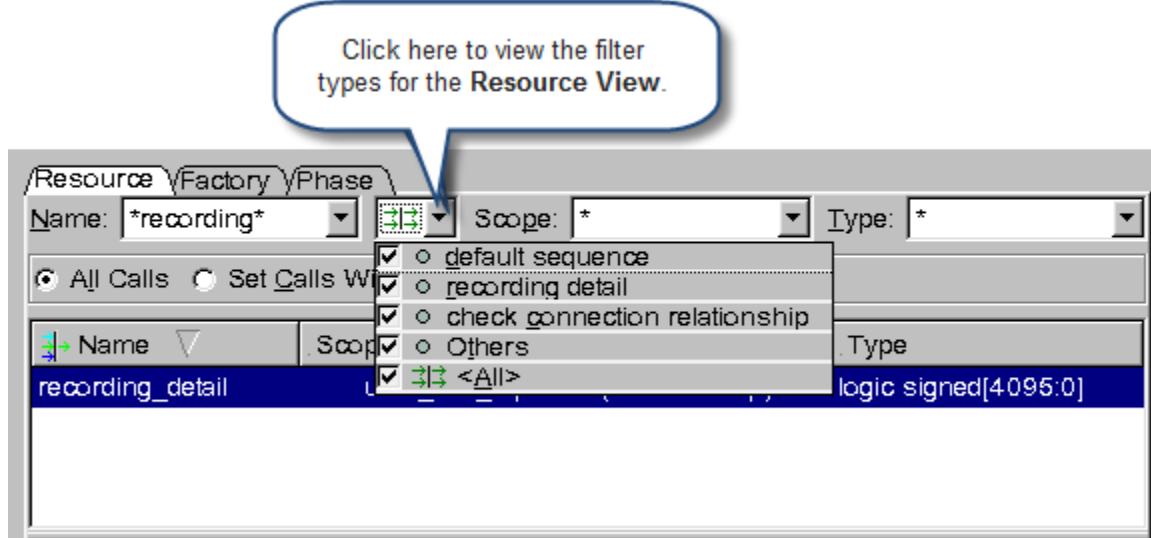
Filter name	Description
Name	Allows you to view the desired function arguments in the Name column.
Scope	Specifies the hierarchical reference path of an object to view only the resource/configuration whose scope pattern matches the specified path. You can select an object from the Object Hierarchy Browser and use Show Resource right-click option to specify it in this filter. You can reset other filters to show only the resource/configuration for this object.
Type	Allows you to view the desired argument type in the Type column. You can specify one of the following types: *, Int, Bit, bit [0:0], bit [1:1], Enum, String, Object, Virtual interface, or Array.

Type Filter for the Resource View

You can click the Type Filter drop-down, as shown in [Figure 11-58](#), to select the filters based on which you can sort the configurations in the **Resource View**.

Type Filter in the **Resource View** allows the filter of `check_connection_relationship`, `default_sequence` and `recording_detail` configuration items. These configurations may contain large number of “(failed lookup)” items.

Figure 11-58 Resource View Type Filter



Using the Resource History View

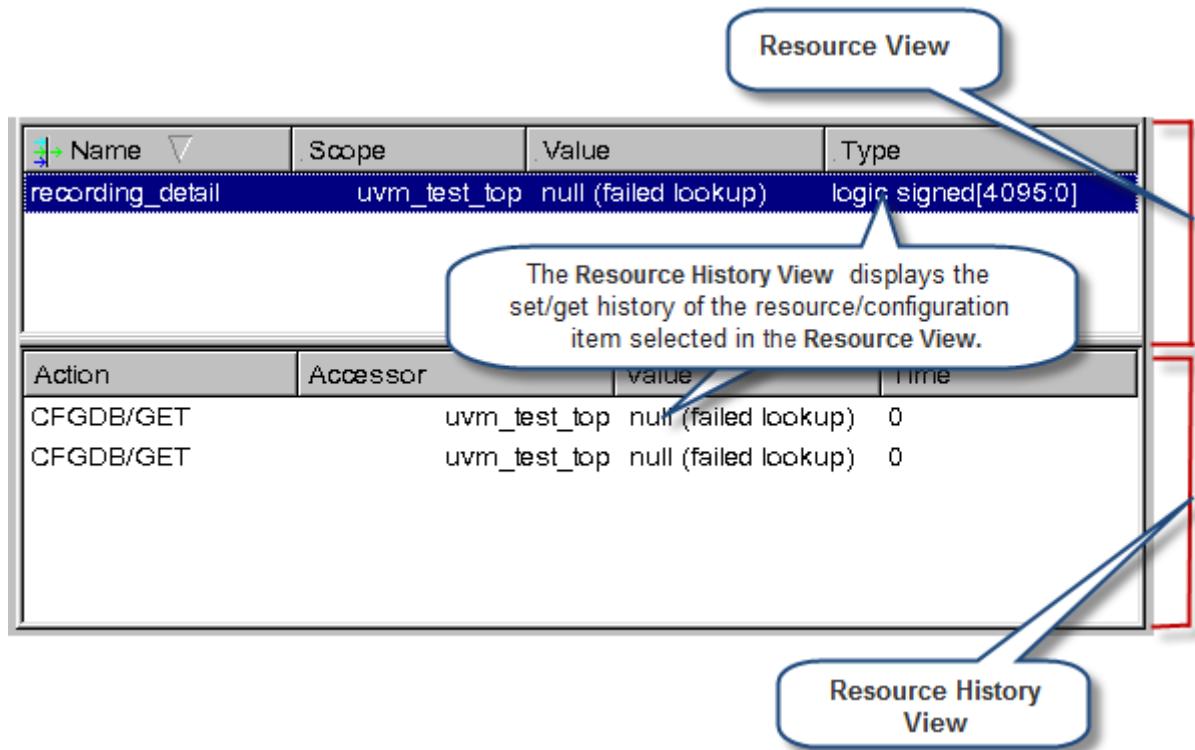
The **Resource History View** displays the set/get history of the resource/configuration item selected in the **Resource View**. This View contains four columns, as shown in [Figure 11-59](#), and you can sort the data in each column. You can double-click an item in the **Resource History View** to view the code from where the set/get function is called. You can select only one item at a time.

[Table 11-16](#) describes the columns in the **Resource View**.

Table 11-16 Resource History View Columns

Column Name	Description
Action	Displays the set/get history of the resource item selected in Resource View
Accessor	Displays the objects that set or get the resource/configuration information
Value	Displays the value or val argument of a function
Time	Displays the time at which action is performed

Figure 11-59 Resource History View



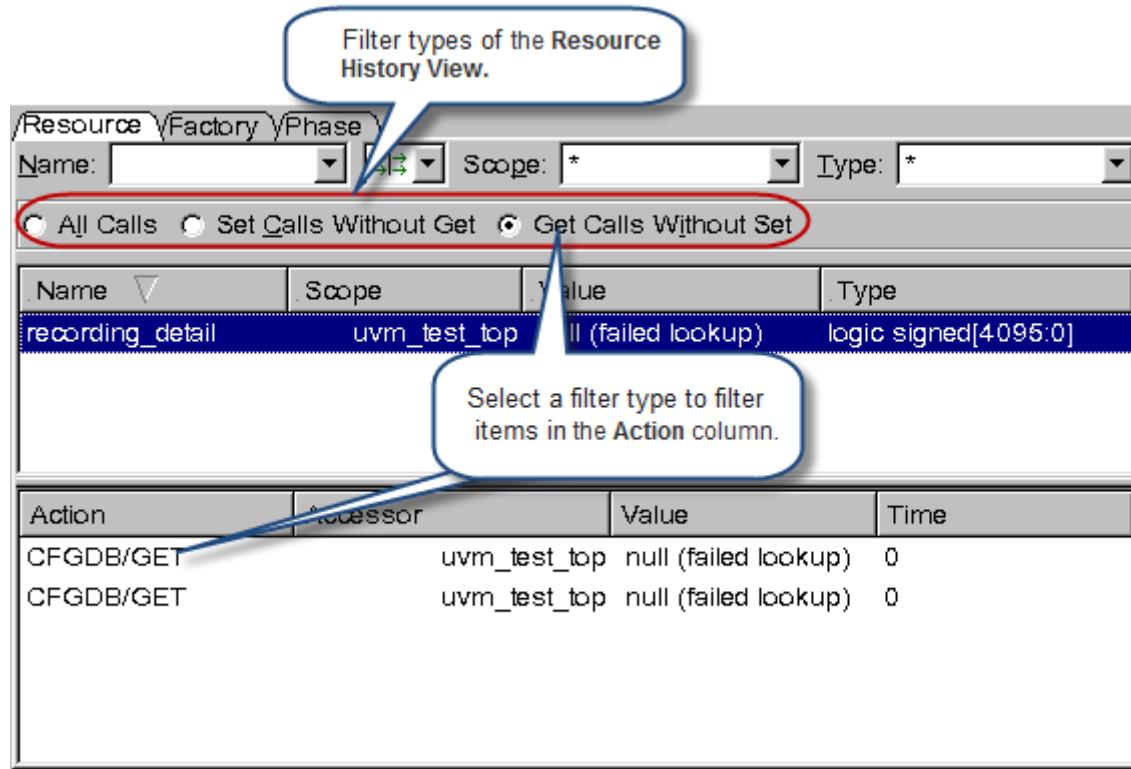
Filtering Action Items in the Resource History View

The Filter Types of the **UVM Resource Browser** (see [Figure 11-60](#)) allow you to configure the type of information to display for the **Action** column in the **Resource History View**. [Table 11-17](#) describes the filter types of the **Resource History View**.

[Table 11-17 Resource History View Filter Types](#)

Filter type	Description
All Calls	Displays the abnormal resource/configuration operations of the entire design. This filter type is selected, by default.
Set Calls Without Get	Displays the set calls of the resource/configuration selected in the Resource View .
Get Calls Without Set	Displays the get calls of the resource/configuration selected in the Resource View .

Figure 11-60 Filtering Action Items in the Resource History View



Right-click Menu Options in the Resource View

Table 11-18 describes the right-click menu options available in the **Resource View**.

Table 11-18 Right-click Menu Options in the Resource View

Option	Description
Show Interface Definition	Navigates you to the SV-interface with which the virtual interface is connected. This option is enabled only when the value type of the selected resource/configuration is virtual interface.
Set radix	Allows you to change the radix of the selected variable (Int, Bit, and Enum). Supported radices include: binary, octal, decimal, and hex. The enum values are shown in the <code>EnumString (value)</code> format. For example, <code>IDLE ('h0)</code> . This option is enabled only when the selected value type is one of the vector types.
Export Set/Get calls	Exports the resource/configuration into a text file.

Right-click Menu Options in the Resource History View

[Table 11-19](#) describes the right-click menu options available in the **Resource History View**.

Table 11-19 Right-click Menu Options in the Resource History View

Option	Description
Show Call in Source Window	Displays the source line of the selected call in the Source View.
Show Accessor in Object Browser	Displays the class object of the selected accessor in the Object Hierarchy Browser .
Show Accessor in Class Browser	Displays the class of the selected accessor in the Class Pane.

UVM Factory View

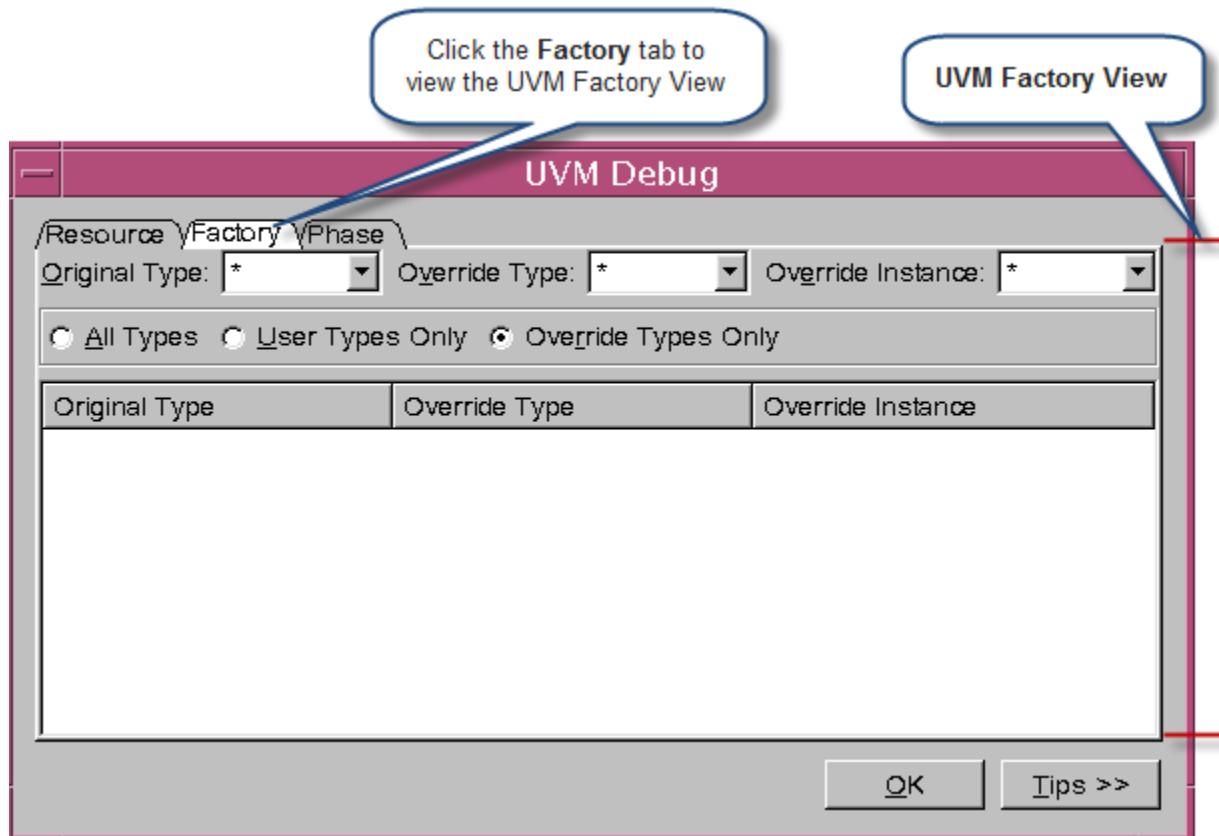
The **UVM Factory View** provides an interface to view the list of all classes or overridden classes that define the UVM factory facility. You can view the UVM Factory View in the **Factory** tab of the UVM

Debug Pane, as shown in [Figure 11-61](#). By default, this view displays only the overridden classes and updates for each simulation stop.

To view the UVM Factory View, perform the following steps:

1. Select **Panes > UVM** from the **Window** menu.
2. Click the **Factory** tab in the UVM Debug Pane, as shown in [Figure 11-61](#).

Figure 11-61 UVM Factory View



The **UVM Factory View** contains three columns, namely, **Original Type**, **Override Type**, and **Override Instance**, as shown in [Figure 11-61](#), and you can sort the data in each column. This view allows you to select only one item at a time.

Table 11-20 describes the columns in the **UVM Factory View**.

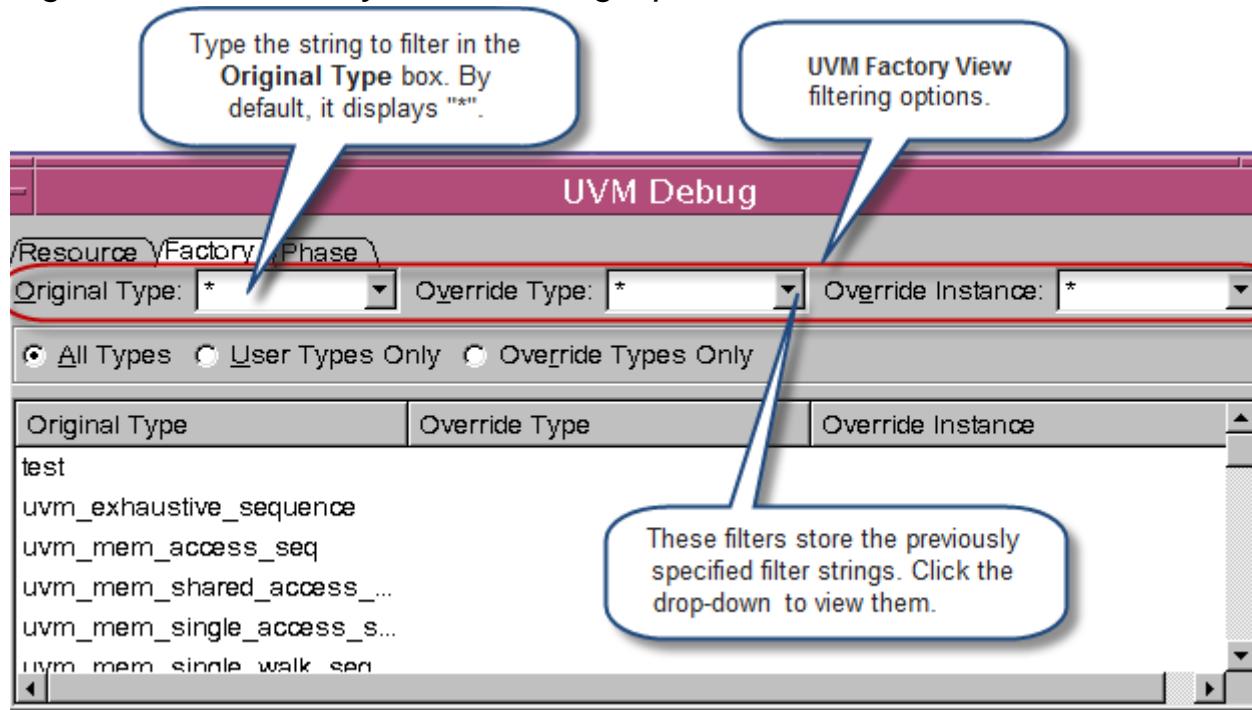
Table 11-20 UVM Factory View Columns

Column name	Description
Original Type	Displays the list of original types
Override Type	Displays the type of instance override
Override Instance	Displays instance overrides

Filtering Types and Instances in the UVM Factory View

Filtering options in the **UVM Factory View** (see [Figure 11-62](#)) allow you to configure the type of information to display for Original Type, Override Type, and Override Instance columns.

Figure 11-62 Factory View Filtering Options



These filters allow you to specify the text to filter and stores the previously specified filter strings. By default, these filters use '*' wildcard character as the filter string. You can change this default setting to a simple string or regular expression by using the **Syntax** drop-down in the **Global** category of the Application Preferences dialog box.

Filtering Items in the UVM Factory View

[Table 11-21](#) describes the filter types of the **UVM Factory View**.

Table 11-21 UVM Factory View Filter Types

Filter type	Description
All Types	Displays the list of all classes
User Types Only	Displays the user-defined classes
Override Types Only	Displays the overridden classes

Right-click Menu Options in UVM Factory View

[Table 11-22](#) describes the right-click menu options available in the **UVM Factory View**.

Table 11-22 Right-click Menu Options in the UVM Resource Browser

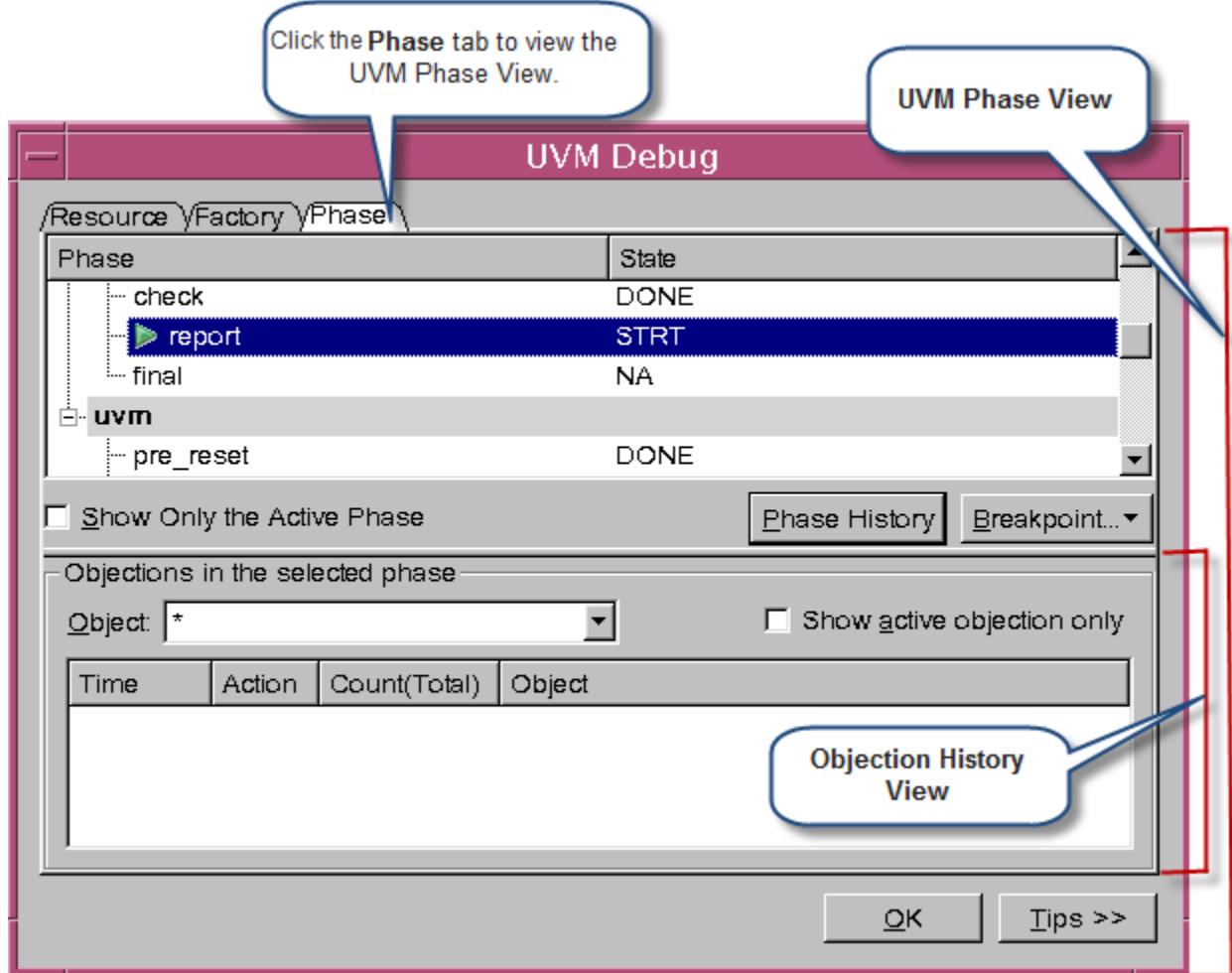
Option	Description	
Show original class in Class Browser		Displays the original class of the selected class in the Class Pane
Show override class in Class Browser		Displays the overridden class of the selected class in the Class Pane
Show Source for	Original Class	Displays the source line of the selected original class in Source View
	Override Class	Displays the source line of the selected override class in Source View
	Override Instance (Enabled only for instance override)	Displays the source line of the selected override instance in Source View

UVM Phase View

UVM provides the objection mechanism to control the phases of simulation. This mechanism raises or drops works for each testbench component until the simulation is complete. The UVM component raises an objection when it is busy doing work and drops its objection when it is idle. [Figure 11-63](#) shows the **UVM Phase View** in the **UVM Debug Pane**. The **UVM Phase View** displays all the predefined phases of common and UVM domain.

Click the **Show only the active phase** check box to view only the executing phases.

Figure 11-63 UVM Phase View



The **State** column displays the status of a phase. The status of a phase can be as follows:

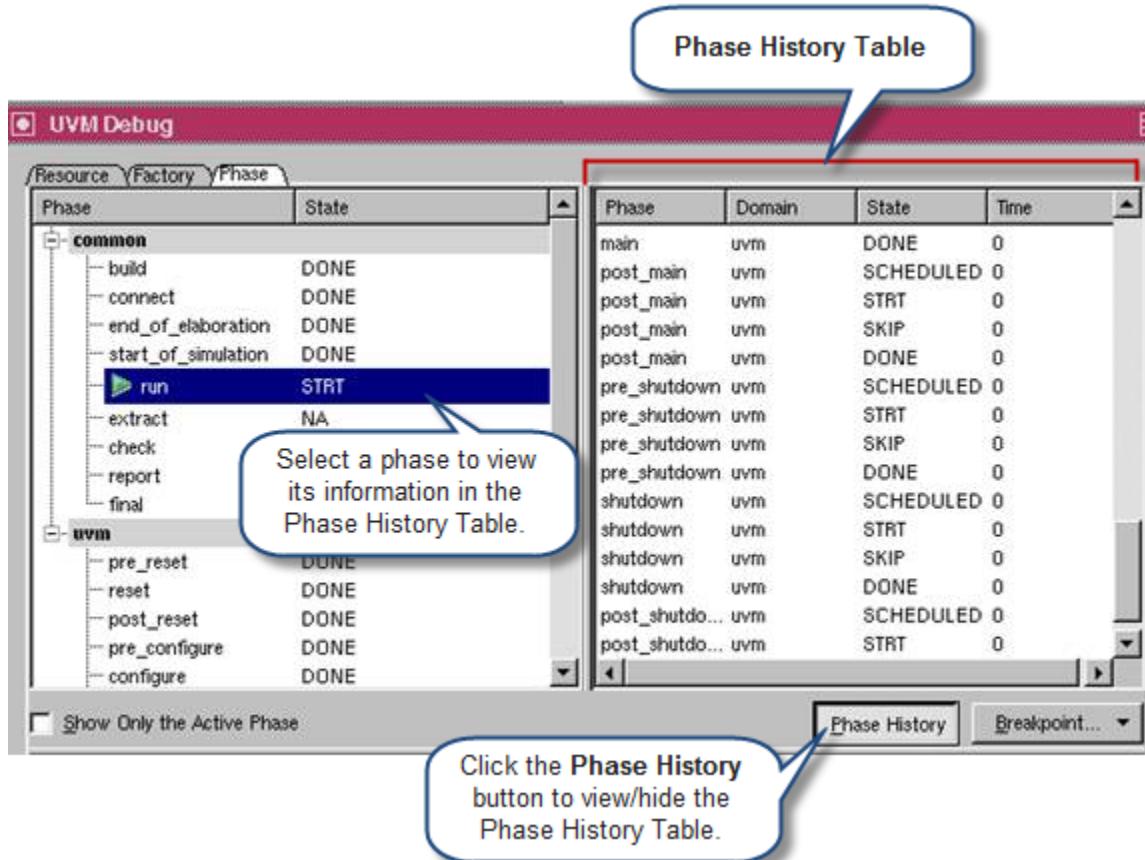
NA — Not started

STRT — Executing

DONE — Completed

You can use the **Phase History** button to view the information of the selected phase in the Phase History Table, as shown in [Figure 11-64](#).

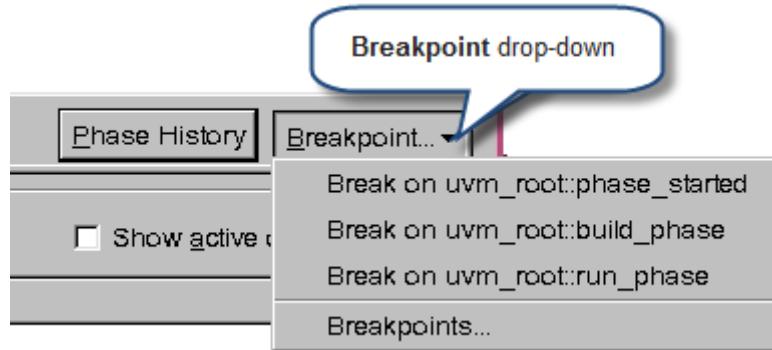
Figure 11-64 Viewing Phase History Table



You can right-click on a phase, select **Set Component Breakpoint** to open the Breakpoints dialog box, and create a breakpoint on the corresponding phase function of `uvm_component` (for example, `uvm_component :: run_phase`). You can specify condition or object options for this component breakpoint.

You can use the **Breakpoint** drop-down, as shown in [Figure 11-65](#), to view the options available to set breakpoints.

Figure 11-65 Options to Set Breakpoints



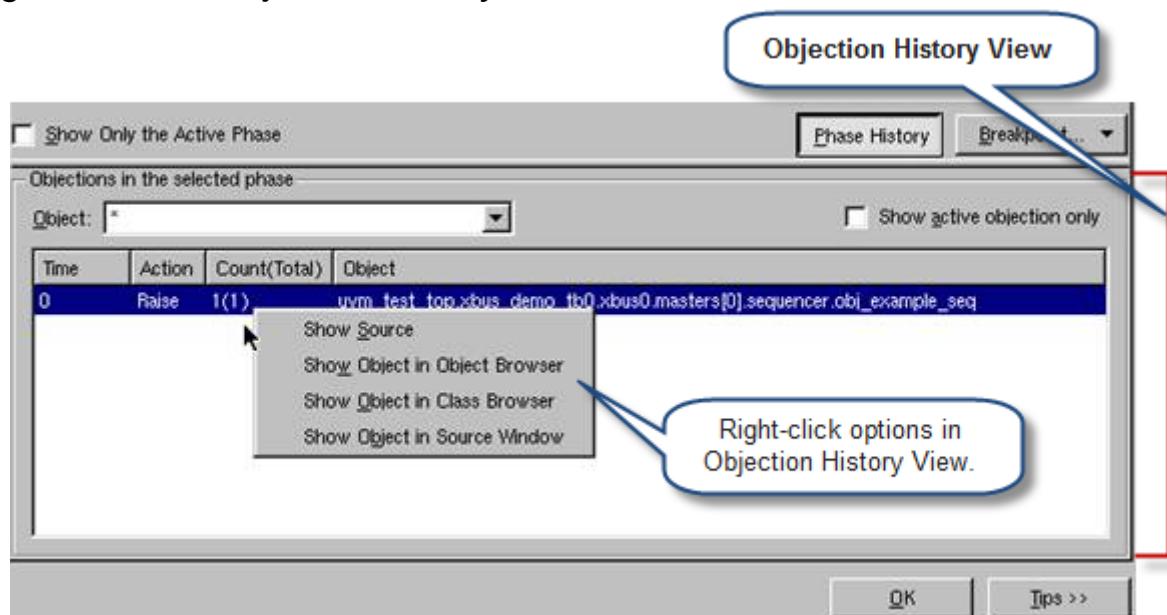
[Table 11-23](#) describes the breakpoint options.

Table 11-23 Breakpoint Options

Breakpoint Option	Description
Break on uvm_root:phase_started	Opens the Breakpoints dialog box and displays Break in the task/function field as uvm_root::phase_started
Break on uvm_root:build_phase	Stops the execution in the uvm_root::build_phase
Break on uvm_root:run_phase	Stops the execution in the uvm_root::run_phase
Breakpoints	Opens the Breakpoints dialog box and displays the breakpoint list

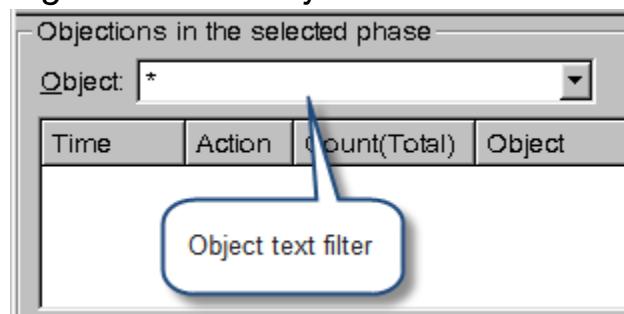
The objections in the selected phase (`uvm_phase::phase_done`) is displayed in the **Objection History View**, as shown in [Figure 11-66](#). For each objection action, its time, type, source object's path, and its object ID are displayed. After the action, the total number of remaining objection count is listed in the **Count(Total)** column.

Figure 11-66 Objection History View



You can use the object text filter to filter the objection list by the object reference path, as shown in Figure 11-67. Click the **Show active objection only** check box to filter the raised objection items that are dropped.

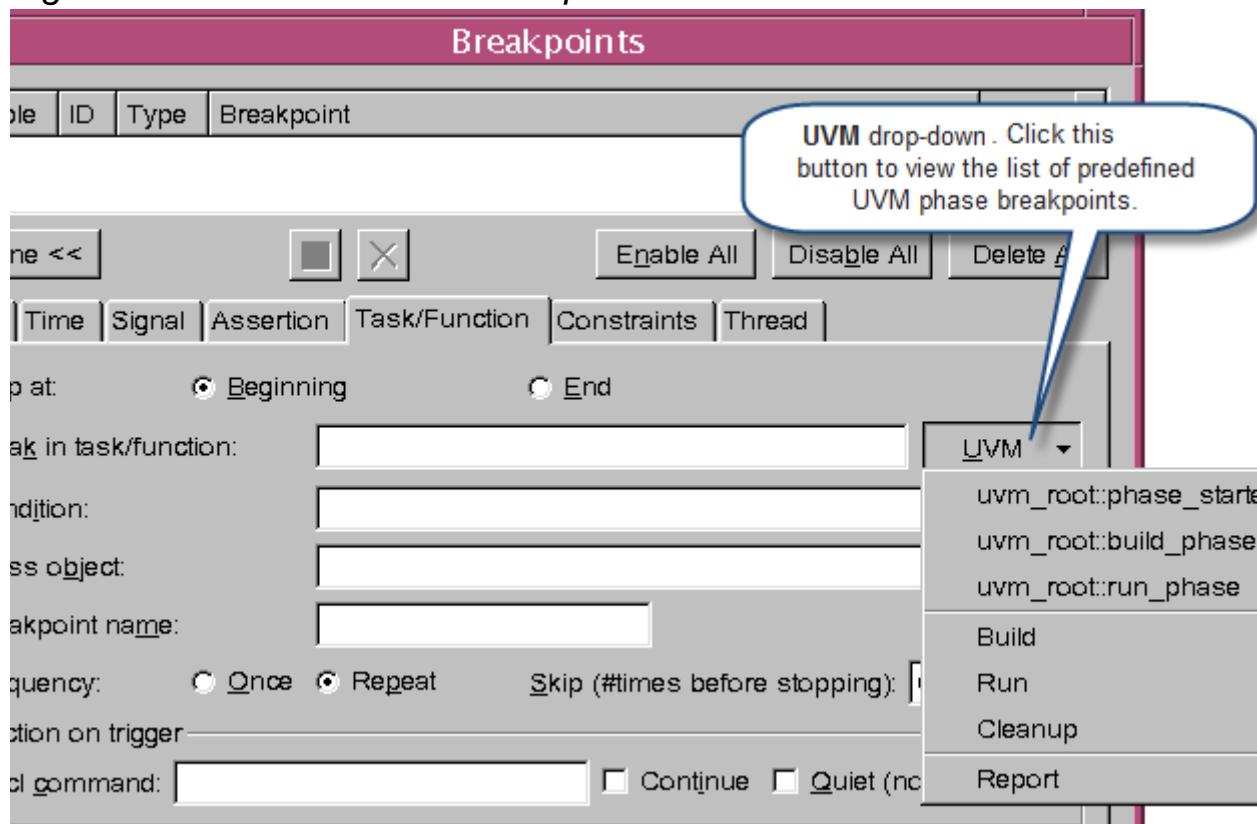
Figure 11-67 Object Text Filter



UVM Phase Breakpoints

You can set breakpoints on the important phases or on the phase methods of `uvm_component` using the **UVM** drop-down in the **Task/Function** tab of the Breakpoints dialog box, as shown in [Figure 11-68](#). Click this drop-down button to view the list of predefined phase breakpoints. Select a phase method from the list, to add it to the **Break in task/function** field.

Figure 11-68 UVM Phase Breakpoints

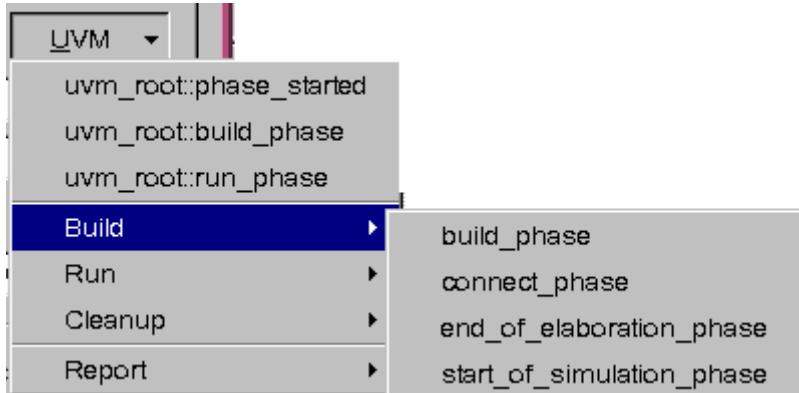


As shown in [Figure 11-68](#), the UVM component breakpoints are divided into three phases, namely, **Build**, **Run**, and **Cleanup**. Each category contains a list of phase methods that can be used as a breakpoint.

The **Build** phase, as shown in [Figure 11-69](#), contains the following phase methods:

`build_phase`, `connect_phase`,
`end_of_elaboration_phase`, and `start_of_sim_phase`

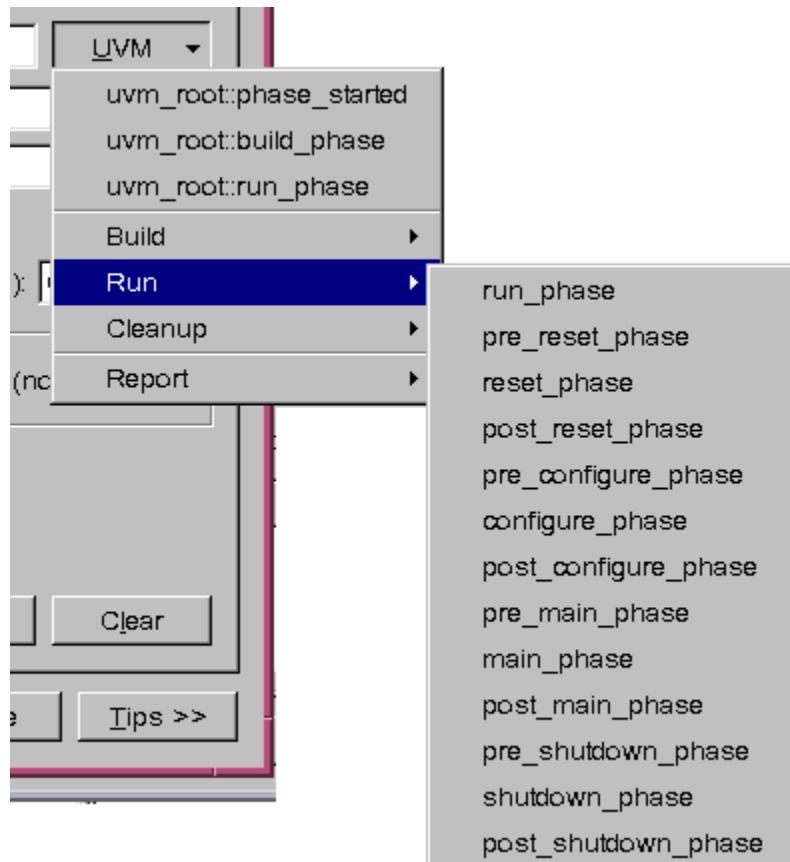
Figure 11-69 Phase Methods in the Build Phase



The **Run** phase, as shown in [Figure 11-70](#), contains the following phase methods:

`run_phase`, `pre_reset_phase`, `reset_phase`,
`post_reset_phase`, `pre_configure_phase`,
`configure_phase`, `post_configure_phase`,
`pre_main_phase`, `main_phase`, `post_main_phase`,
`pre_shutdown_phase`, `shutdown_phase`, and
`post_shutdown_phase`

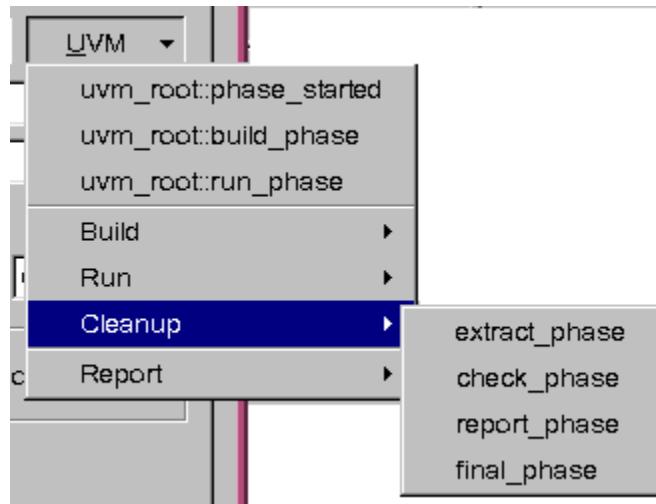
Figure 11-70 Phase Methods in Run Phase



The **Cleanup** phase, as shown in [Figure 11-71](#), contains the following phase methods:

`extract_phase`, `check_phase`, `report_phase`, and
`final_phase`

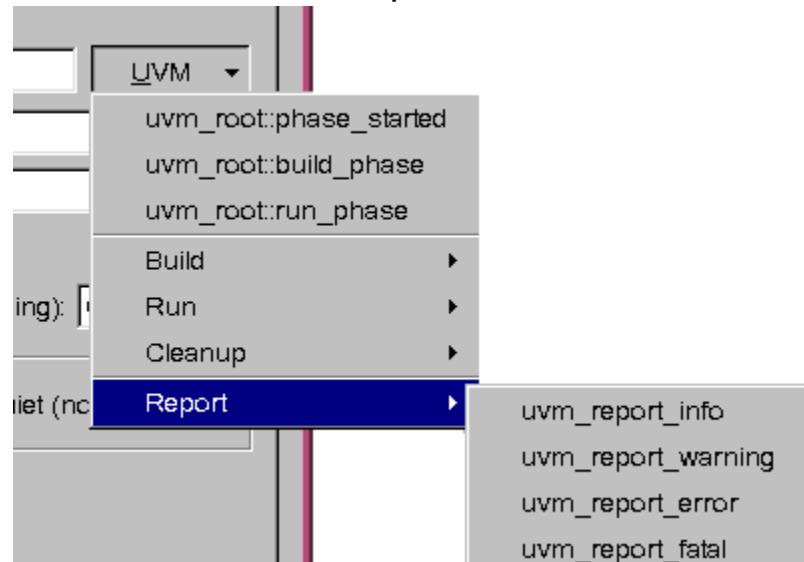
Figure 11-71 Phase Methods in Cleanup Phase



The **Report** phase, as shown in [Figure 11-72](#), contains the following phase methods:

uvm_report_info, uvm_report_warning,
uvm_report_error, and uvm_report_fatal

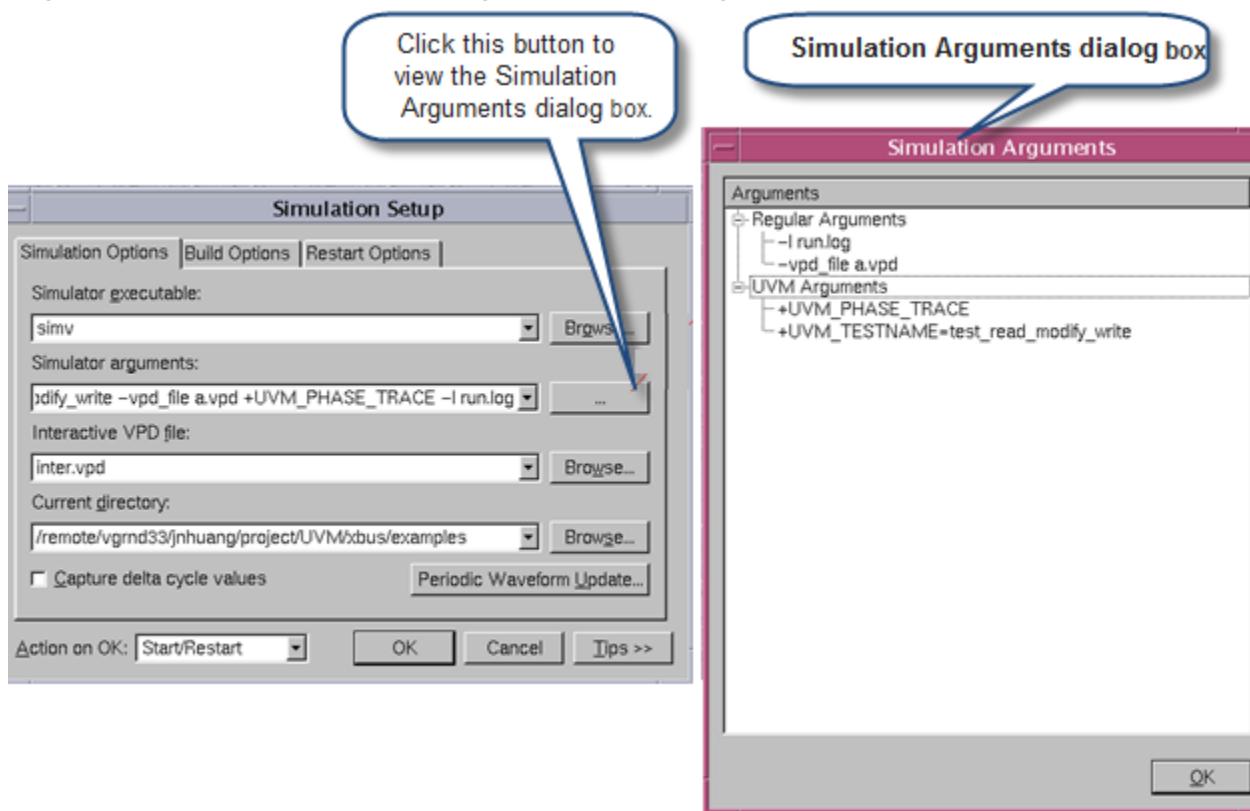
Figure 11-72 Phase Methods in Report Phase



Simulation Arguments Dialog Box

UVM provides the `uvm cmdline processor` class, which helps to parse the simulation runtime options. DVE displays this information in the Simulation Arguments dialog box. You can open this dialog box from the DVE Simulation Setup dialog box, as shown in [Figure 11-73](#).

Figure 11-73 Simulation Arguments Dialog Box



The Simulation Arguments dialog box displays information in a list view, and contains the following two arguments:

- Regular Arguments
- UVM Arguments

The **UVM Arguments** tree includes all the options starting with +UVM (case-insensitive).

The `-f run.f` option contains other simulation options and it can be expanded to view options in it. Same is the case with the `-i` / `-do` options. The `tcl` script content can be expanded to view options in it.

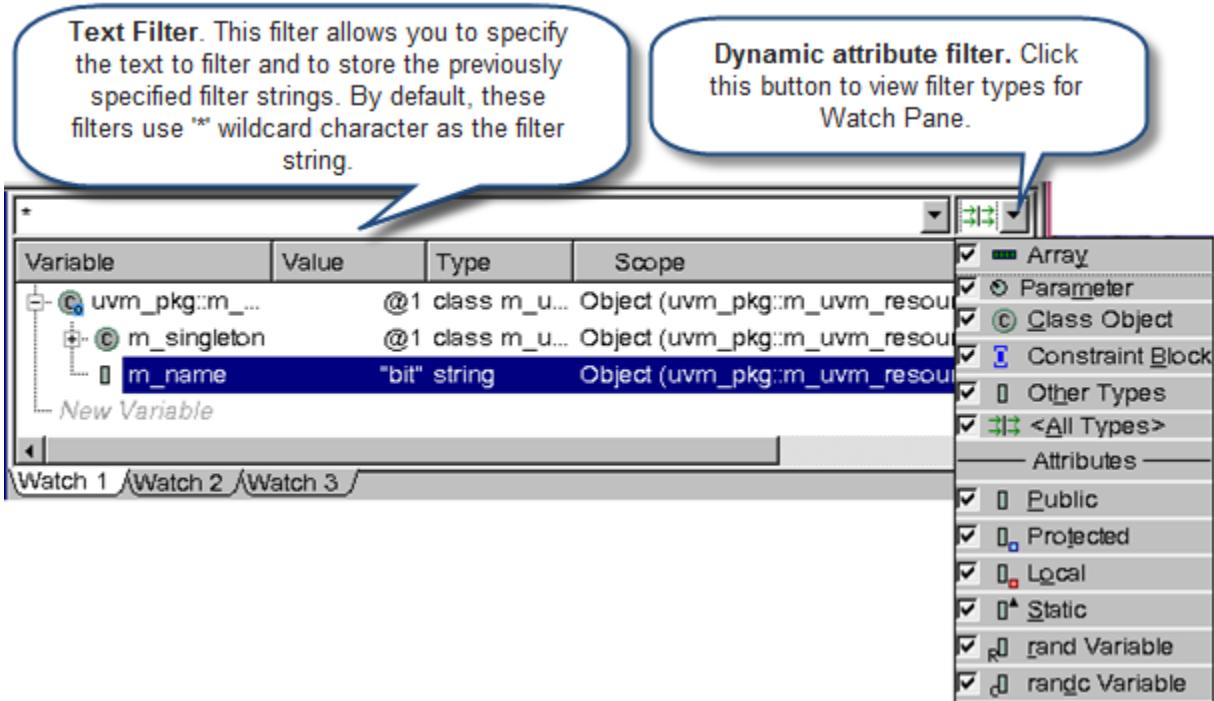
The Simulation Arguments dialog box information is read-only. It updates to the latest `simv` options whenever the dialog box is opened or a different design is selected using the **Designator** box.

Filtering Variables in the Watch Pane

DVE allows you to filter UVM object items in the Watch Pane using the text filter and **Dynamic attribute filter** type filter, as shown in [Figure 11-74](#). Each Watch Pane contains its own text filter and **Dynamic attribute filter** type filter.

These filters work on the visible leaf items. That is, if any leaf item is visible, its parent items will also be displayed (even if they do not match the filters).

Figure 11-74 Watch Pane Filters



12

Debugging Transactions

This chapter contains the following sections:

- “Introduction”
- “Transaction Debug”
- “Transaction Debug in SystemC Designs”
- “Viewing NTB-OV Variables using tblog/msglog”

Introduction

Productive system-level debug necessitates keeping a history of the system evolution that covers the varied modeling abstraction and encapsulation constructs used in both the design and testbench. Moreover, given the mix of abstraction layers and the wealth of data

sources in modern SoC design with IP reuse including user-added messaging, a flexible recording mechanism with an easy to control use-model and sampling mechanism is required.

To address these needs, VCS provides a pair of system tasks `$vcdplustblog` and `$vcdplusmsglog` which is to be called from SystemVerilog. The tasks can be applied in many contexts to record data directly into the VPD file. Both the tasks are based on the transaction abstraction:

- `$vcdplustblog` is intended for design and testbench static and dynamic data recording. It is primarily suited for logging of testbench call frames and for creating dynamic data waveforms essential for post-process debug. `$vcdplustblog` forms the basis of transaction-based debug of dynamic data.
- `$vcdplusmsglog` on the other hand, is intended primarily for recording messages, notes, and most importantly transactions - definition, creation, and relationships on multiple streams. `$vcdplusmsglog` forms the basis of transaction modeling and debug.

Transaction Debug

Using `$vcdplustblog`

The `$vcdplustblog` system task is primarily used for debugging of dynamic data. The data could be variables in the design or testbench code. You can call `$vcdplustblog` in any task, function, or begin block in any static or automatic scope. This sampling level provides a high level of flexibility and is quite matched to testbench recording with the focused selective recording on the call boundaries.

The \$vcdblustblog task when called during simulation dumps the variables it is sampling into the default VPD file (default name in interactive is inter.vpd and in post-process is vcdplus.vpd). Variables to record can be passed as arguments. In addition, \$vcdblustblog can pick up the static and automatic variables in the current frame, and any number of base class variable levels if called in a method, without enumeration. The frame implies the scope and time where the system task was called. Hence, the frame depends on not only the scope but also the time. Multiple \$vcdblustblog task calls at the same timestamp are allowed.

In interactive mode, a \$vcdblustblog call can be invoked with the call command from UCLI or DVE prompt without requiring any code changes. In such usage, \$vcdblustblog uses the UCLI active scope as its reference scope. A sample call follows:

```
ucli% call {$vcdblustblog("Hello World!")}  
Or  
dve> call {$vcdblustblog("Hello World!")}
```

Usage Model

```
$vcdblustblog([frame_class_var_level], [<string>[,<var>]^n])
```

Where,

<frame_class_var_level>

Specifies integer value as follows:

0 — (default). When you specify the integer zero, no frame variables are recorded.

1 — Records frame variables up to one level. If specified in a class method, \$vcdplustblog dumps the class data members.

2 — Records up to two levels, that is the data recorded when integer value 1 is specified + dump 1-level up of base data members if class is an extension.

N — Same as N-1 recording + dump (N-1)-level up of base data members if class is an extension

-1 — Records all the frame variables and all the class data up to the very base class, if class is an extension.

<string>

Specifies any multi-line text or HTML text passed as a string variable or literal.

<var>

Specifies one or more dynamic or static variable provided as an argument (relative to the current scope or with absolute path).

Example 1

The following example shows the usage of \$vcdplustblog in a class method to record data depending on the frame_recording_level passed to \$vcdplustblog. The illustration following the example shows how it is displayed in DVE. A more detailed description of what \$vcdplustblog records into the VPD file, and the DVE pane that is added (Transaction pane) is explained in later sections.

```
program test;
    class class1;
```

```

logic class1_logic;
bit class1_bit;
byte class1_byte;
task class1_fun (int x=0);
    logic class1_task_logic;
    byte class1_task_byte;
    bit class1_task_bit;
endtask
endclass
class class2 extends class1;
    logic class2_logic;
    bit class2_bit;
    byte class2_byte;
    task class2_fun (int x2=0);
        logic class2_task_logic;
        byte class2_task_byte;
        bit class2_task_bit;
    endtask
endclass
class class3 extends class2;
    logic class3_logic;
    bit class3_bit;
    byte class3_byte;
    task class3_fun (int x3=0);
        logic class3_task_logic;
        byte class3_task_byte;
        bit class3_task_bit;
    endtask
endclass
class class4 extends class3;
    logic class4_logic;
    bit class4_bit;
    byte class4_byte;
    task class4_fun (int x3=0);
        logic class4_task_logic;
        byte class4_task_byte;
        bit class4_task_bit;
    endtask
    $vcdplustblog(-1,"Minus-One"); // Case A
    $vcdplustblog(0,"Zero"); // Case B
    $vcdplustblog(1,"One" ); // Case C
    $vcdplustblog(2,"Two" ); // Case D

```

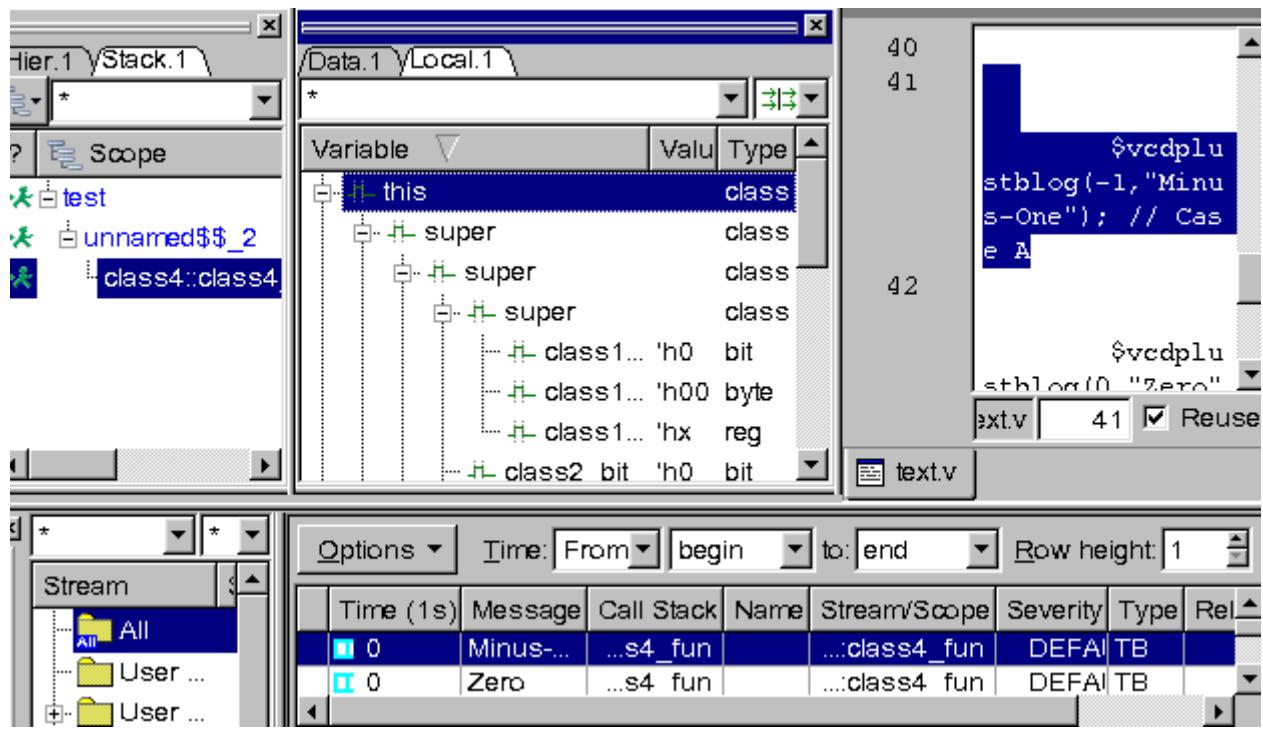
```

        $vcdplustblog(3,"Three");
        $vcdplustblog(4,"Four");
    endtask
endclass
class4 inst=new();
initial
begin
    inst.class4_fun();
    #12 $finish;
end
endprogram

```

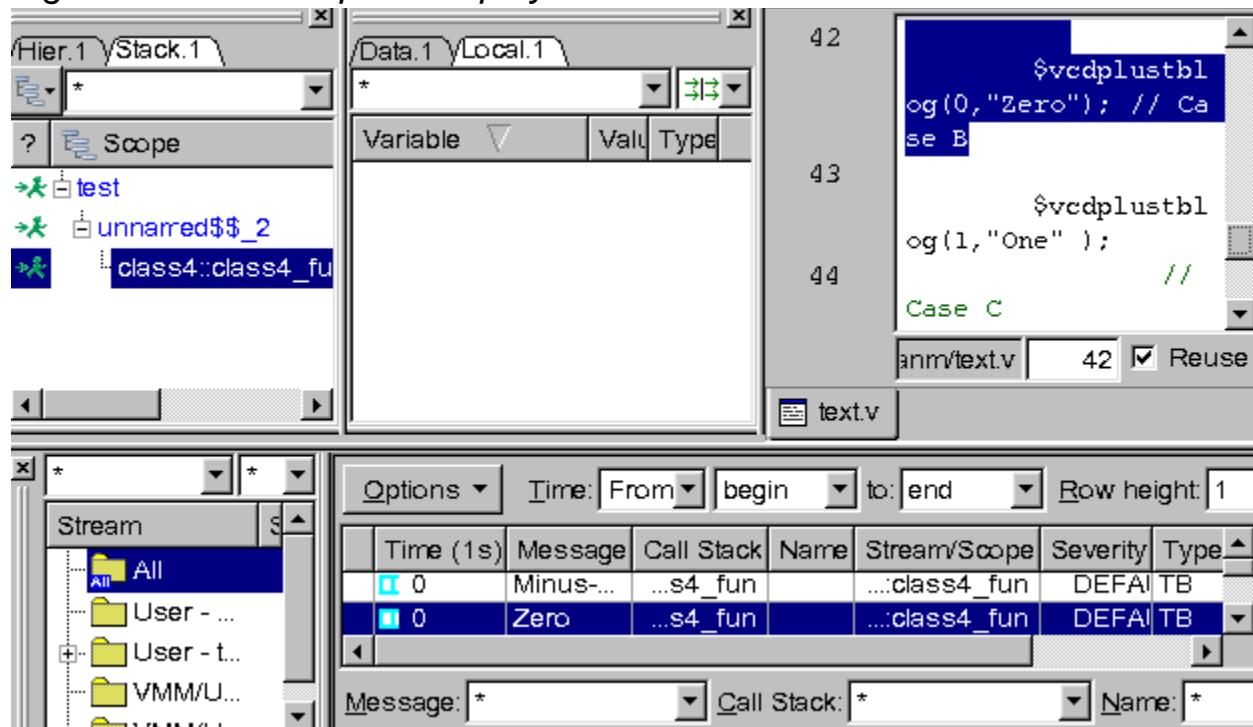
Case A: In the example, when you pass integer minus one (-1) as the frame_recording_level while calling \$vcdplustblog, all the base class data members are recorded as shown in the following figure for the resulting DVE display in the Local pane:

Figure 12-1 Local pane display in DVE



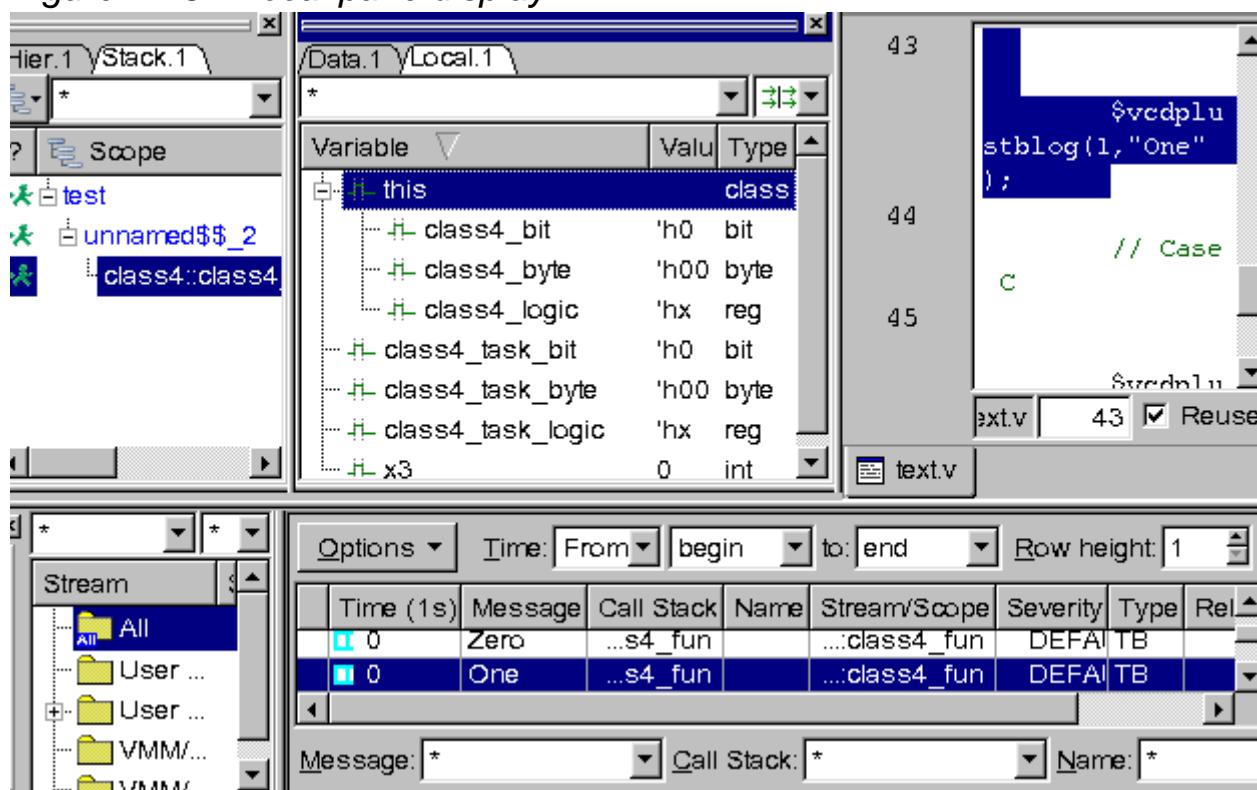
Case B: In the example, when you pass integer zero (0) as the frame_recording_level while calling \$vcndlplusblog, no variables are recorded for this call as shown in the following figure. Note that the Local pane is empty:

Figure 12-2 Local pane display in DVE



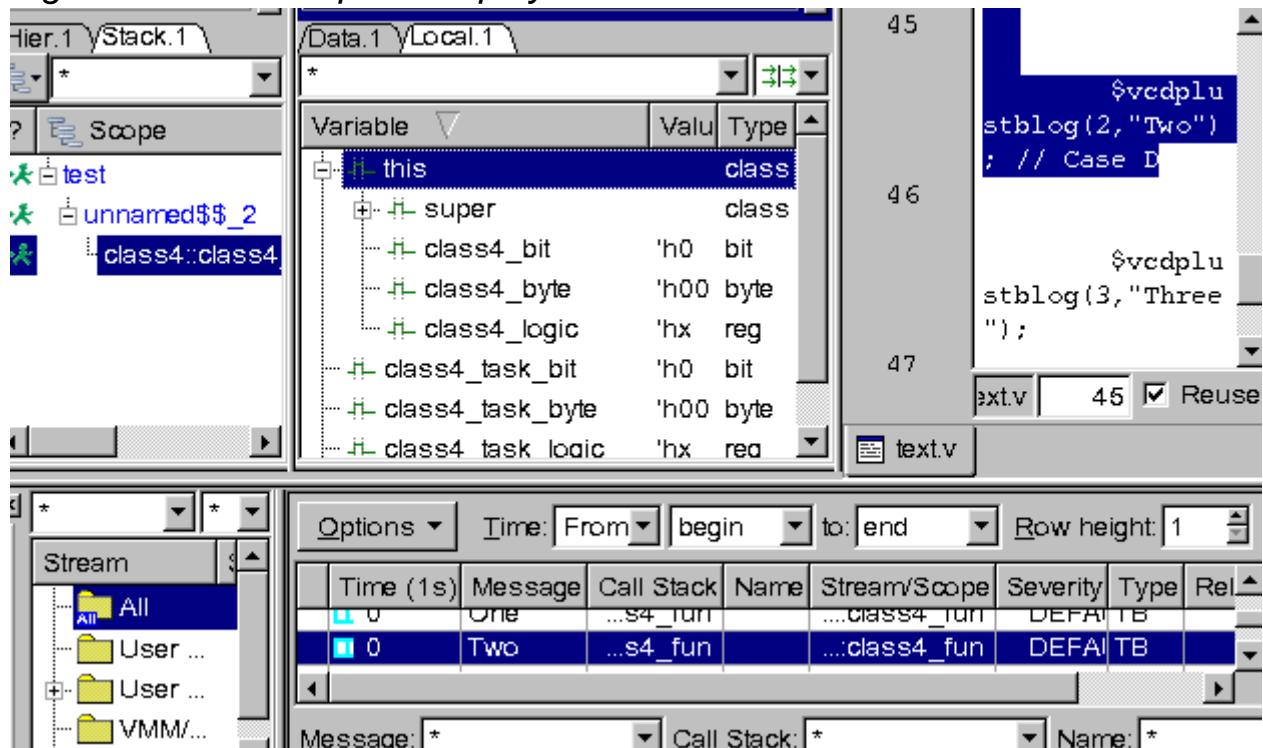
Case C: In the example, when you pass integer one (1) as the frame_recording_level while calling \$vcaplustblog, only the first level that is class 4 and its data members are recorded as shown in the following figure:

Figure 12-3 Local pane display in DVE



Case D: In the example, when you pass integer two (2) as the frame_recording_level, variables in the current extended class where \$vcdplustblog is invoked and the immediate base class (Class 3 and 4) are recorded as shown in the following figure:

Figure 12-4 Local pane display in DVE



VPD Recording

`$vcdplustblog` records the data into the VPD file, to be later read in by DVE. The following information is recorded per call:

- Optionally, frame local variables and class data members if in a method, as controlled by the `<frame_recording_level>`
- Message `<string>` if passed as a variable or literal
- Dynamic or static `<variables>` provided as arguments (name relative to current scope or with absolute path)

In addition, the following items, not passed explicitly as an argument are recorded per call:

- Time of the `$vcdplustblog` call

- Call stack of the `$vcaplustblog` call

`$vcaplustblog` Data Type Limitations

Except for container types, all the data types supported by the DVE Testbench GUI are supported by `$vcaplustblog` recording, when specified in the local frame or passed as arguments. Container types are not recorded as a whole, however individual element like a word or member can be recorded by `$vcaplustblog`. This limitation in container types is intended to avoid large data size.

Turning `$vcaplustblog` Task ON/OFF

You can globally turn the dumping of data on and off. This is useful to limit the dumping for certain time ranges or based on conditions. The following tasks are used for this purpose:

- `$vcaplustblogoff()` — Disables globally `$vcaplustblog` based dumping.
- `$vcaplustblogon()` — Enables globally `$vcaplustblog` based dumping.

Conditions can be specified in code surrounding these task. These tasks do not take any arguments.

Viewing `$vcaplustblog` objects in DVE

For every `$vcaplustblog` call, you can view the dynamic variables depending upon the level specified in the call. Stack pane displays the call stack and Local pane displays the dynamic variables and their values. The Wave view displays the values recorded in the multiple calls in a particular scope over time. The transaction details and messages are displayed in a new pane called the Transaction pane.

Transaction Pane

The Transaction pane displays all the \$vcdplustblog messages in a tabular format. It shows the time, scope of the \$vcdplustblog call, call stack, severity, type, and message. Each row in the Transaction pane corresponds to one \$vcdplustblog call. The type of a \$vcdplustblog call displayed in the transaction pane is TB, and the severity is DEFAULT.

To open the Transaction pane, click the Transaction pane toolbar button or click **Window > Panes > Transaction**.

You can perform the following tasks from the Transaction pane:

- Filter the view using each column — Click the respective filter text field to filter objects. For example, to filter by messages, enter the string in the Messages filter text field.
- View source code of the object — Select a row, right-click and select Show Source.
- Add objects to the Wave view — Select a row, right-click and select Add to Waves.
- Add objects to the List view — Select a row, right-click and select Add to Lists.
- Trace transaction to set context for the transaction message — Select a row, right-click and select Trace Transactions.

By default, DVE displays the current simulation time data in the Stack and Local pane. If you select a previous simulation time from the Transaction pane, the Stack and Local panes display the data recorded by \$vcdplustblog at the selected (previous) time unit, which is nothing but the transaction history.

Viewing the Dynamic Data Types in the Wave view

You can add a particular \$vcdblustblog call, that is a row from the Transaction pane to the Wave View. Wave view would then show a single fabricated variable tied to the particular scope where call exists with values over time across the multiple \$vcdblustblog calls made in said scope. Similarly, variables recorded in a call (whether from a frame or passed as argument) can be shown in the Wave view and they also create a fabricated variable that groups the data into a waveform display (a row) across the multiple samplings made in the scope.

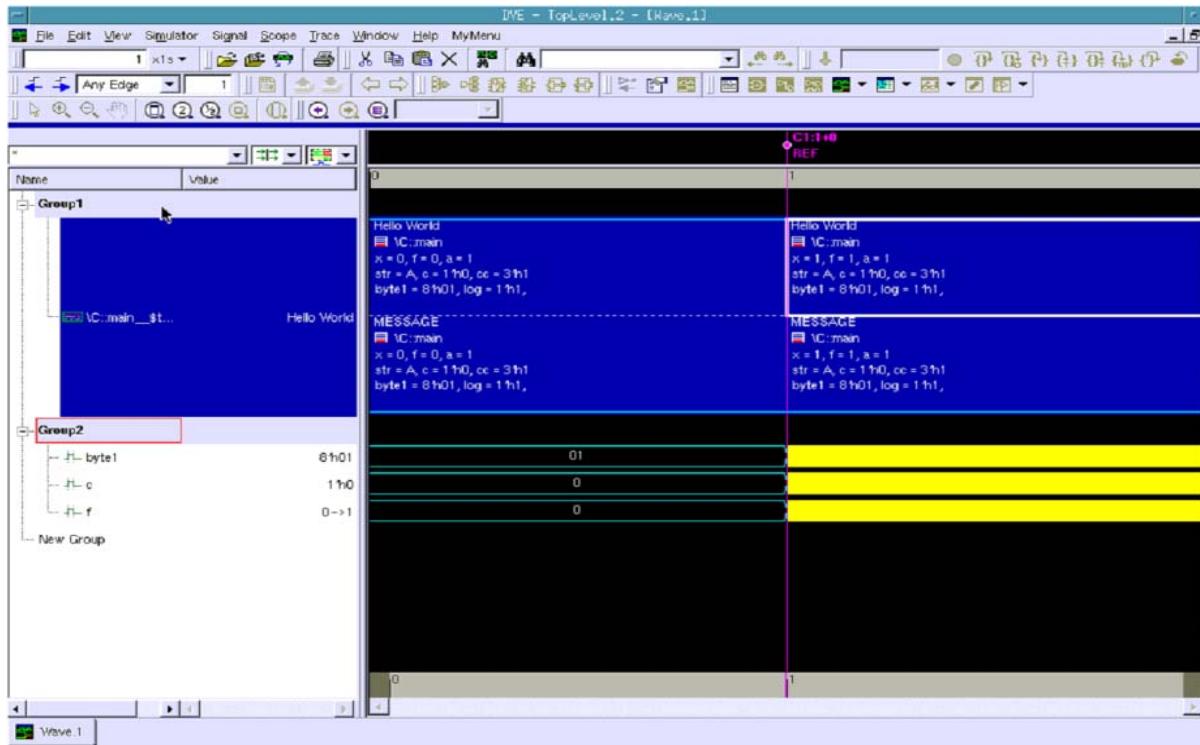
There can only be one \$vcdblustblog variable per call scope representing and grouping all the individual calls made from this scope. The individual calls are displayed as individual values, that is boxes of this \$vcdblustblog variable. The value is held until a subsequent call is made from the same scope, thus forming a new value for the collective \$vcdblustblog scope variable.

In the Wave view,

- The messages are displayed in the format specified in the \$vcdblustblog string variable, multi-line text, or html.
- You can set the display of messages from the Application Preferences dialog box. The messages are always displayed, but you can turn on/off the display of call stack or values.
- You can search the waveforms by giving the string pattern with wildcards, or just scan with the Search Forward and Search Backwards buttons.

The following illustration displays the \$vcdblustblog objects in the DVE Wave view:

Figure 12-5 \$vcddplusblog display in the Wave view



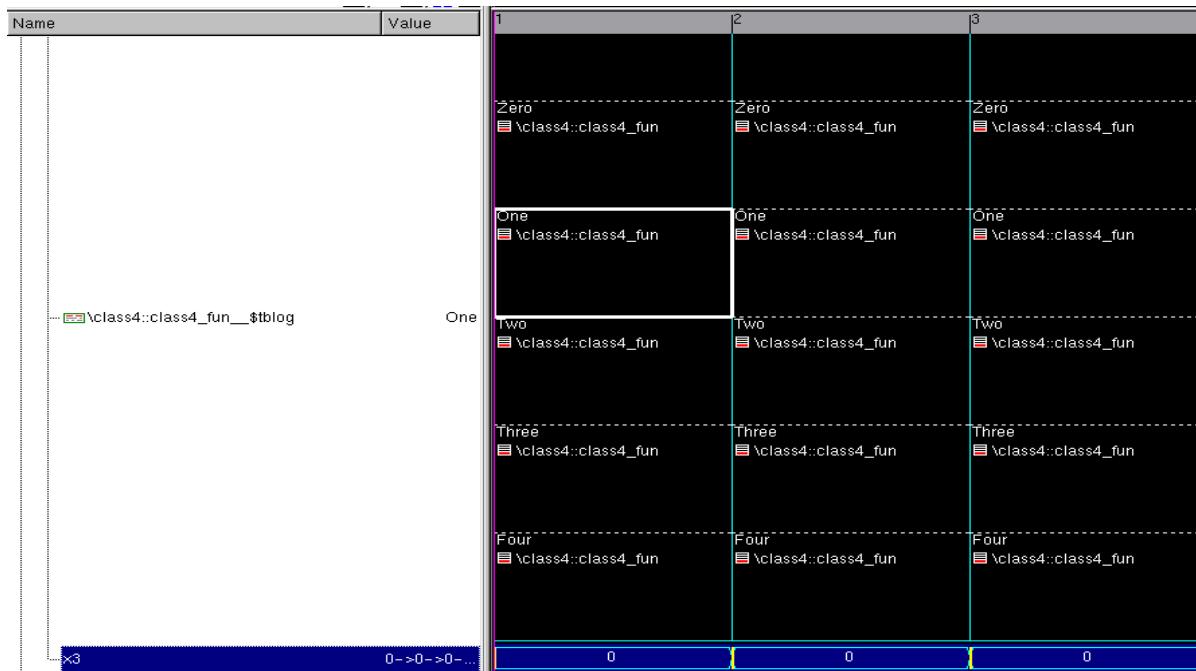
Modified Version of Example 1

To make the Wave display and how it interacts with the Transaction pane clearer, a slightly modified version of the Example 1 is used to generate the figures that follows.

```
...
class4 inst=new();
initial
begin
    for (int i = 0; i < 10; i++) begin
        #1;
        inst.class4_fun();
    end
    #12 $finish;
end
endprogram
```

Selection is for "Case C" mentioned under Example 1, that is when integer value "One" is passed. Note that since all the calls are in the same scope "\class4::class4_fun", a single \$vcdblustblog Wave variable is created and the calls are individual values across time, also note the multiple \$vcdblustblog calls per time slot.

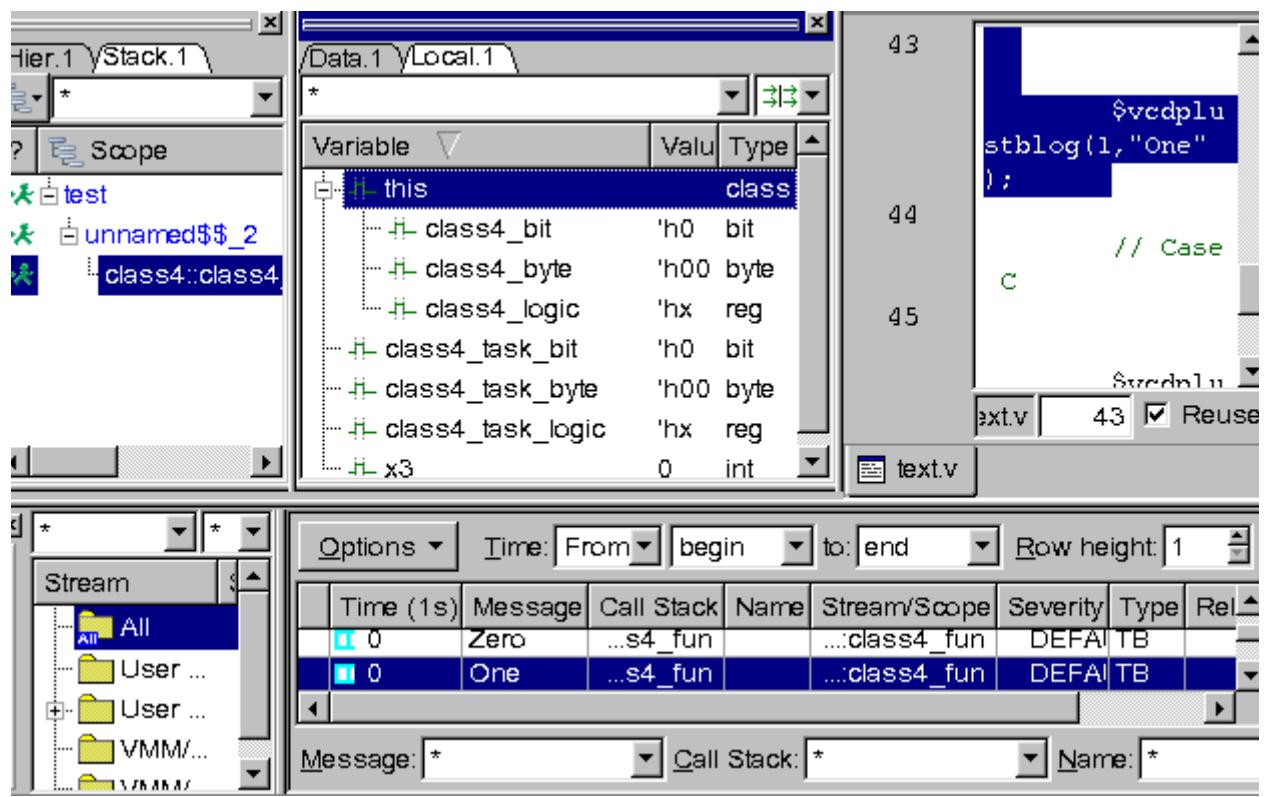
Figure 12-6 Wave view of Case C \$vcdblustblog call with the modified



example

The Transaction pane, context synced with the Wave view, is also shown in the following figure.

Figure 12-7 Transaction Pane of \$vcdblustblog “Case C”



Using \$vcdplusmsglog

\$vcdplusmsglog is primarily designed for transaction recording. \$vcdplusmsglog allows modeling and tracking of transactions on multiple streams. It can be used to define, create (start/extend/finish), and describe transactions including relationships. Similar to \$vcdplustblog, \$vcdplusmsglog can also be called from the UCLI prompt.

Usage Model

The task call syntax is as follows:

```
$vcdplusmsglog ([<frame_class_var_level>,<stream_spec>,<msg_type>,[<msg_name>,<msg_severity>,<message>,<relation_spec>[,<var>]^n])
```

where,

<frame_class_var_level>

Specifies integer value as follows:

0 — (default). When you specify the integer zero, no frame variables are recorded.

1 — Records frame variables up to one level. If specified in a class method, \$vcdplusmsglog dumps the class data members.

2 — Records up to two levels, that is the data recorded when integer value 1 is specified + dump 1-level up of base data members if class is an extension.

N — Same as N-1 recording + dump (N-1)-level up of base data members if class is an extension

-1 — Records all the frame variables and all the class data up to the very base class, if class is an extension.

<stream_spec>

Specifies an optional stream to use in the recording. If no stream is specified, then the call scope is used as stream (like \$vcdblustblog). The stream can be a name passed as literal or a variable. Optionally, you can pass a scope name in which case the stream is created under the said scope.

stream_spec := stream_name [, stream_scope]
where,

stream_scope — <SV identifier literal or string variable>

stream_name — <SV simple (non-escaped) identifier literal or string variable>

<msg_type>

Specifies a transaction message type, such as NOTE, or XACTION. The message type is same as that of vmm_log::types_e that is the type encodings of the enum are same. The constant names however are changed to be more generic. The full listing of the types enum is shown in a later section.

<msg_name>

Specifies the name of the transaction.

<msg_severity>

Specifies the transaction message severity. The message severity is same as that of vmm_log::severities_e, the type encodings of the enum are same. The constant names however are changed to be more generic. The full listing of the severity enum is shown in a later section.

<message>

Specifies the optional message to record with the transaction. The message can consist of a header and optionally a body.

```
message := msg_header [, msg_body]
```

<relation_spec>

Specifies the transaction relation. A relation is a self-relation, such as START or FINISH. \$vcdplusmsglog creates a transaction on the given stream that is the call would be viewed abstractly as:

```
$vcdplusmsglog (<message type, name, severity here>,  
START/FINISH)
```

If relation is not a self-relation, then a target transaction is required and the relation is a uni-directional relation between source and target. The call can be viewed abstractly as:

```
$vcdplusmsglog(<relation_source>, relation,  
<relation_target>)
```

where, the source is specified by the tuple (<stream_spec>, <msg_type>, <msg_name>, <msg_severity>), and target is the transaction target you are relating the source to. The example is as follows:

```
<relation_spec> := [relation] [<user_relation_name>] [,  
<relation_target>]  
where
```

`<user_relation_name>` — string name required only if relation is "USER" enum field.

`<relation_target>` —
[[[stream_scope.]stream_name.]msg_name]

The full listing of the types enum is shown in a later section.

`<var>`

Specifies the transaction attribute variables.

Example

In the following example, multiple calls to `$vcdplusmsglog` is made to record the transactions.

```
`include "msglog.svh" // Package containing the enum
definitions
program p;
import _vcs_msglog::*; // Import package
class C;
    int att1 = 1;
    int att2 = 2;
task read;
    // Create READ transaction on stream "stream1" and
    // start it with attributes: att1, att2
    $vcdplusmsglog("stream1",XACTION,"read",NORMAL,"READ",
                    START, att1, att2);
    #2;
    // Finish READ transaction
    $vcdplusmsglog("stream1",XACTION,"read",NORMAL,"READ",
                    FINISH);
Endtask // read

task response;
    int att3 = 3;
    // transaction RESP is on same stream, and has att3
    $vcdplusmsglog("stream1",XACTION,"resp",NORMAL,"RESP",
                    START, att3);
```

```

#1;
$vcplusmsglog("stream1", XACTION, "resp", NORMAL, "RESP",
    FINISH);
endtask // response
endclass // C

C c = new;
initial begin
    c.read();
    c.response();
    // Relation: RESP is a child of READ.
    $vcplusmsglog("stream1", XACTION, "resp", NORMAL, CHILD,
        "read");
    #1;
    $finish;
end

endprogram

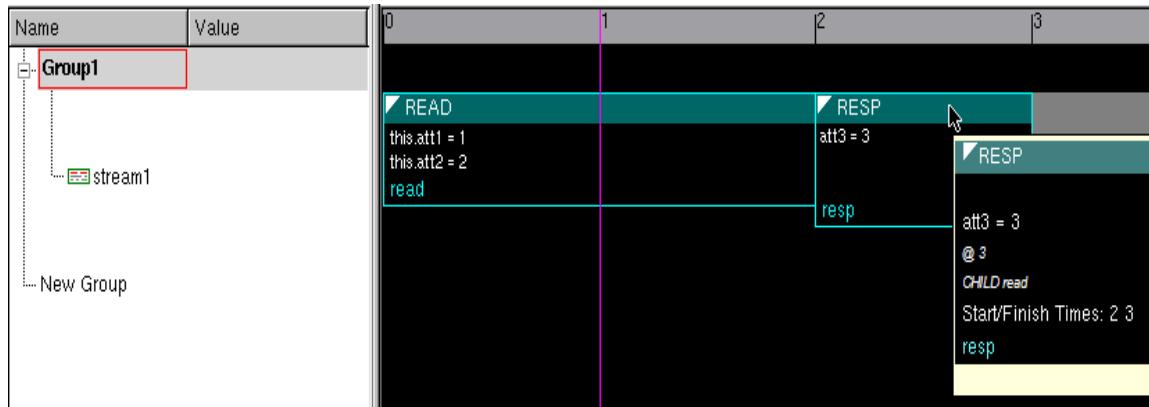
```

In the “read task”, see the two calls that START and FINISH. The transaction called “read” on stream is “stream1”; the type is XACTION, and the severity is NORMAL. The attributes recorded at the START of the message are “att1” and “att2” variable values.

In the “response” task, see the two calls that START and FINISH. The transaction called “resp” on stream is “stream1”, the type is XACTION, and the severity is NORMAL. The transaction attribute recorded at the START of this transaction is “att3”.

The following figure shows the Wave view of DVE displaying the transactions on stream 1. The stream forms a “row” in the Wave view, transactions as values on the stream, transaction names are shown in the lower right corner of the transaction box, and the header message is shown in the box header.

Figure 12-8 \$vcplusmsglog Transaction Recording (Example 1)



In the example, you also create a unidirectional relationship of:

"resp" (on stream1) is a child of "read" (on stream1)

where, "resp" is the source of the relationship, and "read" is the target.

DVE displays the relationship in the source transaction, shown in the tooltip in the figure above.

The first line in the example listing ``include "msglog.svh"' includes the VCS package that defines all the enums for type, severity, and relations. To run this example in VCS and use the package in the msglog.svh file found under include in the VCS release, you will compile as follows:

```
vcs -sverilog -debug_all <testname> +incdir+$VCS_HOME/include
```

Transaction Relationships

The transaction relationship enum is as follows:

```
enum int {
    START = 'h0001,
```

```

    FINISH = 'h0002,
    PRED = 'h0004,
    SUCC = 'h0008,
    SUB = 'h0010,
    PARENT = 'h0020,
    CHILD = 'h0040,
    XTEND = 'h0080,
    USER = 'h0100// USER RELATION
} _MSG_R;

```

START. FINISH, XTEND — are self-relations; they operate on the source and do not need a target transaction. START indicates starting a transaction, and FINISH completes it. XTEND relation allows you to "grow" a transaction (it should start and not finished) with more attributes at any point in its duration.

PARENT/CHILD — Indicates hierarchical transactions.

SUB — Indicates composition sub-part of the transactions.

SUCC/PRED — Indicates causal relationship between the transactions.

These are the built-in relationships. To create any other named relation you want beyond the built-in, you can use the USER relation. If you specify USER, then you must specify an argument as a string name.

VPD Recording

The `$vcdplusmsglog` task records the following information into the VPD file:

- Message header and body, types, severities, and relations
- Dynamic or static variables provided as arguments (relative to current scope or with absolute path)

Similar to `$vcdplustblog`, the following items, not passed explicitly as an argument, are recorded per call:

- Time of the `$vcdplusmsglog` call
- Call stack of the `$vcdplusmsglog` call

`$vcdplusmsglog` however has a different focus than `$vcdplustblog`. `$vcdplusmsglog` is targeted towards transaction modeling and debug, and has no frame recording capability. Its string message consists of a header and a body; the header is expected to be used to describe the kind of the transaction "read", or "write", while the body is used for any generic messaging. The variables are passed in model attributes of the transactions.

Turning `$vcdplusmsglog` Task ON/OFF

You can globally turn the recording of messages on and off. This is useful to limit the recording for certain time ranges or based on conditions. The following tasks are used for this purpose:

- `$vcdplusmsglogoff()` — Disables globally `$vcdplusmsglog` based recording.
- `$vcdplusmsglogon()` — Enables globally `$vcdplusmsglog` based recording.

Filtering `$vcdplusmsglog` Messages in Wave View

You can filter the transaction messages in the Wave View.

Example

In the following example, multiple calls to `$vcdplusmsglog` is made to record the transactions.

test.sv

```
`include "msglog.svh" // Package containing the enum
                      definitions. See the document
                      "Debugging with Transactions" in VCS
                      Online Documentation to see the
                      msglog.svh file.

program p;
class C;
    int att1 = 1;
    int att2 = 2;
task read;
    // Create READ transaction on stream "stream1" and
    // start it with attributes: att1, att2
    $vcplusplusmsglog("stream1",XACTION,"read",NORMAL,"READ",
                      START, att1, att2);
    #2;
    // Finish READ transaction
    $vcplusplusmsglog("stream1",XACTION,"read",FATAL,"READ",
                      FINISH);
endtask // read

task response;
    int att3 = 3;
    // transaction RESP is on same stream, and has att3

$vcplusplusmsglog("stream1",XACTION,"resp",WARNING,"RESP",
                  START, att3);
    #1;
$vcplusplusmsglog("stream1",XACTION,"resp",NORMAL,"RESP",
                  FINISH);
endtask // response
endclass // C

C c = new;
initial begin
    c.read();
    c.response();
    // Relation: RESP is a child of READ.
    $vcplusplusmsglog("stream1",XACTION, "resp",ERROR, CHILD,
                      "read");
```

```
#1;  
$finish;  
end  
  
endprogram
```

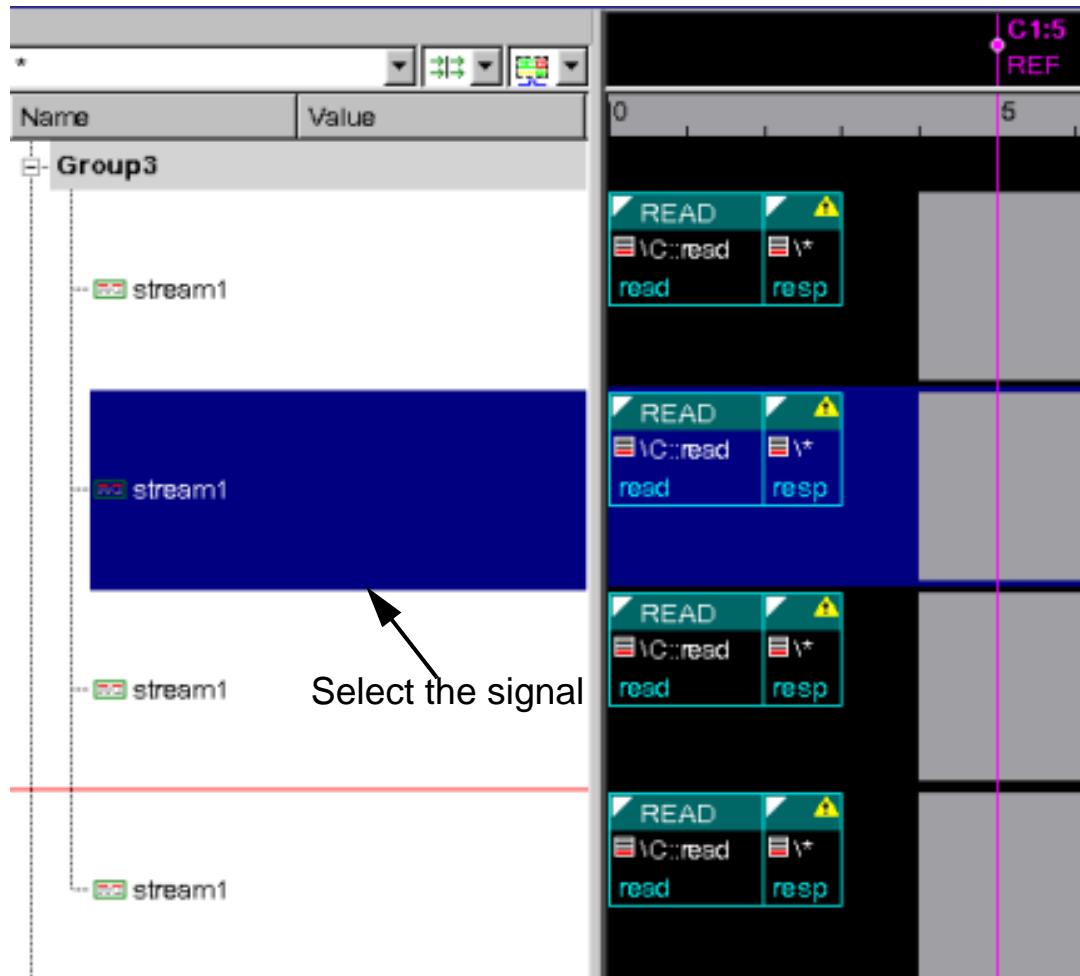
To compile this example code, use the following commands:

```
vcs test.sv -debug_all -sverilog  
simv -gui &
```

To filter the transaction messages in Wave View

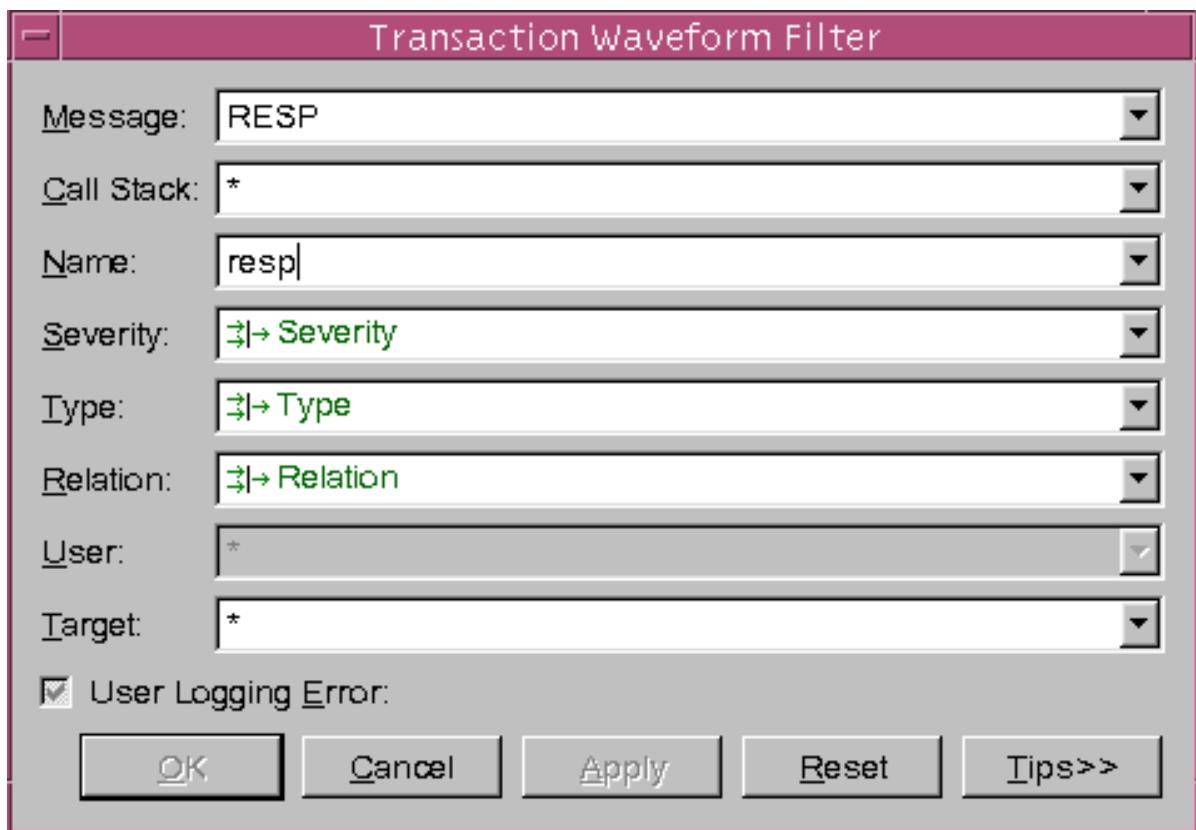
1. Select the messages in the Transaction pane.
2. Right-click and select **Add to Waves**.

The unfiltered transaction messages are added in the Wave View as shown the figure.



3. Select the signal in the Wave View, right-click and select **Transaction Filter**.

The Transaction Waveform Filter dialog box appears.



4. Specify the following information, as required:

- Message — Identifies the transaction message. Type the string or the string with wildcards, for example RESP*, to filter the messages.
- Call Stack — Identifies the caller stack. Type the string or string with wildcards to filter by call stack.
- Name — Identifies the message name. Type the string or string with wildcard to filter by name.
- Severity — Specifies the message severity. Select the check box against the severity using which you want to filter or select the **All** check box to select all the severities.

- Type — Specifies the message type. Select the check box against the message type you want.
 - Relation — Specifies the relation type. Select the check box against each relation to filter based on relation.
 - User — Identifies the user-defined relation type. Type the string to filter.
 - Target — Identifies the message target. Type the string to filter based on target.
 - User Logging Error — Specifies the runtime errors flagged by \$vcplusmsglog call. Select or clear the check box to show or hide the error messages.
5. Click **OK** to apply the filter criteria and close the dialog box.

Click **Apply** to apply the filter criteria and not close the dialog box.

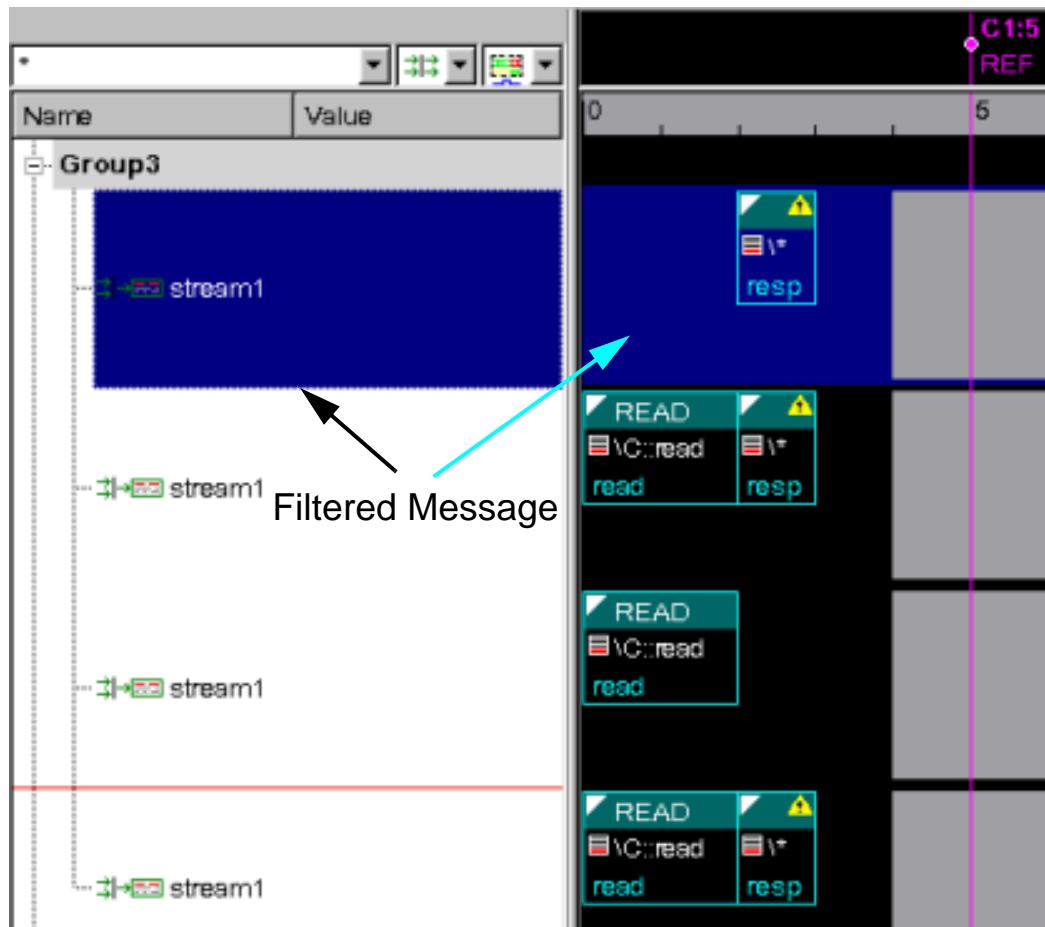
Click **Cancel** to abandon the changes, and close the dialog box.

Click **Tips** to display the tips page on the right.

The messages are filtered in the Wave View as per your filter

criteria. The filtered waveforms display the  icon.

Following figure displays messages filtered by RESP.



Limitations

- Filtering of \$vcdplustblog data is not yet supported.

Viewing Streams and Transaction Relations

You can now use the following features in the Transaction pane:

- Streams list for stream display and the messages within.
- A Transaction Relations dialog box for display and exploration of the transaction relations.

Example

[Example 12-1](#) shows streams and transaction relations.

Example 12-1 Transaction Debug

```
program p;
    import _vcs_msglog::*;
    string OpStream = "OpStream";

    class C;
        int att1 = 1;
        int att2 = 2;

        task read;

            string read1="READ";
            $vcdblustblog(-1,"Read Dynamic data");
            $vcdblusmsglog(OpStream,XACTION,"read",NORMAL,"READ",START,att1, att2);
            #2;
            $vcdblusmsglog(OpStream,XACTION,"read",NORMAL,"READ",FINISH);

        endtask

        task write;

            int att3 = 3;
            $vcdblustblog(-1,"Write Dynamic data");
            $vcdblusmsglog(OpStream,XACTION,"write",NORMAL,"WRITE",START,att3);
            #1;
            $vcdblusmsglog(OpStream,XACTION,"write",NORMAL,"WRITE",FINISH);

        endtask

        task response;

            int att3 = 3;
            $vcdblustblog(-1,"Response Dynamic data");
            $vcdblusmsglog(OpStream,XACTION,"resp",NORMAL,"RESP",START,att3);
            #1;
            $vcdblusmsglog(OpStream,XACTION,"resp",NORMAL,"RESP",FINISH);

        endtask

    endclass

    C c = new;

    initial
    begin
        c.read();
        c.write();
        c.response();
        //Relation between the Transaction

        $vcdblusmsglog(OpStream,XACTION, "resp",FATAL, CHILD,"read");

    end
```

```

$vcplusmsglog(OpStream,XACTION, "read",ERROR,SUCC,"write");
$vcplusmsglog(OpStream,XACTION, "write",WARNING,PARENT,"read");
$vcplusmsglog(OpStream,XACTION, "write",NORMAL,CHILD,"read");
$vcplusmsglog(OpStream,XACTION, "write",TRACE,SUB,"resp");

#1;
$finish;
end
endprogram

```

Steps to compile the example:

```

% vcs -nc -debug_all -sverilog \
$VCS_HOME/include/msglog.svh top.sv

% ./simv

% dve -vpd vcdplus.vpd &

```

Viewing Streams and Messages in Streams List

To view the streams and messages in streams, load the design in DVE.

The Transaction pane displays the stream list on the left and Transaction table on the right.

The screenshot shows the ModelSim Transaction pane. On the left is a tree view labeled 'Streams' with nodes for 'All', 'meglog' (containing 'OpStream'), and 'tblog' (containing 'p\IC:read...', 'p\IC:write...', and 'p\IC:respon...'). On the right is a table labeled 'Transaction' with columns: Time (1s), Message, Call Stack, Name, Stream/Sco, Severity, Type, Relation, and Target. The table contains six rows of transaction logs. Below the table are search fields for 'Message', 'Call Stack', 'Name', and 'User errors'.

Stream	Scope	Time (1s)	Message	Call Stack	Name	Stream/Sco	Severity	Type	Relation	Target
All		0	Readread	p\IC:read	OpStream	DEFAL	TB		
meglog	OpStream	0	READ	...read	read	OpStream	NORM	XACTIO	START	
tblog	p\IC:read...	2	Writewrite		p\IC:write	DEFAL	TB		
tblog	p\IC:write...	2	WRITE	...write	write	OpStream	NORM	XACTIO	START	
tblog	p\IC:respon...	3	Res...onse		...esponse	DEFAL	TB		
tblog	p\IC:respon...	3	RESP	...onse	resp	OpStream	NORM	XACTIO	START	

Streams list

Transaction table

The Streams list shows all the streams created up to the current time. A stream is considered created when the first message for that stream is recorded.

Stream	Scope
All	
msglog	OpStream
tblog	\C::read ... p.\C::read \C::write ... p.\C::write \C::respo... p.\C::respo...

The Stream list contains the following items:

- Stream column — Displays leaf-level name for stream.
- Scope column — Displays the scope where it is defined.

You can filter streams in the Streams list using stream and scope filters on top.

Selection in the Stream list triggers filtering in the Transaction table on the right. You can select multiple streams. When no stream is selected, the Transaction table is empty.

- Nodes — Contains the following three nodes:
 - All — Shows all messages in the Transaction table upon clicking.
 - msglog — Contains all msglog streams. These streams are typically defined using `$vcdpplusmsglog()` calls with a name.

- tblog — Contains all streams generated by tblog.

From the Streams list, you can add the selected streams and dumped variables to the Wave View.

Viewing Messages in the Transaction Table

To view the messages in the Transaction table, click on a node in the Streams list.

Options		Time:	From	begin	to	end	Severity	Type	
							Severity	Type	
✓	Group View	Message	Call Stack	Name	Stream/Scope	Severity	Type	Relation	
	Reset	ID	...\$_2\!C::read	read	OpStream	■ NORMAL	XACTION	START	
		2	Write Dyna...	..._2\!C::write	p.\!C::write	DEFAULT	TB		
		2	WRITE	..._2\!C::write	write	OpStream	■ NORMAL	XACTION	START
		3	Response:response	p.\!C::response	DEFAULT	TB		
		3	RESP	...:response	resp	OpStream	■ NORMAL	XACTION	START

Message: * Call Stack: * Name: * User errors:

The Transaction table contains the following items:

- Options list — Changes table mode.
 - The Group View groups messages related to a single transaction as a single row. You can expand this row to see all self relations for the row. Self relations are START, FINISH, and EXTEND.
 - The Reset View displays a time-ordered message list regardless of relations.

Options		Time:	From	begin	to	end	Severity	Type	
							Severity	Type	
✓	Group View	Message	Call Stack	Name	Stream/Scope	Severity	Type	Relation	
	Reset	ID	...\$_2\!C::read	read	OpStream	■ NORMAL	XACTION	START	
		2	Write Dyna...	..._2\!C::write	p.\!C::write	DEFAULT	TB		
		2	READ	...\$_2\!C::read	read	OpStream	■ NORMAL	XACTION	FINISH
		2	WRITE	..._2\!C::write	write	OpStream	■ NORMAL	XACTION	START
		3	Response:response	p.\!C::response	DEFAULT	TB		

Message: * Call Stack: * Name: * User errors:

- Filters — Control transactions/messages and stream selection in the Streams list. To filter the messages, click the header of any column.

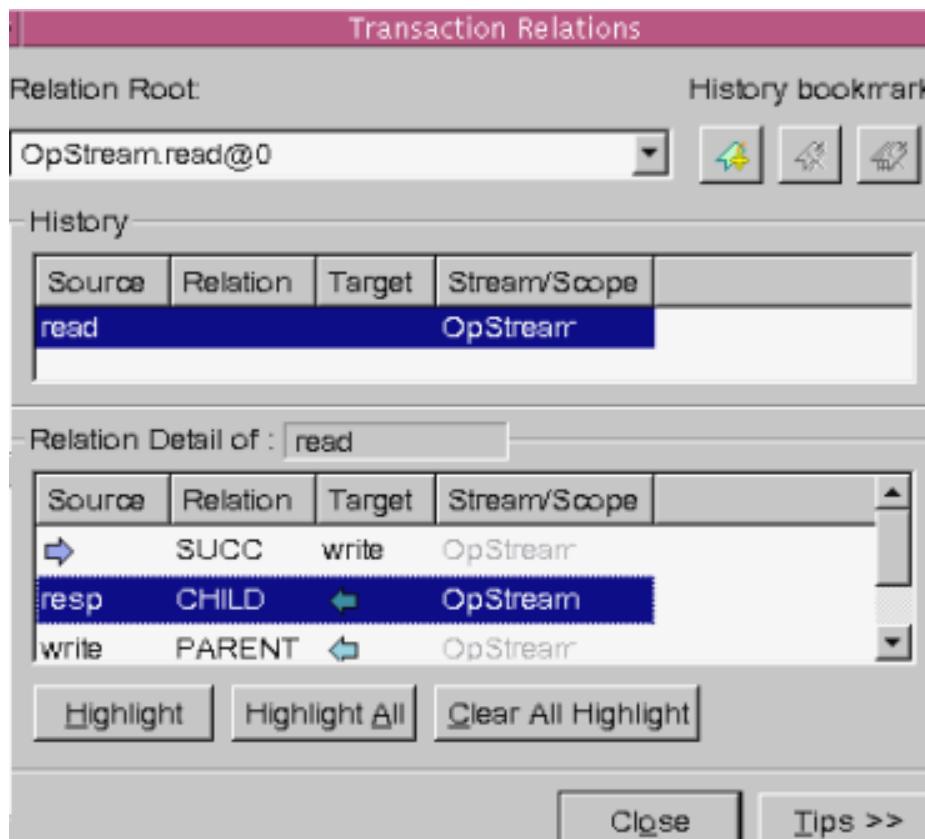
Viewing Transaction Relations

Transaction relation is a unidirectional association between a source and target message name.

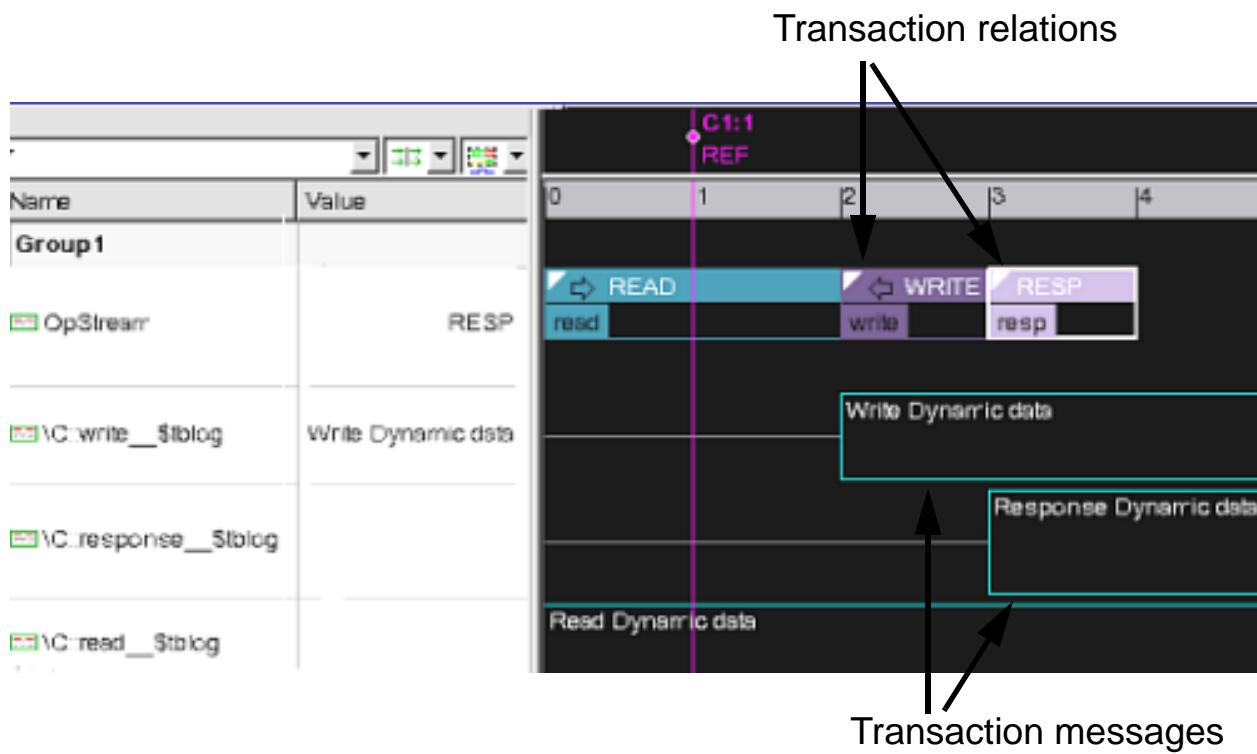
To view transaction relations

1. Select a message in the Transaction table.
2. Right-click and select **Show Transaction Relations**.

The Transaction Relations dialog box appears.



You can view the relation history and its details. You can also highlight the items for easy identification in the Transaction table and Wave View.



3. Click the **Tips>>** button at any time for more details about each field in the Transaction Relations dialog box.

SystemVerilog String Variables dump using \$vcdplustblog() and \$vcdplusmsglog()

You can dump the SystemVerilog String variables in DVE through \$vcdplustblog or \$vcdplusmsglog .

Usage Model

In the following example, SV string datatype is defined.

test.sv

```
package pkg;

class C;
    int i;
    integer p;
    int a1=5;
    string base = "string1";

    task main(int x = 0);
        int f = x;
        int a=1;
        string str = "string2";
        bit c=1'h0;
        bit [2:0] cc = 3'h1;
        byte bytel= 1;
        logic log='h1;

        begin
            $vcdblustblogon();
            $vcdblustblog(-1,"Message",f);
            $vcdblustblogoff();
            $vcdblustblog(-1,"One",a);
            $display("Message");
        end
    endtask
endclass
endpackage // pkg

program prog;
    import pkg::*;

    C inst = new;
    initial
    begin //: A1
        int inti =12;
        inst.main();
        #1;
        inst.main(1);
```

```

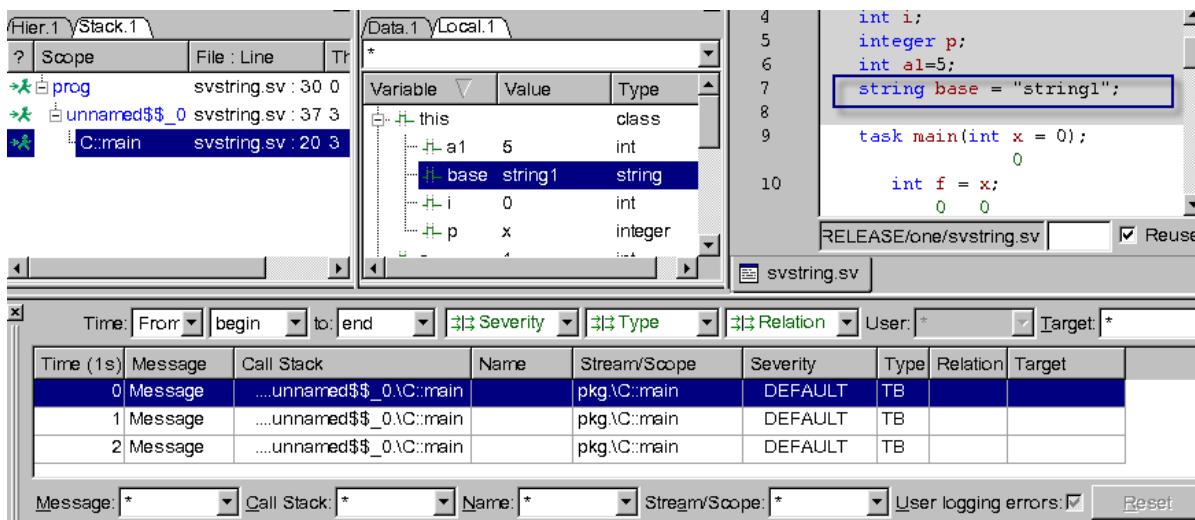
#1;
inst.main(2);
end // : A1
endprogram

```

To compile this example, use the following commands:

```
vcs -R -nc -debug_all -sverilog test.sv
dve -vpd vcdplus.vpd &
```

The following illustration display the SV String variables in the Local pane.



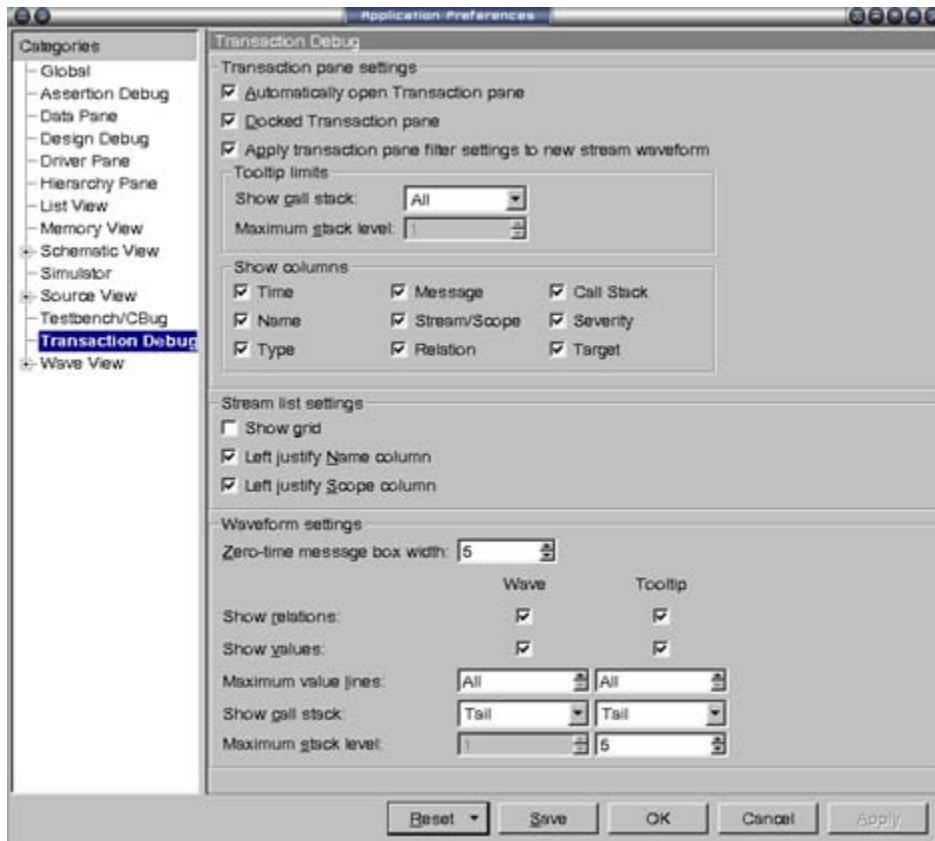
Editing Transaction Debug Preferences

You use the Transaction Debug category in the Application Preferences dialog box to change the settings.

To edit the transaction debug preferences

1. Open DVE.
2. Click **Edit > Preferences**.

The Application Preferences dialog box opens.



3. Click the Transaction Debug category in the left pane and select the following options, as required:
 - Transaction pane settings
 - Automatically open Transaction pane—Opens the Transaction pane automatically when DVE is invoked, if this check box is selected.
 - Docked Transaction pane — Opens the Transaction pane in the same TopLevel window, if this check box is selected.
 - Waveform settings
 - Show caller — Displays the Caller string head, tail, both head and tail, or none in the Wave View and the tooltip.

- Show values — Displays the values of the \$vcdplustblog and \$vcdplusmsglog objects in the Wave View.
 - Show relations — Displays the relations specified in the \$vcdplusmsglog statement.
4. Click one of the following:
- OK** to apply your settings and close the dialog box.
- Apply** to apply your settings and have the dialog box remain open.
- Cancel** to close and not apply the settings.

Using tblog and msglog in DVE Command Prompt

You can use tblog and msglog in the DVE command prompt interactively, without using them in the source code, for debugging the transactions.

Following are the advantages of using tblog/msglog in the interactive mode:

- When you use tblog/msglog in the source code, tblog/msglog gets executed for every call, whereas in interactive mode, tblog/msglog gets executed only when you issue that command.
- When you use tblog/msglog in the source code, you need to recompile the design. whereas in the interactive mode, you don't need to recompile.

Consider the following example:

tbog.sv

```
program prog;
  class C;
```

```

int i;
integer p;
string base = "B";

task main(int x = 0);
    int f = x;
    string str = "A";
    bit c=1'h0;
    logic log='h1;
    begin
        $display("Message %d",x);
    end
endtask
endclass

C inst = new;
initial
begin
    int inti =12;
    inst.main();
    #1;
    inst.main(1);
    #1;
    inst.main(2);
end
endprogram

```

Compile the example using the following commands:

```

vcs -nc -debug_all -sverilog tblog.sv

./simv -gui &

```

The design is loaded in DVE. Type the following commands in the DVE command prompt:

```

dve> stop -file tblog.sv -line 13
dve>run
dve>tblog -l -1 -m {"Run1"}

```

```
dve>run  
dve>tblog -l -1 -m {"Run2"}  
dve>run  
dve>tblog -l -1 -m {"Run3"}  
dve>run
```

The DVE Console pane shows the output as follows:

```
dve> stop -file tblog.sv -line 13  
1  
dve> run  
Stop point #1 @ 0 s;  
dve> tblog -l -1 -m {"Run1"}  
dve> run  
Stop point #1 @ 1 s;  
Message 0  
dve> tblog -l -1 -m {"Run2"}  
dve> run  
Stop point #1 @ 2 s;  
Message 1  
dve> tblog -l -1 -m {"Run3"}  
dve> run  
tblog.sv, 1 : program prog;  
Message 2  
$finish at simulation time 2  
Simulation complete, time is 2.  
V C S   S i m u l a t i o n   R e p o r t  
Time: 2  
CPU Time: 0.120 seconds; Data structure size: 0.0Mb  
Thu Jun 24 23:27:56 2010
```

Now, you can see the tblog data in the Transaction pane

Time (1s)	Message	Call Stack	Name	Stream/Scope	Severity	Type	Relation	User	Target
0	Run1	...0\C.main		prog\C.main	DEFAU	TB			
1	Run2	...0\C.main		prog\C.main	DEFAU	TB			
2	Run3	...0\C.main		prog\C.main	DEFAU	TB			

Message: * Call Stack: * Name: * Stream/Scope: * User logging errors:

Double-click on any record in the Transaction pane to see the corresponding variable in the Local pane, as shown:

The screenshot shows the ModelSim interface with two main windows open. The top window is the Transaction pane, which displays a table of log entries. The bottom window is the Local pane, which shows the state of variables in the current scope. A red dot in the Transaction pane indicates the selected row, which corresponds to the variable list in the Local pane. The Local pane also shows the source code for the task being executed.

Transaction Pane (Top):

Time (1s)	Message	Call Stack	Name	Stream/Scope	Severity	Type	Relation	User	Target
0	Run1	...0\C.main		prog\C.main	DEFAU	TB			
1	Run2	...0\C.main		prog\C.main	DEFAU	TB			
2	Run3	...0\C.main		prog\C.main	DEFAU	TB			

Local Pane (Bottom):

Hier.1\Stack.1\Class.1

Data 1\Local.1\Member.1

Variable	Value	Type
this	class	
b	B	string
i	0	int
p	x	integer
a	'h0	bit
f	0	int
log	'h1	reg
str	A	string
x	0	int

task main(int x = 0);
int f = x;
string str = "A";
bit c=1'h0;
logic log='h1;
begin
\$display("Message %d", x);
end
endtask
endclass

C inst = new;
initial

tblog.sv

File: /slowfs/vgpv2/maheshb/tblog.sv

Time: From begin to end Severity Type Relation User Target

Message: * Call Stack: * Name: * Stream/Scope: * User logging errors:

Transaction Debug in SystemC Designs

Using tblog

The `SC_tblog` class is implemented in SystemC for better transaction level debugging in DVE. This class records formatted string information in the VPD file, which is displayed by DVE.

You can call the `SC_tblog` class within any `SC_THREAD`, `SC_CTHREAD`, `SC_METHOD`, function, or constructor.

This chapter consists of the following sections:

- “Use Model” on page 44
- “SC tblog On/Off Control” on page 45
- “Recorded Information in VPD File” on page 46
- “Example” on page 47
- “Viewing the Recorded Information in DVE” on page 51

Use Model

Use the following command for better transaction level debugging in DVE :

```
sc_snps::tblog << "String" <<
    TBLOG_VAR(var) <<
    sc_snps::end;
```

Where,

- `sc_snps::tblog` is the class, which is defined in the `tli_tb.log.h` file.

- `String` is the string, which is recorded in the VPD file.
- `TBLOG_VAR` is a macro, which displays the following information in DVE:
 - Name of the `var` for declaration, where `var` is the dynamic or static variable provided as argument (relative to current scope or with absolute path).
 - `typeid(var)`, converted to an enum as required, for declaration.
 - Value of the variable which is encoded as required.

Note:

You must include `tli_tblog.h` to use `sc_snps::tblog`
`#include "tli_tblog.h".`

SC tblog On/Off Control

The `SC tblog` class supports a global mechanism to turn dumping on and off. This is useful to limit the recording for certain time ranges based on conditions.

Disabling tblog-based recording

You can use the following API call to disable `tblog`-based recording:

```
sc_snps::tblogoff();
```

Enabling tblog-based recording

You can use the following API call to enable `tblog`-based recording:

```
sc_snps::tblogon();
```

Note:

Enabling or disabling tblog-based recording is global for both SystemVerilog and SystemC. If you turn it ON or OFF with \$vcdplustblog in SystemVerilog, then the SystemC recording is also turned ON or OFF and vice versa.

Enabling or disabling tblog-based recording by passing an argument

You can use the following API call to pass in an argument, for example, from a variable:

```
sc_snps::tblogon(bool)
```

Example:

- sc_snps::tblogon()

OR

```
sc_snps::tblogon(true) or sc_snps::tblogon(1)
```

- sc_snps::tblogoff()

OR

```
sc_snps::tblogon(false) or sc_snps::tblogon(0)
```

Recorded Information in VPD File

The following information is recorded in VPD file:

- Current simulation time
- Call stack

- Constant string message
- Name of the variable
- Value of the variable

Note:

- If the space within the waveform is limited, hovering the mouse pointer over a certain transaction shows its full text in a separate window.
- If you double-click a log record, it opens the call stack in the Stack pane.

Example

```
=====Mem.h=====
#define SC_INCLUDE_DYNAMIC_PROCESSES

#include "systemc"
using namespace sc_core;
using namespace sc_dt;
using namespace std;
#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h"
#include "tlm_utils/simple_target_socket.h"
#include "tli_tblogger.h"
SC_MODULE(Initiator)
{
    tlm_utils::simple_initiator_socket<Initiator> socket;
    SC_CTOR(Initiator)
        : socket("socket") {
            SC_THREAD(init_process);
        }
    void init_process()
    {
        tlm::tlm_generic_payload* trans = new
        tlm::tlm_generic_payload;
        sc_time delay = sc_time(20, SC_NS);
        for (int i = 1; i < 40; i += 4)
        {
```

```

        tlm::tlm_command cmd =
static_cast<tlm::tlm_command>(rand() % 2);
        if (cmd == tlm::TLM_WRITE_COMMAND) data = i;

                trans->set_command( cmd );
        trans->set_address(i);
        trans->set_data_ptr(reinterpret_cast<unsigned
char*>(&data) );
        trans->set_data_length( 4 );
        trans->set_streaming_width( 4 );
                trans->set_byte_enable_ptr( 0 );
        trans->set_dmi_allowed( false );
        trans->set_response_status(
        tlm::TLM_INCOMPLETE_RESPONSE );
        socket->b_transport( *trans, delay );

                cout << "trans = { " << (cmd ? "Write" :
"Read") << ", " << hex << i
                << " } , data = " << hex << data << " at time "
<< sc_time_stamp()
                << endl;
sc_snps::tblog<<"trans"
                << TBLOG_VAR(*trans) << TBLOG_VAR(data)
                << sc_snps::end;
        sc_snps::tblogon();
        wait(delay);
    }
}

int data;
};

SC_MODULE(Memory)
{
    tlm_utils::simple_target_socket<Memory> socket;

    SC_CTOR(Memory)
    : socket("socket")
    {
        // Register callback for incoming b_transport interface

```

```

method call
    socket.register_b_transport(this, &Memory::b_transport);

    // Initialize memory with random data
    for (int i = 0; i < 256; i++)
        mem[i] = 0x1010 | (rand() % 256);
}

// TLM-2 blocking transport method
virtual void b_transport( tlm::tlm_generic_payload& trans,
sc_time& delay )
{
    tlm::tlm_command cmd = trans.get_command();
    sc_dt::uint64 adr = trans.get_address() ;
    unsigned char* ptr = trans.get_data_ptr();
    unsigned int len = trans.get_data_length();
    unsigned char* byt = trans.get_byte_enable_ptr();
    unsigned int wid = trans.get_streaming_width();

    if ( cmd == tlm::TLM_READ_COMMAND )
        {
            memcpy(ptr, &mem[adr], len);
sc_snps::tblog<<"Mem_READ"
            << TBLOG_VAR(trans) << TBLOG_VAR(mem[adr])
            << sc_snps::end;
}
    else if ( cmd == tlm::TLM_WRITE_COMMAND )
        {
            memcpy(&mem[adr], ptr, len);
sc_snps::tblog<<"Mem_WRITE"
            << TBLOG_VAR(trans) << TBLOG_VAR(mem[adr])
            << sc_snps::end;
}

    // Obliged to set response status to indicate successful
completion
    trans.set_response_status( tlm::TLM_OK_RESPONSE );

}

int mem[256];
};

```

```

SC_MODULE(Top)
{
Initiator initiator;
Memory      memory;

SC_CTOR(Top) : initiator("initiator"), memory("memory")
{
    initiator.socket.bind( memory.socket );
}
};

int sc_main(int argc, char* argv[])
{
    Top top("top");
    sc_start();
    return 0;
}

=====
=====Mem.cpp=====
#include "Mem.h"
=====

To compile this example code, use the following commands:
```

```

./clean.csh

syscan Mem.cpp -cflags -g -t1m2
vcs -sysc=22 sc_main -cflags -g -debug_all
simv
```

To run this example code, use the following commands:

```

./simv

dve -vpd vcdplus.vpd &
```

Viewing the Recorded Information in DVE

To view the transactions using tblog

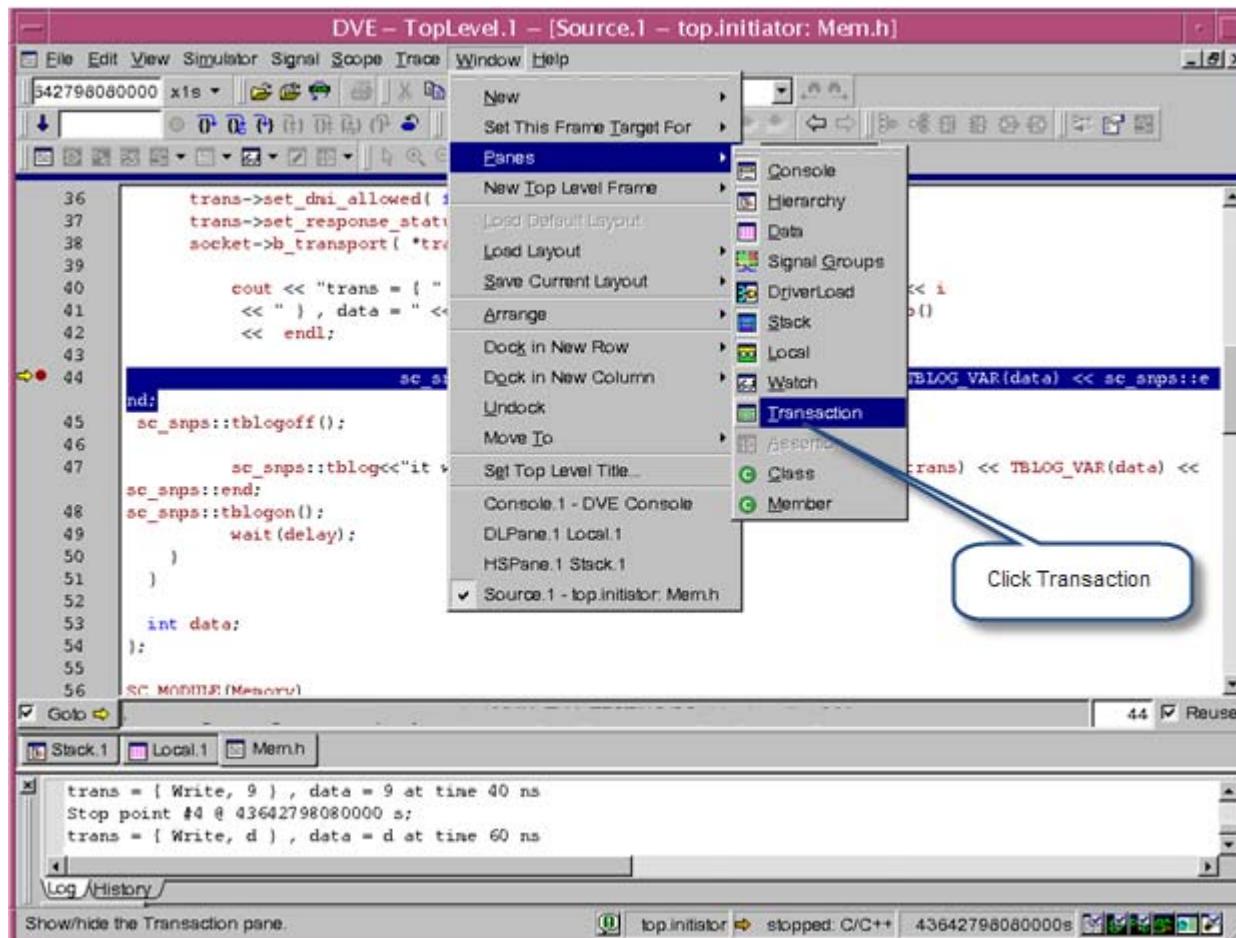
1. Open simv in DVE using the %simv -gui & command.
2. Double-click on the SystemC code where you have tblog in the Hierarchy pane.
3. Set the breakpoint at sc_snps::tblob

The screenshot shows the DVE (Digital Verification Environment) interface. The main window displays a portion of the Mem.h source code. A callout bubble points to the line 'sc_snps::tblob' with the text 'Set the breakpoint here'. The code includes comments explaining the logging of transactions and their details. Below the code editor, there are tabs for Stack, Local, and Mem. A message box at the bottom left states: 'Signal 'top.initiator.initiator_tblob'' cannot be found.. Stop point #1 @ 14547599360000 s: trans = { Read, 5 }, data = 10ff at time 20 ns'. The status bar at the bottom right indicates the simulation has stopped at C/C++ line 14547599360000.

```
36     trans->set_dmi_allowed( false );
37     trans->set_response_status( tlm::TLM_INCOMPLETE_RESPONSE );
38     socket->b_transport( *trans, delay );
39
40     cout << "trans = [ " << (cmd ? "Write" : "Read") << ", " << hex << i
41     << " ], data = " << hex << data << " at time " << sc_time_stamp()
42     << endl;
43
44     sc_snps::tblob<<"trans" << TBLOG_VAR(*trans) << TBLOG_VAR(data) << sc_snps::e
nd;
45     sc_snps::tbloboff();
46
47     sc_snps::tblob<<"it will not be displayed in DVE" << TBLOG_VAR(*trans) << TBLOG_VAR(data) <<
sc_snps::end;
48     sc_snps::tblobon();
49     wait(delay);
50   }
51 }
52
53 int data;
54 };
55
56 #endif // MEM.H
```

4. Run the simulation.
5. Click Next.

6. Click **Window > Panes > Transaction** to open the Transaction pane.



7. In the Transaction pane, double-click on the desired transaction, to view:
 - Stack function in the Stack Pane, as shown in [Figure 12-9](#).
 - Data members in the Local Pane, as shown in [Figure 12-10](#).

Figure 12-9 Stack Function in the Stack Pane

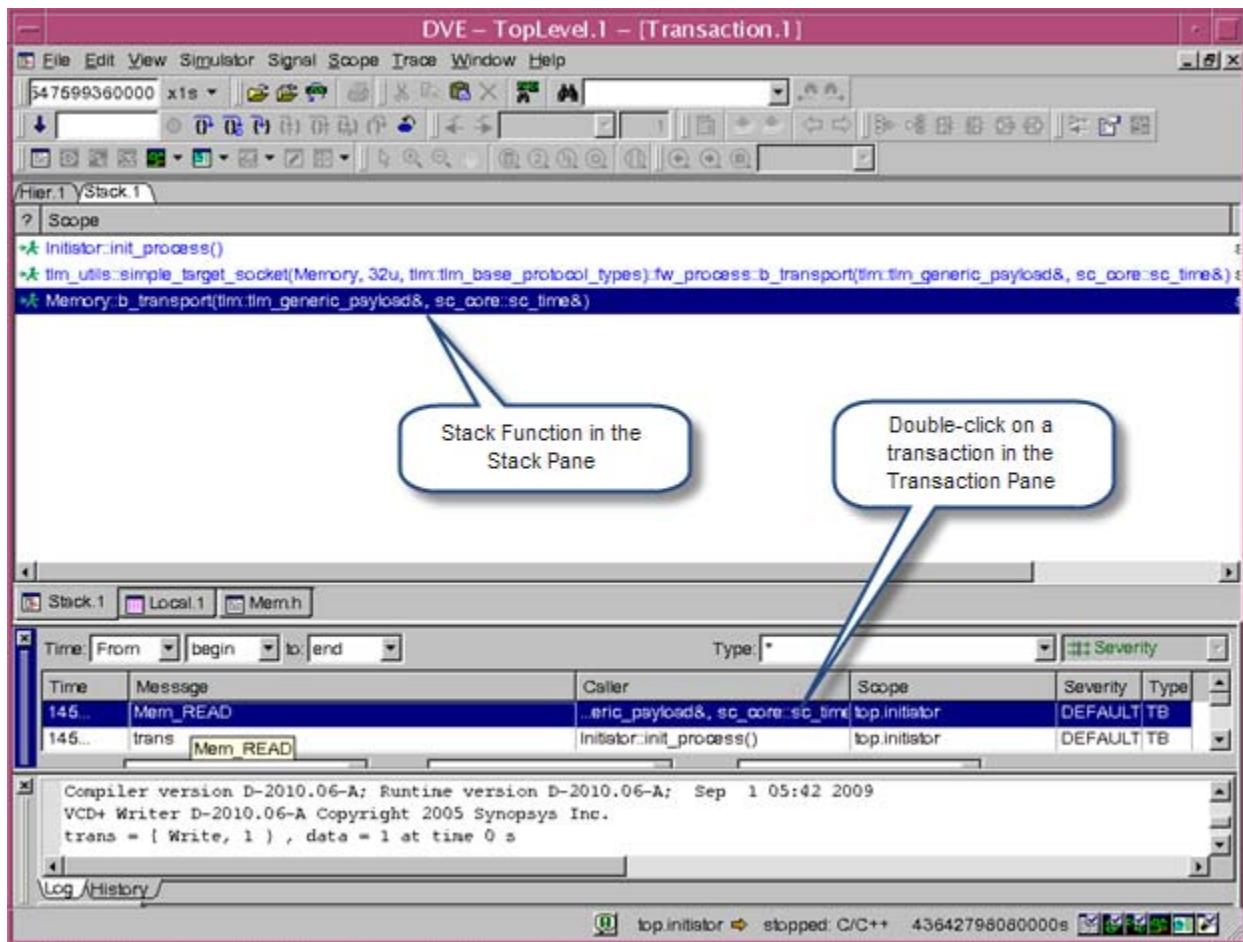
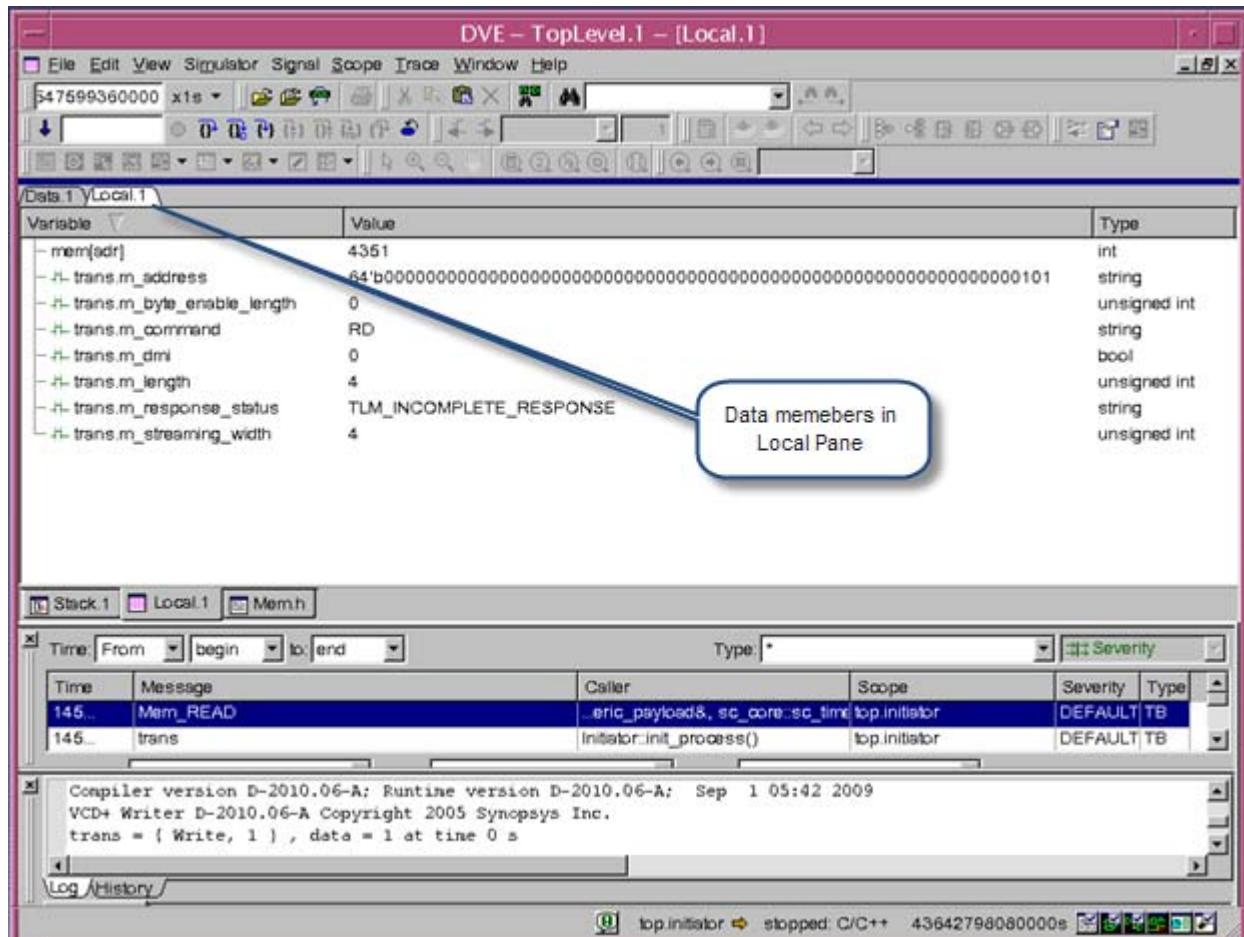


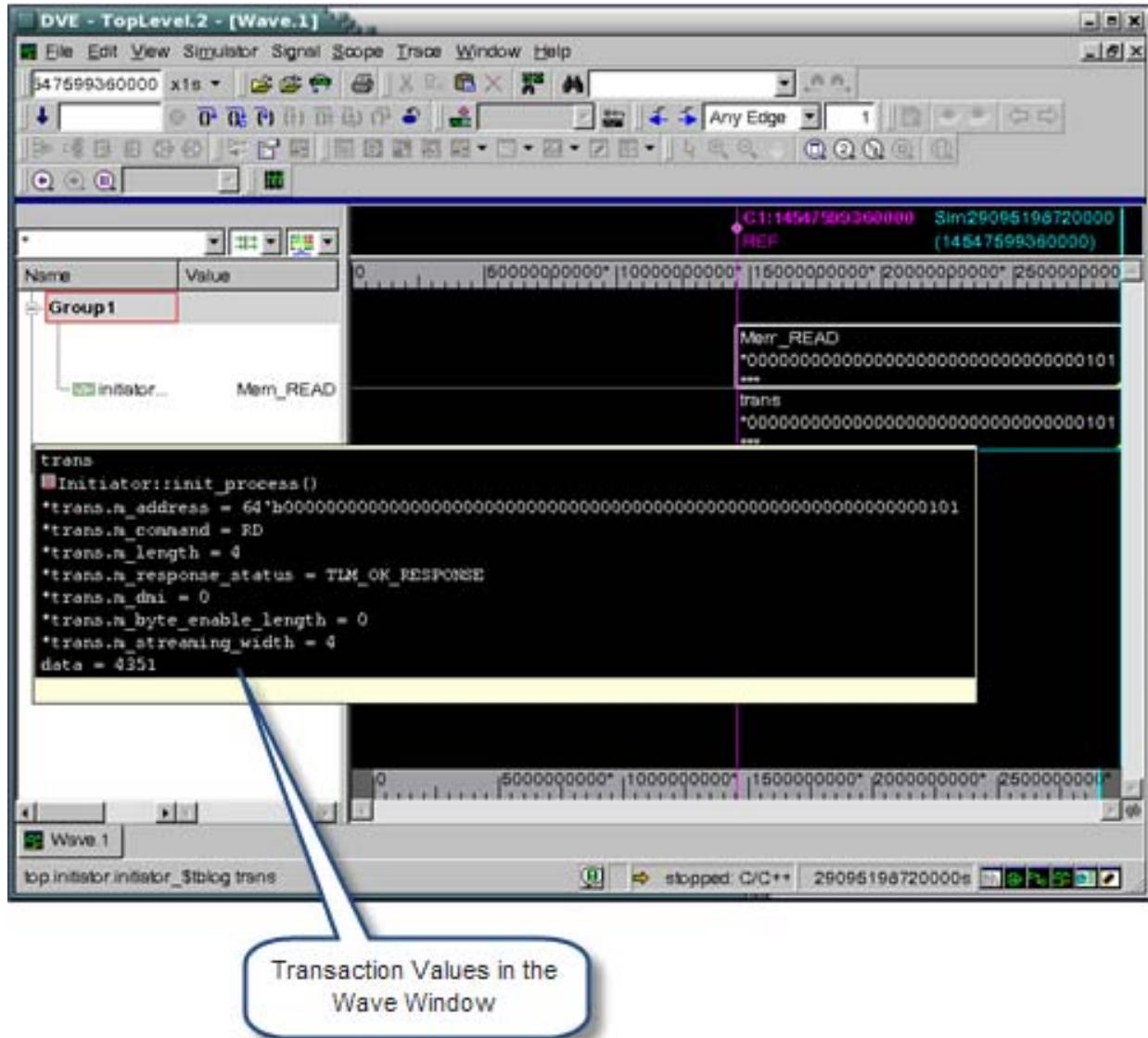
Figure 12-10 Data Members in the Local Pane



8. Hover the mouse pointer over the transaction to view its transaction values in the Wave View.

Note:

You can add any transaction in the Wave View.



Using msglog

DVE supports recording, displaying, and debugging of transactions for SystemVerilog and SystemC/C++. In SystemVerilog, transactions are captured using the `$vcplusmsglog` API. In the

SystemC/C++ domain, transactions are also captured with the msglog API. This document describes how to use the msglog API inside SystemC/C++ source code.

The SystemC msglog API in SystemC provides better transaction level debugging in DVE. It allows modeling and tracking of transactions on multiple streams. It can be used to define, create (start, extend, or finish) and describe transactions including relationships.

Usage

The following syntax describes the usage of the SystemC msglog API:

```
sc_snps::msglog [<< stream_name [<< stream_scope]
    << sc_snps::MSG_T [<< msg_name]
    << sc_snps::MSG_S [<< msg_header [<< msg_body
        [<< msg_body_more...]^n ]
    << sc_snps::MSG_R [<< relation]
    [<< [[stream_scope.]stream_name.]msg_name]
    [<< MSGLOG_VAR(<var>)]^n
    << sc_snps::end;

stream_scope ::= <SV identifier literal or string var>
stream_name ::= msg_name ::= <SV simple (non-escaped)
identifier literal or string var>

msg_header ::= msg_body ::= <html string literal or string
var>
relation ::= <Any literal string or var>
```

Note:

`sc_snps::msglog` is defined in the `tli_msglog.h` file. Its arguments are order dependent, as shown in below table:

Table 0-1. Arguments of `sc_snps::msglog`

Argument	Description	Example
<code>sc_snps::msglog << ...</code>	Starts the collection of arguments. The value given before <code>sc_snps::MSG_T</code> goes into {Streamname}	
<code>sc_snps::MSG_T</code>	Defines the message type. Value given after <code>sc_snps::MSG_T</code> goes into {Msg_label}	<code>sc_snps::XACTION,</code> <code>sc_snps::DEBUG</code>
<code>sc_snps::MSG_S</code>	Defines the message severity. The first string object after <code>sc_snps::MSG_S</code> goes into {Header}, everything else given after <code>sc_snps::MSG_S</code> goes into {Body}	<code>sc_snps::TRACE,</code> <code>sc_snps::ERROR</code>
<code>sc_snps::MSG_R</code>	Defines the message relation. The first string object after <code>sc_snps::MSG_R</code> goes into {Relation-Label} and the optional second string before <code>MSGLOG_VAR</code> goes into the <code>relation_target</code> , which is "[[stream_scope.] stream_name.] msg_name".	<code>sc_snps::START,</code> <code>sc_snps::FINISH</code>
<code>MSGLOG_VAR</code>	All variables for which you want to see transactions in the Transaction pane should be defined in <code>MSGLOG_VAR</code> .	<code><<MSGLOG_VAR (*trans)</code>
<code>... << sc_snps::end</code>	Finishes the collection of arguments, and writes the data to VPD.	

- `MSG_T` (Message Type) is of type enum. It consists of the following values .

```
enum E_MSG_T {
```

```

    FAILURE = 0x0001,
    NOTE = 0x0002,
    DEBUG = 0x0004,
    REPORT = 0x0008,
    NOTIFY = 0x0010,
    TIMING = 0x0020,
    XHANDLING = 0x0040,
    XACTION = 0x0080,
    PROTOCOL = 0x0100,
    COMMAND = 0x0200,
    CYCLE = 0x0400
} ;

```

- **MSG_S (Message Severity)** is of type enum. It consists of the following values.

```

enum E_MSG_S {
    FATAL = 0x0001,
    ERROR = 0x0002,
    WARNING = 0x0004,
    NORMAL = 0x0008,
    TRACE = 0x0010,
    DEBUGS = 0x0020,
    VERBOSE = 0x0040,
    HIDDEN = 0x0080,
    IGNORE = 0x0100
} ;

```

- **MSG_R (Message Relation)** is of type enum. It consists of the following values.

```

enum E_MSG_R {
    START = 0x0001,
    FINISH = 0x0002,
    PRED = 0x0004,
    SUCC = 0x0008,
    SUB = 0x0010,
    PARENT = 0x0020,
    CHILD = 0x0040,
    XTEND = 0x0080,
    USER = 0x0100
}

```

```
 } ;
```

You must include the `tli_msglog.h` header file to use `sc_snps::msglog`. This file is located at the following path:

```
$VCS_HOME/etc/systemc/tlm/tli/tli_msglog.h
```

To make the header visible, compile your SystemC source file with the `tlm2` option, as shown in following example:

```
syscan ... -tlm2 ... myfile.cpp
```

Alternatively, add the include path as shown below:

```
syscan ... -cflags -I$VCS_HOME/etc/systemc/tlm/tli  
... myfile.cpp
```

Example

```
=====Mem.h=====  
#define SC_INCLUDE_DYNAMIC_PROCESSES

#include "systemc"
using namespace sc_core;
using namespace sc_dt;
using namespace std;
#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h"
#include "tlm_utils/simple_target_socket.h"
#include "tli_tblogger.h"
using namespace sc_snps;
#include "tli_msglog.h" //msglog API is declared in this file
SC_MODULE(Initiator)
{
    tlm_utils::simple_initiator_socket<Initiator> socket;
    SC_CTOR(Initiator)
    : socket("socket") {
        SC_THREAD(init_process);
    }
    void init_process()
```

```

    {
        tlm::tlm_generic_payload* trans = new
        tlm::tlm_generic_payload;
        sc_time delay = sc_time(20, SC_NS);
        for (int i = 1; i < 40; i += 4)
        {
            tlm::tlm_command cmd =
            static_cast<tlm::tlm_command>(rand() % 2);
            if (cmd == tlm::TLM_WRITE_COMMAND) data = i;
            trans->set_command(cmd);
            trans->set_address(i);
            trans->set_data_ptr(reinterpret_cast<unsigned
char*>(&data));
            trans->set_data_length(4);
            trans->set_streaming_width(4);
            trans->set_byte_enable_ptr(0);
            trans->set_dmi_allowed(false);
            trans->set_response_status(
            tlm::TLM_INCOMPLETE_RESPONSE);
            socket->b_transport(*trans, delay);

            cout << "trans = { " << (cmd ? "Write" : "Read") << ", "
            << hex << i << " } , data = " << hex << data << " at time "
            << sc_time_stamp() << endl;

            wait(delay);
        }
    }

    int data;
};

SC_MODULE(Memory)
{
    tlm_utils::simple_target_socket<Memory> socket;

    SC_CTOR(Memory)
    : socket("socket")
    {
        // Register callback for incoming b_transport interface
        method call

```

```

socket.register_b_transport(this, &Memory::b_transport);

// Initialize memory with random data
for (int i = 0; i < 256; i++)
    mem[i] = 0x1010 | (rand() % 256);
}

// TLM-2 blocking transport method
virtual void b_transport( tlm::tlm_generic_payload& trans,
sc_time& delay )
{
    tlm::tlm_command cmd = trans.get_command();
    sc_dt::uint64 adr = trans.get_address() ;
    unsigned char* ptr = trans.get_data_ptr();
    unsigned int len = trans.get_data_length();
    unsigned char* byt = trans.get_byte_enable_ptr();
    unsigned int wid = trans.get_streaming_width();

    if ( cmd == tlm::TLM_READ_COMMAND )
    {
        memcpy(ptr, &mem[adr], len);
        sc_snps::msglog << "stream"
            << sc_snps::NOTE << "CYCLE"
            << sc_snps::TRACE << "write Mem" << "writing
mem transactions"
            << sc_snps::START
            << MSGLOG_VAR(trans)
            << end;
    }

    else if ( cmd == tlm::TLM_WRITE_COMMAND )
    {
        memcpy(&mem[adr], ptr, len);
        sc_snps::msglog << "stream"
            << sc_snps::NOTE << "CYCLE"
            << sc_snps::TRACE << "write Mem" <<
"writing mem transactions"
            << sc_snps::FINISH
            << MSGLOG_VAR(trans)
            << end;
    }
}

```

```

        // Obliged to set response status to indicate successful
completion
        trans.set_response_status( tlm::TLM_OK_RESPONSE );
    }

    int mem[256];
};

SC_MODULE(Top)
{
Initiator initiator;
Memory     memory;

SC_CTOR(Top) : initiator("initiator"), memory("memory")
{
    initiator.socket.bind( memory.socket );
}
};

int sc_main(int argc, char* argv[])
{
    Top top("top");
    sc_start();
    return 0;
}

=====
=====Mem.cpp=====
#include "Mem.h"
=====
```

To compile this example code, use the following commands:

```

./clean.csh

syscan Mem.cpp -cflags -g -tlm2
vcs sc_main -cflags -g -debug_all
simv
```

To run this example code, use the following commands:

```
./simv -gui&
```

SystemC msglog On/Off Control

The SystemC `msglog` class supports a global mechanism to turn `msglog` dumping on and off.

Disabling msglog-based recording

You can use the following API call to disable `msglog`-based recording:

```
sc_snps::msglogoff();
```

Enabling msglog-based recording

You can use the following API call to enable `msglog`-based recording:

```
sc_snps::msglogon();
```

Enabling or disabling msglog-based recording by passing an argument

You can use the following API call to pass in an argument, for example, from a variable:

```
sc_snps::msglogon(bool)
```

where,

`sc_snps::msglogon()` is equivalent to
`sc_snps::msglogon(true)` or `sc_snps::msglogon(1)`

and

`sc_snps::msglogoff()` is equivalent to
`sc_snps::msglogon(false)` or `sc_snps::msglogon(0)`

Note:

Default setting of msglog recording is `msglogon()`.

Recorded Information in VPD File

The following information is recorded in VPD file:

- Current simulation time
- Call stack
- Constant string message
- Message Labels
- Message Types
- Message Severities
- Message Relation
- Dynamic or Static variables

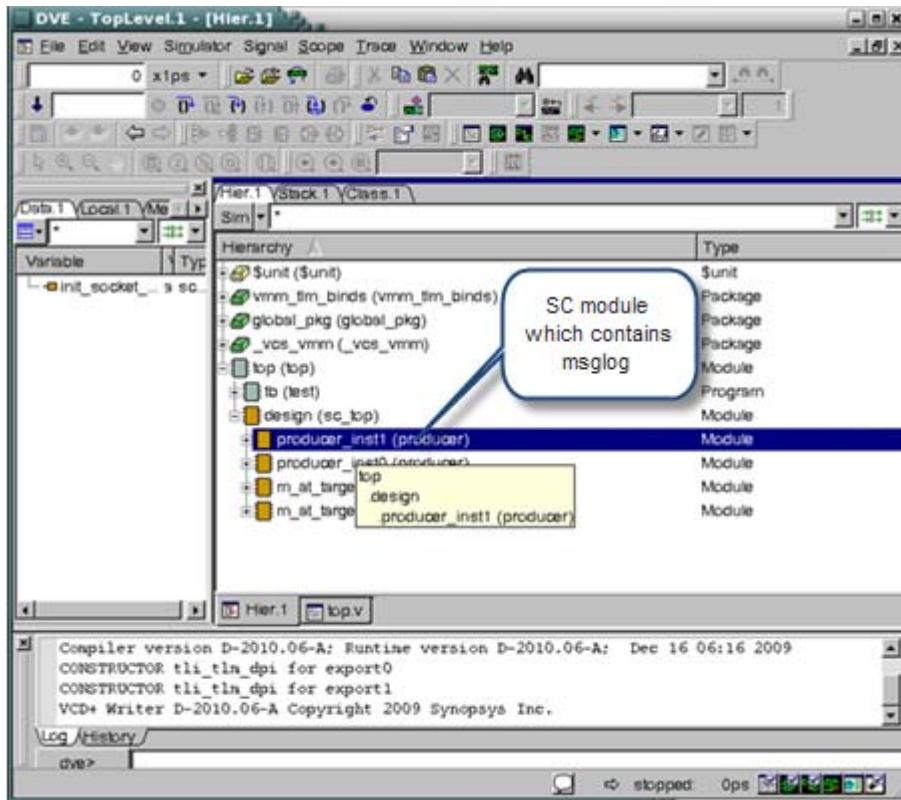
Note:

- If the space within the waveform is limited, hovering the mouse pointer over a certain transaction shows its full text in a separate window.
- If you double-click a log record, it opens the call stack in the Stack pane.

Viewing the Recorded Information in DVE

To view the transactions using msglog

1. Open simv in DVE using the %simv -gui & command.
2. In the Hierarchy pane, double-click on the SystemC code which contains msglog.



3. Run the simulation and stop in the portion of the code, where msglog API's are present.
4. Click **Window > Panes > Transaction** to open the Transaction pane:
5. In the Transaction pane, double-click on the desired transaction, to view:
 - Stack function in the Stack Pane, as shown in [Figure 12-11](#).

- Data members in the Local pane, as shown in Figure 12-12.
 - Hover the mouse pointer over the transaction to view its transaction values in the Wave View, as shown in Figure 12-13.

Figure 12-11 Stack Function in the Stack Pane

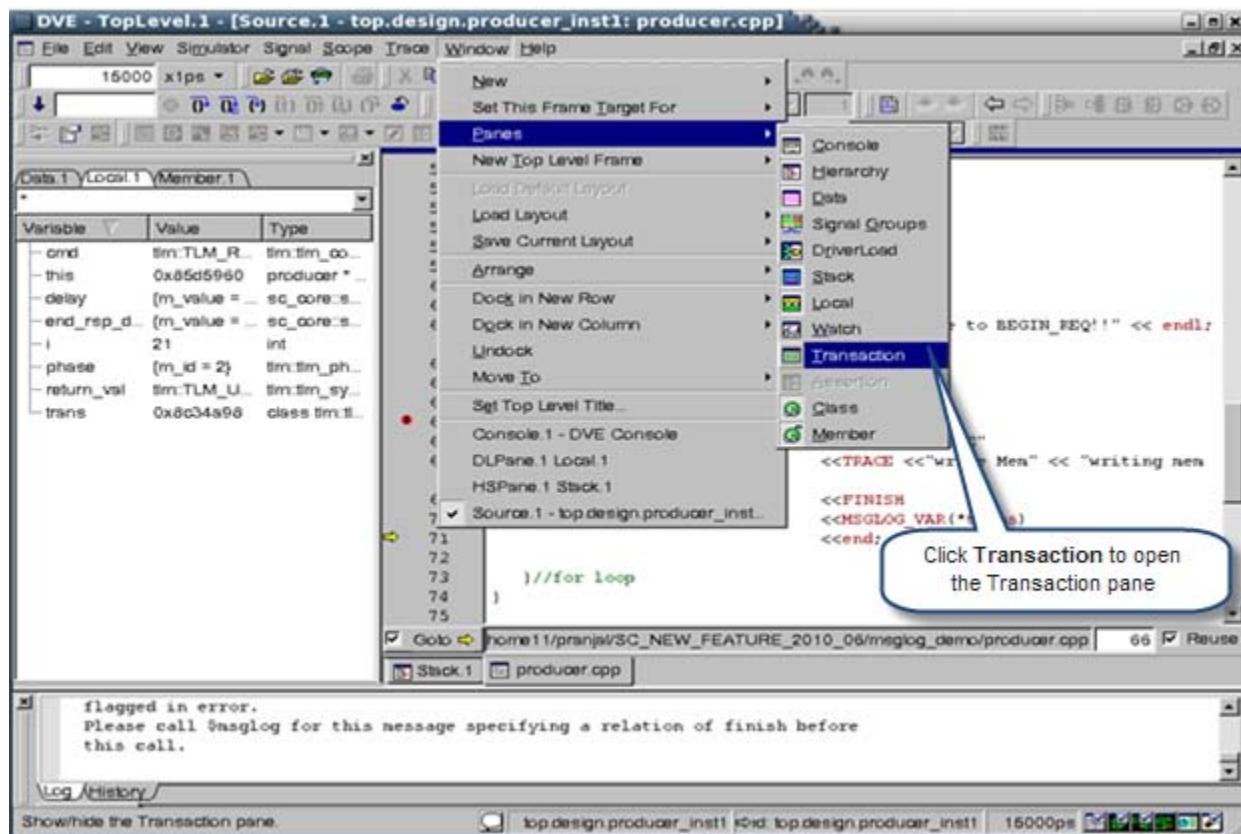


Figure 12-12 Data Members in the Local Pane

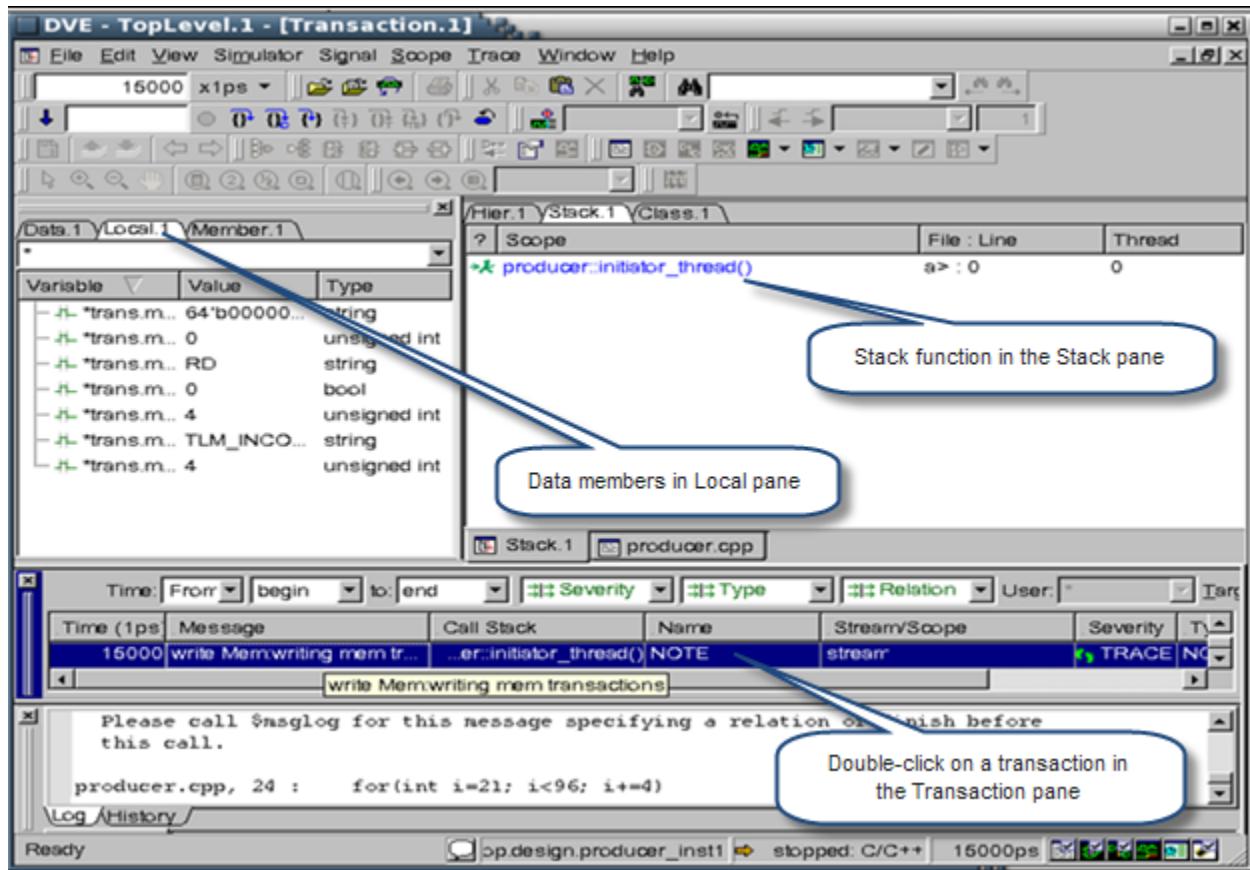
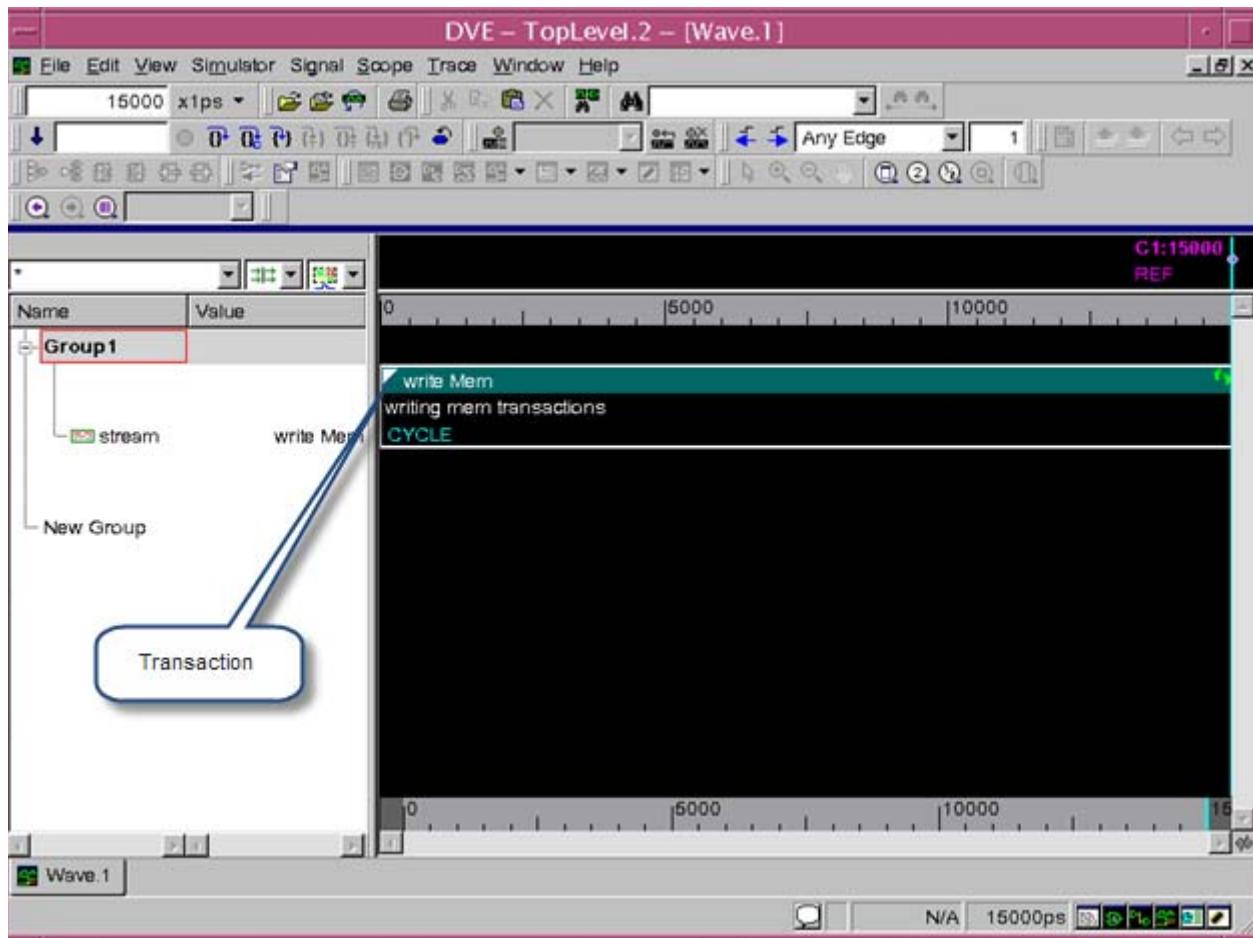
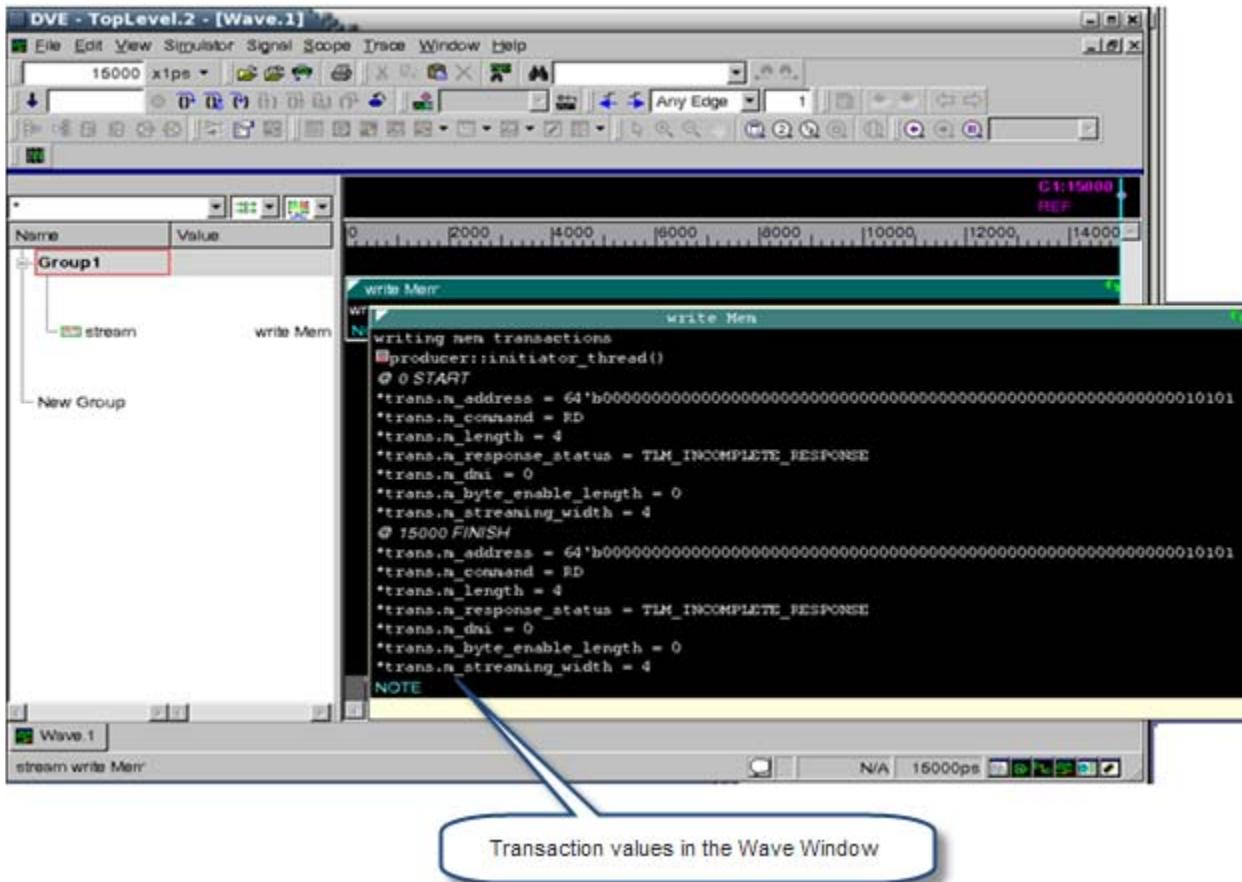


Figure 12-13 Transaction Values in the Wave View



Note: You can add any number of transactions in the Wave view



Limitations

Following is the limitation of SystemC msglog class:

The stack shows SystemC/C++ frames with their functions names only. Source file or Source line information is not available.

Viewing NTB-OV Variables using tblog/msglog

You can view NTB/OV variables in DVE using the tblog and msglog system tasks. The syntax for tblog/msglog for NTB/OV is similar to \$vcaplustblog/\$vcplusmsglog, as shown below:

\$vcaplustblog for SV:

```
$vcaplustblog(-1,"Write", i);
```

tbog for OV:

```
tbog(-1,"Write",i);
```

\$vcplusmsglog for SV:

```
$vcplusmsglog(OpStream, XACTION, "SYSTEM", NORMAL, "System  
Msg", "I am a parent", START, i);
```

msglog for OV:

```
msglog(OpStream, XACTION_T, "SYSTEM", NORMAL_S, "System  
Msg", "I am a parent", START_R, i);
```

In the syntax, note the text in mauve. This is the type of severity or relation.

An example for this feature has been copied in the \$VCS_HOME/doc/examples/debug/transaction_debug/ntb_ov_msglog directory.

For msglog, #include "msglog.vrh" file is required. This file is available in \$VCS_HOME/include.

The representation in DVE is as follows:

Figure 12-14 Visualization of msglog in Wave view

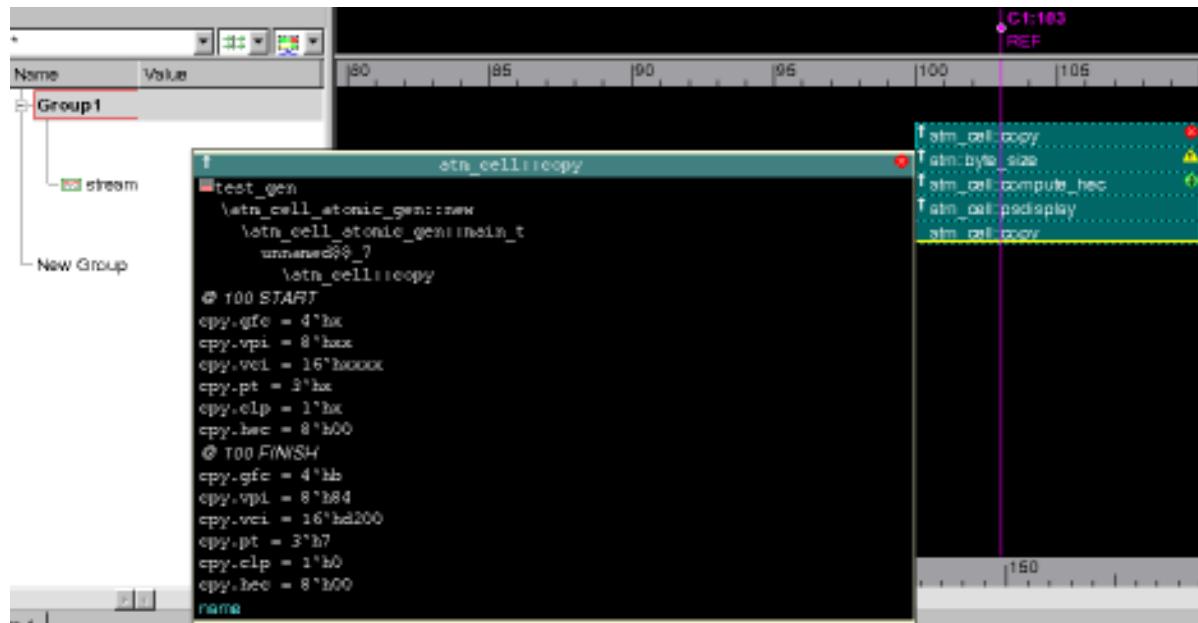


Figure 12-15 Visualization of tblog in Wave view

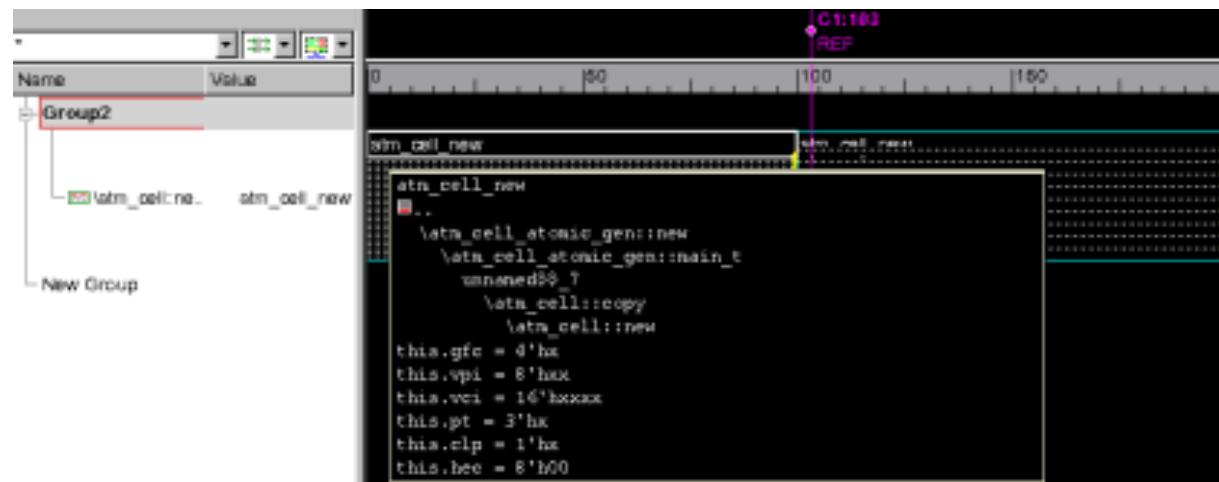
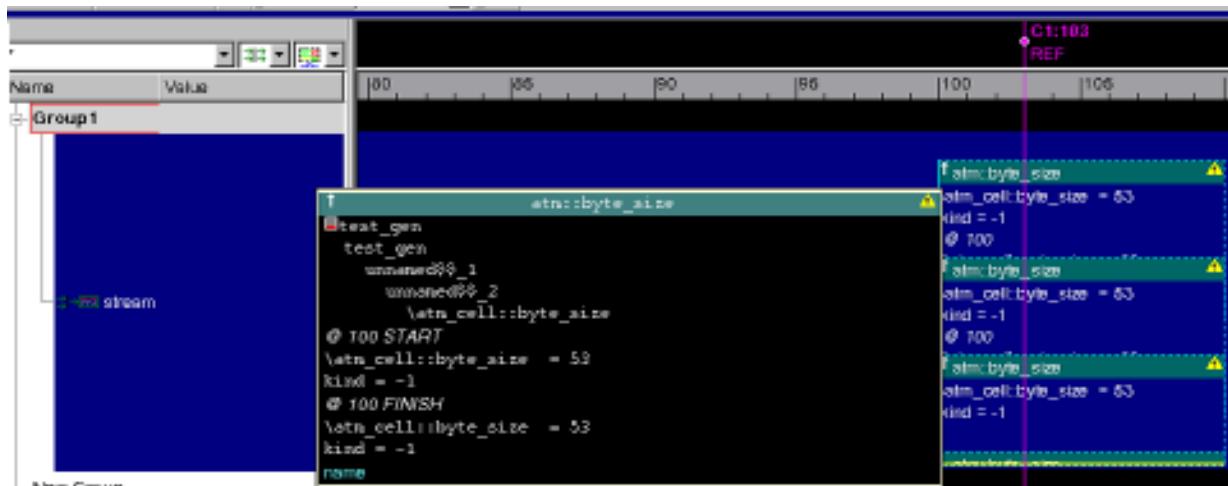


Figure 12-16 Visualization after applying the filter: filter by severity warning



13

Using the C, C++, and SystemC Debugger

This chapter describes debugging VCS and VCS MX designs that include C, C++, and SystemC modules with DVE. It contains the following sections.

- “[Getting Started](#)”
- “[Commands Supported by the C Debugger](#)”
- “[Common Design Hierarchy](#)”
- “[Interaction with the Simulator](#)”
- “[Configuring CBug](#)”
- “[VPD Dumping for SC_FIFO Channels](#)”
- “[Supported platforms](#)”
- “[Example: A Simple Timer](#)”

- “Viewing SystemC Source and OSCI Names in DVE”
 - “Using CBug to Display Instance Name of Target Instance in TLM-2.0”
 - “CBug Stepping Improvements”
-

Getting Started

This section describes how to get started using DVE to debug designs that include C, C++, and SystemC modules.

Using a Specific gdb Version

Debugging of C, C++ and SystemC source files relies upon a gdb installation with specific patches. This gdb is shipped as part of the VCS image and is used per default when CBug is attached. No manual setup nor installation of gdb is needed.

Attaching the C-Source Debugger in DVE

You can debug designs containing C-source modules with or without the C debugger running. However, you must attach the C-source debugger to view and debug C-source code within the design.

Note that the `-debug_all` flag enables line breakpoints for the HDL (Verilog, VHDL) parts only not for C files. You must compile the C files with the `"-g"` C compiler option as follows:

- When invoking the C/C++ compiler directly:

```
gcc ... -g ...
```

```
g++ ... -g ...
```

- When invoking one of the VCS tools:

```
vcs     ... -cflags -g ...
syscan ... -cflags -g ...
syscsim ... -cflags -g ...
```

The following steps describe attaching the C-source debugger to run DVE to debug VCS or VCS MX (Verilog, VHDL, and mixed) simulations containing C, C++, and SystemC source code.

1. Compile your VCS or VCS MX with C, C++, or SystemC modules as you normally would, making sure to compile all C files you want to debug in DVE.

For example, with a design with Verilog on top of a C or C++ module:

```
gcc -g [options] -c my_pli_code.c
vcs +vc -debug_all -P my_pli_code.tab my_pli_code.o
```

Or with a design with Verilog on top of a SystemC model:

```
syscan -cpp g++ -cflags "-g" my_module.cpp:my_module
vcs -cpp g++ -sysc -debug_all top.v
```

Note that you must use `-debug` or `-debug_all` to enable debugging.

2. Open DVE.
3. Click  to start the simulation.
4. Select **Simulator > C/C++ Debugging**.

Or

Enter `cbug` on the console command line.

Debugging of C, C++, and SystemC source code is enabled and you see the following message in the console History tab:

```
CBug - Copyright Synopsys Inc. 2003-2006.
```

Note:

The C-source debugger will automatically attach when you set a breakpoint in a C / C++ file (extensions .c, .cc, .cpp, and .h are recognized).

Detaching the C-source Debugger

You can detach and reattach the C-source debugger at any time during your session.

To detach the C-source debugger, toggle the debugger off by selecting **Simulator > C/C++ Debugging** or enter the following command on the console:

```
cbug -detach
```

Displaying C Source Files in the Source View

There are three ways to display a C source file in the Source view:

- Automatically when the simulation stops in a given C file due to a breakpoint or cross-step
- Explicitly through **File > Open**.

Note:

Select **C/C++ files** as File type in the Open Source File dialog box.

Double-click a SystemC process in the design hierarchy. DVE automatically opens the source file if the file can be found and was compiled with `-g`; otherwise you are prompted to proceed.

Commands Supported by the C Debugger

These commands are supported by the C debugger:

- `continue`
- `run`
- `next`
- `next -end`
- `step`
- `get variable_name` (Returns the variable value)
- `finish`
- `stack`
- `dump` (of SystemC objects)
- `cbug`

Note:

Save/restore is also supported for simulations that contain SystemC or other user-written C/C++ code (e.g. DPI, PLI, VPI, VhPI, DirectC), however, there are restrictions. See the description of the 'save' and 'restore' command in the UCLI User Guide for full details. CBug has to be detached during a 'save' or 'restore' command but can be re-attached afterwards.

The following commands are not supported:

- `force` (applied to C or SystemC signals)
- `release` (applied to C or SystemC signals)
- `drivers` (applied to C or SystemC signals)
- `loads` (applied to C or SystemC signals)

Note:

This section uses the full UCLI command names. If you are using a command alias file, such as the Synopsys-supplied alias file, enter the alias on the UCLI command line. See the *UCLI User Guide* for more information.

scope Command

The `scope` command is supported for SystemC instances.

show Command

`show [-instances|-signals|-ports]` is supported for SystemC instances, for example "`show -ports top.inst1`". Any other type such as `-scopes`, `-variables`, `-virtual` is not supported for SystemC instances. A radix is ignored.

change Command

The `change` command is supported within these two strict limitations:

- Only variables that are visible in the current scope of the C function (e.g. local variables, global variables, class members.) can be changed. Hierarchical path names like `top.inst1.myport` are not supported.

- The type must be a simple ANSI type like `int`, `char`, `bool`. Changing SystemC bit-vector types like `sc_int<>` or user-defined types is not supported. Any attempt to set an unsupported datatype will trigger the error message "Unsupported type for setting variable".

stack Command

When you are stopped in C code, then you can see the stack list. Each entry of the list tells the source file, line number, function name. The function where you are stopped right now appears at the top of the list. If the source code for a given function has been compiled without compiler flag `-g`, then the file/line number information is not available. CBug selects `without-g.txt` in this case.

Command `stack -up | -down` move the active scope up or down. The source file corresponding to the active scope is shown and get command applies to this scope.

Accessing C/C++/SystemC Elements with the get Command

Note:

When you use the "get" command for SystemC variables, the value of radix types hex and bin is represented with a prefix '0' and optionally with a '0x' or '0b' format specifier. The prefix '0' is added if the value field does not start with a '0'. This is visible in the UCLI get output and in DVE.

For example, a 16bit value of ('C' notation) `0x8888` appears as (SystemC notation) `0x08888`, and a decimal '3' (11) in a two bit variable appears as '`0b011`' in binary radix.

When stopped at a C source location, certain elements are visible and can be queried with the `ucli::get` command:

- Function arguments
- Global variables
- Local variables
- Class members (if the current scope if a method)
- Ports, sc_signal and plain members of SystemC modules anywhere within the combined HDL+SystemC instance hierarchy.
- Arbitrary expression including function calls, pointers, array indices etc. Note that some characters such as '[]' need to be enclosed by '{}' or escaped with '\' otherwise Tcl will interpret them.

Examples

- ucli::get myint
- ucli::get this->m_counters
- ucli::get {this->m_counters[2]}
- ucli::get strlen(this->name)

The <name> given with a synopsys::get <name> argument refers to the scope in the C source where the simulation stopped (the active scope). This is important to keep in mind because C source may have multiple objects with the same name but in different scopes. Which one is visible depends on the active scope. This means that <name> may not be accessible anymore once you step out of a C/C++ function.

Accessing SystemC Elements with the get Command through an hierarchical Path Name

The argument of `synopsys::get` may refer to an instance within the combined HDL/SystemC instance hierarchy. All ports, `sc_signals` and also all plain member variables of an SystemC instance can be accessed with `synopsys::get` at any time. Access is possible independent of where the simulation is currently stopped, even if it is stopped in a different C/C++ source file or not in C/C++ at all.

Example

Assume this instance hierarchy

```
top          (Verilog)
middle       (Verilog)
bottom0     (SystemC)
```

where "bottom0" is an instance of this SC module:

```
SC_MODULE(Bottom) {
    sc_in<int> I; // SC port
    sc_signal<sc_logic> S; // SC signal
    int PM1; // "plain" member variable, ANSI type
    str PM2; // "plain" member variable, user-def type
};
struct str {
    int a;
    char* b;
};
```

These accesses are possible:

```
synopsys::get top.middle.bottom0.I
synopsys::get top.middle.bottom0.S
synopsys::get top.middle.bottom0.PM1
synopsys::get top.middle.bottom0.PM2
synopsys::get top.middle.bottom0.PM2.a
```

Access is possible at any point in time, independent of where the simulation stopped. Note that this is different to accessing local variable of C/C++ functions. They can only be accessed if the simulation is stopped within that function.

Note that accessing plain member variables of SystemC instances is only possible with `synopsys::get` but not with `synopsys::dump`.

Format / Radix:

The C debugger will ignore any implicitly or explicitly specified radix. The format of the value returned is exactly as it is given by gdb (only SystemC data types are specially dealt with). Besides integers, you can also query the value of pointers, strings, structures, or any other object that gdb can query.

SystemC Datatypes:

The C debugger offers specific support for SystemC datatypes, for example, an `sc_signal<sc_bv<8>>`. When you do a `print` of such a value, gdb usually returns the value of the underlying SystemC data structure that is used to implement the data type. This is normally by no means what you want to see and is generally useless. The C debugger recognizes certain native SystemC data types and prints the value in an intuitive format. For example, it will print the value of the vector in binary format for an `sc_signal<sc_bv<8>>`.

The following native SystemC types are recognized.

Templatized channel types `C<T1>`:

```
C := { sc_in_clk, sc_in, sc_inout, sc_out, sc_signal,  
ccss_param }
```

```
T1 := { bool, [[un]signed] char, [unsigned] [long|short] int,
        [[long] double] float, sc_logic, sc_lv, sc_bit, sc_bv,
        sc_[u]int, sc_int_base, sc_big[u]int, sc_[un]signed,
        sc_fxval[_fast], sc_[u]fix[ed][_fast], sc_string,
        char*, void*, struct X* }
```

When the value of an object O of such a type C is to be printed, then the C debugger prints the value of O.read() rather than O itself.

Native SystemC data types:

```
T2 := { sc_logic, sc_lv, sc_bit, sc_bv,
        sc_[u]int, sc_int_base, sc_big[u]int, sc_[un]signed,
        sc_fxval[_fast], sc_[u]fix[ed][_fast], sc_string }
```

The C debugger prints values of these data types in an intuitive format. Decimal format is taken for sc_[u]int, sc_int_base, sc_big[u]int, sc_[un]signed, binary format for sc_logic, sc_lv, sc_bit, sc_bv.

Example

SystemC source code:

```
sc_in <int> A
sc_out<sc_bv<8>>B;
sc_signal <void*>;
int D;
synopsys::get A
17
synopsys::getB
01100001
synopsys::getC
0x123abc
synopsys::getD
12
```

Changing Values of SystemC and Local C Objects with synopsys::change

CBug supports changing the values of C variables and SystemC `sc_signal` using the UCLI `change` command.

Example:

```
change my_var 42
change top.inst0.signal_0 "1100ZZZZ"
```

Changing SystemC Objects

The value change on any SystemC `sc_signal`, either from C++ code or using the `change` command, modifies only the next value, but not the current value.

The current value is updated only with the next SystemC delta cycle. Therefore, you may not view the effect of the `change` command directly. If you query the value with the UCLI `get` command, then you will see the next value because the `get` command retrieves the next value, but not the current value for `sc_signal`.

However, accessing the `sc_signal` with `read()` inside the C++ code, displays the current value until the next SystemC delta cycle occurs. CBug generates a message explaining that the assignment is delayed until the next delta cycle.

Note:

A change may compete with other accesses inside the C++ code. If a signal is first modified by the `change` command, but later on, if a `write()` happens within the same delta-cycle, then `write()` cancels the effect of the earlier `change` command.

The format of the value specified with the `change` command is defined by the corresponding SystemC datatype. ANSI integer types expect decimal literals. Native SystemC bit-vector types accept integer literal and bit-string literals.

Examples

```
SystemC module 'top.inst_0' has
sc_signal<int>           sig_int
sc_signal<sc_int<8> > sig_sc_int
sc_signal<sc_lv<40> > sig_sc_lv

change top.inst_0.sig_int    42      // assign decimal 42

change top.inst_0.sig_sc_int 0d015    // assign decimal 15
change top.inst_0.sig_sc_int 0b0111ZZXX //assign bin value
change top.inst_0.sig_sc_int 0x0ffab   // assign hex value
change top.inst_0.sig_sc_int 15        // assign decimal 15
change top.inst_0.sig_sc_int -15       // assign decimal -15

change top.inst_0.sig_sc_lv  0d015    // assign decimal 15
change top.inst_0.sig_sc_lv -0d015    // assign decimal -15
change top.inst_0.sig_sc_lv 0b0111ZZXX // assign bin value
change top.inst_0.sig_sc_lv 0x0ffab   // assign hex value
change top.inst_0.sig_sc_lv 0011ZZXX // assign bin value
```

Supported Datatypes

The following datatypes are supported:

- All types of ANSI integer types, for example, `int`, `long long`, `unsigned char`, `bool`, and so on.
- Native SystemC bit-vector types: `sc_logic`, `sc_lv`, `sc_bv`, `sc_int`, `sc_uint`, `sc_bigint`, and `sc_biguint`.

Limitations of Changing SystemC Objects

- Only SystemC objects `sc_signal` and `sc_buffer` can be changed. Changing the value of ports, `sc_fifo`, or any other SystemC object is not supported.
- You must address SystemC objects by their full hierarchical path name or by a name relative to the current scope.

Example:

```
scope top.inst1.sub_inst
change top.inst0.signal_0 42 // correct
change signal_0 42 // wrong, local path not supported
for SystemC

scope top.inst0
change signal_0 43 // correct, scope + local
```

- User-defined datatypes are not supported.
- A permanent change (`force -freeze`) is not supported.

Changing Local C Variables

Local C variables are the variables that are visible within the current C/C++ stack frame. This is the location where the simulation stops. However, you can change the frame by using the UCLI `stack -up` or `stack-down` command, or by double-clicking on a specific frame in the DVE stack pane.

Local C variables are the:

- Formal arguments of functions or methods
- Local variables declared inside a function or method

- Member variables visible in the current member function and global C variables

Example

```

100 void G(int I)
101 {
102     char* S = strdup("abcdefg");
103     ...
104 }
105
106 void F()
107 {
108     int I=42;
109     G(100);
110     ...
111 }
```

Assume that the simulation stops in function G at line 103.

```

change I 102 //change formal arg I from G defined in line 100
change I 0xFF
change S "hij kl"
change {S[1]} 'I'
scope -up
change I 42 // change variable I from F defined in line 108
```

Limitations of Changing Local C Variables

- You must attach CBug.
- You can change only simple ANSI types like: bool, all kinds of integers (for example, signed char, int, long long), char*, and pointers. Arrays of these types are supported if only a single element is changed.
- The format of the value is defined by gdb, for example, 42, 0x2a, 'a', "this is a test".

- SystemC types are not supported, for example, `sc_int`, `sc_lv` is not supported.
- STL types such as `std::string`, `std::vector`, and so on, are not supported.
- Using the full path name (for example, `top.inst_0.my_int`) is not supported. You can use only local names (for example, `my_int` or `this->my_int`).

Using Breakpoints

You can set line breakpoints on C / C++ / SystemC source files using the Source view, the Breakpoints dialog box, or the command line. Breakpoints in C-source code support line breakpoints.

Set a Breakpoint from the Breakpoints Dialog Box

You can set a line, time, or signal breakpoint using the Breakpoints dialog box. See the section “[Managing Breakpoints from the Dialog Box](#)” on page 4-15 for more information.

Control Line Breakpoints in the Source view

You can control line breakpoints in the Source pane in two ways:

- Clicking on the circular breakpoint indicator in the line attribute area.
- Selecting a line breakpoint, right-clicking from the attribute area, then selecting a context-sensitive menu command.

For more information, see the section “[Control Line Breakpoints in the Source view](#)” on page 4-13.

Set a Breakpoint from the Command Line

To create a line breakpoint from the command line, enter the stop command into the console command line using the following syntax:

```
stop -file filename -line linenumber
```

For example:

```
stop -file B.c -line 10
stop -file module.cpp -line 101
stop -in my_c_func
stop -in timer::clock_action()
```

Instance Specific Breakpoints

Instance specific breakpoints are supported with respect to SystemC instances only. Specifying no instance or instance name "-all" means to always stop, no matter what the current scope is,

If the debugger reaches a line in C, C++, SystemC source code for which a instance-specific breakpoint has been set, then it will stop only if the following two conditions are met:

- The corresponding function was called directly or indirectly from a SystemC SC_METHOD, SC_THREAD or SC_CTHREAD process.
- The name of the SystemC instance to which the SystemC process belongs matches the instance name of the breakpoint.

Note that C functions called through the DPI, PLI, DirectC or VhPI interface will never stop in an instance-specific breakpoint because there is no corresponding SystemC process.

You must use the name of the Systemc module instance and not the name of the SystemC process itself.

Breakpoints in Functions

You can also define a breakpoint by its C/C++ function name with the

```
stop -in function  
command.
```

Examples

```
stop -in my_c_function  
stop -in stimuli::clock_action()
```

Restriction

If multiple active breakpoints are set in the same line of a C, C++ or SystemC source code file, then the simulation will stop only once.

Deleting a Line Breakpoint

To delete a line breakpoint, do either of the following:

- Click the red button in the Source view to disable the breakpoint.
- Select **View > Breakpoints**, select the breakpoint to delete, then click **Delete**.
- On the console command line, enter

```
stop -delete <index>
```

then press **Enter**.

Stepping Through C-source Code

Stepping within, into, and out of C code during simulation is accomplished using the `step` and `next` commands. Extra arguments to either `step` or `next`, such as `-lang` or `-thread` are not supported for C code. Only `next -end` is allowed.

Stepping within C Sources

You can step over a function call with `next` or step into a function with `step`.

Note:

Stepping into a function that was not compiled with `-g` is generally supported by gdb and also the C debugger. However, in some cases gdb becomes confused on where to stop next and may proceed further than anticipated. In such cases, it is recommended to set a breakpoint on a C source that should be reached soon after the called function finishes and then issue the command `synopsys ::continue`.

Use the `stack -up` command to open the source code location where you want to stop, set a breakpoint, and then continue.

Cross-stepping between HDL and C Code

Cross-stepping is supported in many but not all cases where C code is invoked from Verilog or VHDL code. These cases are supported:

- From Verilog caller into a PLI C function - Note that this is only supported for the "call" function, but not that "misc" or "check" function and also only if the PLI function was statically registered.

- From the PLI C function back into the Verilog caller.
- From Verilog caller into DirectC function and also back to Verilog.
- From VHDL caller into an VhPI "foreign" C function that mimics a VHDL function and also back to VHDL. Note that the cross-step is not supported on the very first occasion when the C function is executed. Cross-stepping is possible for 2nd, 3rd and any later call of that function.
- From Verilog caller into an export "DPI" C function and also back to Verilog.
- At the end of a Verilog export "DPI" task or function back into the calling C function. Note that this cross-step HDL > C is only possible if the Verilog code was reached via a cross-step from C->HDL in the first place.

All cross-stepping is only possible if the C code has been compiled with debug information (gcc -g).

Cross-stepping in and out of Verilog PLI Functions

When you steps through HDL code and come to a call of a user-provided C function, such as a PLI function like \$myprintf, then the `next` command will step over this function. But the `step` command will step into the C source code of this function.

Consequent `step/next` commands walk through the C function and finally you return to the HDL source. Seamless stepping HDL->C > HDL is thus possible. This feature is called cross-stepping.

Cross-stepping is supported only for function that meet this criteria:

- PLI function
- Statically registered through a tab file

- The call only (but not `misc` or `check`)

Cross-stepping into other Verilog PLI functions is not supported. However, an explicit breakpoint can be set into these function which will achieve the same effect.

Cross-Stepping in and out of VhPI Functions

Cross-stepping from VHDL code into a C function that is mapped through the VhPI interface to a VHDL function is supported with certain restrictions:

- Cross-step in is not possible on the very first occasion when the C function is executed. Only later calls are supported. A cross-step out of C back into VHDL code is always supported.
- Cross-stepping is not supported for C code mapped through the VhPI interface onto a VHDL *entity*.
- Cross-stepping from Verilog into a DirectC function is supported, also cross-step back out. There are no restrictions.
- Cross-stepping between [System]Verilog and import/export DPI functions is supported with a few restrictions:
- Cross-step from an import DPI function back into the calling Verilog source code is supported only if this DPI function was entered with a cross-step in the first place. That means doing continuously step commands will lead from the Verilog caller, into and through the import DPI function and back to the Verilog caller statement into the import DPI function, through that function and finally back into the calling Verilog statement.

However, if the DPI function was entered through a `run` command and the simulation stopped in the import C function due to a breakpoint, then the cross-step out of the import DPI function into the calling Verilog statement is *not* supported. The simulation will advance until the next breakpoint is reached.

- Cross-step from an export DPI task/function back into the calling C source code is supported only if this DPI task/function was entered with a cross-step in the first place. That means doing continuously step commands will lead from the C caller, into and through the import DPI task/function and back to the C caller.

However, if the export DPI task/function was entered through a `run` command and the simulation stopped in the export task/function due to a breakpoint, then the cross-step out of the export DPI function into the calling C statement is *not* supported. The simulation will advance until the next breakpoint is reached.

Cross-stepping from C into HDL

Stepping from C code (that is called as a PLI function) into HDL code is generally supported. There are two ways to do this.

- If the C function was reached by previously cross-stepping from HDL into C, then CBug is able to automatically transfer control back to the HDL side once you step out of the C function. In this case, just type `step` or `next` in C code.

- In all other cases, CBug is not able to detect that the C domain is exited and needs an explicit command to transfer control back to the HDL side. When you do a `step` or `next` command that leaves the last statement of a C function called from HDL, then the simulation will stop in a location that belongs to the simulator kernel. There will be usually no source line information available since the simulator kernel is generally not compiled with `-g`, so you will not see a specific line/file information. Instead, file `without-g.txt` will be displayed.

If this happens, you can proceed as follows:

`synopsys::continue` or `run`

or

`next -end`

The `continue` will bring you to the next breakpoint which could be in either HDL or C source code. The `next -end` command will stop as soon as possible in the next HDL statement or the next breakpoint in C code, whichever comes first. Again, use commands `synopsys::continue` or `synopsys::next -end` to proceed.

Cross-Stepping in and out of SystemC Processes

CBug offers specific support for stepping between SystemC or HDL processes:

- When you leave a function that defines a `SC_METHOD` process, then a `step` or `next` command will automatically stop in the next SystemC or HDL process, whatever comes next.

- Similarly, when you use the `step` or `next` command over a 'wait' statement that belongs to an SC_THREAD process, then you will stop in the next SystemC or HDL process, whatever comes next.
- Doing a `step` or `next` in Verilog or VHDL will automatically stop in a SystemC process if that process happens to be next SystemC or HDL process to be executed.

That means doing `step` or `next` repeatedly will follow to flow of SystemC and HDL processes in the exact order in which the simulator executes them.

Direct gdb Commands

You can send certain commands directly to the underlying gdb through UCLI command `cbug ::gdb`. The command will be executed right away and the UCLI command will return the response from gdb.

The command is

```
cbug ::gdb gdb-cmd
```

gdb-cmd is an arbitrary command accepted by gdb including an arbitrary number of arguments, for example `info sources`. Doing `cbug ::gdb` will automatically attach CBug, send `<gdb-cmd>` to gdb and return the response from gdb as the return result of the Tcl routine. The result may have one or multiple lines.

The routine returns successfully in most cases, even if gdb itself gives an error response. The routine gives an Tcl error response only when *gdb-cmd* has the wrong format, for example when it is empty.

Only a small subset of gdb commands are always allowed. These are commands that for sure do not change the state of gdb or simv, e.g. commands `show`, `info`, `disassemble`, `x`, etc. Other command `make cbug::gdb` return with error saying cannot execute this gdb command because it would break CBug.

Example:

```
ucli% cbug::gdb info sources
```

Source files for which symbols have been read in:

```
./pythag.c, rmapats.c, ctype-info.c, C-ctype.c, C_name.c,  
./../gcc/libgcc2.c
```

Source files for which symbols will be read in on demand:

```
ucli% cbug::gdb whatis pythag  
type = int (int, int, int)  
ucli%
```

Add Directories to Search for Source Files

This is directly done with the `gdb dir dir-name` command. For example:

```
ucli% gdb dir /u/joe/proj/abc/src
```

Use this command to check which directories are searched:

```
ucli% gdb show dir  
Source directories searched:  
/u/joe/proj/abc/src:$cdir:$cwd
```

Adding directories may be needed to locate the absolute location of some source files.

Example:

```
ucli% cbug::expand_path_of_source_file foo.cpp
    Could not locate full pathname, try "gdb list
    sc_fxval.h:1" followed by "gdb info source" for more
    details. Add directories to search path with "gdb dir
    <src-dir>".

ucli% gdb dir /u/joe/proj/abc/src

ucli% cbug::expand_path_of_source_file foo.cpp
/u/joe/proj/abc/src/foo.cpp
```

Note that adding a directory partially invalidates the cache used to store absolute pathnames. Files for which the absolute path name has already been successfully found and cached are not affected. But files for which the pathname could not be located so far will be tried again the next time if a new directory was added after the last try.

Common Design Hierarchy

An important part of debugging simulations containing SystemC and HDL is the ability to view the common design hierarchy and common VPD trace file.

The common design hierarchy shows the logical hierarchy of SystemC and HDL instances in the way it is specified by the user. See also the VCS / DKI documentation for more information how to add SystemC modules to a simulation.

The common hierarchy shows these elements for SystemC objects:

- Modules (instances)
- Processes:
 - `SC_METHOD`, `SC_THREAD`, `SC_CTHREAD`
- Ports: `sc_in`, `sc_out`, `sc_inout`,
 - `sc_in<T>`
 - `sc_out<T>`
 - `sc_inout<T>`
 - `sc_in_clk` (= `sc_in<bool>`)
 - `sc_in_resolved`
 - `sc_in_rv<N>`
 - `sc_out_resolved`
 - `sc_out_rv<N>`
 - `sc_inout_resolved`
 - `sc_inout_rv<N>`
- Channels:
 - `sc_signal<T>`
 - `sc_signal_resolved`
 - `sc_signal_rv<N>`
 - `sc_buffer<T>`
 - `sc_clock`

- `rvm_sc_sig<T>`
- `rvm_sc_var<T>`
- `rvm_sc_event`
- With datatype `T` being one of
 - `bool`
 - `signed char`
 - `[unsigned] char`
 - `signed short`
 - `unsigned short`
 - `signed int`
 - `unsigned int`
 - `signed long`
 - `unsigned long`
 - `sc_logic`
 - `sc_int<N>`
 - `sc_uint<N>`
 - `sc_bigint<N>`
 - `sc_bignum<N>`
 - `sc_bv<N>`
 - `sc_lv<N>`
 - `sc_string`

All these objects can also be traced in the common VPD trace file. Port or channels that have a different type, for example a user-defined struct, will be shown in the hierarchy but cannot be traced.

The common design hierarchy is generally supported for all combinations of SystemC, Verilog, and VHDL. The pure-SystemC flow (the simulation contains only SystemC but neither VHDL nor Verilog modules) is also supported.

All these objects can also be traced in the common VPD trace file. Interaction between CBug and the Simulator

The common design hierarchy is supported in the following combinations:

SystemC top
Vlog down

and

SystemC top
Vlog down
VHDL down

and

VHDL top
SystemC down
Vlog down

and

Vlog top
SystemC down

and

Vlog top
SystemC down
VHDL down

Common VPD tracing and other debugging features are not supported in the following combinations:

SystemC-top
VHDL down

and also not

VHDL top
Verilog down
SystemC instantiated from Verilog

and also not

sc_main()
SystemC top
Verilog top

Post-processing Debug Flow

One way to use DVE is to first let the simulation run, create a VPD file and then look at the VPD file afterwards. This is called post-processing mode. All data will be contained in a VPD file.

There are different ways to create a VPD file. Not all are supported for common VPD with SystemC:

Supported

- Interactive using DVE and the **Add to Waves...** command.
- Run the simulation in `-ucli` mode and apply `synopsys ::dump` command.

Not Supported

- With `$vcdblusion()` statement(s) in Verilog code.
- With VCS option `+vpdfile`.

If you create a VPD file in one of the unsupported ways, then you will not see SystemC objects at all. Instead you will find dummy Verilog or VHDL instances at the place were the SystemC instances are expected. The simulation will print warning that SystemC objects are not traced.

Use these commands to create a VPD file when SystemC is part of the simulation:

```
Create file dumpall.ucli :  
cbug::config add_sc_source_info always <-- this line is  
                                         optional, *1  
synopsys::cbugsynopsys::cbug <--this line is optional, *1  
synopsys::scope .  
    set fid [synopsys::dump -file dump.vpd -type VPD]  
    puts "Creating VPD file dump.vpd"  
    synopsys::dump -add "." -depth 0 -fid $fid  
    synopsys::continue
```

Then run simulation like this:

```
simv -ucli < dumpall.ucli
```

The line `synopsys : cbug` is optional. If specified, then CBug will attach and store in the VPD file the source file/line information for SystemC instances that are dumped. This is convenient for post-processing: a double-click on a SystemC instance or process will open the source-code file.

Note that all source code must be compiled with compiler flag `-g` which will slow down the simulation speed to some extend (how much varies greatly with each design). Furthermore, attaching CBug will take some CPU time during which the underlying gdb reads all debug information. This seconds runtime overhead is constant. Last, attaching CBug creates a gdb process that may need a large amount of memory if the design contains many C/C++ files compiled with `-g` flag. In summary, adding the `synopsys : cbug` is a tradeoff between better debugging support and runtime overhead.

Interaction with the Simulator

Usually the C debugger and the simulator (the tool, e.g. simv) work together unnoticed. However, there are a few occasions when the C debugger and the tool cannot fully cooperate and when this is visible. These cases depend on whether the active point (the point where the simulation stopped, for example due to a BP) is in the C domain or HDL domain.

Prompt Indicates Current Domain

The prompt reflects if the simulation is stopped in the HDL or C domain.

- *ucli%* -> HDL domain
- *CBug%* -> C domain

Commands affecting the C domain:

Commands that apply to the C domain, for example setting a breakpoint in C source code, can always be issued, no matter in which domain the current point lies.

Some commands, however, can only be applied when the simulation is stopped in the C domain:

- The stack command to show which C/C++ functions are currently active.

- Reading a value from C domain (e.g. a class member) with the `synopsys::get` command is sensitive to the C function where the simulation is currently stopped. Only variables visible in this C scope can be accessed.

That means it is not possible to access, for example, local variables of a C/C++ function or C++ class members when stopped in HDL domain. Only global C variables can always be read.

Combined Error Message

When the C debugger is attached and you enter a command, such as `get xyz`, then UCLI issues the command to both the simulator and the C debugger (starting with the one where the active point lies, e.g., starting with the tool in case the simulation is stopped in the HDL domain). If the first one responds without error, then the command is not issued again to the second one. However, if both tool and the C debugger produce an error message, the UCLI combines both error messages into a new one which is then printed.

Example:

```
Error: {
    {tool: Error: Unknown object}
    {cbug: Error: No symbol "xyz" in current context.;}
}
```

Update of Time, Scope, and Traces

Any time, when the simulation is stopped in C code, the following information is updated:

- Correct simulation time

- Scope variable (accessible with synopsys::env scope) is either set to a valid HDL scope or to string "<calling-C-domain>"
 - If you stop in C/C++ code while executing a SystemC process, then the scope of this process is reported.
 - String "<calling-C-domain>" is reported when the HDL scope that calls the C function is not known. This happens, for example, in case of DPI, PLI, VhPI or DirectC functions.
- All traces (VPD file) are flushed

Configuring CBug

Use the `cbug::config` UCLI command to configure the CBug behavior. The following modes are supported:

Startup Mode

When CBug attaches to a simulation, then there are two different modes to choose from. Enter the UCLI command:

```
cbug::config startup fast_and_sloppy|slow_and_thorough
```

to select the mode before attaching CBug.

Mode 'slow_and_thorough' is the default and may consume much CPU time and virtual memory for the underlying gdb in case of large C/C++/SystemC source code bases with many 1000 lines of C/C++ source code.

Mode 'fast_and_sloppy' will reduce the CPU and memory needed, however, it comes on the expense that not all debug information is available to CBug right away. Most debugging features will still work fine, but there may be occasional problems, for example, setting breakpoints in header files may not work.

Attach Mode

```
cbug::config attach auto|always|explicit
```

Mode 'attach' defines when CBug attaches. Value 'auto' is the default and attaches CBug in some situations, for example when you set a breakpoint in a C/C++ source file and when double-clicking a SystemC instance. Value 'always' will attach CBug whenever the simulation starts. If value 'explicit' is selected, then CBug is never attached automatically.

cbug::config add_sc_source_info auto|always|explicit

The `cbug::add_sc_source_info` command stores source file/line information for all SystemC instances and processes in the VPD file. Doing that may take a long time but is useful for post-processing a VPD file after the simulation ended. Value 'auto' invokes `cbug::add_sc_source_info` automatically when CBug attaches and the simulation runs without the DVE GUI; 'always' invokes `cbug::add_sc_source_info` automatically whenever CBug attaches; 'explicit' never invokes it automatically. Default is 'auto.'

VPD Dumping for SC_FIFO Channels

CBug supports VPD dumping of SystemC sc_fifo channels. It also supports printing the content of FIFOs with the UCLI get command. The format of how the data is printed can be configured.

FIFO objects that can be Dumped or Printed

CBug supports VPD dumping for the following:

- sc_fifo channels = objects of type `sc_core::sc_fifo`.
- User-defined classes or structs using the `<<` operator (see “[Support for Data Types](#)”).
- Classes derived from `sc_fifo`.

Not supported:

- `sc_fifo_in` ports
- `sc_fifo_out` ports

Only SystemC 2.2 is supported. SystemC 2.0.1 and 2.1 are not supported.

Displaying Data in SC_FIFO

- The number of available tokens currently stored in the FIFO is always displayed.
- The values of tokens can be optionally displayed. You can set a limit on the number of tokens that are displayed.

- The list of processes that are currently waiting for a FIFO data_read or data_written event can also be displayed optionally.

Example

The current status of a FIFO of type sc_fifo of size 20 is printed. The dumping of this FIFO has been configured to show processes and tokens, but no more than four tokens:

```
20 available
0 free

Waiting for read:
    TOP.inst0.Stim
Waiting for write:

Data: [0]=oldest value
[0] 8902
[1] 13
...
[18] 9999
[19] 1111
```

The FIFO is currently full; all 20 slots are filled with tokens, 0 slots are free. The oldest token (the one that will be removed by the next `read()` call) is at position '[0]', and has value 8902. The most recently written token is at position '[19]'.

Configuring Dumping of a FIFO

The format on how the content of a FIFO is dumped can be configured from DVE, with an UCLI command, from the DVE Preferences dialog, or by a C interface.

Configuring with UCLI

The UCLI command `cbug::config fifo_dump` specifies how FIFOs are dumped. The number of tokens currently available in the FIFO is always printed in the first line.

Printing of tokens stored in the FIFO is controlled using the following command.

```
cbug::config fifo_dump data (on|off|<limit>)
[<object_path>]
```

Value `on` specifies to dump all tokens regardless of their number. Value `off` (the default) turns off printing tokens altogether. An integer specifies the maximum number of tokens to be printed. If more values are available, then half the limit is printed above the “...” and the other half after the “...”, as shown in the above example.

If a SystemC FIFO object is specified with `<object-path>`, then the limit applies to this FIFO only. If no object is specified, then the limit is the default limit for all other FIFOs.

Example

```
cbug::config fifo_dump data on
cbug::config fifo_dump data 10 Top.fifo1
cbug::config fifo_dump data off Top.A fifo3
cbug::config fifo_dump data 4
```

A SystemC process that calls the blocking method `sc_fifo::read()` blocks when the FIFO is empty. The process dynamically waits until the `data_written` event of the FIFO triggers, which happens on the next `write()` or `nb_write()`. Similarly, a process writing `sc_fifo::write()` blocks when the

FIFO is full. You can configure displaying processes that are suspended in the dynamic lists of the `data_read` and `data_written` events with the following command:

```
cbug::config fifo_dump proc (on|off)
```

The setting applies to all FIFOs. The processes are not printed if the printing tokens are disabled. Printing processes are disabled by default.

Configuring with DVE

To configure with DVE:

1. Click **Edit > Preferences**.

The Application Preferences dialog box opens.

2. Select **Testbench/CBug**.

The SystemC `sc_fifo` Dumping appears on the right-hand side. Select the desired options.

Note:

Specifying the limit of #tokens for a particular `sc_fifo` in the DVE GUI is not possible. You can do this only with the UCLI `cbug::config` command.

Configuring from SystemC Source Code

The same configurations can also be done from the user's SystemC source code using the following C functions.

```
extern "C" int sc_snps_cbug_fifo_set_limit_num_objects \
(int num, sc_core::sc_object* fifo_ptr=0);
```

The Value -1 on the first argument is equivalent to off and defines to print no data. Value 0 is equivalent to on and defines to print all data without limit. A positive number sets the limit of tokens to be printed. Specifying `fifo_ptr=0` sets the default limit for all FIFOs.

```
extern "C" int sc_snps_cbug_fifo_set_show_proc \
(int enableShowingProcesses);
```

Both functions are declared in the `systemc_user.h` file, which you need to include as follows:

```
#include <cosim/bf/systemc_user.h>
```

Support for Data Types

Native ANSI and SystemC types

Native ANSI types (for example, `bool`, `int`, `long long`) are fully supported. They are displayed in the same way as an `sc_signal` object of the same type. You cannot change the radix.

Native SystemC bit-vector types (`sc_lv`, `sc_bv`, `sc_int`, ...) are also fully supported. They are printed in the same way as the `<<` operator. Bit-vectors are printed in full length with no restrictions. You cannot change the radix.

String type `std::string` (and `char*`) are fully supported. The full length of the string is displayed.

User-defined Types

Printing tokens is based on the `<<` operator. Dumping of user-defined types (for example, classes or structs), is fully supported if the `<<` operator for this type is defined properly.

Note that OSCI SystemC makes it mandatory to define the `<<` operator of any type `T` used in an `sc_fifo`, so that there is no extra work needed for debugging. However, the operator may be defined as a stub, in which case tokens are not properly displayed.

The number of available tokens and processes (if enabled) is always properly displayed, even if the `<<` operator is just a stub.

Change Bars in Waveform

A FIFO trace in the waveform shows a change bar whenever the FIFO has activity (`read()`, `write()`, `nb_read()`, `nb_write()` calls).

A change bar is also printed when the visible status does not change. For example, imagine that both `read()` and `write()` happen at the same time step. A change bar is printed even though the number of available tokens does not change.

UCLI 'get' Command

The UCLI `get` command prints the same information as VPD dumping. Both of them share the same functionality, and are subject to the same configuration.

Speed Impact

If a FIFO is not dumped, then no runtime overhead is incurred. If it is dumped, then the overhead varies greatly with the configuration, number of tokens to be printed, and type.

You can reduce the runtime overhead by first configuring to print only the minimal amount of information (no data, no processes). Then, when the simulation reaches a certain time, change the configuration to display more data.

Supported platforms

Interactive debugging with CBug is supported on the following platforms:

- RHEL32/Suse, 32-bit
- RHEL64/Suse, 64-bit (VCS flag -full64)
- Solaris 5.9/5.10, 32-bit

Interactive debugging with CBug is not supported on these platforms:

- Solaris, 64-bit
- -comp64 flow of VCS, all platforms
- any other platform

An explicit error message is printed when you try to attach CBug on a platform that is not supported.

For Solaris 32-bit (sparcOS5), interactive debugging with CBug is not possible in:

- Any kind of C/C++ code if the simulation has been compiled with `vcs -vera` command.
- C/C++ code that is located in a shared library (e.g. `mylib.so`) and explicitly loaded with an `dlopen()` command. All VhPI applications fall into this situations. Debugging of C/C++ source files located inside such a shared library is not possible.

If you set a breakpoint and stop there, then `gdb` and CBug fails and `simv` terminates without any error message. Stopping in other C/C++ source files that are not part of the shared library is supported. Note that this restriction is specific to Solaris, RHEL32 has no such restriction.

For Solaris 64-bit, debugging of SystemC modules is only possible in the post-processing flow. Port/signals of SystemC modules can be dumped in a VPD file and later displayed by DVE. Note that this specific platform does not allow to store source file/line information for SystemC instances; doing a double-click an SystemC instance or process will not open the corresponding source file.

Using `SYSTEMC_OVERRIDE`

VCS ships with multiple SystemC versions (2.0.1, 2.1, 2.2) which are used per default. In rare cases, it might be necessary to use a different SystemC installation that you compiled on your own. This can be done by setting the `SYSTEMC_OVERRIDE` environment variable (see the *VCS / VCSI User Guide*).

If you use `SYSTEMC_OVERRIDE`, then some or all of the SystemC specific CBug features are not available.

These features are not available:

- Tracing of SystemC objects (ports, sc_signals).
- Printing of SystemC native datatypes like sc_int in an intuitive format. Instead you will see the usual form how gdb prints the data which is generally not useful for SystemC objects.
- Stopping in the next SystemC user process with 'next' or 'step'.

These feature may or may not work depending on how much different the SystemC installation is compared to an OSCI installation:

- Showing SystemC objects (instances, processes, ports) in the common hierarchy (Hierarchy pane in DVE).
- Double-clicking an SystemC instance or processes to open the source file.
- Cross-stepping in or out of SystemC user processes and HDL code

Any other SystemC specific CBug feature:

The following non-SystemC specific CBug feature will always work:

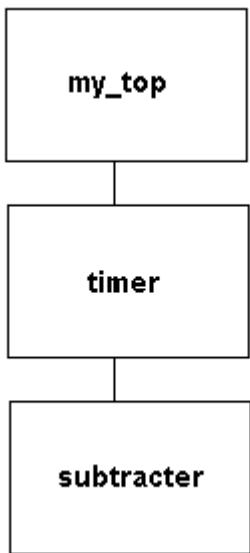
- Setting breakpoints in SystemC source code (you may have to open the source file with File/Open File in DVE, though).
- Stepping through SystemC source code. Note that stepping out of one SC user processes and stopping in the next one without a breakpoint is not supported).

- Access a variable/class member with `synopsys::get`. The variable needs to be visible in scope of the C function where the simulation is currently stopped. Note that enhanced printing of native SystemC types is not available.

Example: A Simple Timer

The design is a simple timer that decrements a count starting from an initial value and signals an interrupt when the count reaches 0.

The test case is a Verilog top + SystemC down test case.



`my_top` is the driving module that drives the clock, reset, `inval_valid`, `inval`, `current_val`, and `interrupt` and samples the `timer_val` (current timer value and interrupt from the SystemC module).

Where:

- `reset` restores the initial value of the timer.

- `inval` is the timer initial value.
- `inval_valid` indicates to the timer that the input value is valid.
- `timer_val` is the current timer value.

The SystemC module for timer instantiates the `sc_subtracter` module that has `sc_current_value` as the input and returns `sc_current_value-1` as the result (`sc_current_value_minus_one`).

The `current_value-1` that is returned by the subtracter module is feedback to the subtracter module at every succeeding clock edge till the final value reaches 0 and interrupt is triggered.

1. Compile the testcase.

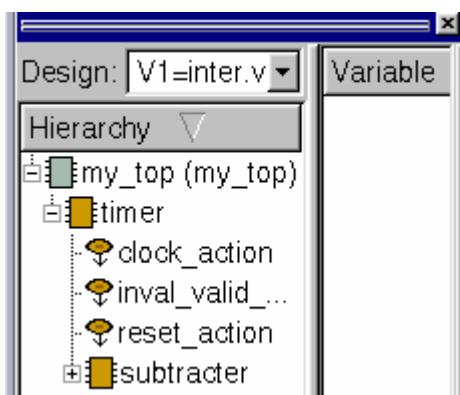
```
syscan -cpp g++ -cflags "-g" timer.cpp:timer sc_subtracter.cpp
vcs -cpp g++ -sysc top.v -debug_all $VCS_HOME/lib/ucli.o -timescale=1ps/1ps
```

2. Start DVE.

dve

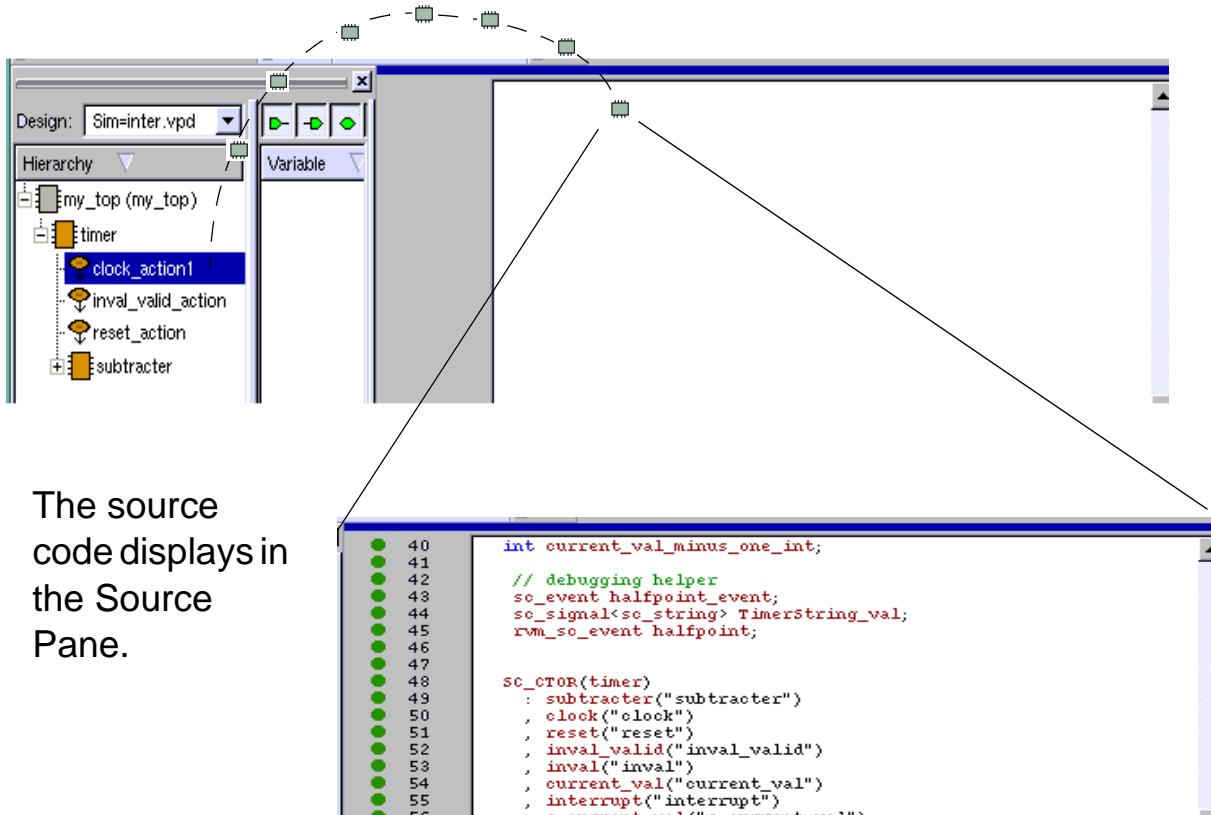
3. Click to start the simulation.

The DVE Hierarchy pane displays the design. Note that SystemC modules display in orange.



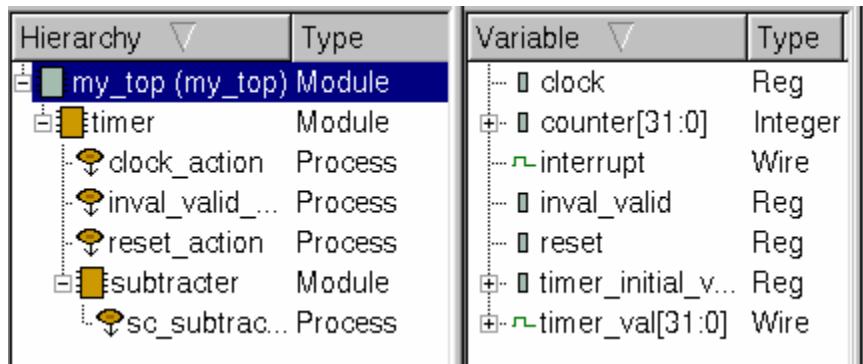
4. To display the source code of a process (), drag and drop the process from the Hierarchy pane to the Source view.

Drag a process to the Source Pane



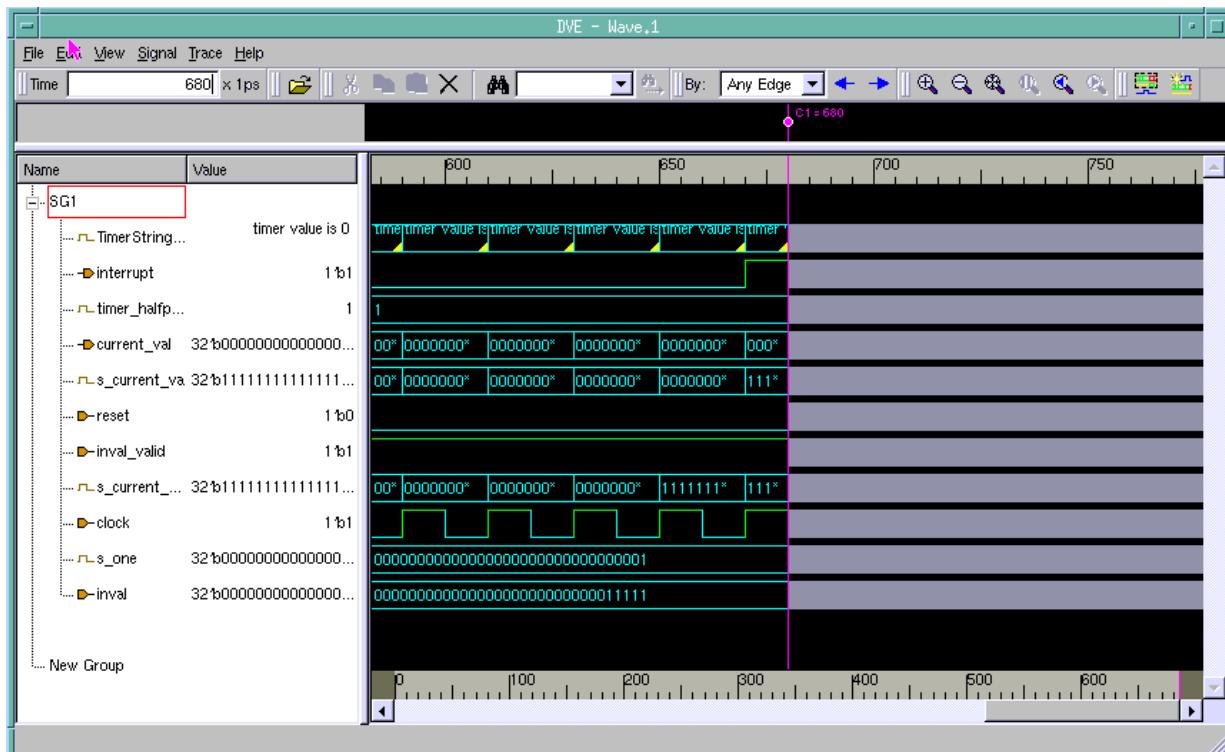
Note that anytime a C source code is opened or a breakpoint is set on a C source code, the C debugger is launched automatically.

5. Click on the design to display variables in the Variable Pane.



6. In the Hierarchy Pane, select the top level of the design (**my_top (my_top)**), right-click, then select **Add to Waves** from the context-sensitive menu.

The Wave view displays waveforms up to the current simulation time. . .

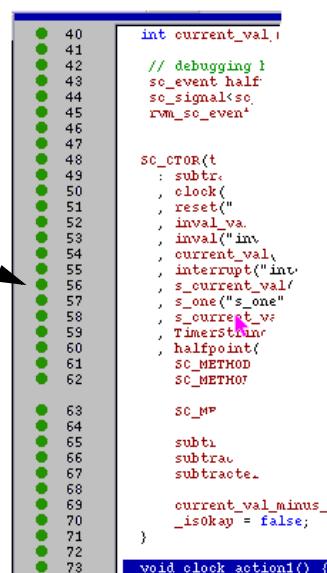
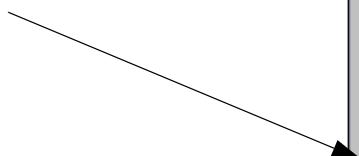


Note:

You cannot display the waveform while the simulator is stopped in C code.

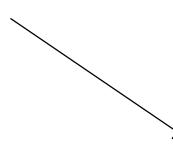
7. To set line breakpoint, click any green circle in the Line Attribute area of the Source view as follows.

Click any breakable line indicator to activate line break.



```
int current_val;  
// debugging 1  
sc_event half;  
sc_signal<sc_rv> sc_rv_sc_event;  
  
SC_CTOR(t : subtracter)  
<> :  
<> clock;  
<> reset("reset");  
<> inval("inval");  
<> current_val("current_val");  
<> interrupt("interrupt");  
<> s_current_val("s_current_val");  
<> s_one("s_one");  
<> s_current_val_v; // Timersthing  
<> halfpoint(sc_METHOD(sc_METHOD));  
  
SC_MP  
<> subtr1;  
<> subtr2;  
<> subtracter;  
  
current_val_minus_ _isOkay = false;  
}  
  
void clock action1() {
```

Red circle indicates activated line breakpoint.



Line	Line Attribute
64	
65	
66	Red circle
67	
68	
69	

Note:

Similar to setting a breakpoint in the C code, you can set a breakpoint in HDL code and step back and forth between C and HDL code using the debugger.

8. Click  (Continue) in the Top Level Window toolbar.

The simulation advances to the breakpoint.

9. Enter finish in the console command line.

The simulation runs to the end.

Viewing SystemC Source and OSCI Names in DVE

This chapter describes how the DVE CBug automatically displays SystemC port and signal names with its source name.

This chapter consists of the following sections:

- “Use Model”
- “Source and OSCI Names”
- “Displaying Source and OSCI Names in DVE”
- “Limitations”

Use Model

- The following example provides the recommended conventions for naming signals and ports in the SystemC constructor:

Example-1:

```
sc_in CLK;  
SC_CTOR(top) : CLK("CLK") { . . . }
```

In this case, the port will be visible, as expected, in the design hierarchy with its proper name `top.CLK`.

- If the port name is not provided, then a random port name such as `port_0` will be generated and assigned automatically, as shown in the following example:

Example-2:

```
sc_in CLK;  
  
SC_CTOR(top) { ... }  
  
// Initializer for CLK is missing
```

This naming convention is not recommended because the hierarchy refers to the port or signal as `top.port_0`, whereas the source code knows only `CLK`.

- You can provide a different name explicitly, as shown in the following example:

Example-3:

```
SC_CTOR(top) : CLK("BCLK") { ... }
```

This naming convention is not recommended, because the hierarchy refers to the port or signal as `top.BCLK`, whereas the source code recognizes only `CLK`.

Source and OSCI Names

Source Name

Source name is the name of an instance in the SystemC source code. In [Example-1](#), CLK is the source name.

OSCI Name

OSCI name is the name given to the constructor. In [Example-2](#), port_0 is the OSCI name, and in [Example-3](#), BCLK is the OSCI name.

Note:

The SystemC language recommends you to provide same names for Source and OSCI.

Displaying Source and OSCI Names in DVE

DVE supports display of both source and OSCI name. To activate this feature, do the following:

1. Compile and build the simulation executable with debug flags.

```
% syscan -cflags -g and
```

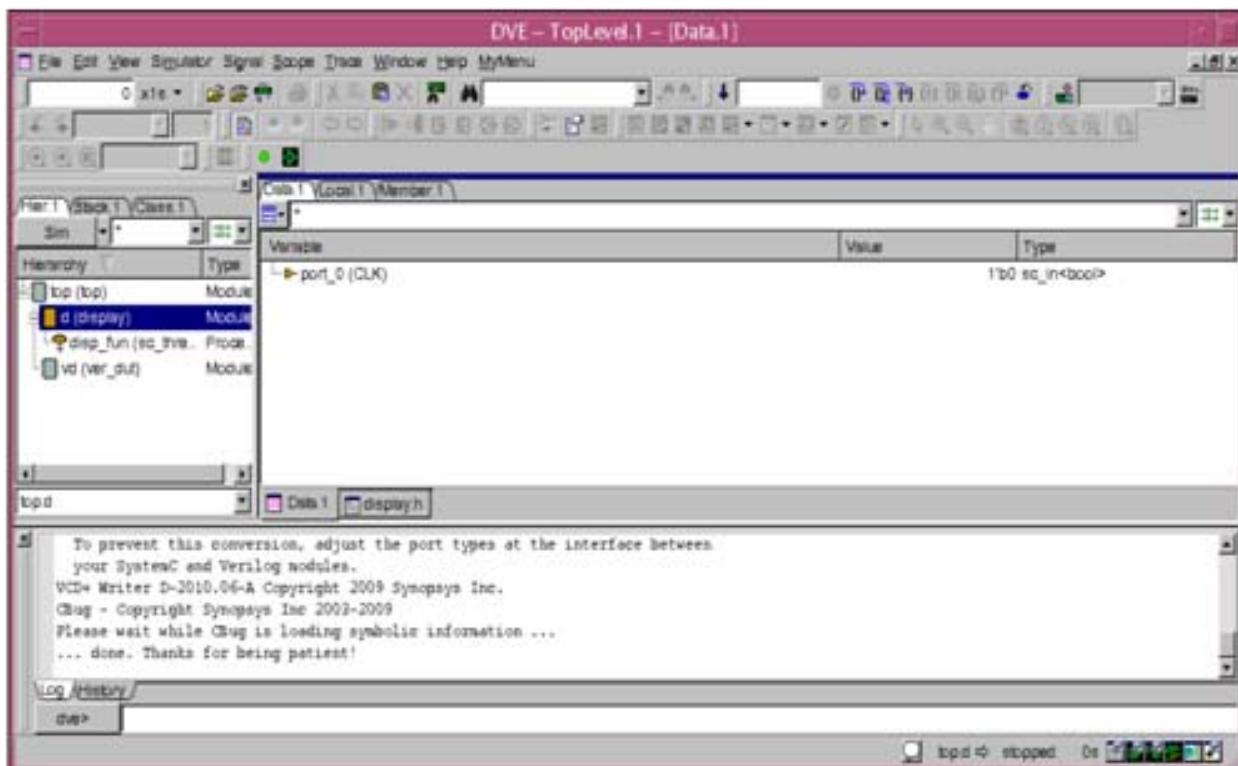
```
% vcs -debug_all
```

2. In DVE, select **Edit > Preferences > Testbench/CBug > CBUG: Store SystemC class member variables which are not derived from SystemC base classes in VPD (performance impact). This setting also enables displaying better descriptions for names like 'port_0'.**

3. Enable CBug, using any one of the following methods:
 - Type `cbug` on the DVE command-line
 - In DVE, select **Simulator > C/C++ Debugging > [x] Enable**

For [Example-2](#), DVE shows:

*`port_0` (CLK) * in Data, Wave, Watch, and List panes. The following figure shows the `port_0` (CLK) port in the Data pane:



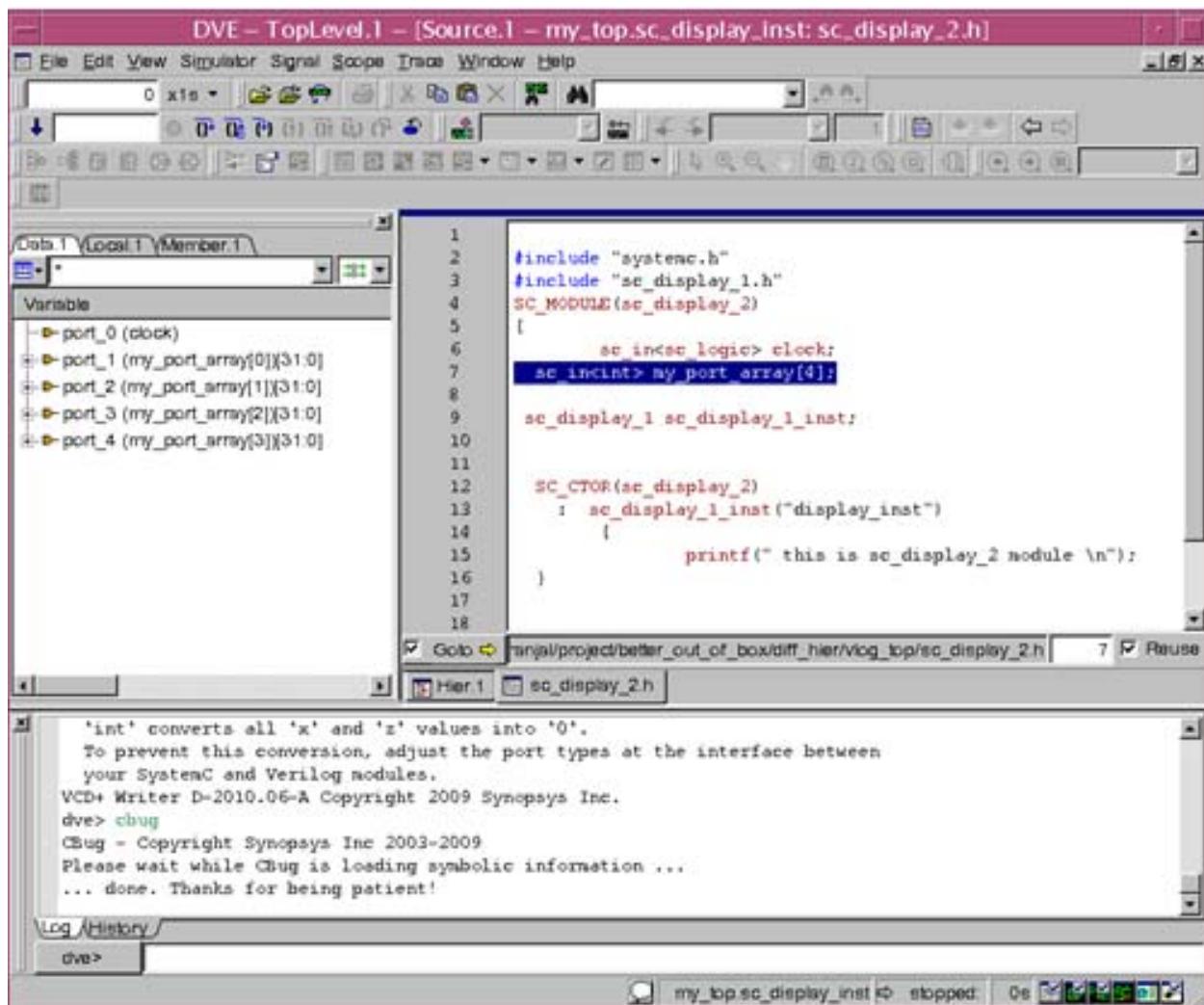
Example-4:

For the following source code, as shown in the figure given below:

```
sc_in<int> my_port_array[4];
```

DVE displays the following ports in the Data pane:

```
* port_1 (my_port_array[0])
* port_2 (my_port_array[1])
* port_3 (my_port_array[2])
* port_4 (my_port_array[3])
```



Limitations

The following are the limitations of viewing SystemC Source and OSCI Names in DVE:

- DVE depends on the capability of the GNU Project Debugger (GDB). At present, GDB cannot access variables defined in certain namespace scenarios. This breaks the UCLI get command for plain member variables in modules that contain namespace.
GDB cannot read the dimensions of multi-dimensional arrays in certain template scenarios. For multi-dimensional arrays, DVE generates a warning message and displays the array content as a linear array.
- This feature is available only on SystemC 2.2 and newer versions (if any), but not on versions 2.1 and 2.0.1.

Using CBug to Display Instance Name of Target Instance in TLM-2.0

This section describes how to use CBug to display the instance name of a target instance in Transaction-level Modeling (TLM).

When the simulation stops inside SystemC source code, then UCLI or DVE reports where it stopped. This scope refers to a SystemC module from the static design hierarchy. Use the UCLI command `senv activeScope` to view the active scope. In DVE, the scope is also shown at the bottom-right.

The following are various cases where CBug reports the SystemC module as a scope:

- If the simulation stops inside a member function of a SystemC module, then this module is reported.
- If stopped inside a C function, CBug looks in the call stack for the nearest SystemC module and reports it.

Note:

CBug analyzes the call stack only up to a maximum depth of five levels.

- CBug also looks into the call stack if it is stopped inside a C++ member that is not related (derived) to a SystemC module.
- If the SystemC kernel is currently executing a SystemC process, and the call stack does not reveal a SystemC module, then the SystemC module which owns this process is reported.

- If neither the call stack nor the active process reveals a module, then an empty scope is reported. DVE displays Calling C domain in this case. This can happen with DPI or PLI calls initiated from the HDL side.

Example

Consider the following SystemC module `mem_tb`:

```
....  
mem_tb_socket -> b_transport(trans, delay);  
....
```

This module calls the `b_transport` function through the `mem_tb_socket` socket, which is an initiator socket.

Consider another SystemC module `memory`, in which `b_transport` is defined, as given below:

```
void memory :: b_transport (tlm::tlm_generic_payload&trans,  
sc_time & delay)  
{  
....  
....  
}
```

If control steps in or stops at the `b_transport` function, then the `senv activeScope` command displays the `memory` as an active scope.

Limitations of Displaying Instance Name of Target Instance in TLM-2.0

If you call ordinary C functions or member functions of ordinary C++ classes that do not belong to a scope (that is, that are not derived from an `sc_object`), then the calling scope is displayed only if the caller with a scope is less than five levels away on the call stack. This happens because the visibility of the caller scope ends at level five of the call stack frame.

CBug Stepping Improvements

This section describes the enhancements made to CBug to make stepping smarter in the following topics:

- “[Using Step-out Feature](#)” on page 59
- “[Automatic Step-through for SystemC](#)” on page 60

Using Step-out Feature

You can use the step-out feature to advance the simulation to leave the current C stack frame. If a step-out leaves the current SystemC process and returns into the SystemC or HDL kernel, then simulation stops on the next SystemC or HDL process activation, as usual, with a sequence of `next` command.

CBug currently supports the existing `next -end` UCLI command. This command is used to advance the simulation until you reach the next break point or exit the C domain, and then you are back into the HDL domain.

The behavior of this command is changed to support the step-out functionality. This command is now equivalent to the `gdb finish` command. This feature will be continued under a new UCLI command `next -hdl`.

Note:

The step-out feature does not apply in an HDL context.

Automatic Step-through for SystemC

The following are some of the typical scenarios where you can step into SystemC kernel functions:

- `Read()` or `Write()` functions for ports or signals
- Assignment operator gets into the overloaded operator call
- `sc_fifo`, `tlm_fifo`, `sc_time` and other built-in data type member functions or constructors
- `wait()` calls and different variants of `wait()` calls
- Performing addition or other operations on ports gets inside the kernel function when you do a step. This happens if you have a function call as part of one of its arguments to the `add` function.

A step should step-through to the next line in the user code or at least outside the Standard Template Library (STL), but should not stop within the STL method. CBug does a step-through for any method of the following STL classes:

- STL containers like `std::string`, `std::hash`
- Other STL classes like `vector`, `dequeue`, `list`, `stack`, `queue`, `priority_queue`, `set`, `multiset`, `map`, `multimap`, and `bitset`

Enabling and Disabling Step-through Feature

Use the following command to enable the step-through feature:

```
cbug::config step_through on
```

Use the following command to disable the step-through feature:

```
cbug::config step_through off
```

If step-through is disabled and UCLI `step` ends in a SystemC kernel or STL code, then an information message is generated if you use `next -end` (=gdb `finish`). This message states that `cbug::config enable stepover` exists, and may be useful. This message is generated only once while CBug is attached.

Recovering from Error Conditions

In some cases, it is possible that an automatic step-through does not quickly stop at a statement, but triggers another step-through, followed by another step-through, and so on. In this case you notice that DVE or UCLI hangs, but may not be aware that the step-through is still active.

CBug must recognize this situation and take action. This happens if a step-through does not stop on its own after 10 consecutive iterations of internal `finish` or `step`.

CBug can either stop the chain of internal `finish` or `step` sequences on its own, and report a warning which states that the automatic step-through is aborted and how to disable it.

14

Debugging Constraints

Debugging constraints is a challenge because incorrect constraints can result in no legal solution. Also, constraints may contain complex conditions and reflect relations between variables that may be difficult to understand.

Constraints can fail due to failed randomization of a class object, conflicting constraints, or unexpected values assigned to variables. Better debug capabilities can help you understand the constraints better. In addition, interactive debug can help you understand how a certain randomize call was solved, sources of inefficiency, and the reason for a solver failure.

DVE supports constraint debugging by allowing you to:

- Analyze a randomize call after the randomization has been completed and before the `post_randomize()` function is executed.

- Set breakpoints to stop simulation at a certain randomize call.
- View static information about constraints in the Member Pane.
- Browse constraint objects in the Local Pane.
- View constraint details and how the constraints were solved in the Constraints dialog box.
- Understand randomization by querying any variable or constraint.
- Change the radix type of a variable value and values of a constraint expression in the Solver Pane and Relation Pane of the Constraints dialog box.
- Drag-and-drop a variable or constraint block item from the Solver Pane or Relation Pane into the Search field of the Constraints dialog box.
- Use the **Add To Search** right-click option in the Solver Pane and Relation Pane, to specify a variable or constraint block item in the Search field.
- View object ID information of a class object in the **Value** column of the Solver Pane.
- Extract test case of a Partition and Randomize Call item.
- Control `rand_mode/constraint_mode` and randomization from UCLI/DVE.

This chapter consists of the following sections:

- “Enabling Constraint Solver for Debugging”
- “Invoking the Constraint Solver Debugger GUI”
- “Debugging Constraint-Related Problems”

- “Constraint Browsing in Class Hierarchy Browser”
- “Browsing Objects in Local Pane”
- “Using the Constraints Dialog”
- “Inconsistent Constraints”
- “Debugging Constraints Example”
- “Changing Radix Type of a Variable or Constraint Expression in Constraints Dialog Box”
- “Drag-and-Drop Support for Constraints Debug”
- “Viewing Object ID Information of a Class in Solver Pane”
- “Cross Probing”
- “Extracting Test Case”
- “Controlling rand_mode/constraint_mode and Randomization from UCLI/DVE”
- “Constraints Debug Limitations”

Enabling Constraint Solver for Debugging

To enable debugging capabilities for the constraint solver, you must specify the `-debug_all` switch, along with your compilation command. For example:

```
% vcs -debug_all test.v
```

Invoking the Constraint Solver Debugger GUI

You can start the constraint solver debugger from the command line, and then run the simulation from the GUI:

```
% simv -gui
```

Debugging Constraint-Related Problems

To debug constraint-related problems, follow these guidelines:

- “[Breaking Execution at a Randomize Call](#)” on page 4
- “[Analyzing a Randomization Call](#)” on page 18

Breaking Execution at a Randomize Call

You can set a breakpoint to stop the simulation at a certain randomize call. A breakpoint is a setting on a line of code that tells DVE to stop the simulation immediately before the line of code on which it is set so that you can examine that line of code before continuing.

You can use the UCLI commands shown in [Table 14-1](#) or the DVE Breakpoints dialog box to set a breakpoint at a certain randomize call (see [Figure 14-1](#)).

Table 14-1 Commands to Set Breakpoint at a Randomize Call

UCLI Command	Equivalent Setting in DVE Breakpoints Dialog Box	Description
<code>stop -file <file> -line <line> -skip <num></code>	See Figure 14-1	Allows you to stop at any randomize call and skip any number of intermediate randomize calls.
<code>stop -solver</code>	See Figure 14-2	Stops the execution at all subsequent randomize calls.
<code>stop -solver -once</code>	See Figure 14-2	Stops the execution at the next randomize call encountered.
<code>stop -solver -serial <num></code>	See Figure 14-2	Stops the execution at a randomize call with a certain solver serial number.

Note:

You must specify the above-mentioned UCLI commands in the DVE console.

To open the Breakpoints Dialog, select **Simulator > Breakpoints**. In the Breakpoints dialog box, click **Define>>** to display the breakpoint creation tabs, as shown in [Figure 14-1](#).

Figure 14-1 Setting Breakpoints in the Line Tab

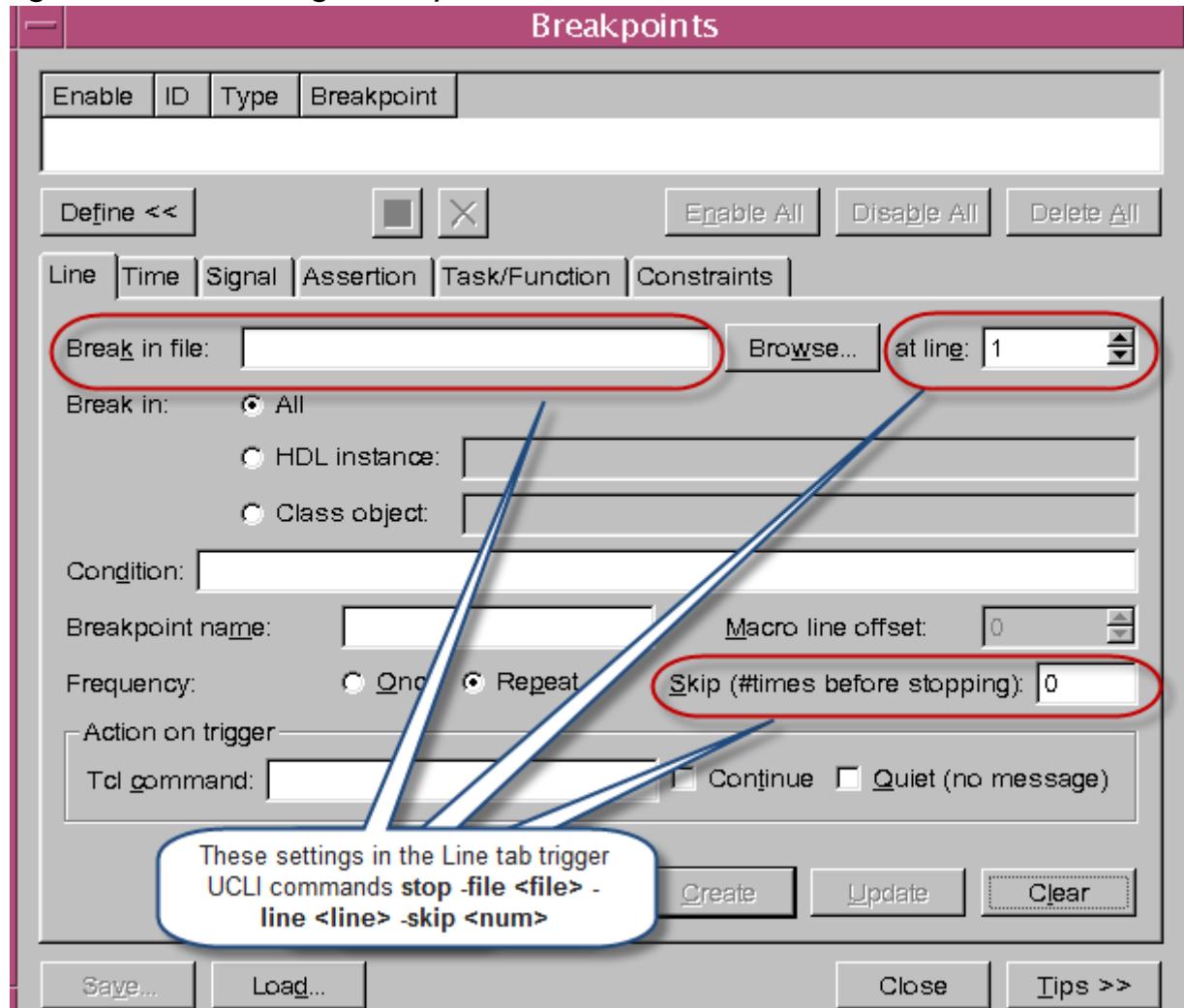
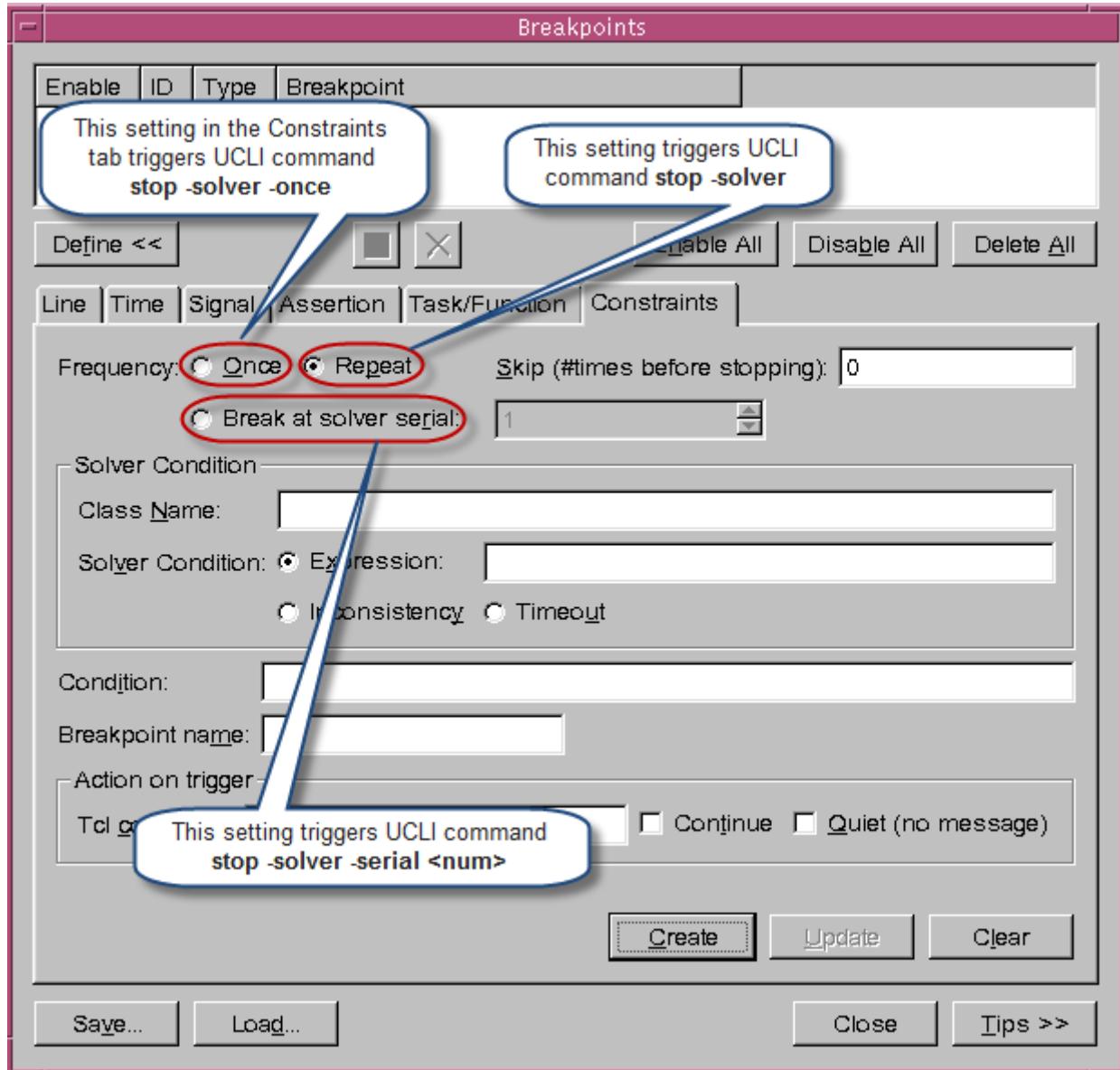


Figure 14-2 Setting Breakpoints in the Constraints Tab

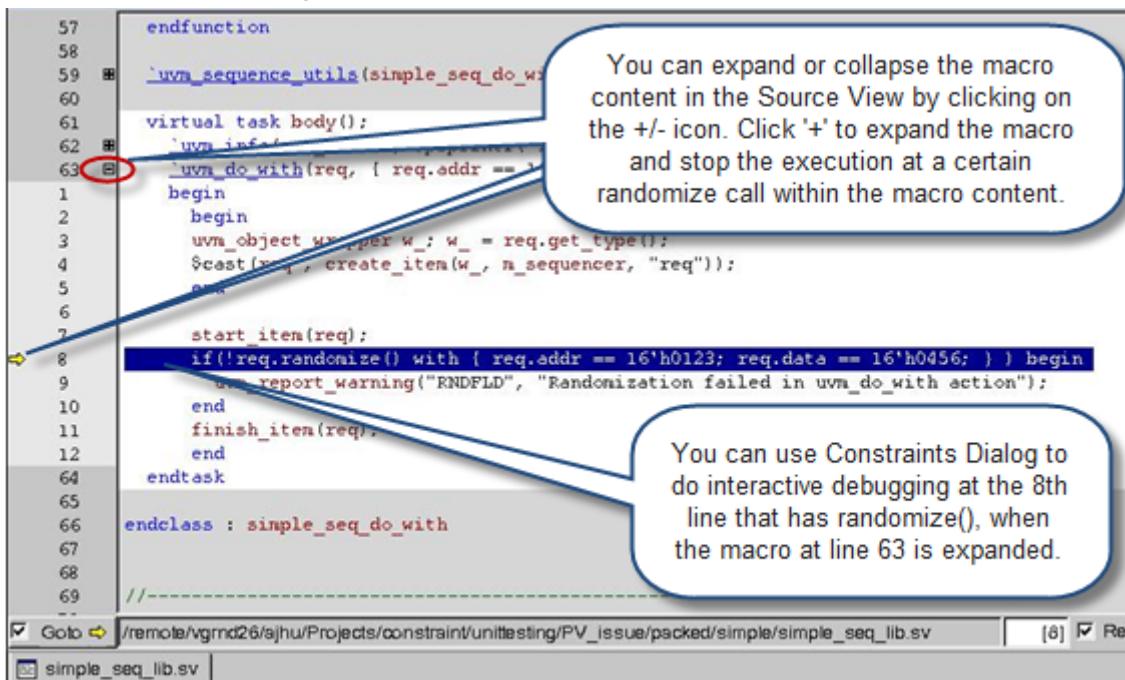


Stopping at Randomize Calls Within Macros

DVE supports debugging of randomize calls within the macro content. You must expand the macro, as shown in [Figure 14-3](#), to stop the execution at a certain randomize call within the macro

content. Then click the **Step in Constraint Solver** toolbar icon  to open the Constraints dialog box (see [Figure 14-26](#)). DVE highlights the randomize call in the macro content.

Figure 14-3 Stopping Execution at Randomize Calls within Macros



The screenshot shows a DVE Source View window displaying a Verilog-like code snippet. The code defines a class `simple_seq_do_with` with an `endfunction` and a `virtual task body()`. The task body contains several statements, including a `begin` block with a `begin` block inside it. The inner `begin` block contains a `uvm_object` assignment and a `create_item` call. A call to `start_item` is followed by a conditional statement `if(!req.randomize())` which includes a constraint `with [req.addr == 16'h0123; req.data == 16'h0456;]`. This line is highlighted with a blue rectangle. A yellow arrow points from the left margin to this line. A red circle highlights the collapse icon (a plus sign) at line 63, which is part of the `endtask` keyword. Two callout boxes provide instructions: one for expanding/collapsing macros and another for using the Constraints Dialog.

```

57 endfunction
58
59 uvm_sequence_utils(simple_seq_do_wi
60
61 virtual task body();
62     `uvm_info(`UVM_INFO, $sformatf("Randomization started for sequence %s", get_name()), get_sequencer())
63     uvm_do_with(req, { req.addr == 16'h0123; req.data == 16'h0456; })
64     begin
65         begin
66             uvm_object#(req) w_; w_ = req.get_type();
67             $cast(w_, create_item(w_, n_sequencer, "req"));
68         end
69     end
70     start_item(req);
71     if(!req.randomize()) with [ req.addr == 16'h0123; req.data == 16'h0456; ] begin
72         `uvm_report_warning("RNDFLD", "Randomization failed in uvm_do_with action");
73     end
74     finish_item(req);
75 end
76 endtask
77
78 endclass : simple_seq_do_with
79
80 
```

You can expand or collapse the macro content in the Source View by clicking on the +/- icon. Click '+' to expand the macro and stop the execution at a certain randomize call within the macro content.

You can use Constraints Dialog to do interactive debugging at the 8th line that has randomize(), when the macro at line 63 is expanded.

Creating Solver Conditional Breakpoint at Randomize Calls

You can use the `stop -solver` UCLI command along with options `-class`, `-solver_cond` or the DVE Breakpoints dialog box to create a solver conditional breakpoint at a certain randomize call.

Creating Solver Conditional Breakpoint Using `stop -solver` UCLI command

Use the following syntax:

```
stop -solver -class <class_name> -solver_cond <expr>
```

where,

`<class_name>` — The name of class definition to which the built-in `randomize()` method belongs. For example, `obj1.randomize()`, where `obj1` is the instantiated object of the `A` class definition. You can specify `-class A` for a `randomize()` method call from any instantiated objects of `A`.

`<expr>` — Following is the syntax of `<expr>`:

```
expression ::= { condition [logical_op condition] }

condition ::= variable [relational_op value]

variable ::= [classobj_member.|struct_member.|union_member.]
            simple_name[.size] | 1

value ::= decimal_number|binary_number|hex_number
        (radix_base ::= 'b | 'h )

logical_op ::= && | || | or | and
relational_op ::= == | < | > | = | !=
```

Key points to note:

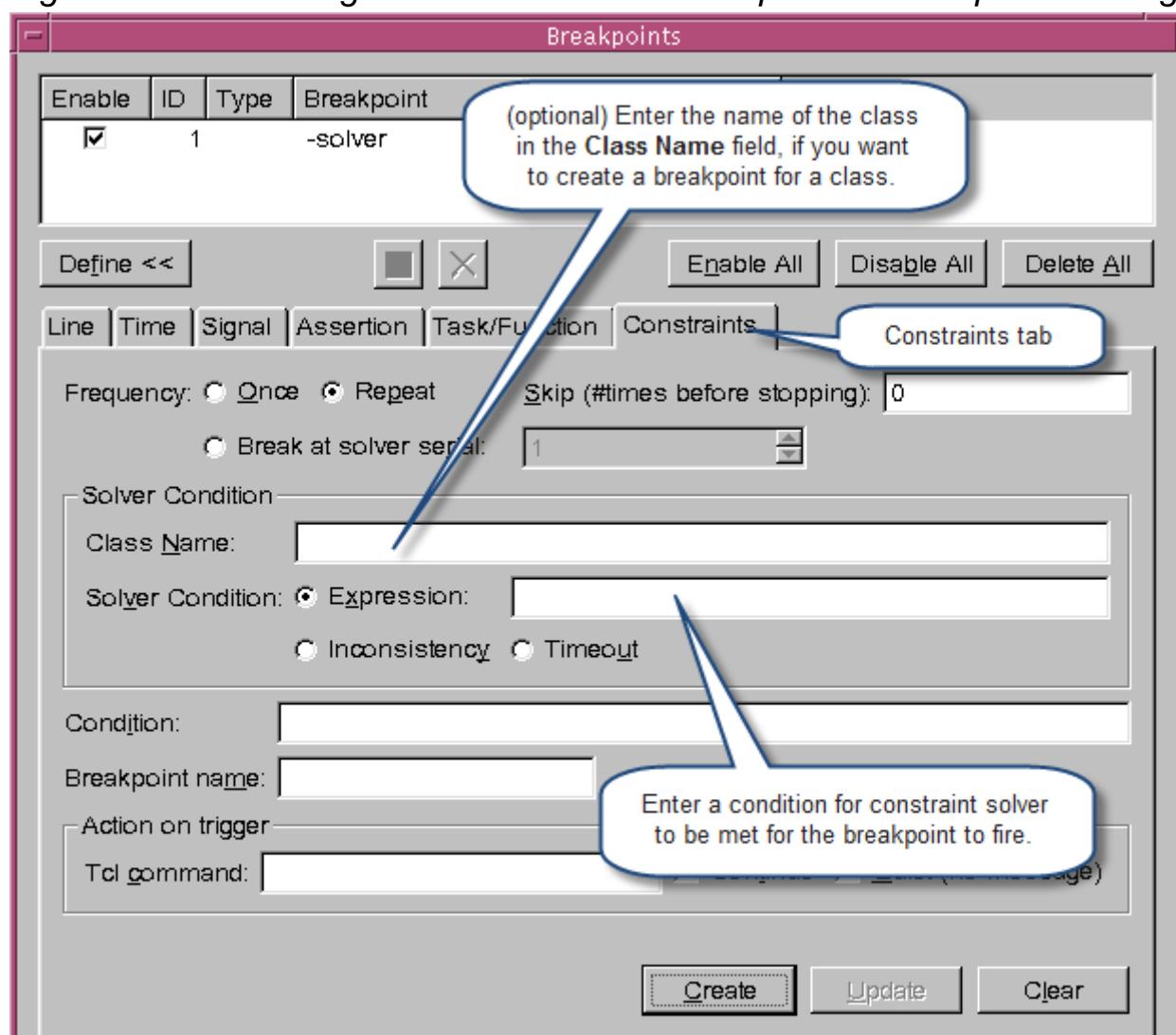
- Only variables of type `rand` whose modes are `ON`, are taken care in solver expression. That is, `-solver_cond` is different from `-condition`.
- The `-class` command is optional. If no class is specified on the command line, the `-solver_cond` command applies to the `std::randomize()` call. You can specify `rand` variables from the randomization data of the current scope (LRM 12.11).
- If you do not specify `rand_value`, the `rand` variable can be solved to any non-zero value.

- If `-class` is specified without `-solver_cond`, the breakpoint is hit when any rand variable of the specified class is solved.

Creating Solver Conditional Breakpoint Using DVE Breakpoints Dialog

[Figure 14-4](#) illustrates creating a solver conditional breakpoint in the Constraints tab of the DVE Breakpoints dialog box.

Figure 14-4 Creating Solver Conditional Breakpoint in Breakpoints Dialog



Setting Breakpoint on Parameterized Classes

You can set a breakpoint on parameterized classes from the command line or the Class Browser.

To set breakpoints on parameterized classes from the command line, use the following syntax:

```
%dve> stop -solver -class {class_name} -solver_cond {expr}
```

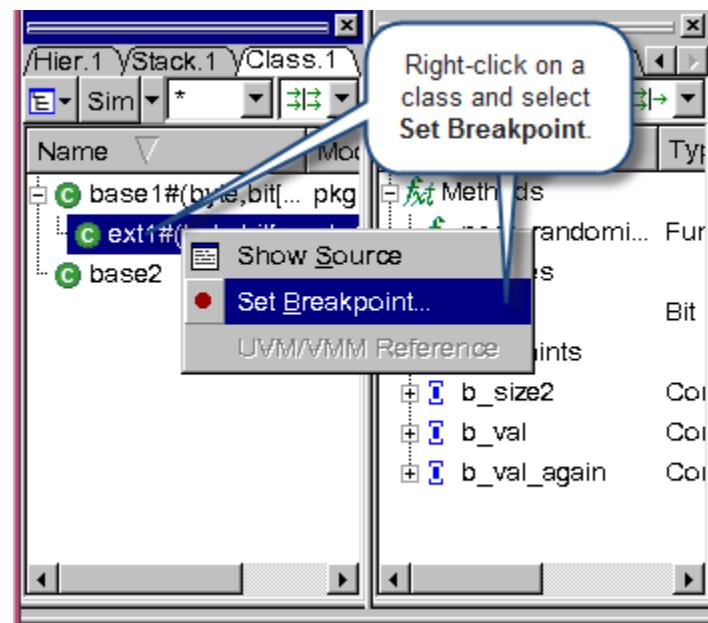
For example:

```
%dve> stop -solver -class {ext1#(byte,bit[31:0],bit)} \  
-solver_cond { a > 0 }
```

To set breakpoints on parameterized classes from the Class Browser:

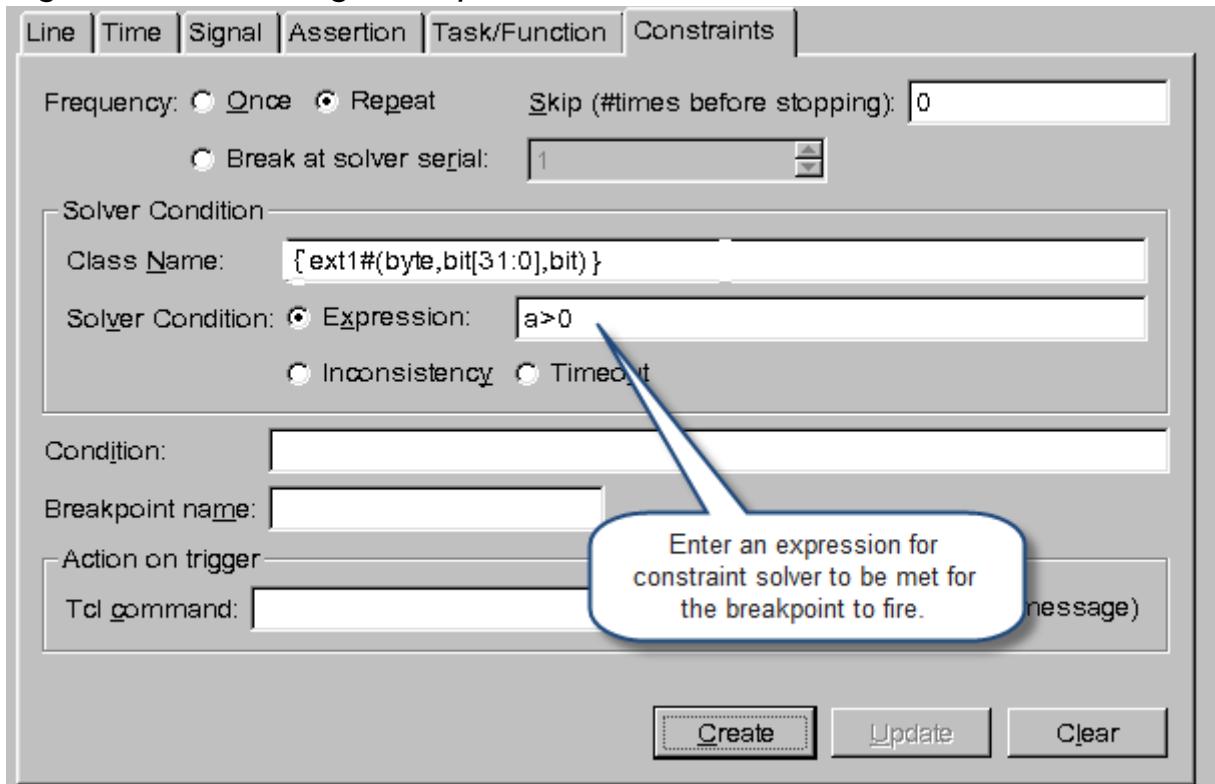
1. Right-click on a class and select **Set Breakpoint**, as shown in the [Figure 14-5](#).

Figure 14-5 Setting Breakpoint on Parameterized Classes



2. Enter an expression for the constraint solver in the **Expression** field of the **Breakpoints** dialog box, as shown in [Figure 14-6](#).

Figure 14-6 Setting Breakpoint on Parameterized Classes



3. Click the **Create** button.

Setting Breakpoints on Constraint Inconsistency and Timeout

You can set a breakpoint on constraint inconsistency and timeout from the command line or from the Breakpoints dialog box. This section describes the following topics:

Use the following syntax to set a breakpoint on constraint inconsistencies and timeout from the command line:

```
stop -solver [-class <class_name>] [-inconsistency|-timeout  
| -solver_cond <expr>]
```

The following points describe the usage of this syntax:

- **-class** is optional. If you don't specify a class, then **-inconsistency** and **-timeout** will apply to the `std::randomize()` call.
- **<class_name>** is the name of class definition to which the built-in `randomize()` method belongs. For example, `obj1.randomize()`. Here, `obj1` is instantiated object of the `A` class definition. You can specify **-class A** for the `randomize()` method call from any instantiated objects of `A`. If you specify **<class_name>** as `*`, then all classes will be considered.
- **-inconsistency** specifies the condition that will be satisfied when any inconsistent constraints exist in the specified class.
- **-timeout** specifies the condition that will be satisfied when any timeout constraints exist in the specified class. The value of `timeout` is set by using `simv` option
`+ntb_solver_cpu_limit=<timeout value>`.
- You cannot use `solver_cond`, `-inconsistency`, and `-timeout` together. VCS generates an error message, if these options are used together.
- `solver_cond`, `-inconsistency`, and `-timeout` are optional. If none of them are specified, then VCS stops at all `randomize` calls of the specified class.

Examples

Use the following command to detect potential constraint inconsistencies from class C:

```
% dve> stop -solver -class C -inconsistency
```

Use the following command to detect potential constraint timeout from class C:

```
% dve> stop -solver -class C -timeout
```

Use the following command to detect potential constraint inconsistencies from any class:

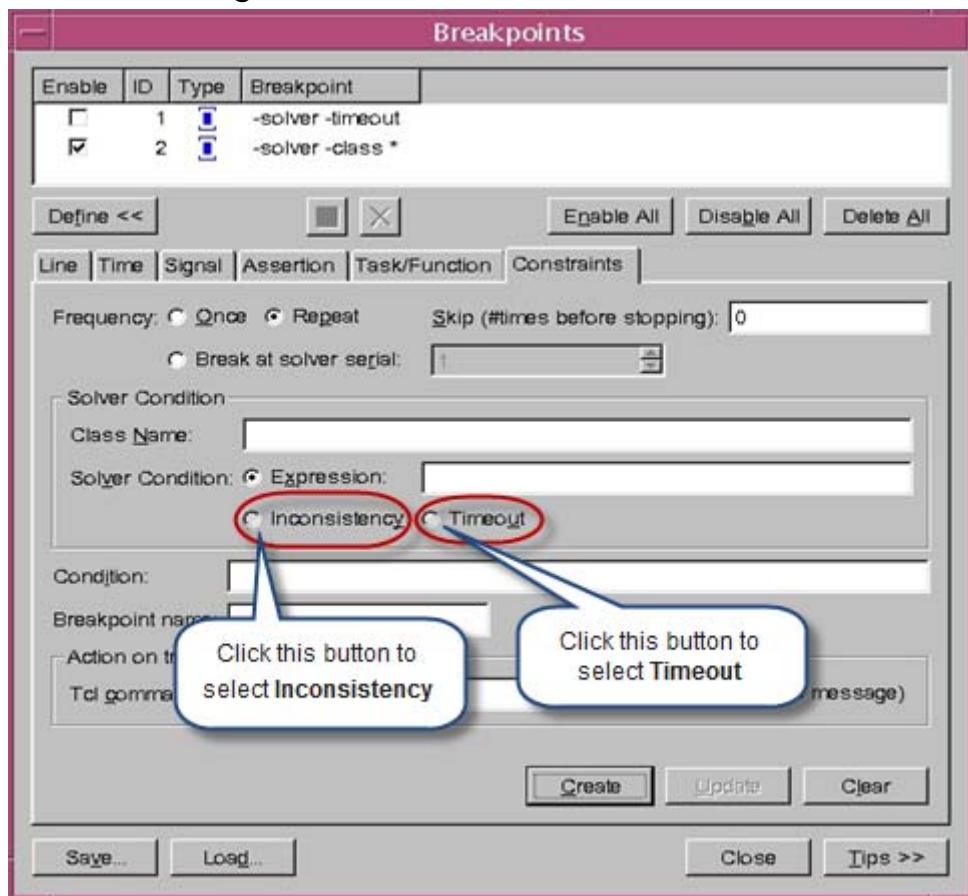
```
% dve> stop -solver -class * -inconsistency
```

Use the following command to detect potential constraint inconsistencies from any class:

```
% dve> stop -solver -class * -timeout
```

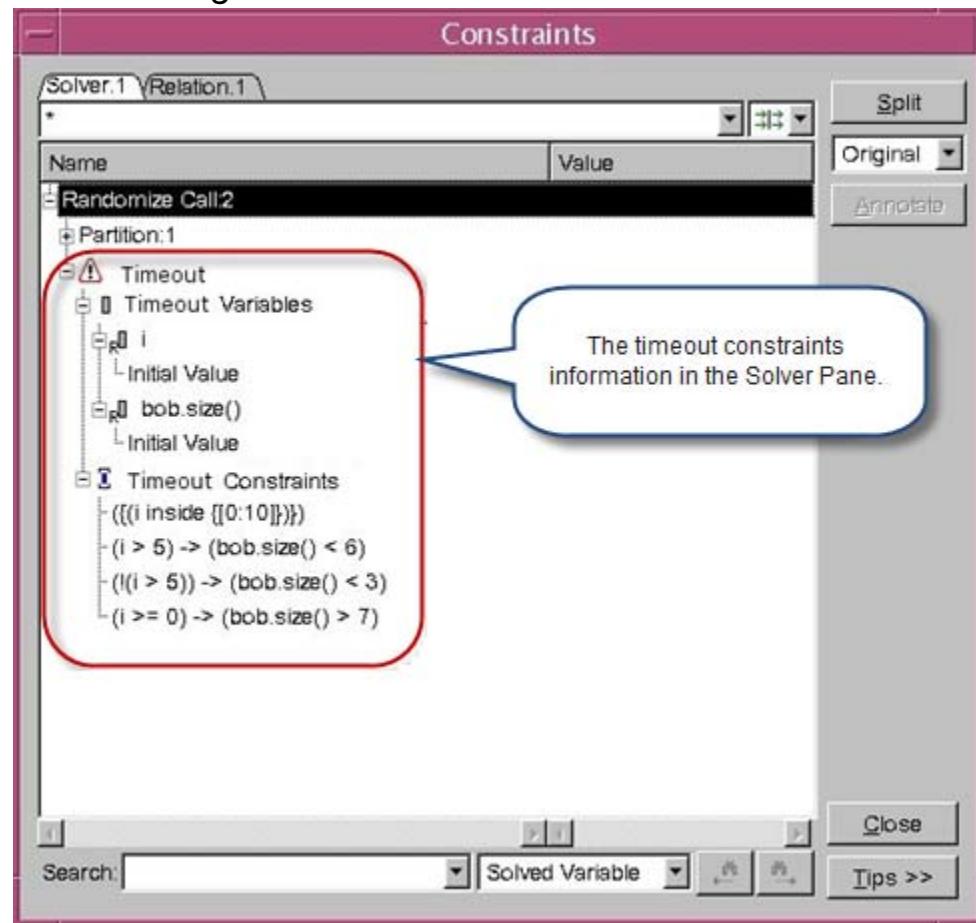
To set a breakpoint on constraint inconsistency and timeout from the Breakpoints dialog box, select **Inconsistency** or **Timeout**, as shown in [Figure 14-7](#).

Figure 14-7 Detecting Constraint Inconsistencies and Timeout



You can view the timeout constraints information in a separate partition of the Solver Pane, as shown in the following figure. For more information on Solver Pane, see “[Using the Solver Pane](#)” on page 25.

Figure 14-8 Viewing Timeout Information

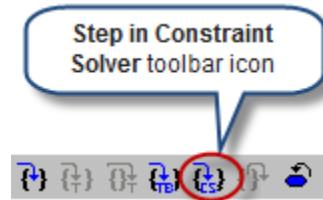


Analyzing a Randomization Call

Once you are at a randomize call, you can:

- Click the **Next** or **Continue** toolbar icon to continue execution. This does not open the Constraints dialog box.
- Use the `step -solver` command or its equivalent toolbar icon (see [Figure 14-9](#)) to open the Constraints dialog box.

Figure 14-9 Step in Constraint Solver Toolbar Icon



You can apply this command only on the current line to be executed. This command takes you directly into constraint debug mode if the current line has a randomize call. Otherwise, it goes to the next line.

The following steps explain how to analyze a randomization call when you use the `step -solver` command:

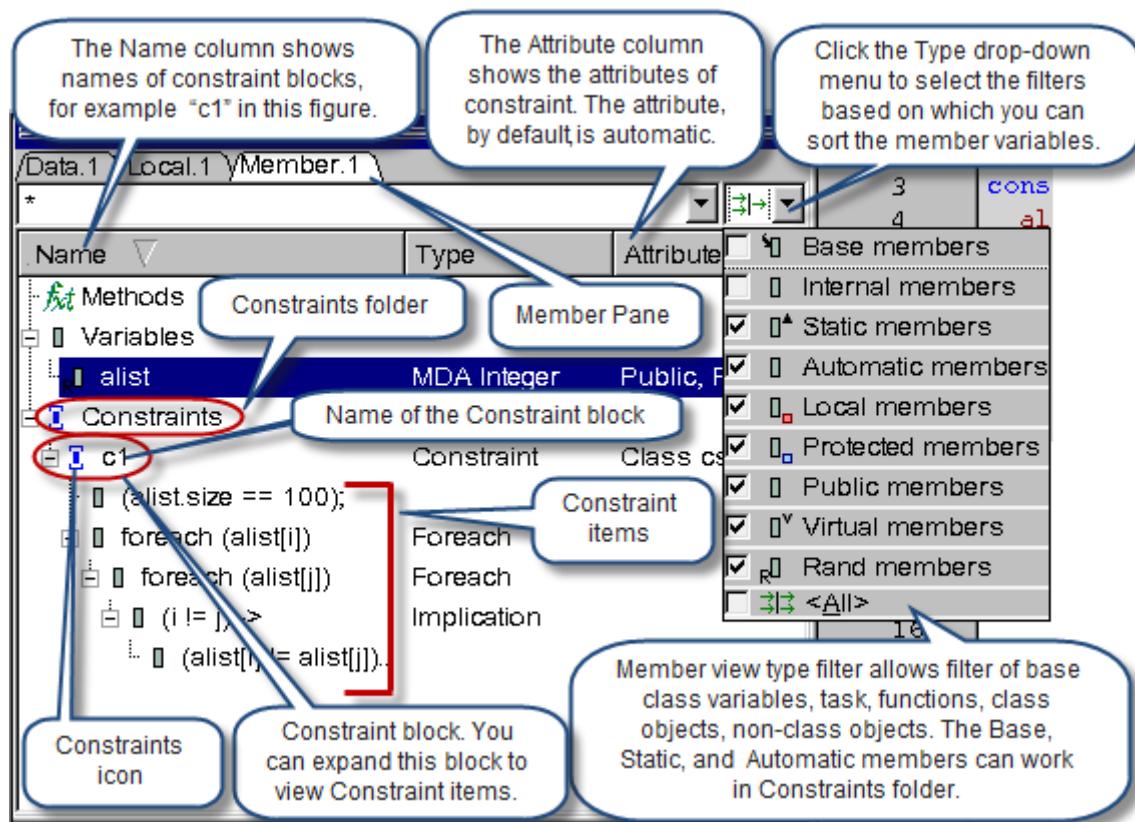
1. Execution breaks at the beginning of the `post_randomize()` call, if you have overwritten it. If you have not overwritten the `post_randomize()` call, execution breaks at the end of the randomize call.
2. In DVE, the Constraints dialog box opens up automatically if you execute the `step -solver` command or click the **Step in Constraint Solver** toolbar icon. It includes two tabs: Solver Pane and Relation Pane, as shown in [Figure 14-16](#). You can also select **Simulator > Constraints** to open this dialog box when constraints data is ready.

When you finish analyzing a randomize call, you can continue execution as usual. Program execution stops at the next breakpoint or runs to completion.

Constraint Browsing in Class Hierarchy Browser

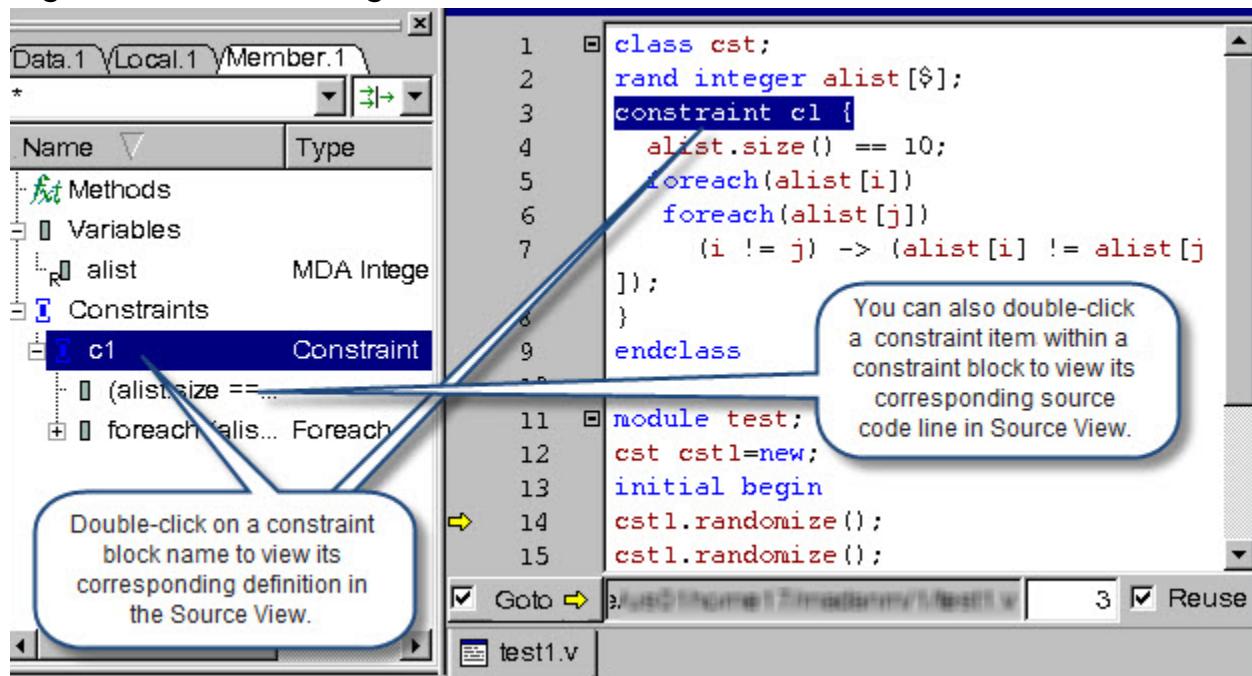
You can use the Constraints folder in the Member Pane to view static information about constraints in DVE. This folder displays all constraints under it which are defined in the selected class, as shown in [Figure 14-10](#).

Figure 14-10 Viewing Static Constraint Information in Member Pane



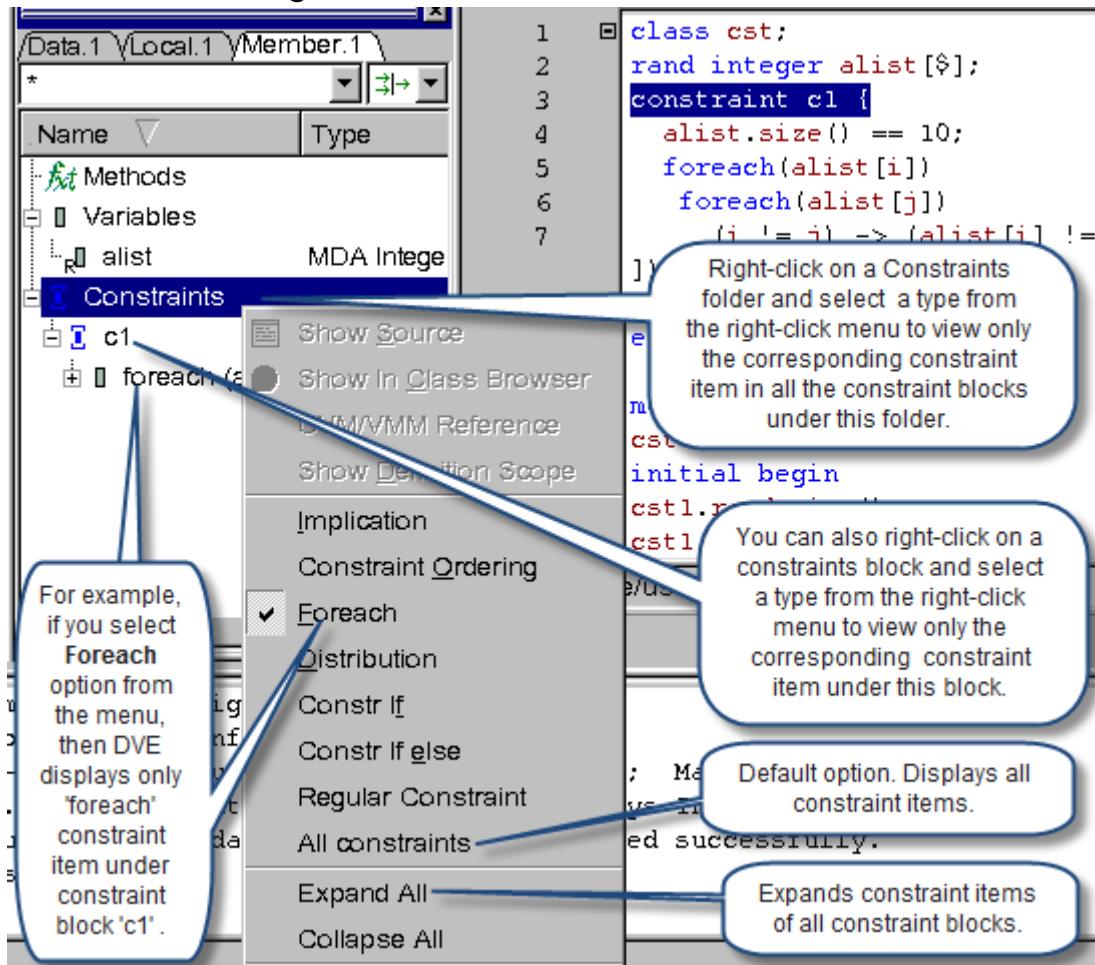
DVE allows you to view the corresponding definition of a constraint block in the Source View, as shown in [Figure 14-11](#).

Figure 14-11 Viewing Definition of a Constraint Block in Source View



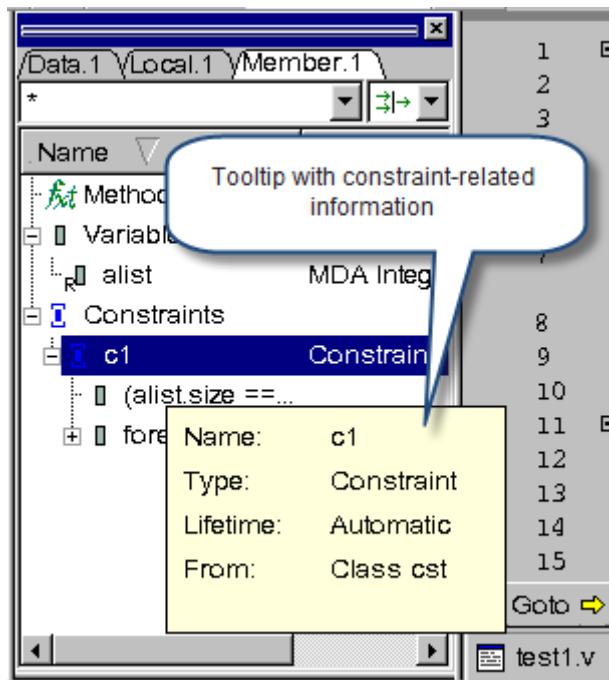
DVE allows you to select the constraint items to be displayed in the constraint block, as shown in [Figure 14-12](#). By default, all constraint items are displayed in the Member Pane.

Figure 14-12 Viewing Constraint Items



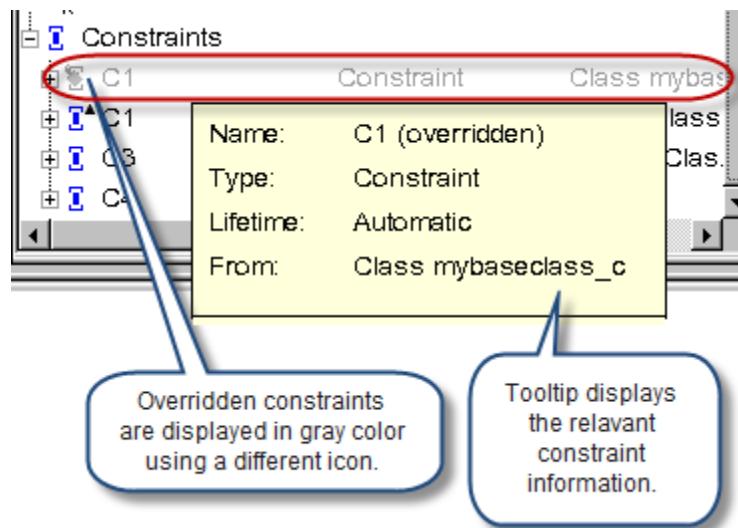
Mouse over the Constraints folder, constraint block, or a constraint item to view a ToolTip with relevant constraint information, as shown in Figure 14-13.

Figure 14-13 ToolTip with Constraint-related Information



Overridden constraints are displayed in gray using a different icon. For example, if a constraint is overridden in a derived class, you can select **Base members** in the Type filter to view the constraint from the base class, as shown in [Figure 14-14](#).

Figure 14-14 Viewing Overridden Constraints



Browsing Objects in Local Pane

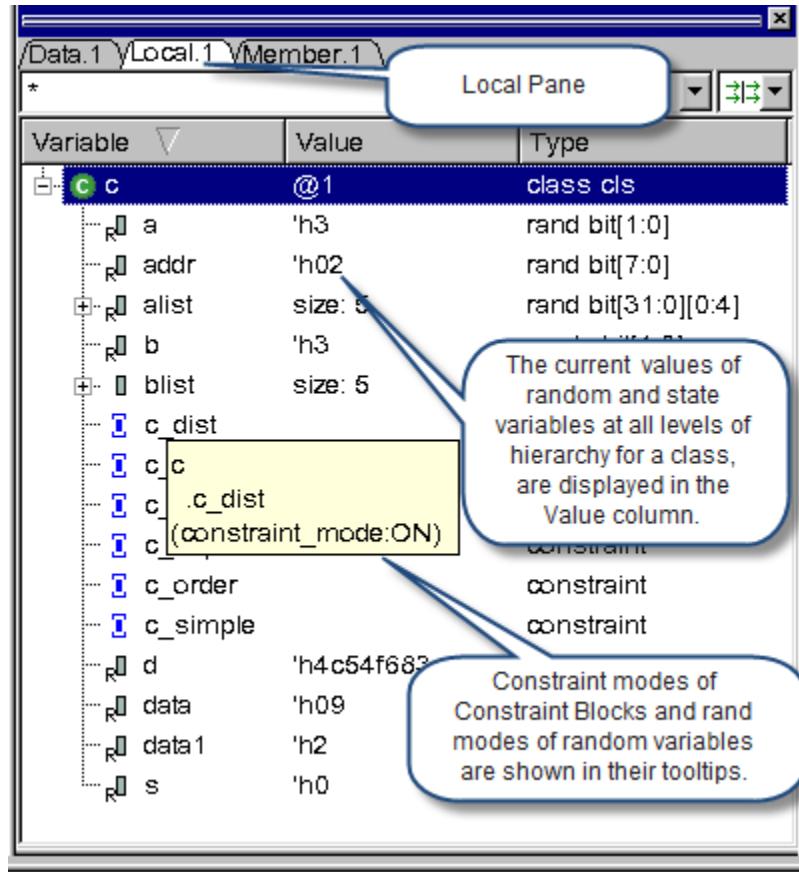
The Local Pane shares a tabbed view with the Data tab. The Local Pane displays variables of a selected scope in the Stack Pane.

Note the following points about how constraint object browsing works in the Local Pane (shown in [Figure 14-15](#)):

- For `foo.randomize()`, you can only browse `foo` and parent hierarchies of `foo`. In the Local Pane, DVE displays random variables, state variables, constraint blocks, and constraint expressions in each class at each level of hierarchy.
- For random fields, DVE displays the values of their `rand_modes` in the ToolTips.
- For `foo`, DVE displays the current values of random and state variables at all levels of hierarchy, in the Value column of the Local Pane.

- Constraint modes of Constraint Blocks are displayed in their ToolTips.
- For a `std::randomize` call, you can browse each argument as a separate object in addition to `foo`.
- DVE also displays the following information about constraints in the Local Pane:
 - The icon for a variable is displayed in gray if `rand_mode` is off.
 - The icon for a constraint block is displayed in gray if `constraint_mode` is off.
 - Constraint blocks defined inside the base or parent class are displayed under the super folder in the Data Pane.
 - You can use the `show -randomize` command to get the serial number for the last randomize call.

Figure 14-15 Object Browsing in Local Pane



Using the Constraints Dialog

DVE displays the Constraints dialog box when you execute the `step -solver` command or click the **Step in Constraint Solver** toolbar icon . This dialog box includes two tabs: Solver Pane and Relation Pane, as shown in Figure 14-16.

Using the Solver Pane

The following debug data of a certain randomize call is displayed in the Solver Pane (see Figure 14-16):

- The serial number of the randomize call (Randomize Call)
- Problem partition numbers
- The call of function (Function Call)
- The name and mode (On/Off) of constraint block (Constraint Block)
- Constraint expression (Constraint Expression)
- The name, type (Rand/RandC/State), mode (On/Off if type is Rand/RandC), value, and initial value range of variable (Variable)
- The name and value of function call argument (Argument)
- Object ID information of a class
- If there is inconsistency, then name, type, mode of variables, and constraint expressions are displayed under the inconsistent folder, as shown in [Figure 14-23](#).

Icon and Text Color in Constraints Dialog

- The icon and text for soft/dropped constraints is displayed in gray.
- The icon and text for off constraint blocks and all constraints inside off constraint blocks is displayed in gray.
- All other constraints are displayed in black.
- The icon for rand off variables is displayed in gray.

Figure 14-16 Constraints Dialog Solver Pane

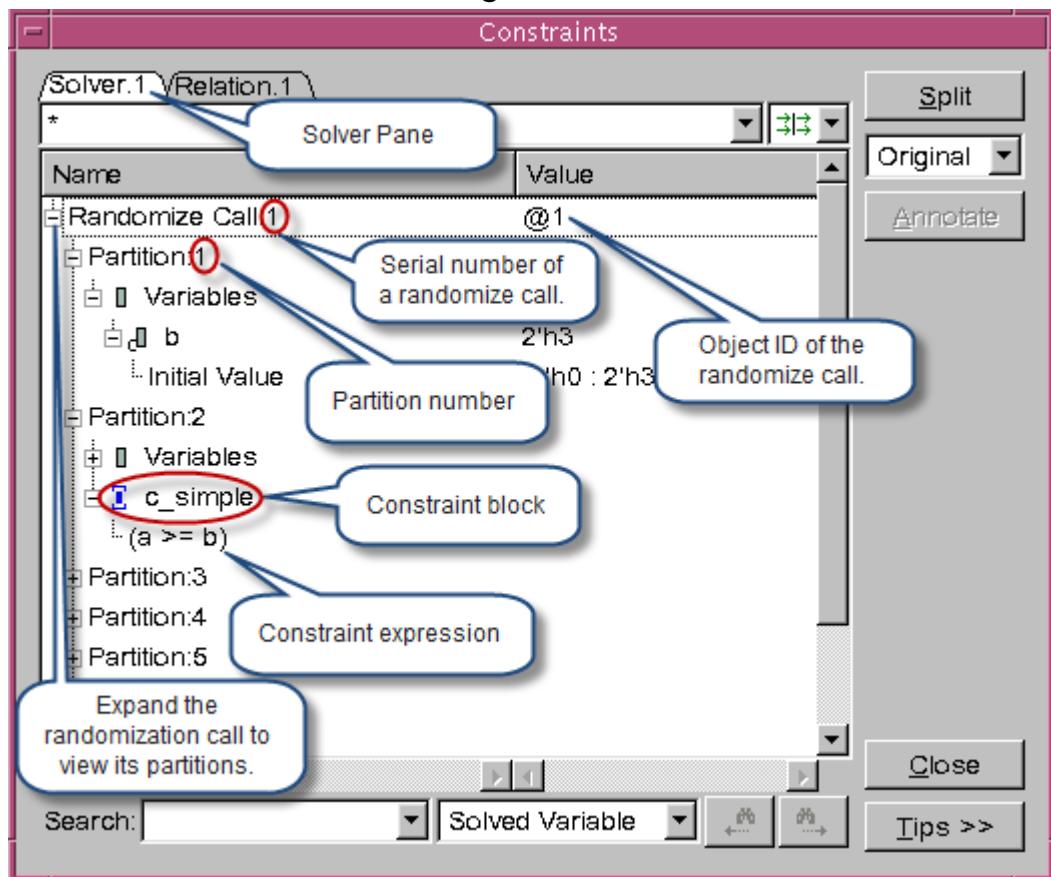
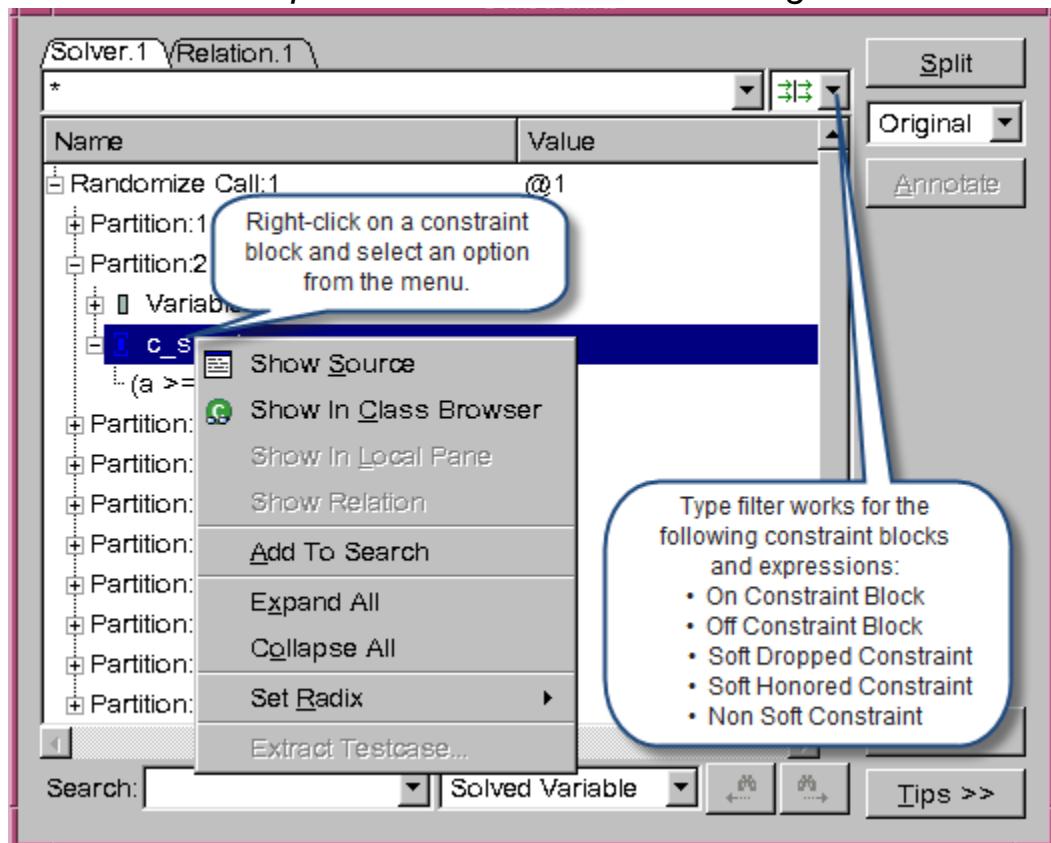


Table 14-2 explains the right-click menu options in the Constraints dialog box Solver Pane:

Table 14-2 Right-click Options in Constraints Solver Dialog Pane

Option Name	Description
Show Source	This option works for Randomize Call, Function Call, Constraint Block, Constraint Expression, and Variable. It opens the corresponding file and highlights the corresponding line in the Source View.
Show In Class Browser	This option works for Variable and Constraint Block. It locates the definition in the Class Browser (Class Pane and Member Pane).
Show Relation	This option works only for Variable. The selected variable and related variables are displayed in the Relation Pane.
Set Radix	Allows you to change the radix type of a variable value. For more information, see “ Changing Radix Type of a Variable or Constraint Expression in Constraints Dialog Box ” .
Extract Testcase	Allows you to extract partition-level test cases. For more information, see “ Extracting Test Case ” .

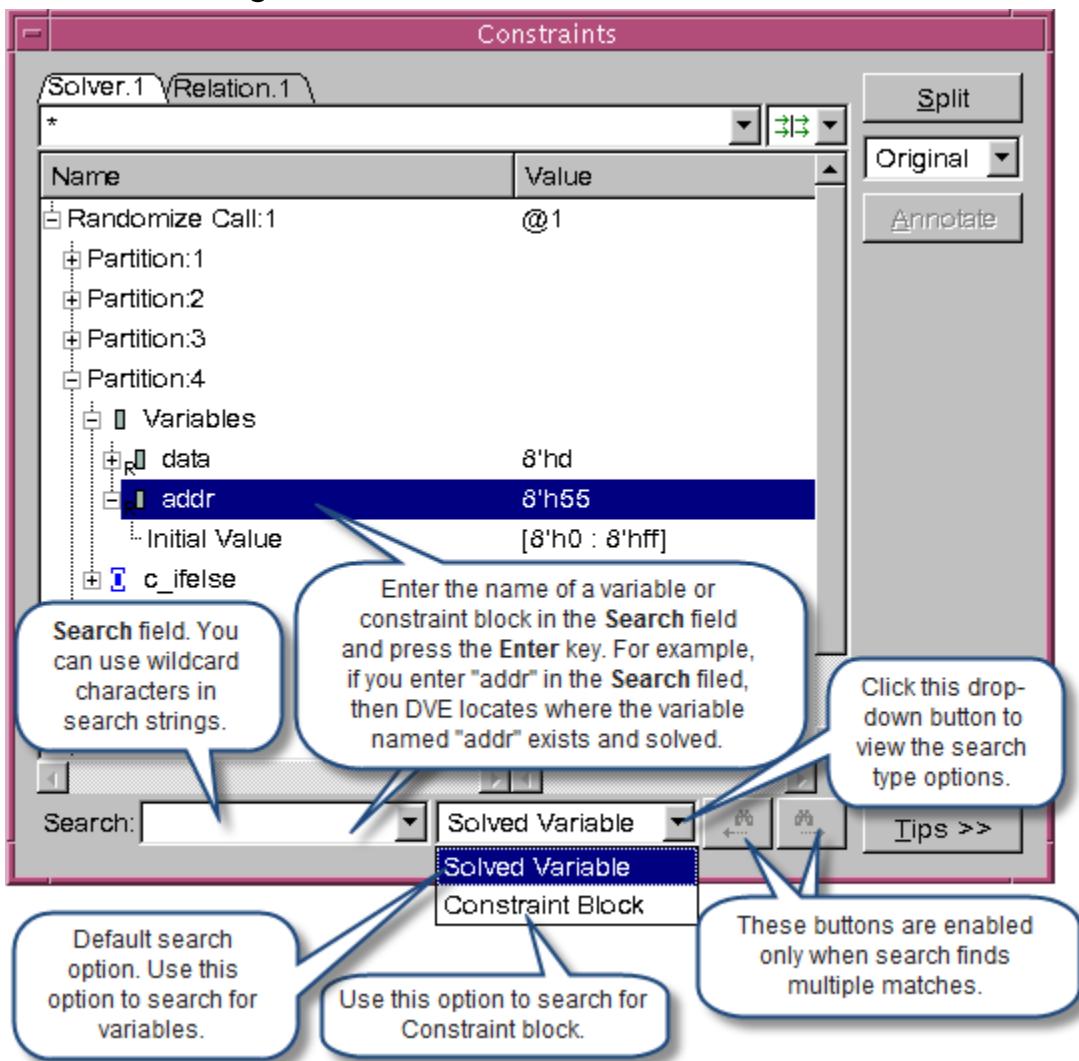
Figure 14-17 Menu Options in the Constraints Dialog Solver Pane



Searching Variables or Constraints in Solver Pane

You can use the **Search** field to search for variables or constraint blocks in the Solver Pane (see [Figure 14-18](#)).

Figure 14-18 Using the Solver Pane Search



Using the Relation Pane

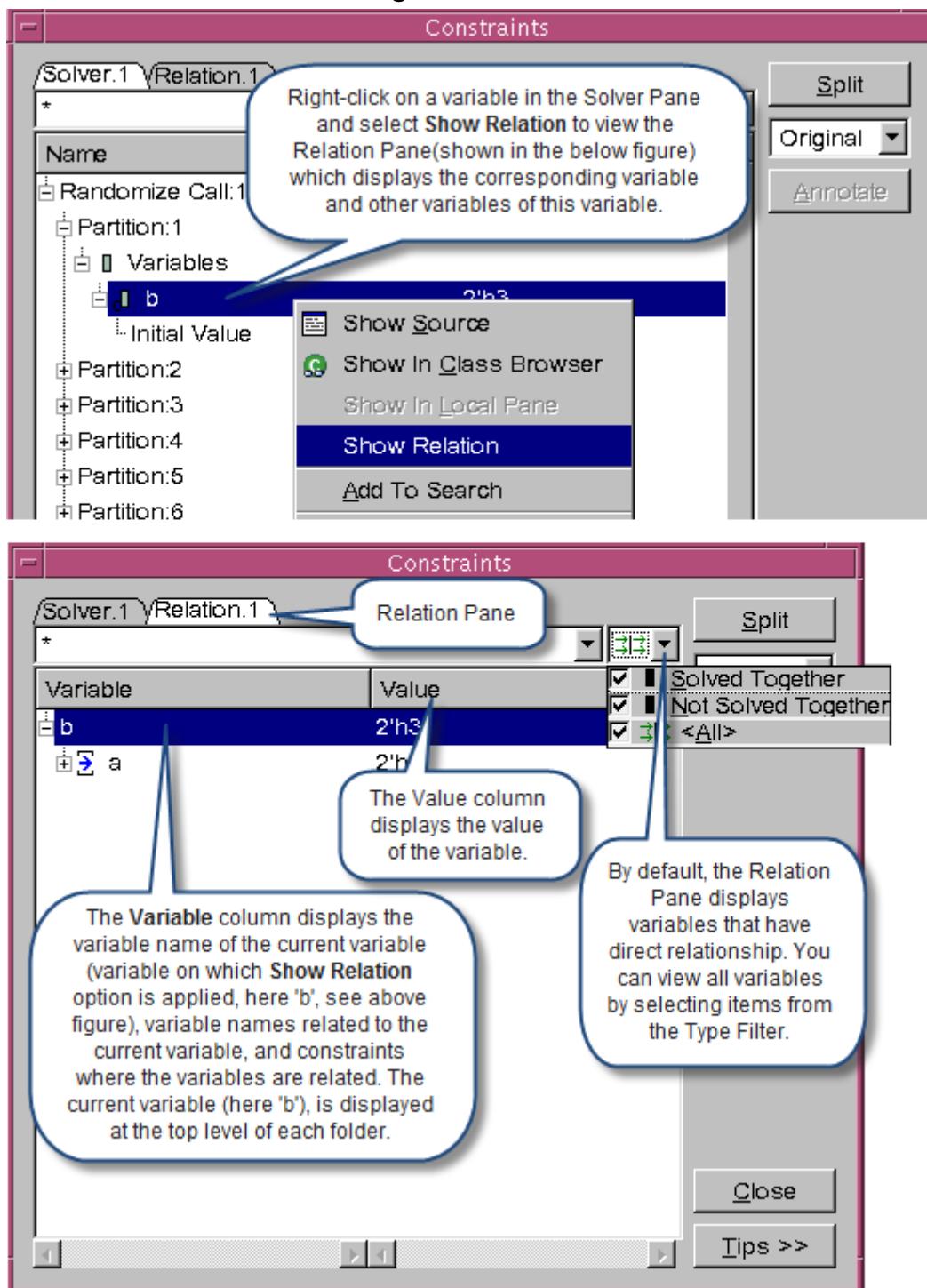
When you right-click on any variable in the Solver Pane and select the **Show Relation** option, the Relation Pane appears and the corresponding variable and other variables related to this variable are displayed in the Relation Pane (see [Figure 14-19](#)).

[Table 14-3](#) explains the right-click menu options in the Constraints dialog box Relation Pane:

Table 14-3 Right-click Options in Relation Pane

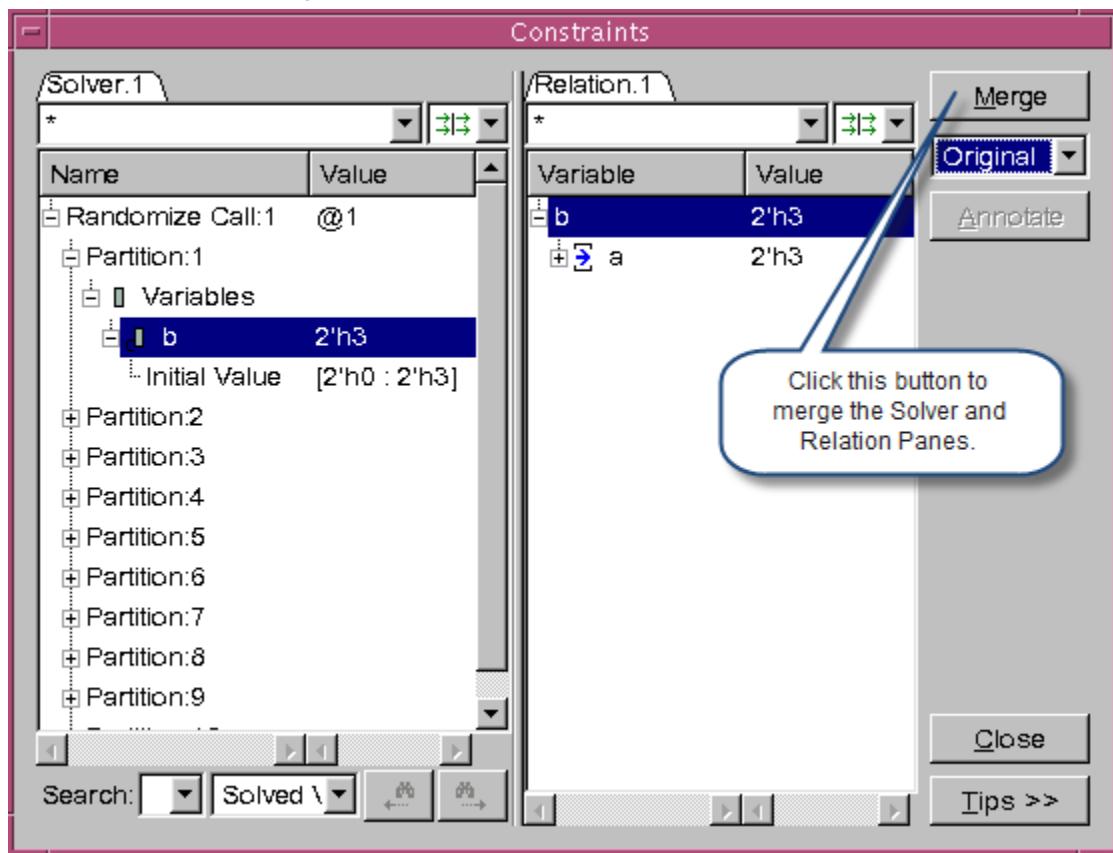
Option Name	Description
Show Source	This option works for Constraint Expression and Variable. It opens the corresponding file and highlights the corresponding line in the Source View.
Show In Class Browser	This option works for Variable. It locates the definition in the Class Browser (Class Pane and Member Pane).
Show Relation	This option works for a Variable not on the top level. The selected variable and related variables are displayed in the Relation Pane.
Show In Solver	This option works for Constraint Expression. It highlights the selected constraint expression in the Solver Pane.
Set Radix	Allows you to change the radix type of a variable value. For more information, see “Changing Radix Type of a Variable or Constraint Expression in Constraints Dialog Box” .
Delete	This option deletes the currently selected root variable from the Relation Pane.

Figure 14-19 Constraints Dialog Relation Pane



To view the Solver Pane and Relation Pane side by side, as shown in [Figure 14-20](#), click the **Split** button near the top right-hand side of the Constraints dialog box.

Figure 14-20 Viewing Solver Pane and Relation Pane Side by Side



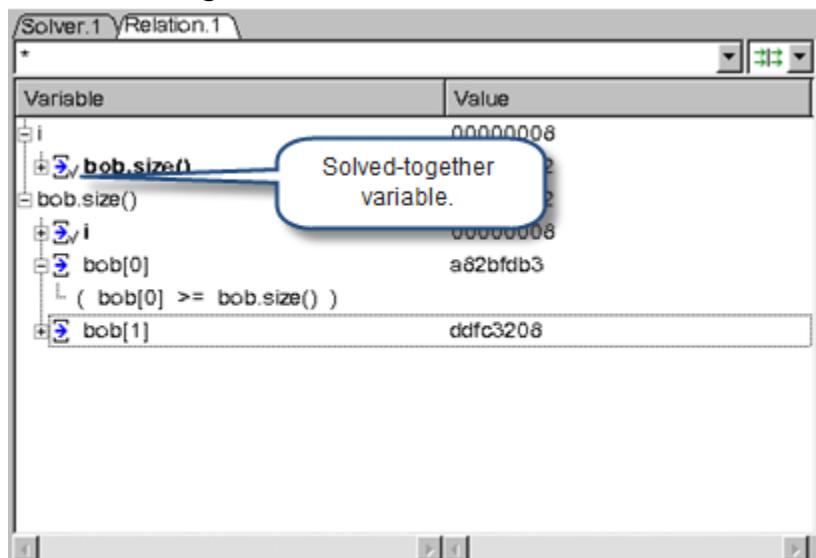
DVE displays the solved-together variables using a different icon in the corresponding line, as shown in [Figure 14-21](#).

[Table 14-4](#) shows the icons used by DVE to indicate solved-together variables.

Table 14-4 DVE Icons for Solved-together Variables:

Icon	Indicates
	Solved-together variable
	Not solved-together variable

Figure 14-21 Solved-together and Related Variables



Inconsistent Constraints

If the solver finds inconsistent constraints while executing the `step -solver` command, DVE displays a message box (see [Figure 14-22](#)). Click **OK** to go to the Constraint dialog box Solver Pane, where minimal inconsistent constraints are displayed, as shown in [Figure 14-23](#).

Figure 14-22 Message Box

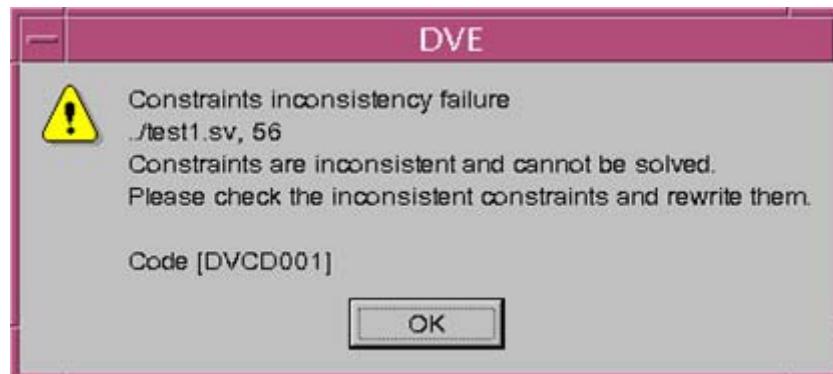
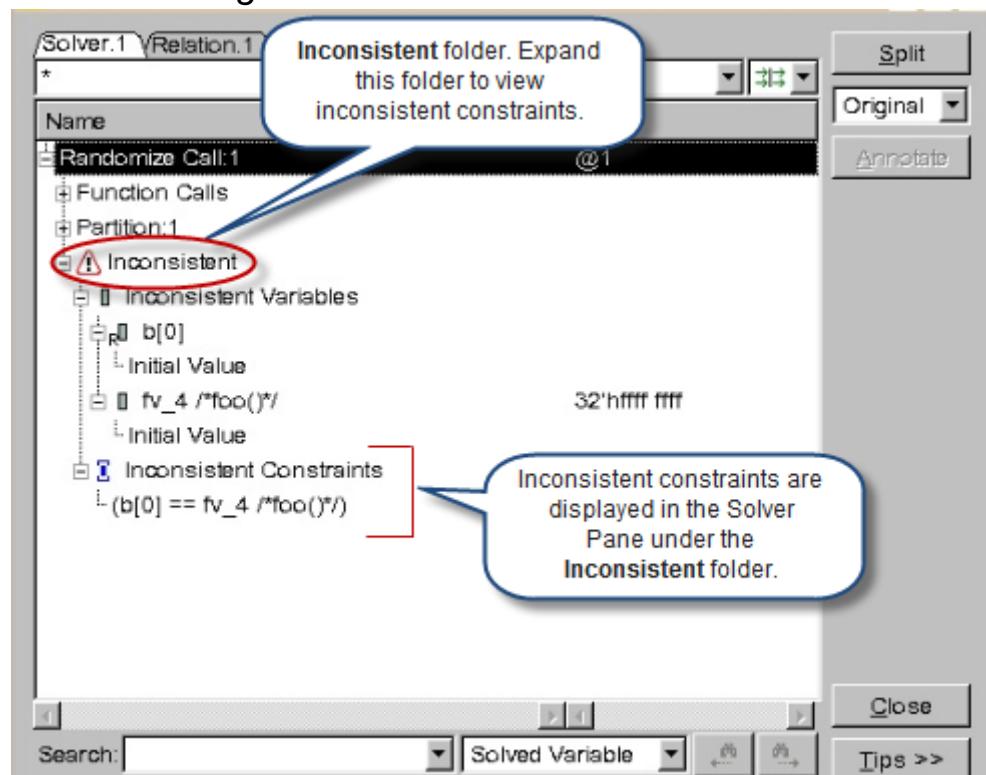


Figure 14-23 Viewing Inconsistent Constraints in Solver Pane



Debugging Constraints Example

This section explains a constraints debugging example. [Example 14-1](#) shows how constraints are defined in a sample design file. You can view these constraints in the DVE Member Pane, Local Pane, and Constraints dialog box.

Example 14-1 Design File with Constraints (cstr_debug.sv)

```
class cls;
    rand bit[1:0] a;
    randc bit[1:0] b;
    rand bit[3:0] data1;
    rand bit[31:0] alist[5];
    bit[31:0] blist[5];
    rand bit[7:0] addr, data;
    rand bit s;
    rand bit[31:0] d;

    constraint c_simple {
        a >= b;
    }

    constraint c_dist {
        data1 dist {4'b0010 := 1, 4'b1000 :=2, 4'b1010 :=5};
    }

    constraint c_foreach {
        foreach(alist[i])
            foreach(blist[j])
                (alist[i] != blist[j]);
    }

    constraint c_ifelse {
        if (addr <= 7)
            data < 10;
        else if (addr > 7)
            (data >=10 && data <= 15);
    }
```

```

constraint c_implication { s -> d == 0; }
constraint c_order { solve s before d; }

endclass

module test;
  cls c=new;
  initial begin
    repeat(10) c.randomize();
  end
endmodule

```

Compile the `cstr_debug.sv` code, shown in [Example 14-1](#), as follows:

```
% vcs -sverilog -debug_all cstr_debug.sv
```

Invoke the DVE GUI:

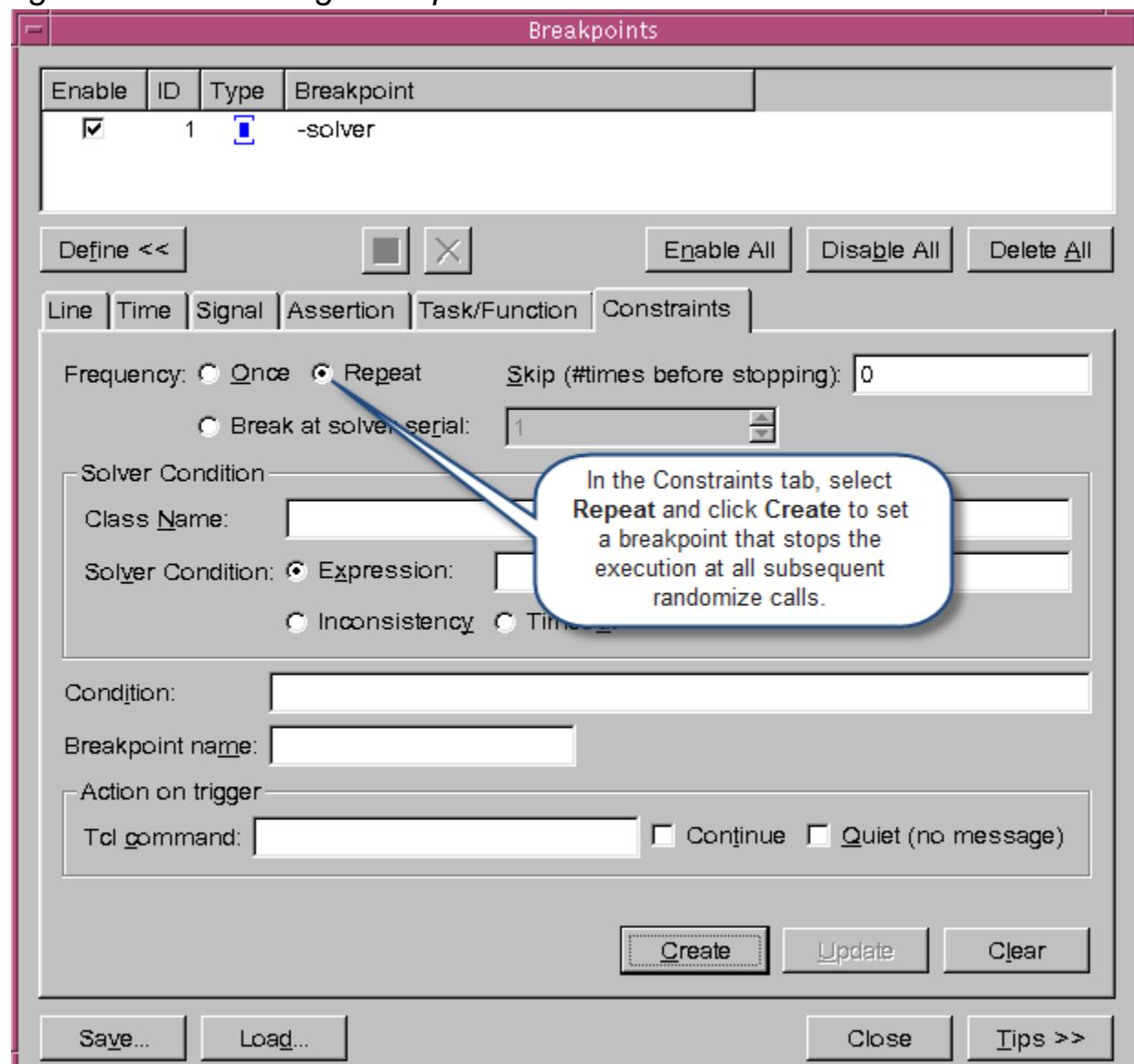
```
% simv -gui&
```

To debug the constraints in DVE, follow these steps:

1. Set breakpoints in the Breakpoints dialog box to stop the simulation at a certain randomize call. To open the Breakpoints dialog box, select **Simulator > Breakpoints** and click **Define>>** to display the breakpoint creation tabs.

For more information on the types of breakpoints that you can set in the Breakpoints dialog box, see [“Breaking Execution at a Randomize Call” on page 4](#). For example, [Figure 14-24](#) shows how to set a breakpoint that stops execution at all subsequent Randomize Calls.

Figure 14-24 Setting Breakpoints in the Constraints Tab



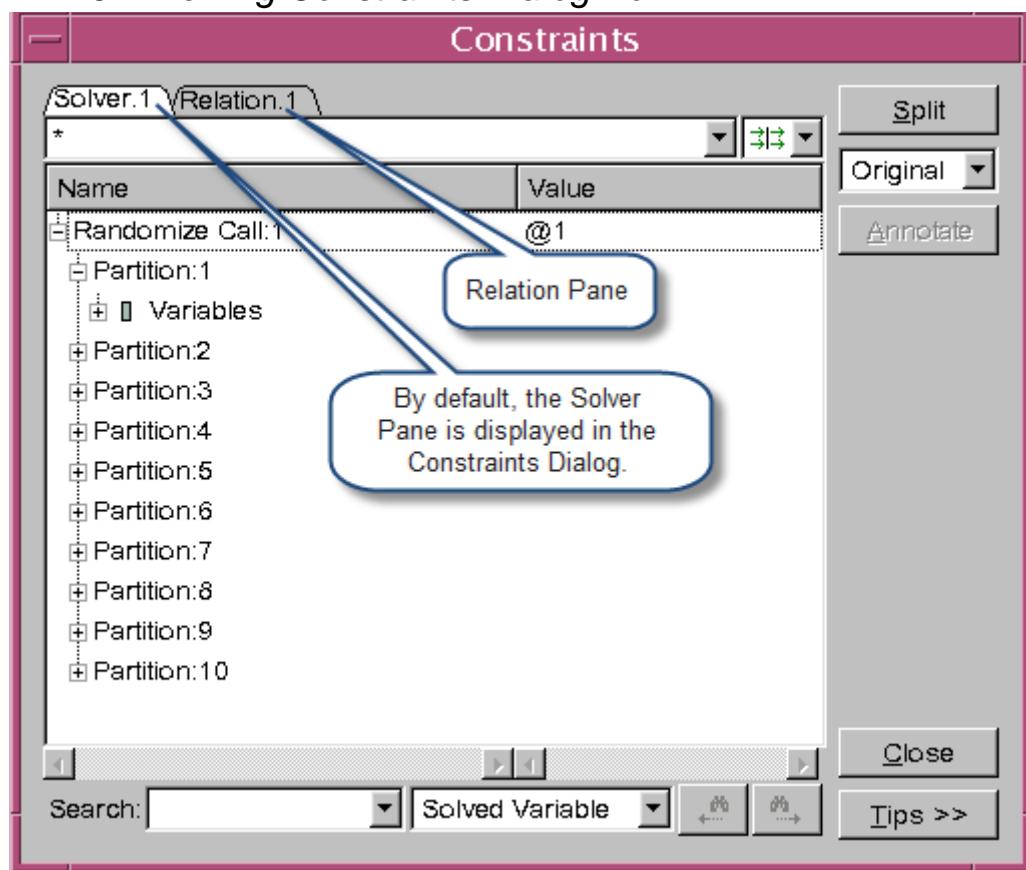
2. Click to start the simulation. The current active line stops at the first Randomize Call, as shown in Figure 14-25.

Figure 14-25 Execution Stopping at the First Randomize Call

```
1  class cls;
36
37 module test;
38   cls = new();
39   initial begin
40     repeat(10) c.randomize();
41   end
42 endmodule
```

3. To view the Constraints Folder and constraint blocks in the Member Pane, select a class in the Class Pane.
4. To see the constraint variables in the Local Pane, select a scope in the Stack Pane.
5. To open the Constraints dialog box, execute the `step -solver` command in the DVE console, or click the **Step in Constraint Solver** toolbar icon  .

Figure 14-26 Viewing Constraints Dialog Box



Changing Radix Type of a Variable or Constraint Expression in Constraints Dialog Box

DVE allows you to change the radix type (the numeric base used to display integer values) of a variable value or values of a constraint expression in the Solver Pane and Relation Pane of the Constraints dialog box. You can use one of the following two methods to do this:

- [Using Constraints Dialog Box](#)
- [Using Tcl command](#)

Supported Radix Types

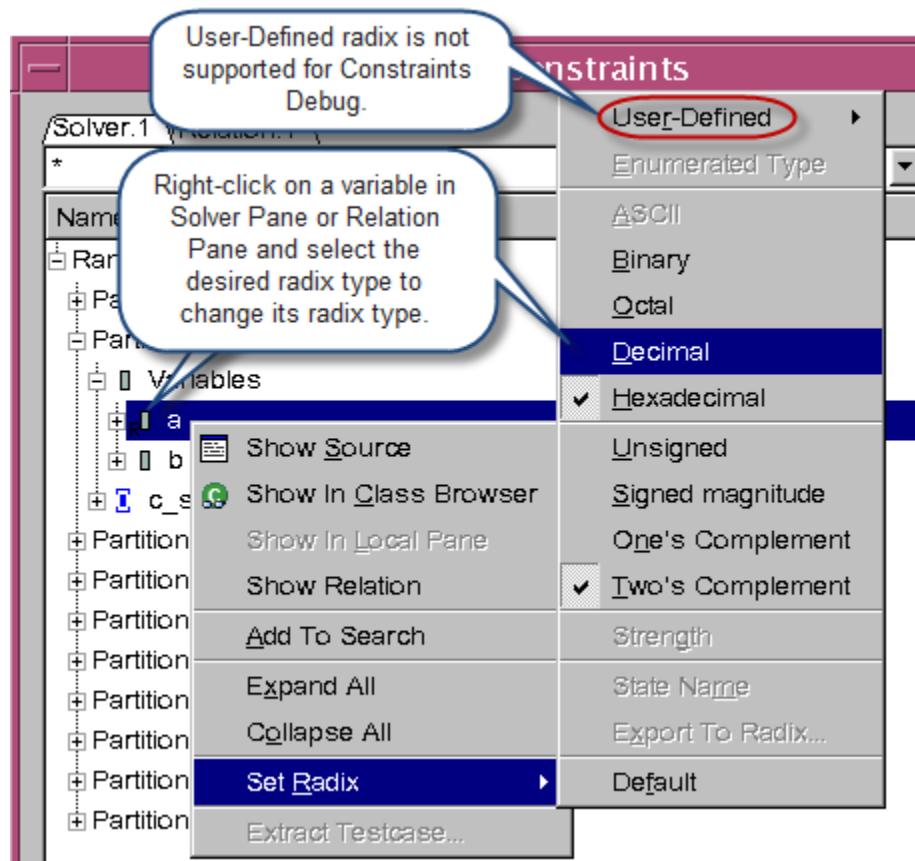
- Binary
- Octal
- Decimal
- Hexadecimal

Using Constraints Dialog Box to Change the Radix Type of a Variable or Constraint Expression

Changing Radix Type of a Variable

To change the radix type of a variable value, right-click on the desired variable name in Solver Pane or Relation Pane, click **Set Radix**, and then select the desired radix type from the list, as shown in [Figure 14-27](#). If this variable is present in different places (for example, in both Solver and Relation panes), then changes will be applied to selected variable only.

Figure 14-27 Changing the Radix Type of a Variable

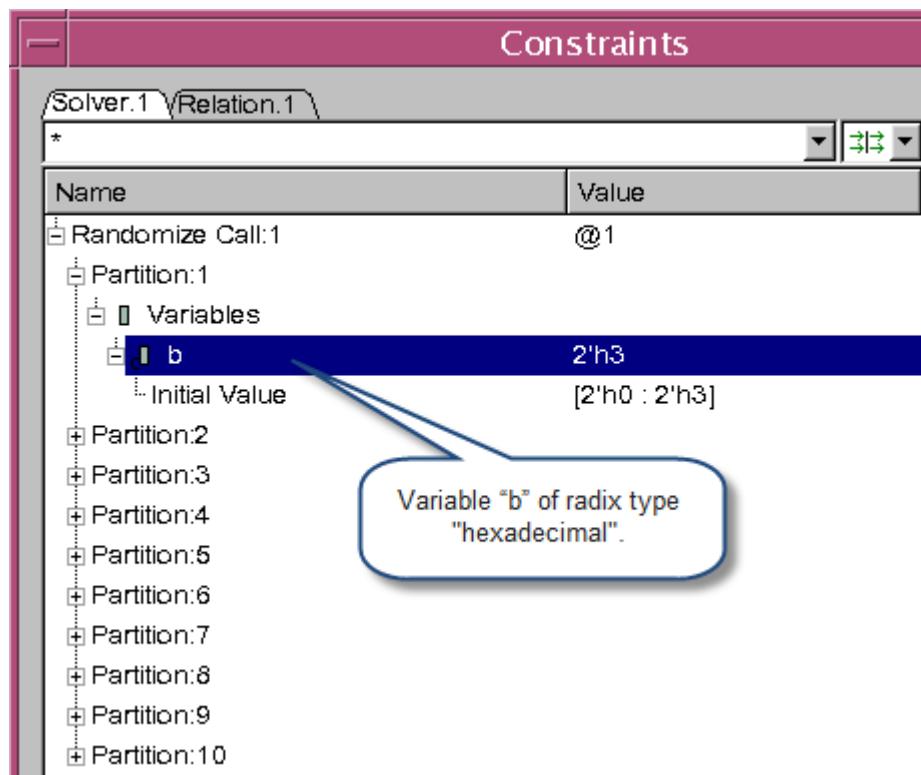


Note:

- Changing radix of a variable in Constraints dialog box will not affect it in other views (Local Pane, Member Pane, and so on), and vice versa.
- This feature does not support values of type enum.
- The “User-Defined” radix option (see [Figure 14-27](#)) is not supported.

For example, in the Constraints dialog box shown in [Figure 14-28](#), consider variable “b” whose radix type is “Hexadecimal”.

Figure 14-28 Changing the Radix Type of a Variable Value



[Table 14-5](#) shows how DVE displays the value of this variable in other radix formats.

Table 14-5 Value of a Variable “b” in Different Radix Formats

Radix	Final Value	Initial Value Range
Decimal	3	[0 : 3]
Binary	2'b11	[2'b0 : 2'b11]
Hexadecimal	2'h3	[2'h0 : 2'h3]
Octal	2'o3	[2'o0 : 2'o3]

Changing Radix Type of Constraint Expression

To change the radix type of a constraint expression, right-click on the desired constraint expression in Solver Pane or Relation Pane, click **Set Radix**, and then select the desired radix type from the list. Every constant string inside the expression will be transformed according to the new radix. If this constraint expression is present in different places, then changes will be applied to selected constraint expression only.

For example, consider the constraint expression highlighted in [Figure 14-29](#):

Figure 14-29 Changing the Radix Type of a Constraint Expression

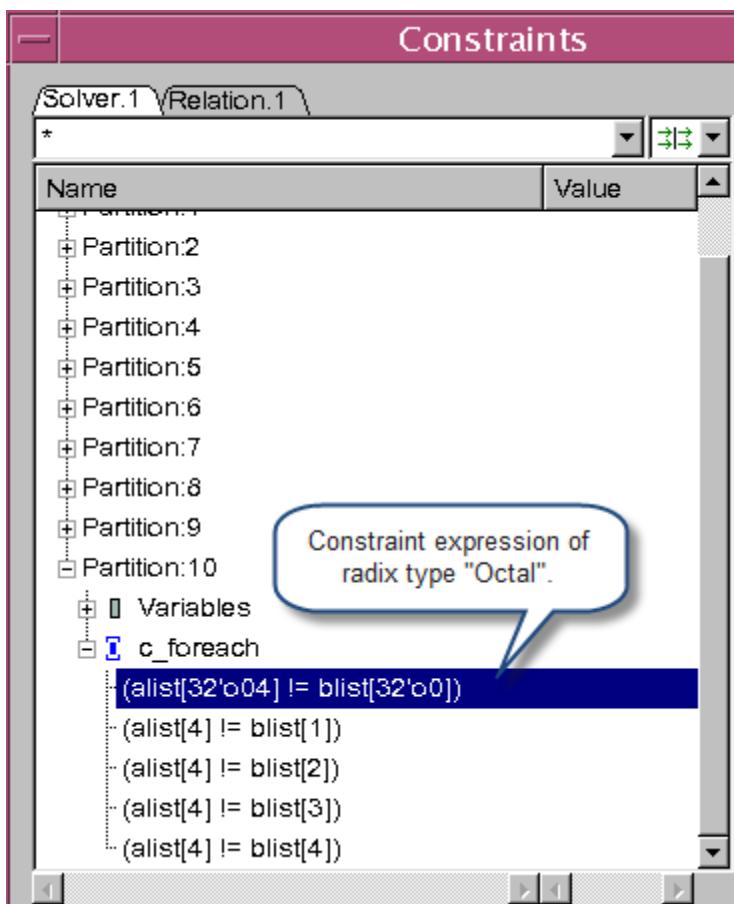


Table 14-6 shows this expression in different radix formats

Table 14-6 Changing Radix Type of a Constraint Expression

Expected Radix	Result
Decimal	(alist[4] != blist[0])
Binary	(alist[32'b100] != blist[32'b0])
Hexadecimal	(alist[32'h4] != blist[32'h0])
Octal	(alist[32'o04] != blist[32'o0])

Using Tcl Command to Change the Radix Type of a Variable or Constraint Expression

You can use the following Tcl command at the DVE command line to change the radix type of a variable or constraint expression:

```
gui_constraint_set_radix [-id windowid] -radix  
string [-vars list] [-exprs list]
```

Where,

`gui_constraint_set_radix` — Enables radix change for a variable or constraint expression in Constraints dialog box

`id windowid` — Window identifier.

`radix string` — Radix Type

`vars list` — Variables to set radix

`exprs list` — Constraint expressions to set radix

If the desired variable or constraint expression is present in different places, then changes will be applied to selected variable or constraint expression only.

Examples

```
gui_constraint_set_radix -id Solver.1 -radix hex -  
vars {x}// changes the radix type of variable "x"  
to "Hexadecimal".
```

```
gui_constraint_set_radix -id Solver.1 -radix oct -
exprs {x=1}//changes the radix type of all
constant strings inside the expression
"x=1" to "Octal".
```

Drag-and-Drop Support for Constraints Debug

This section describes drag-and-drop support for Constraints Debug under the following topics:

- [“Drag-and-Drop Support in Constraints Dialog Box”](#)
- [“Drag-and-Drop Items from Class Browser and Member Pane to Breakpoint Dialog Box”](#)

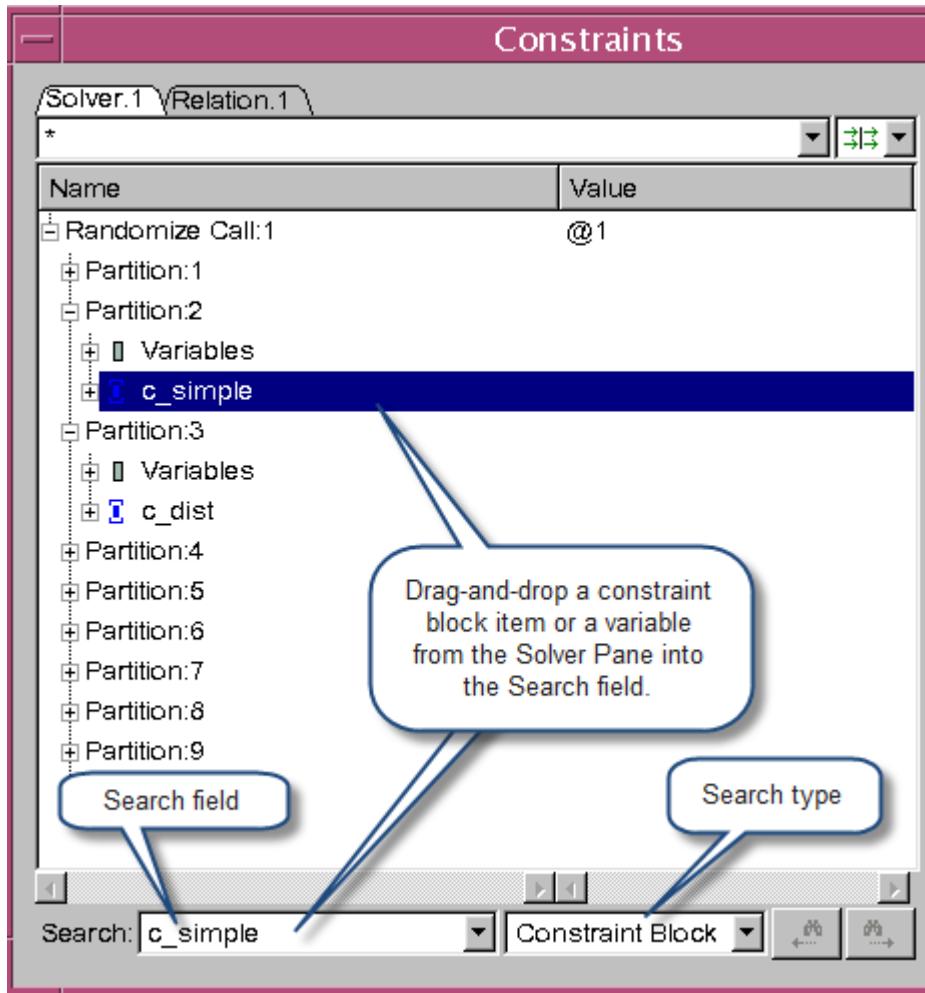
Drag-and-Drop Support in Constraints Dialog Box

The Constraints dialog box allows you to drag-and-drop a variable or constraint block item from the Solver Pane or Relation Pane into the Search field.

When you drag-and-drop an item from Solver Pane or Relation Pane into the Search field, the Constraints dialog box automatically updates its Search type as per the item you dropped into Search field. For example, if current Search type is “Constraint Block”, but the item dropped into Search field is a variable, then the Search type automatically changes to “Solved Variable”.

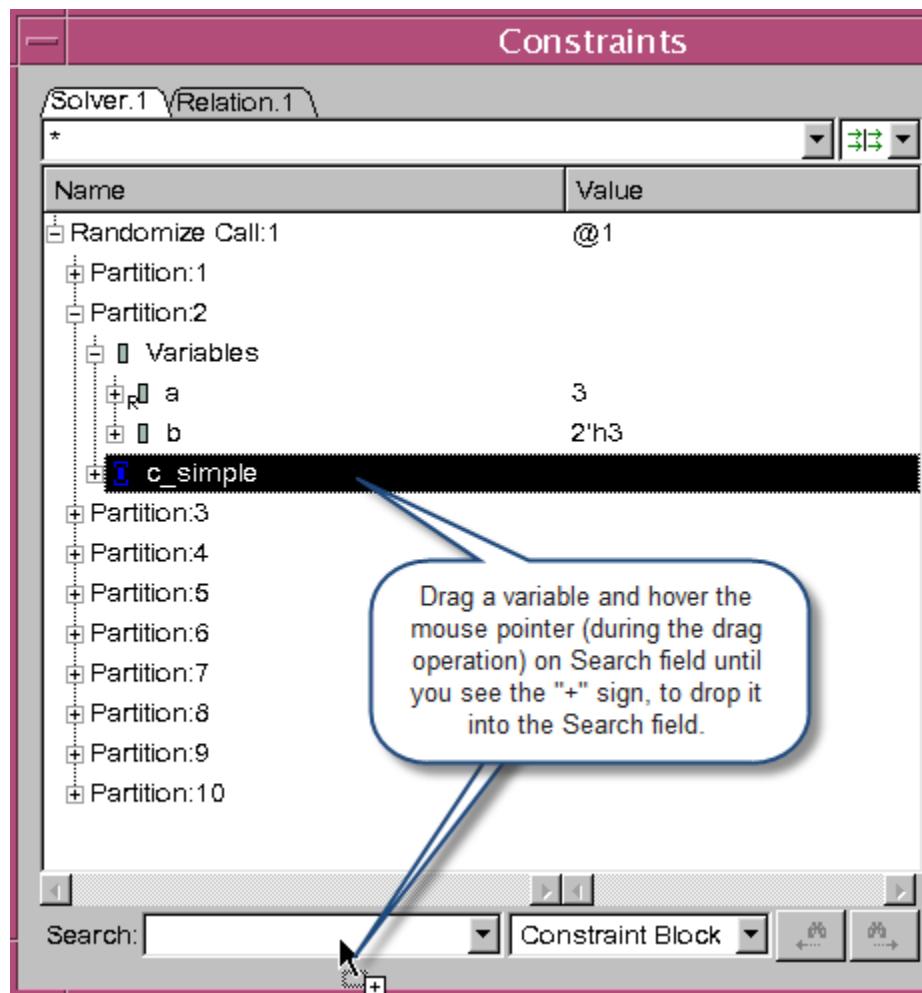
Performing Drag-and-Drop Operation in the Solver Pane

Figure 14-30 Performing Drag-and-Drop Operation in the Solver Pane



Drag an item from the Solver Pane or Relation Pane and hover the mouse pointer over the Search field, as shown in [Figure 14-31](#), to drop it into the Search field.

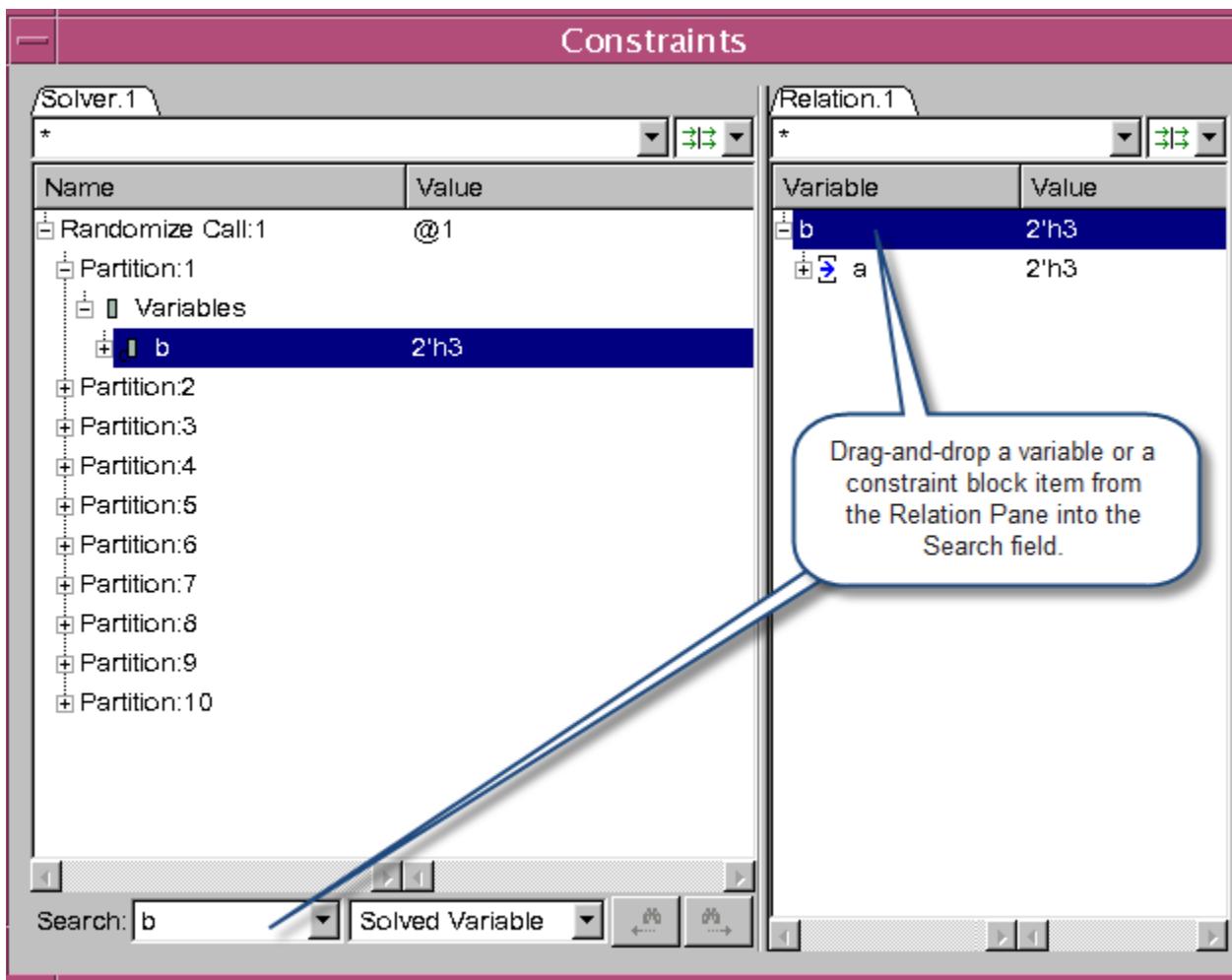
Figure 14-31 Performing Drag-and-Drop Operation



Performing Drag-and-Drop Operation in the Relation Pane

You must click the **Split** button in the Constraints dialog box to perform drag-and-drop operation in Relation Pane. Click the **Split** button near the top right-hand side of the Constraints dialog box to view the Solver Pane and Relation Pane side-by-side, as shown in [Figure 14-32](#).

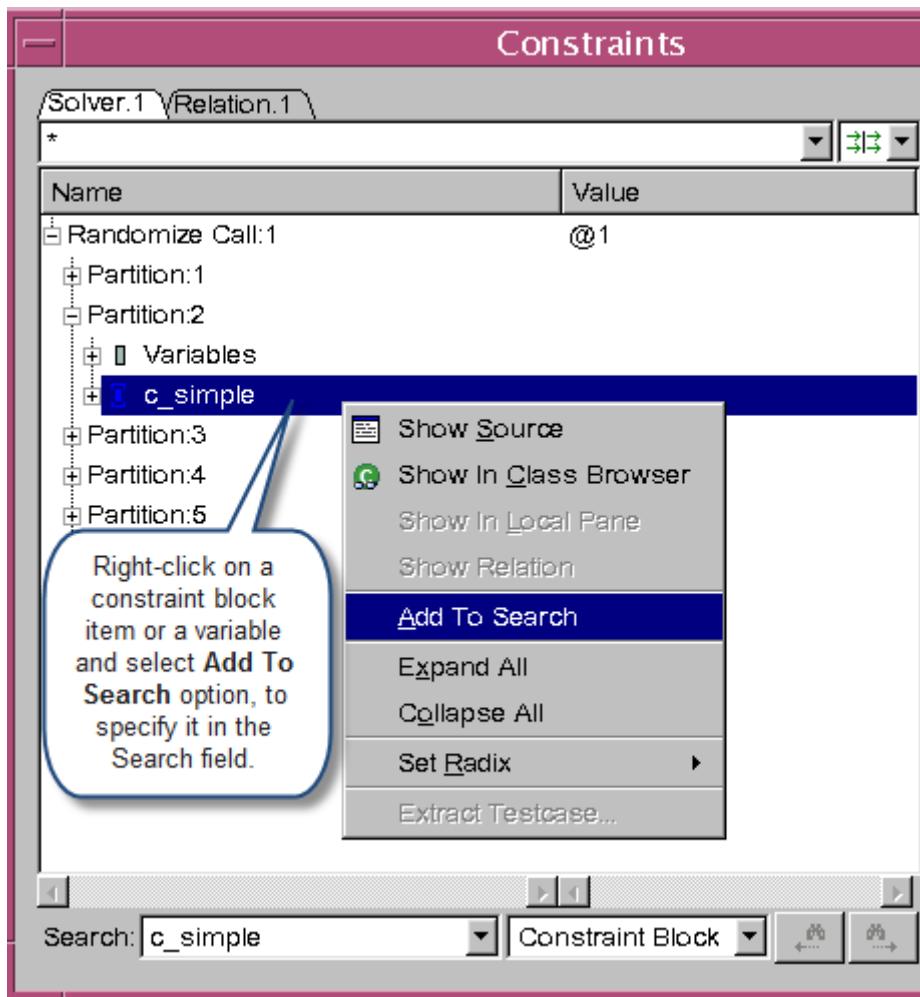
Figure 14-32 Performing Drag-and-Drop Operation in the Relation Pane



Using “Add To Search” Option

You can also use the **Add To Search** right-click option in the Solver Pane and Relation Pane, to specify a variable or constraint block item in the Search field. Right-click on a variable or constraint block item and select **Add To Search** option, as shown in [Figure 14-33](#), to specify it in the Search field.

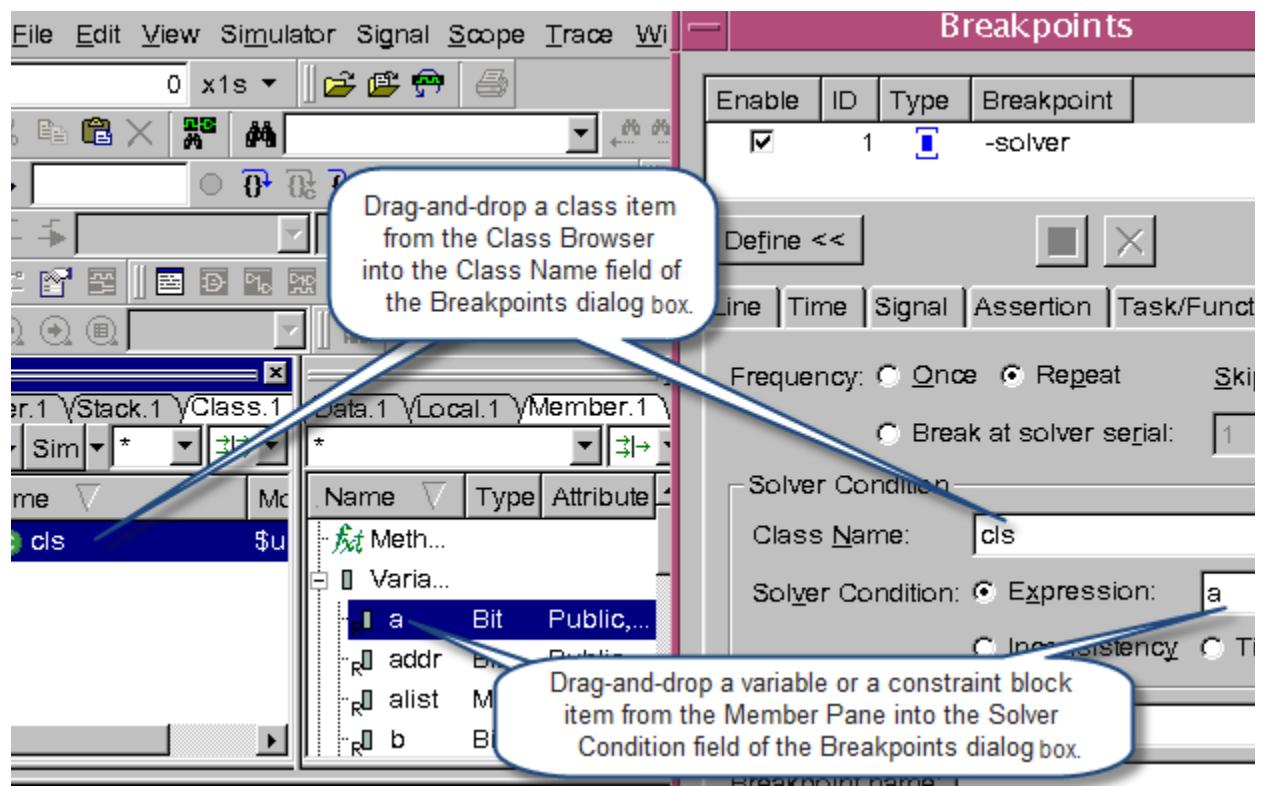
Figure 14-33 Using “Add To Search” Option



Drag-and-Drop Items from Class Browser and Member Pane to Breakpoint Dialog Box

DVE allows you to drag-and-drop Class Browser items (class name) and Member Pane items (variable and constraint block item) into the Breakpoints dialog box, as shown in [Figure 14-34](#).

Figure 14-34 Drag-and-Drop Items to the Breakpoint Dialog Box



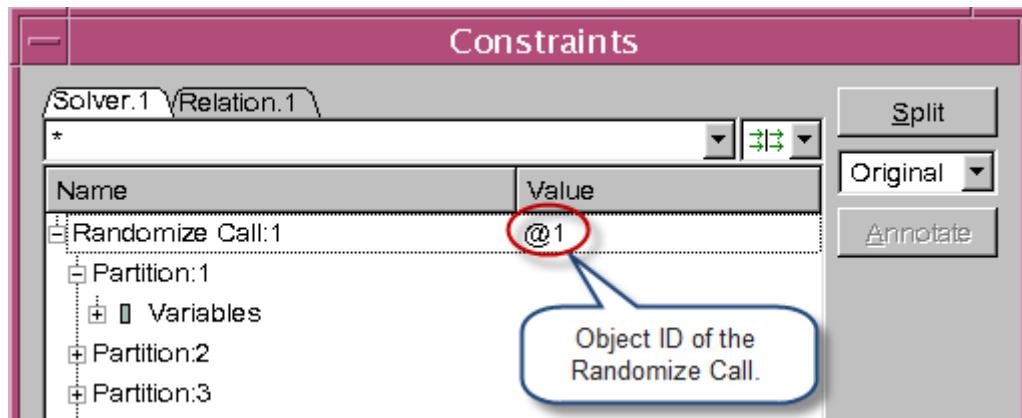
Viewing Object ID Information of a Class in Solver Pane

An object ID uniquely identifies an object within the entire database. Objects are referred to via their unique object IDs. DVE displays object ID of a class object in the Value column of the Solver Pane, as shown in [Figure 14-35](#). You can use this information to find corresponding object-related information in the Local Pane. DVE displays the object ID information in the following format:

@<object index>

Example: @1

Figure 14-35 Viewing Object ID Information of a Class in the Solver Pane



Cross Probing

Cross probing is the ability to select items in the Constraints dialog box and have them mapped to the corresponding items in the Local Pane.

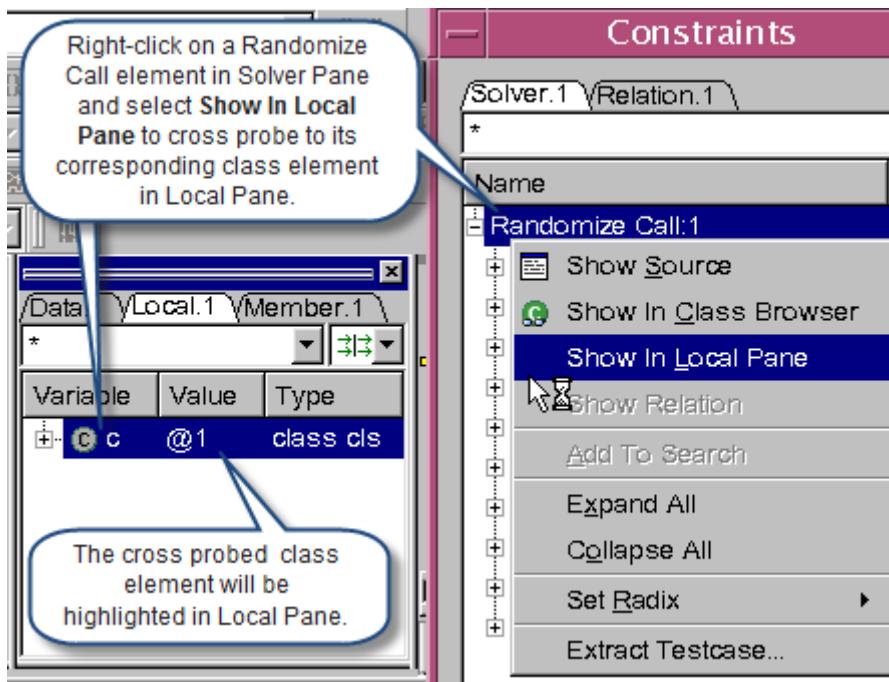
Cross Probing to Local Pane

You can use the **Show In Local Pane** right-click option on a Randomize Call item in the Solver Pane, to cross probe its corresponding class item in Local Pane, as shown in [Figure 14-36](#).

Note:

- You can use this option only on current root class item (Randomize Call item) in Solver Pane.
- This option will be enabled on root class item (Randomize Call), if it contains object ID information.
- This option will be disabled, if current Randomize Call is coming from standard randomize, and does not contain object ID information.

Figure 14-36 Cross Probing to the Local Pane



Cross Probing to Class Browser from Randomize Call

You can use the **Show In Class Browser** right-click option on a Randomize Call item in Solver Pane, to cross probe its corresponding class item in Class Browser.

Extracting Test Case

DVE allows you to extract test case of a Partition and Randomize Call item. This section describes the following topics:

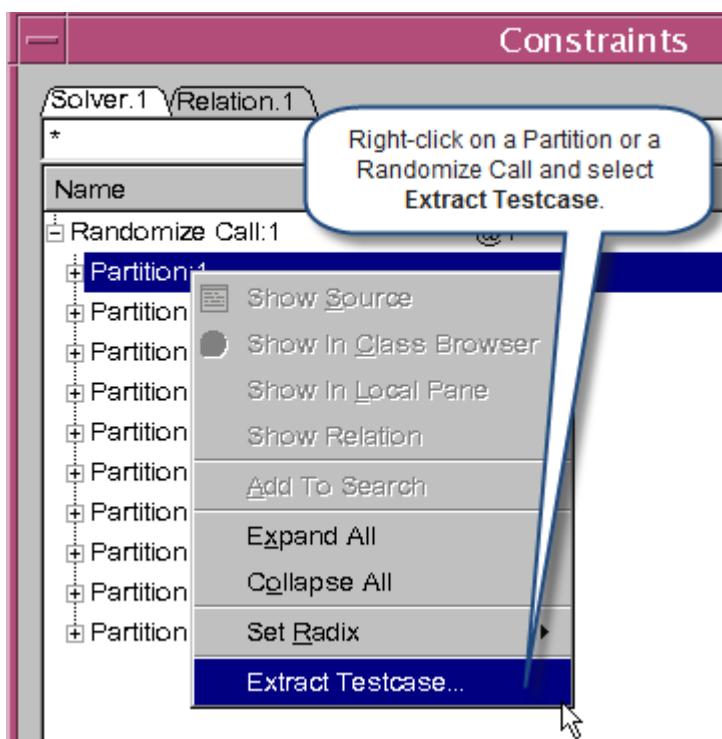
- “[Extracting Test Cases from DVE](#)”
- “[Extracting Test Cases Using UCLI Command](#)”

Extracting Test Cases from DVE

DVE allows you to extract partition-level test cases using the **Extract Testcase** right-click option in the Solver Pane. This allows you to create test case for a desired Partition when needed, thereby saving you debug time.

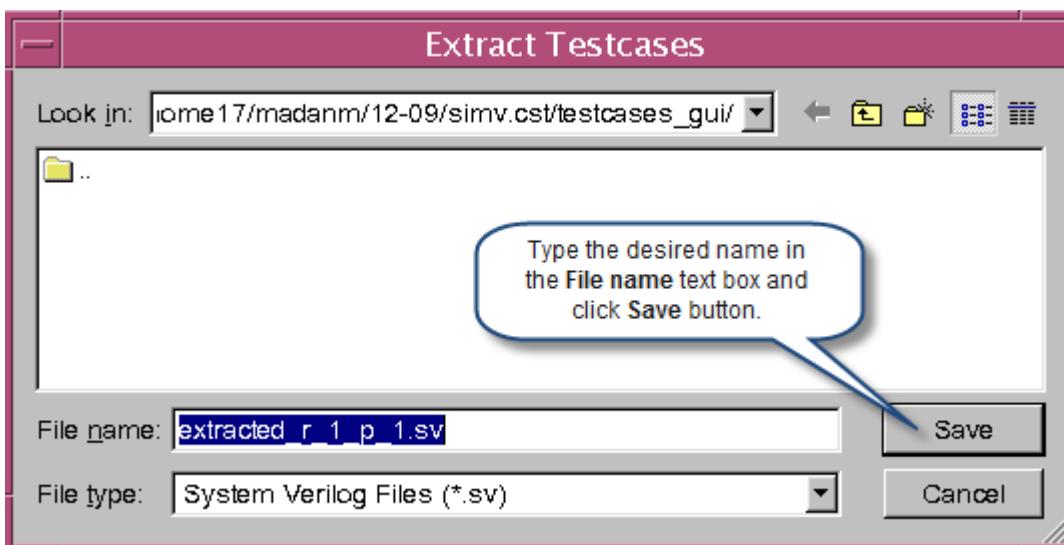
You can use the **Extract Testcase** right-click option on a Randomize Call or a Partition, as shown in [Figure 14-37](#).

Figure 14-37 Extracting Test case from DVE



DVE displays **Extract Testcases** dialog box (directory dialog box for Randomize Call item and filename dialog box for Partition item), as shown in [Figure 14-38](#), when you use the **Extract Testcase** option. Click Save button in the **Extract Testcases** dialog box to extract test case to specified directory or file.

Figure 14-38 Viewing Extracting Testcases Dialog Box



Using “Extract Testcase” Option on a Randomize Call Item

This operation extracts test cases for all Partitions under the Randomize Call, where each Partition generates a test case. DVE displays **Extract Testcases** dialog box (directory dialog box) which allows you to specify directory name. By default, the directory will be <working_dir>/simv.cst/testcases_gui

Using “Extract Testcase” Option on a Partition Item

This operation extracts test case for the selected Partition. DVE displays **Extract Testcases** dialog box (filename dialog box) which allows you to specify directory name and file name. By default, the path of the file will be:

```
<working_dir>/simv.cst/testcases_gui/  
extracted_r_<serial_num>_p_<partition_num>.sv  
<working_dir>/simv.cst/testcases_gui/  
extracted_r_<serial_num>_p_<partition_num>_inconsistent_co  
nstraints.sv
```

Extracting Test Cases Using UCLI Command

You can use the following UCLI command to extract test cases from DVE:

```
constraints extract [-all] [-partition <n>] [-dir <dirname>] [-file <filename>]
```

Where,

<n> — Specify the number of a certain FOG Partition or GP Partition

<dirname> — Specified path that output data will be dumped into

<filename> — Specified file that output data will be dumped into

Controlling rand_mode/constraint_mode and Randomization from UCLI/DVE

DVE and UCLI allows you to control `rand_mode()` or `constraint_mode()` methods and rerun randomization call. This helps you to find the root cause for constraint failure and perform interactive debugging without exiting the DVE.

The `rand_mode()` method can be used to control whether a random variable is active or inactive. Inactive variables are not randomized by the `randomize()` method, and their values are treated as state variables by the solver.

The `constraint_mode()` method can be used to control whether a constraint is active or inactive. When a constraint is inactive, it is not considered by the `randomize()` method.

Controlling rand_mode/constraint_mode from UCLI

You can use the UCLI call command to control `rand_mode()` and `constraint_mode()` methods from UCLI or from DVE console. The following use model describes the commands to control these modes.

Use Model

`rand_mode()`

Following is the syntax for calling the `rand_mode()` method:

```
ucli% call [object.] [rand_variable.]rand_mode(0|1)
```

or

```
ucli% call [object.]rand_variable.rand_mode( )
```

Where,

`object` — Optional. Name of an allocated class object. If this argument is not specified, then the action is applied to the current (`this`) object.

`rand_variable` — Name of a random variable to which the operation is applied. This argument is optional. If you do not specify this argument, then the operation is applied to all random variables within the specified object.

`rand_mode` — Allows you to specify the randomization mode. The mode can be `0` (OFF) or `1` (ON) (see Table 1). If no argument is specified, then this command returns the current active state of the specified random variable. It returns `1` if the variable is active (ON), and returns `0` if the variable is inactive (OFF).

Table 14-7 rand_mode Arguments

value	Meaning	Description
0	OFF	Sets the specified variables to inactive, so that they are not randomized on subsequent calls to the randomize() method.
1	ON	Sets the specified variables to active, so that they are randomized on subsequent calls to the randomize() method.

Note:

VCS generates an error message, if the specified object/variable does not exist or the object is not allocated.

constraint_mode()

Following is the syntax for calling the constraint_mode() method:

```
ucli% call [object.] [constraint_block.] constraint_mode(0|1)
```

or

```
ucli% call [object.] constraint_block.constraint_mode( )
```

Where,

object — Optional. Name of an allocated class object. If this argument is not specified, then the action is applied to the current (*this*) object.

constraint_block — Name of a constraint block. The constraint block name can be the name of any constraint block in the class hierarchy. If you do not specify this argument, then the operation is applied to all constraints within the specified object.

`constraint_mode`—Allows you to specify the constraint mode. The mode can be `0` (OFF) or `1` (ON) (see Table 1). If no argument is specified, then this command returns the current active state of the specified random variable. It returns `1` if the constraint is active (ON), and returns `0` if the constraint is inactive (OFF).

Table 14-8 constraint_mode Arguments

value	Meaning	Description
<code>0</code>	OFF	Sets the specified constraint block to inactive, so that it is not enforced by subsequent calls to the <code>randomize()</code> method.
<code>1</code>	ON	Sets the specified constraint block to active, so that it is considered by subsequent calls to the <code>randomize()</code> method.

Note:

VCS generates an error message, if the specified object/constraint block does not exist or the object is not allocated.

Controlling `rand_mode`/`constraint_mode` from DVE

DVE allows you to enable or disable `rand_mode()` and `constraint_mode()` variables in the Local Pane using the **Set Rand Mode** and **Set Constraint Mode** right-click options.

[Figure 14-39](#) and [Figure 14-40](#) shows how to set `rand_mode` for random variable and `constraint_mode` for a constraint variable. By default, rand mode for all variables is set to **On**. You can select **Off** to disable the selected variable. [Figure 14-41](#) and [Figure 14-42](#) shows the random and constraint variables when the mode is set to OFF.

Figure 14-39 Enabling/Disabling Rand Mode

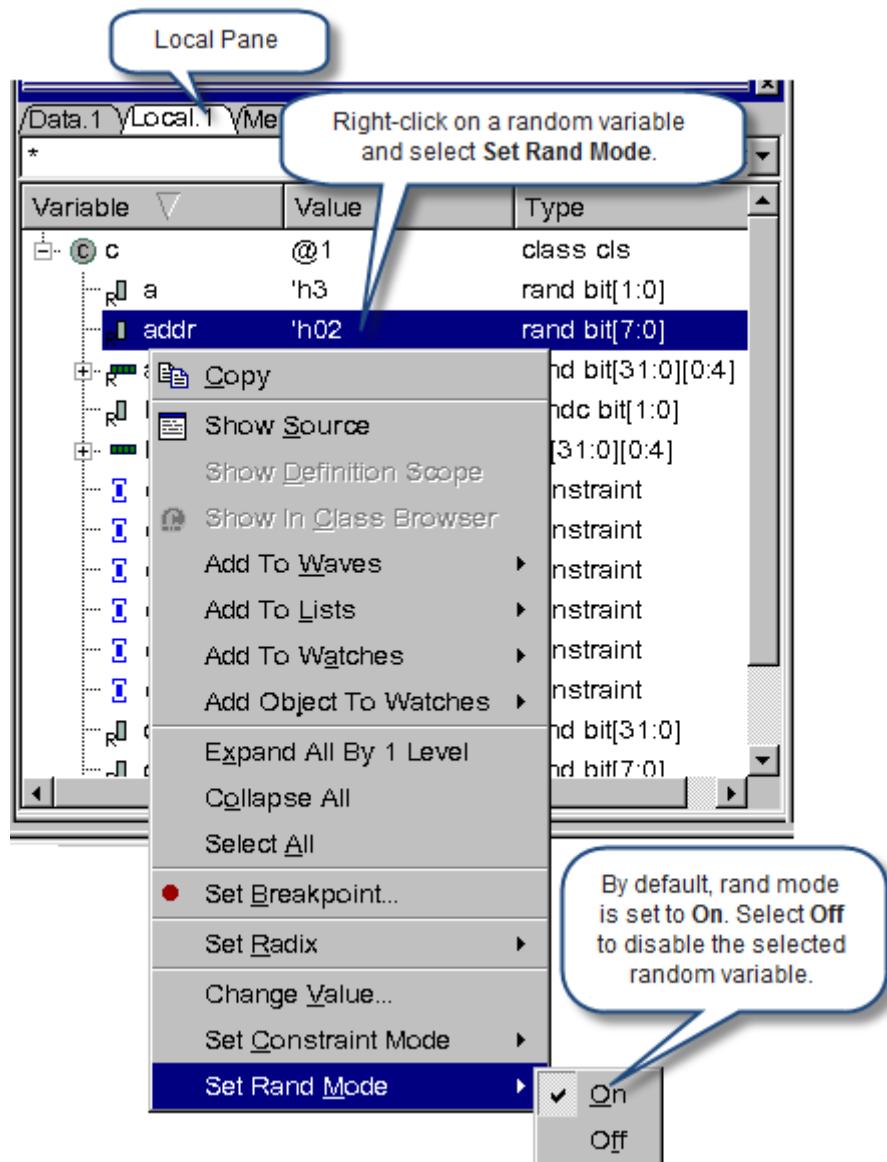


Figure 14-40 Enabling/Disabling Constraint Mode

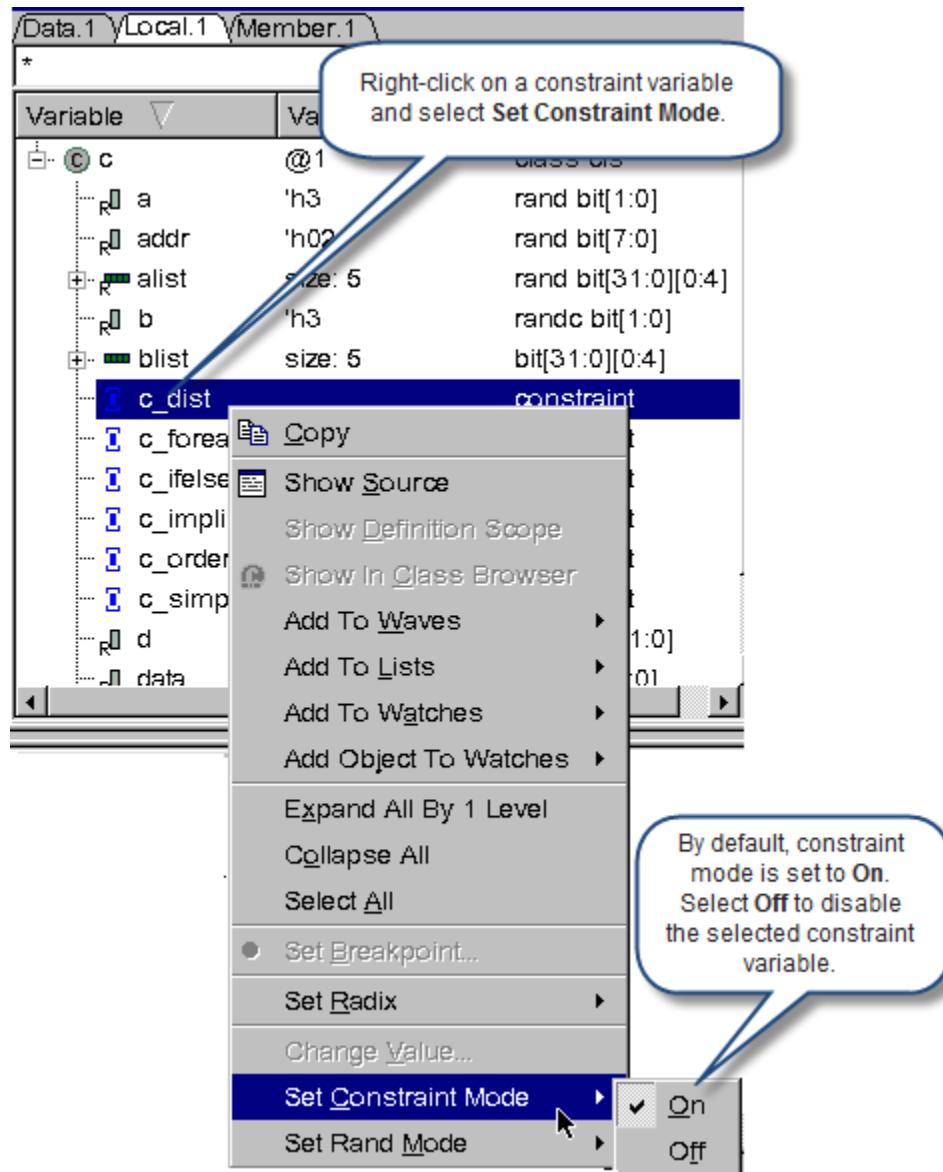


Figure 14-41 Disabled Random Variable

This icon indicates that the random variable is disabled (OFF).

Hover the mouse pointer on a varable to view its mode.

Variable	Value	Type
c c	@1	class cls
- R a	'h3	rand bit[1:0]
- R addr	'h02	rand bit[7:0]
+ R alist	size: 5	rand bit[31:0][0:4]
- R b		randc bit[1:0]
+ R blist	size: 5	bit[31:0][0:4]
- I c_dist		constraint
- I c.foreach		constraint

Figure 14-42 Disabled Constraint Variable

This icon indicates that the constraint variable is disabled (OFF).

Variable	Value	Type
c c		class cls
- R a	'h3	rand bit[1:0]
- R addr	'h02	rand bit[7:0]
+ R alist	size: 5	rand bit[31:0][0:4]
- R b	'h3	randc bit[1:0]
+ R blist	size: 5	bit[31:0][0:4]
- I c_dist		constraint
- I c.foreach		constraint
- I c_if		constraint
- I c_imper...		constraint
- I c_order		constraint
- I c_simple		constraint
- R d	'h4c54f683	rand bit[31:0]
- R data	'h09	rand bit[7:0]

Rerandomization from DVE/UCLI

When there is a solver failure due to inconsistent constraints, you should identify the source of the conflict and modify the inconsistent constraints using the call commands described above or `force` command, and then you can perform rerandomization from the UCLI or DVE console to generate a new constraint database for further debugging.

Rerandomization from UCLI

You can use the following UCLI command for rerandomization from UCLI or DVE console:

```
cstr% step -solver -re_randomize
```

This command works only in constraint debug mode (that is, it can be executed only when the `cstr%` prompt is generated after you execute `step -solver`). VCS generates an error message, if you call this command in non-constraint debug mode.

When you execute the above command, a new constraint debug database will be generated for this rerandomization. A new Solver View will be displayed in DVE. You can compare the data in the two views.

If `simv` goes to the next line or exits from the constraint debug mode, then VCS removes both constraint debug databases.

For example, consider the following testcase(`test.sv`):

Example 14-2 Design File with Constraints (`test.sv`)

```
1: class A;
2:     rand bit[3:0] x,y,z;
3:     constraint C1 {
```

```

4:           x + y == z;
5:       }
6:   constraint C2 {
7:       solve x before y;
8:   }
9: endclass
10:
11: program test;
12:     initial begin
13:         A obj = new;
14:         obj.x = 4'b1111;
15:
16:         $display("before randomize()");
17:         obj.randomize(); //set breakpoint here
18:         $display("x = %d", obj.x);
19:
20:         $display("after randomize()");
21:
22:     end
23: endprogram

```

Set the breakpoint at line 17 and run the simulation.

Execute the following commands:

```

ucli% step -solver // generate debug db for obj.randomize
at Line 17
cstr% call obj.x.rand_mode() // the current active state 1
cstr% call obj.x.rand_mode(0) // set rand_mode as inactive
cstr% call obj.x.rand_mode() // return the current state 0
cstr% call obj.C2.constraint_mode(0) // disable C2
cstr% step -solver -re_randomize // generate a new db for
re-randomizing at Line 17
cstr% next // the both dbs are removed
test.v, 18: $display("x = %d", obj.x);
ucli%

```

Rerandomization from DVE

Use the **Redo Randomize Call and Step in Constraint Solver**

toolbar icon  to perform rerandomization from DVE.

For example, consider [Example 14-2](#).

Compile the test.sv code shown in [Example 14-2](#), as follows:

```
% vcs -sverilog -debug_all test.sv
```

Invoke the DVE GUI, as follows:

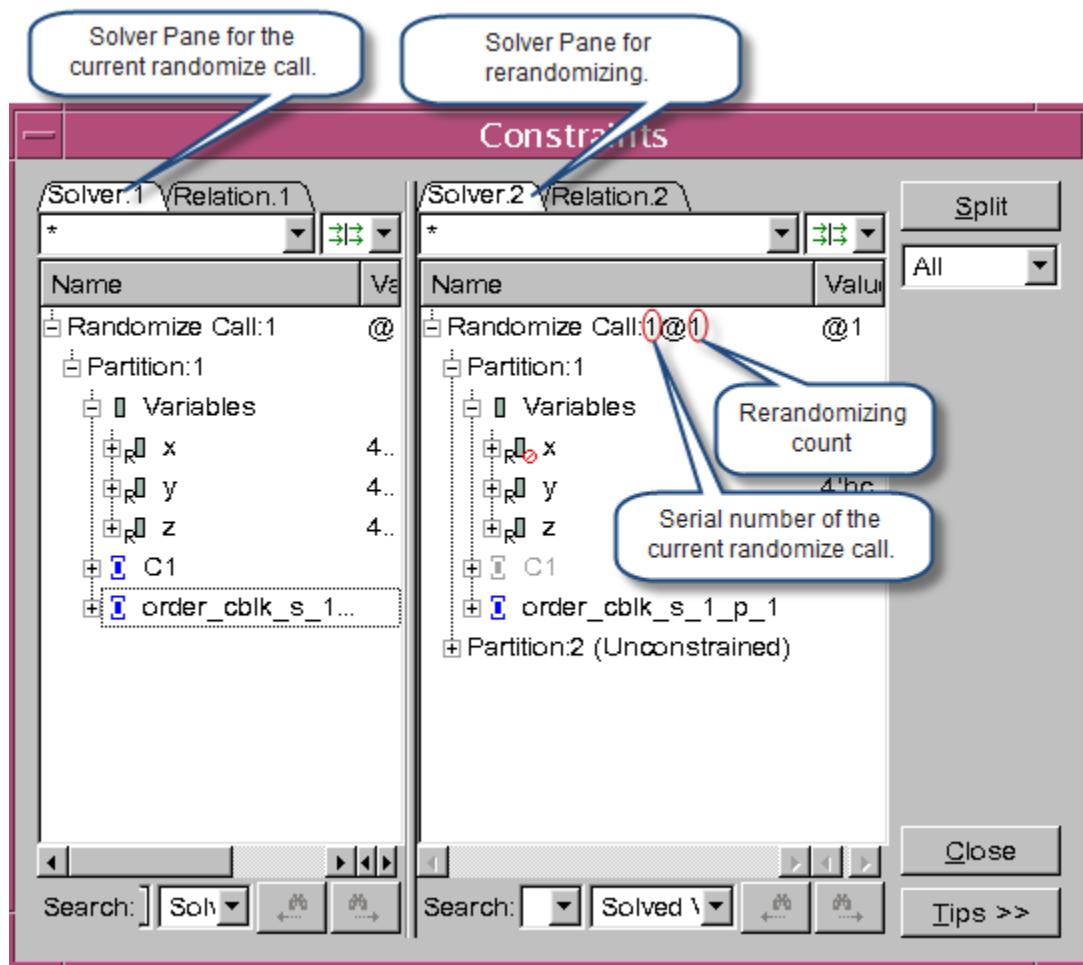
```
% ./simv -gui&
```

Perform the following steps:

1. Set the breakpoint at line 17 and run the simulator.
2. To open the Constraints dialog box, execute the step -solver command in the DVE console, or click the **Step in Constraint Solver** toolbar icon .

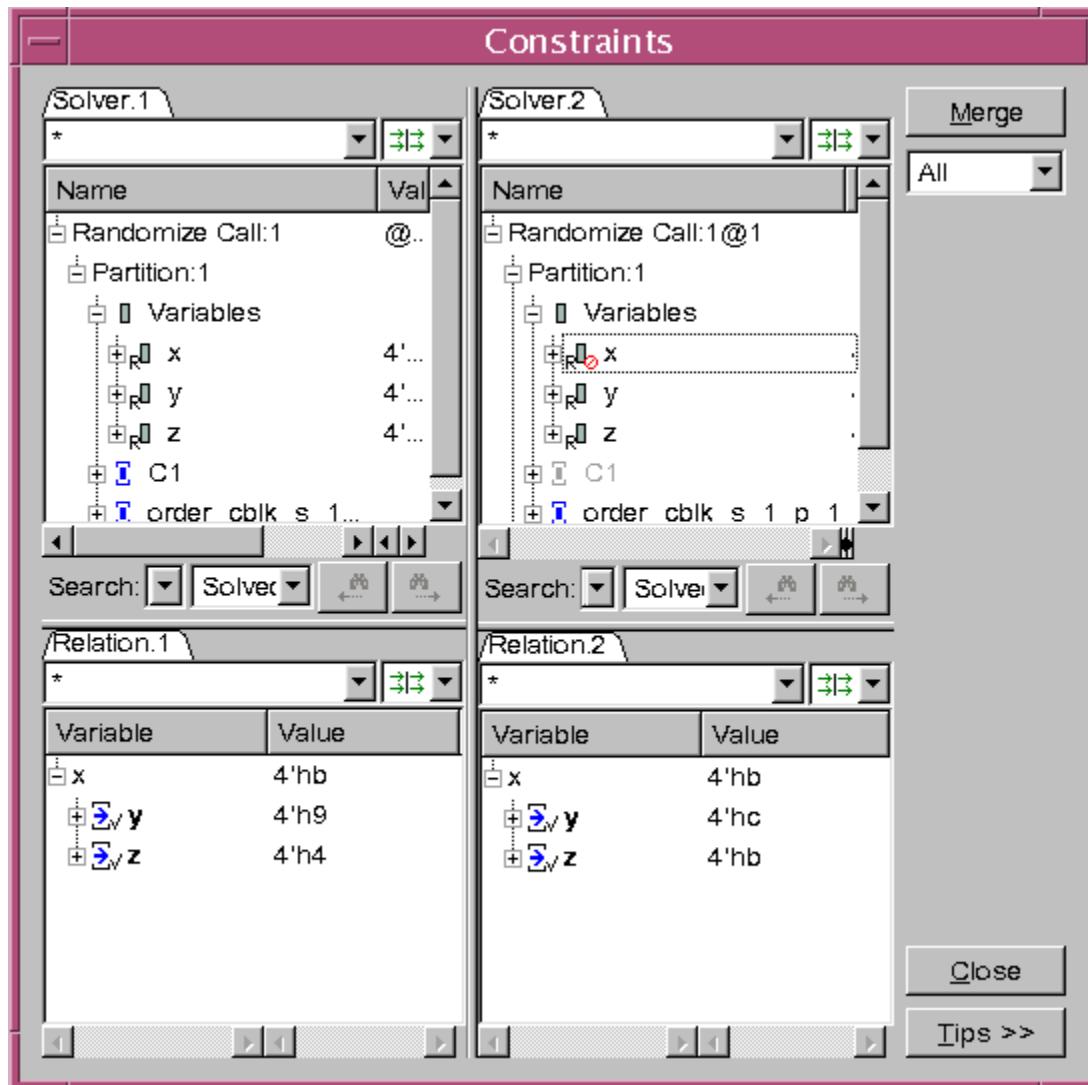
If there is unexpected data or any inconsistencies in the Constraints dialog box Solver Pane, change the mode of desired rand variables or constraint blocks, and click the  toolbar icon to open the new Constraints dialog box which contains both current and new Solver Panes side-by-side (see [Figure 14-43](#)). You can use this side-by-side comparison view to compare both Solver Panes for further debugging:

Figure 14-43 Constraints Dialog Box Side-by-Side Comparison View



Click the **Split** button to view the Relation Panes under corresponding Solver Panes, as shown in [Figure 14-44](#).

Figure 14-44 Relation Panes Under Corresponding Solver Panes



You can use the Switch View Mode drop-down, as shown in [Figure 14-45](#), in the Constraints dialog box, to view either original randomize or rerandomize views.

Figure 14-45 Switch View Mode Drop-Down

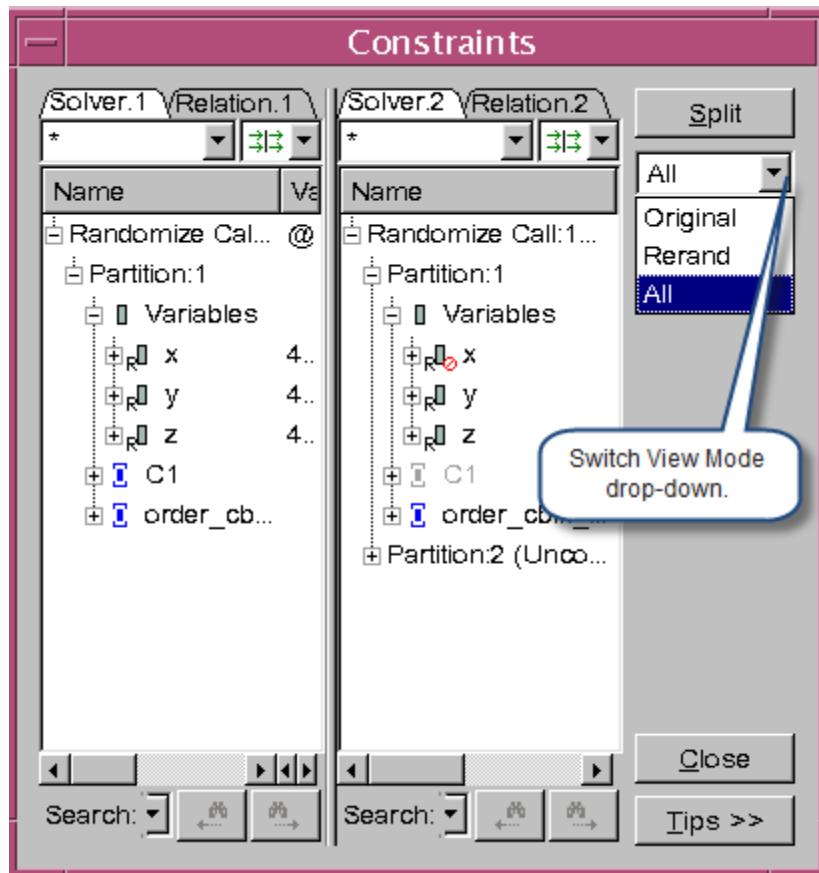


Table 14-9 describes the Switch View Mode drop-down options.

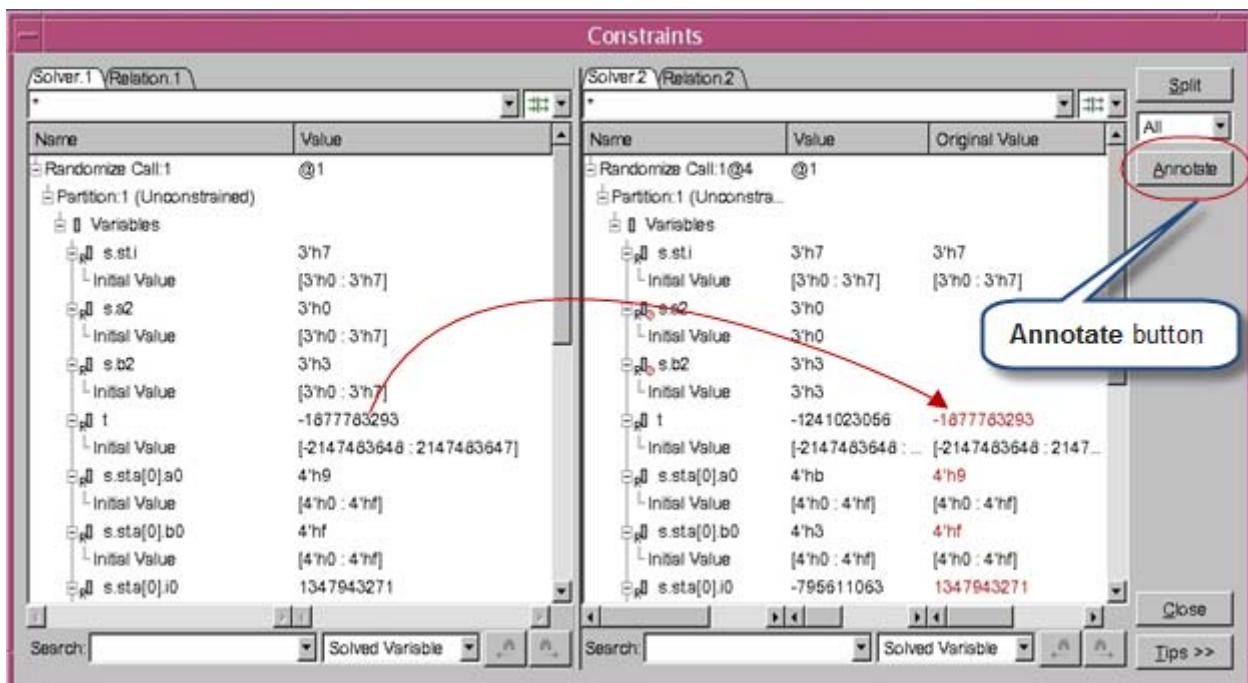
Table 14-9 Switch View Mode Drop-Down Options

Option	Description
Original	Displays Solver Pane for the current randomize call.
Rerand	Displays Solver Pane for rerandomizing.
All	Default option. Displays both original randomize and rerandomize views.

Value Annotation

You can use the **Annotate** button to annotate the different solved value for random variables on rerandomize Solver Pane. This feature helps you to understand the difference between original randomize result and re-randomize result.

Figure 14-46 Different Solved Value Annotation



Click the **Annotate** button, for each solved variable in rerandomize tree(Solver.2), DVE searches the same solved variable in original randomize tree(Solver.1), and annotates the difference of solved value and initial value range in red (see [Figure 14-46](#)), in rerandomize tree.

Constraints Debug Limitations

- Cross-probing to the Member Pane on a hierarchical variable or constraint (`obj.variable`/`obj.constraint`) goes only to the top-level object (`obj`).

15

Debugging Macros in DVE

During debugging, if you encounter a macro, viewing its definition and signal values could be a challenge. Also, it is difficult to understand its cause and effect because you cannot view the value annotated in its content.

DVE now supports macro debugging capability by allowing you to:

- Expand and collapse the macro content
- View signal value annotations in the macro content
- View the macro content in a tooltip
- Jump to the definition of a macro in the source code
- Change the background color of line attribute area for expanded macros
- View text indentation in expanded macro and tooltip

- Set a breakpoint inside the macro content
- Step in and out of the macro content
- Trace drivers and loads of a signal inside the macro content
- View the macro definition for a nested macro

Enabling Macro Debug

The syntax to enable macro debug features is:

VCS

```
% vcs -debug_all <filename> <other_options>
```

Expanding and Collapsing the Macro Content

You can expand or collapse the macro in the Source View by clicking on the +/- icon that appears to the left side of the macro, as shown in [Figure 15-2](#). However, if there is a nested macro, it will be expanded and parameters will be resolved automatically.

Viewing Signal Value Annotations in the Macro Content

DVE annotates the values dumped during simulation for signals inside the macro content, as shown in [Figure 15-2](#). Follow the below steps to use this capability.

1. Select **Edit > Preferences**.

The Applications Preferences dialog box appears.

2. In the Source View category, select **Show value annotation** and click **OK**.
-

Viewing the Macro Content in a Tooltip

Hovering the mouse cursor on the macro displays its content in a tooltip, as shown in [Figure 15-2](#). If there is a nested macro, it will be expanded and parameters will be resolved automatically. Hovering the mouse cursor on a signal displays its type and value annotation information in a tooltip.

Note:

Tooltip displays value annotations only if the macro content contains a single signal. It does not display value annotations if the macro content contains multiple signals.

Viewing the Definition of a Macro in the Source Code

DVE now displays the macro as a hyperlink. You can click the hyperlink to go to the macro definition in the source code.

Note:

You cannot view hyperlink for macros used inside the macro definition.

You can use the toolbar icon shown below to go back to the location where the macro is used.



Viewing Text Indentation in Expanded Macro and Tooltip

The expanded macros and tooltip displays the macro content with same indentation as defined in the macro definition.

For example, if you define a macro in source code with indentation, as shown below:

```
'define MULTI2 \
if (a) \
    b = 0; \
else \
    b = 1;
```

Then this macro is displayed in expanded macro and tooltip, as shown in [Figure 15-1](#):

Figure 15-1 Viewing Text Indentation

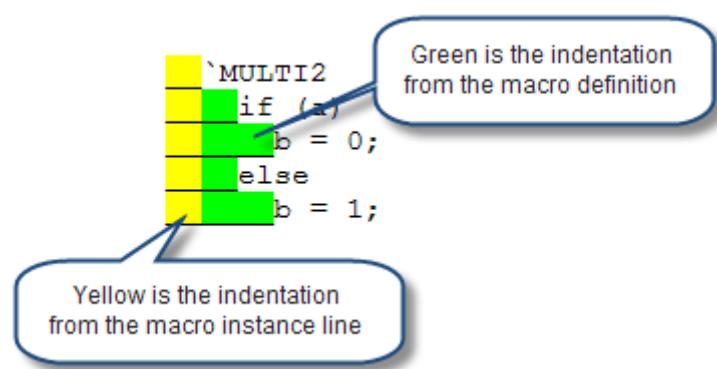
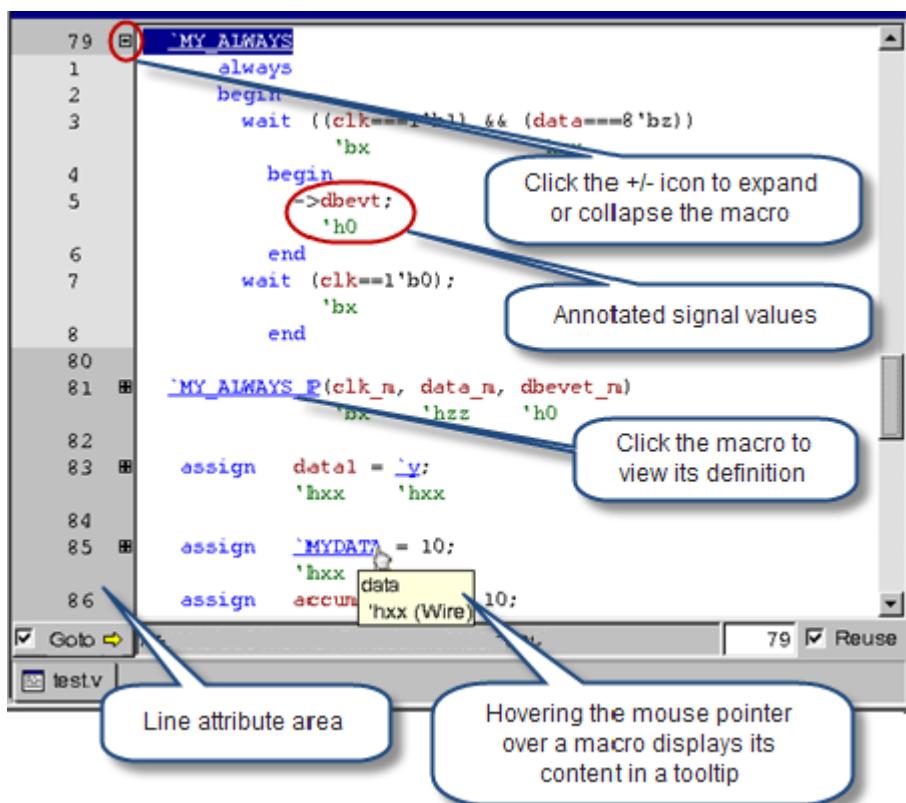


Figure 15-2 Macro Debug in DVE

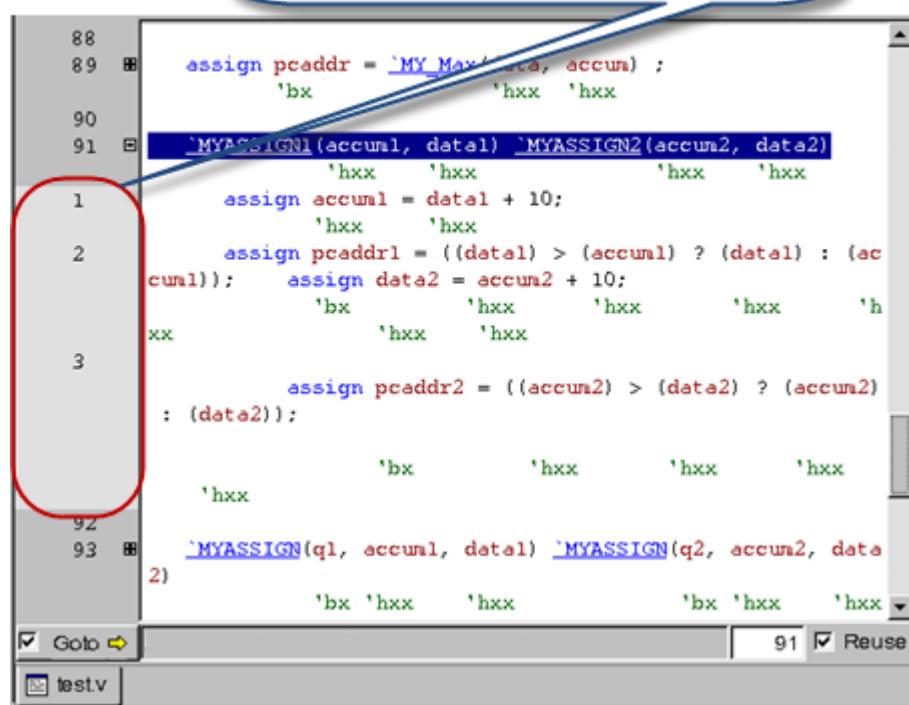


Changing Background Color of Line Attribute Area for Expanded Macros

You can change the background color of line attribute area for expanded macros. The default background color is gray.

Figure 15-3 Changing Background Color

You can set different background color in the line attribute area for expanded macros.



```
88 assign pcaddr = `MY_ASSIGN1(accum, data1) `MY_ASSIGN2(accum2, data2)
89           'bx      'hxx  'hxx
90
91           'hxx  'hxx      'hxx  'hxx
1 assign accum1 = data1 + 10;
2   'hxx  'hxx
3 assign pcaddr1 = ((data1) > (accum1) ? (data1) : (ac
cum1)); assign data2 = accum2 + 10;
4   'bx      'hxx  'hxx      'hxx  'h
xx      'hxx  'hxx
5
6 assign pcaddr2 = ((accum2) > (data2) ? (accum2)
: (data2));
7   'bx      'hxx  'hxx      'hxx
8   'hxx
9 MY_ASSIGN(q1, accum1, data1) `MY_ASSIGN(q2, accum2, data
2)
10  'bx 'hxx  'hxx      'bx 'hxx  'hxx
```

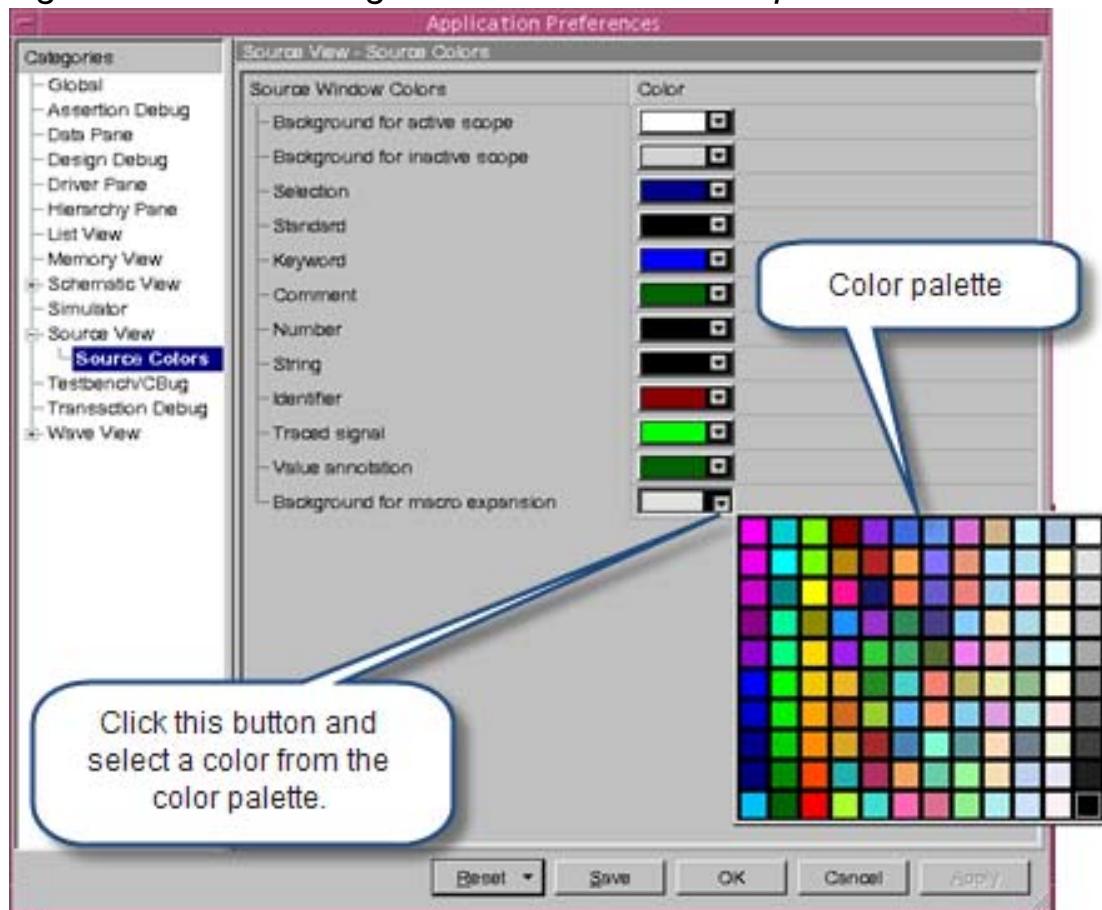
To change background color of line attribute area for expanded macros:

1. Select **Edit > Preferences**.

The Applications Preferences dialog box appears.

2. In the **Source View - Source Colors** category, click the **Background for macro expansion** drop-down, as shown in [Figure 15-4](#).
3. Select a color from the color palette and click **Apply**.
4. Click **OK**.

Figure 15-4 Selecting a Color from the Color palette



For example, if you select green color from the palette, then the color of line attribute area changes to green, as shown in [Figure 15-5](#).

Figure 15-5 Changing the Color of Line Attribute Area

```

88
89  assign pcaddr = `MY_Max(data, accum) ;
      'bx          'hxx  'hxx
90
91  `MYASSIGN(accum1, data1) `MYASSIGN2(accum2, data2)
      'hxx  'hxx          'hxx  'hxx
1    assign accum1 = data1 + 10;
      'hxx  'hxx
2    assign pcaddr1 = ((data1) > (accum1) ? (data1) : (ac
cum1));  assign data2 = accum2 + 10;
      'bx          'hxx  'hxx          'hxx  'h
xx          'hxx  'hxx
3
      assign pcaddr2 = ((accum2) > (data2) ? (accum2)
: (data2));
      'bx          'hxx  'hxx          'hxx
      'hxx
92
93  `MYASSIGN(q1, accum1, data1) `MYASSIGN(q2, accum2, data
2)
      'bx 'hxx  'hxx          'bx 'hxx  'hxx

```

Goto /remote/macro/test.v 91 Reuse
test.v

Examples

Consider the following example testcase test.v:

```

`define MYDATA data

`define MAX 32
`define NMAX 37
`define NASSIGN nsignal = `NMAX

`define MY_ALWAYS    always \
begin \
    wait ((clk==1'b1) && (^MYDATA==8'bz)) \
        begin \
            ->dbevt; \
        end \
    wait (clk==1'b0); \

```

```

    end

`define CHECKCLK(theclk) (theclk==1'b1)
`define CHECKDATA(thedata) (thedata==8'bz)

`define MY_ALWAYS_P(pclk,pdata, pevent)    always \
begin \
    wait (^CHECKCLK(pclk) && ^CHECKDATA(pdata)) \
begin \
    ->pevent; \
end \
    wait (clk==1'b0); \
end

`define MYASSIGN1(x, y) \
assign x = y + 10; \
assign pcaddr1 = `MY_Max(y, x);

`define MYASSIGN2(x, y) \
assign y = x + 10; \
assign pcaddr2 = `MY_Max(x, y);

`define MYASSIGN(q, x, y) \
assign x = y + 10; \
assign q = `MY_Max(y, x);

`define y data2

`define RHS rhs

`define LHS lhs

`define MY_Max(a,b) ((a) > (b) ? (a) : (b))

module cpu;
    wire [7:0] data;
    wire [7:0] accum;
    reg        pcaddr;
    reg        clk;
    reg        myArr[`MAX:0];

    wire [7:0] data_m;

```

```

reg      clk_m;
event   dbevet_m;

wire [7:0] data1;
wire [7:0] accum1;
reg      pcaddr1;

wire [7:0] data2;
wire [7:0] accum2;
reg      pcaddr2;

reg      q1;
reg      q2;

wire [7:0] data1;

reg      lhs, rhs;
reg      lhs1, rhs1;

event dbevt;

`MY_ALWAYS

`MY_ALWAYS_P(clk_m, data_m, dbevet_m)

assign   data1 = `y;

assign   `MYDATA = 10;
assign   accum = data + 10;
assign   clk = ~clk;

assign pcaddr = `MY_Max(data, accum) ;

`MYASSIGN1(accum1, data1) `MYASSIGN2(accum2, data2)

`MYASSIGN(q1, accum1, data1) `MYASSIGN(q2, accum2, data2)

assign `LHS = `RHS;

assign lhs1 = rhs1;

```

```
endmodule
```

Steps to compile the example:

```
% vcs -sverilog -debug_all test.v
```

```
% simv -gui&
```

The following examples demonstrate how to debug a macro in DVE.

Example-1: Simple Macro Defined as Signal Name

Example 15-1 Simple Macro Defined as Signal Name

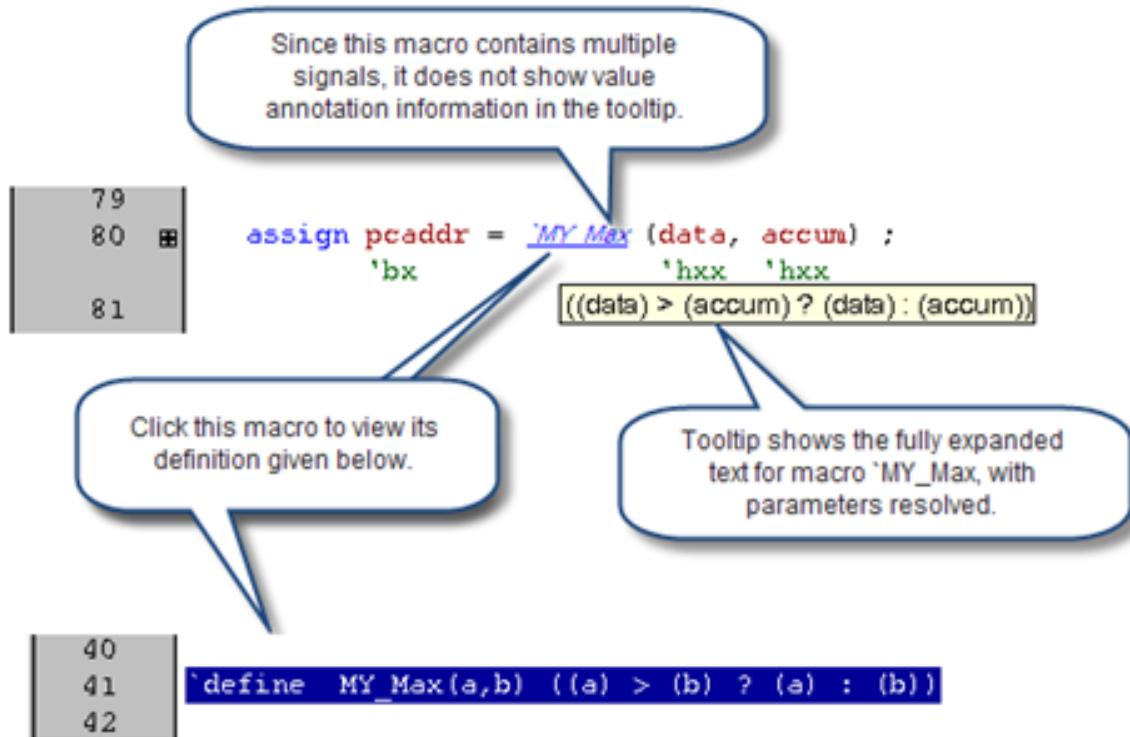
Signal value annotations
are displayed under the
signal.

Since this macro content contains a
single signal, it displays tooltip with its
value annotation information when you
hover the mouse pointer on it.

A screenshot of a simulation tool interface. On the left, there is a code editor window showing lines 84, 85, and 86 of a Verilog script. Line 84 contains a 'bx' macro definition. Line 85 shows an assignment statement. Line 86 is blank. To the right of the code editor, a tooltip is displayed over the assignment statement. The tooltip contains the text 'assign LHS = RHS;' with the 'LHS' and 'RHS' parts underlined. Below this, the word 'bx' is highlighted with a red oval. To the right of the 'bx' label, another 'bx' label is shown in green. A yellow box labeled 'rhs' with the sub-label 'bx (Reg)' is also part of the tooltip.

Example-2: Single Line Macro with Parameters

Example 15-2 Single Line Macro with Parameters



Usage Example

Consider the test case shown in [Example 15-3](#).

Example 15-3 Macro Debugging Test File (test.v)

```
`define MYDATA data

`define MAX 32
`define NMAX 37
`define NASSIGN nsignal = `NMAX

`define MY_ALWAYS    always \
begin \
    wait ((clk==1'b1) && (^MYDATA==8'bz)) \
    begin \

```

```

        ->dbevt; \
    end \
    wait (clk==1'b0); \
    end

`define CHECKCLK(theclk) (theclk==1'b1)
`define CHECKDATA(thedata) (thedata==8'bz)

`define MY_ALWAYS_P(pclk,pdata, pevent)    always \
begin \
    wait (^CHECKCLK(pclk) && ^CHECKDATA(pdata)) \
begin \
    ->pevent; \
end \
    wait (clk==1'b0); \
end

`define MYASSIGN1(x, y) \
assign x = y + 10; \
assign pcaddr1 = `MY_Max(y, x);

`define MYASSIGN2(x, y) \
assign y = x + 10; \
assign pcaddr2 = `MY_Max(x, y);

`define MYASSIGN(q, x, y) \
assign x = y + 10; \
assign q = `MY_Max(y, x);

`define y data2

`define RHS rhs

`define LHS lhs

`define MY_Max(a,b) ((a) > (b) ? (a) : (b))

module cpu;
    wire [7:0] data;
    wire [7:0] accum;
    reg         pcaddr;
    reg         clk;

```

```

reg           myArr [`MAX:0] ;

wire [7:0]  data_m;
reg        clk_m;
event     dbevet_m;

wire [7:0]  data1;
wire [7:0]  accum1;
reg        pcaddr1;

wire [7:0]  data2;
wire [7:0]  accum2;
reg        pcaddr2;

reg        q1;
reg        q2;

wire [7:0]  data1;

reg        lhs, rhs;
reg        lhs1, rhs1;

event dbevt;

`MY_ALWAYS

`MY_ALWAYS_P(clk_m, data_m, dbevet_m)

assign   data1 = `y;

assign   `MYDATA = 10;
assign   accum = data + 10;
assign   clk = ~clk;

assign pcaddr = `MY_Max(data, accum) ;

`MYASSIGN1(accum1, data1) `MYASSIGN2(accum2, data2)

`MYASSIGN(q1, accum1, data1) `MYASSIGN(q2, accum2, data2)

assign `LHS = `RHS;
assign lhs1 = rhs1;

```

```
endmodule
```

Compile the test.v design file shown in [Example 15-3](#):

```
% vcs -sverilog -debug_all test.v
```

Invoke the DVE GUI:

```
% simv -gui&
```

Setting Breakpoints in the Macro Content

A breakpoint is a setting on a line of code that tells DVE to stop the simulation immediately before the line of code on which it is set, so that you can examine it before continuing. DVE allows you to debug macros by setting breakpoints inside your macro content.

To set a breakpoint inside your macro content, you can:

- Click the line attributes area of the Source view, next to an executable line, or
- Right-click the line attributes area of the Source view, and select **Set Breakpoint**.

A solid red circle indicates that a line breakpoint is set (see [Figure 15-6](#)).

Figure 15-6 Setting Breakpoint Inside the Macro Content

The screenshot shows a Verilog source code editor with the following code:

```
47 reg pcaddr1;
48 wire [7:0] data2;
49 wire [7:0] accum2;
50 reg pc_addr2;
51 reg q1;           Line attribute area
52 reg q2;
53 wire [7:0] data1;
54 reg lhs, rhs;
55 reg lhs1, rhs1;
56 eventdbevt;
57 `MY_ALWAYS
 1 always
 2 begin
 3   wait ((clk==1'b1) && (data==8'bz)) Breakpoint inside
 4   begin the macro content
 5     ->dbevt;
 6   end
 7   wait (clk==1'b0);
 8 end
58 `MY_ALWAYS_P(clk_m, data_m, dbevet_m)
59 assign data1 = 'y;
60 assign `MYDATA = 10;
61 assign accum = data + 10;
62 assign clk = ~clk;
63 assign pcaddr = `MY_Max(data, accum) ;
64 `MYASSIGN1(accum1, data1) `MYASSIGN2(accum2, data2)
65 `MYASSIGN(q1, accum1, data1) `MYASSIGN(q2, accum2, data2)
```

A red dot marks the line number 3, which is part of the macro content. A callout bubble points to this line with the text "Breakpoint inside the macro content". Another callout bubble points to the line attribute area (lines 51-52) with the text "Line attribute area".

Note:

You can only set a line breakpoint on an executable line. If a line is not executable, then no breakpoint is set when you click next to it.

When you run the simulator, the line where the simulation stopped is marked by a yellow arrow (current active line) in the Source View, as shown in [Figure 15-7](#).

Figure 15-7 Yellow Arrow Indicating the Line where Simulation Stopped

```
54 reg lhs, rhs;
55 reg lhs1, rhs1;
56 eventdbevt;
57 #MY_ALWAYS
1 always
2 begin
3 wait ((clk==1'b1) && (data==8'bz))
4 begin
5 ->dbevt;
6 end
7 wait (clk==1'b0);
8 end
58 #MY_ALWAYS_P(clk_m, data_m, dbevet_m)
59 assign data1 = `y;
```

Yellow arrow (current active line) indicates the point at which simulation was suspended

In Figure 15-7, you can see that the simulation stopped at the breakpoint. When the simulator stops inside macros, you can use the Next icon (shown in Table 15-1), to run the simulator step-by-step: inspecting functions, stepping line-by-line, setting new breakpoints, and so on.

To disable a breakpoint inside the macro content

Click the solid red breakpoint circle to disable it.

To delete a breakpoint inside the macro content

Double-click on a solid red circle. The red circle disappears, indicating that the breakpoint is deleted.

To delete all breakpoints inside the macro content

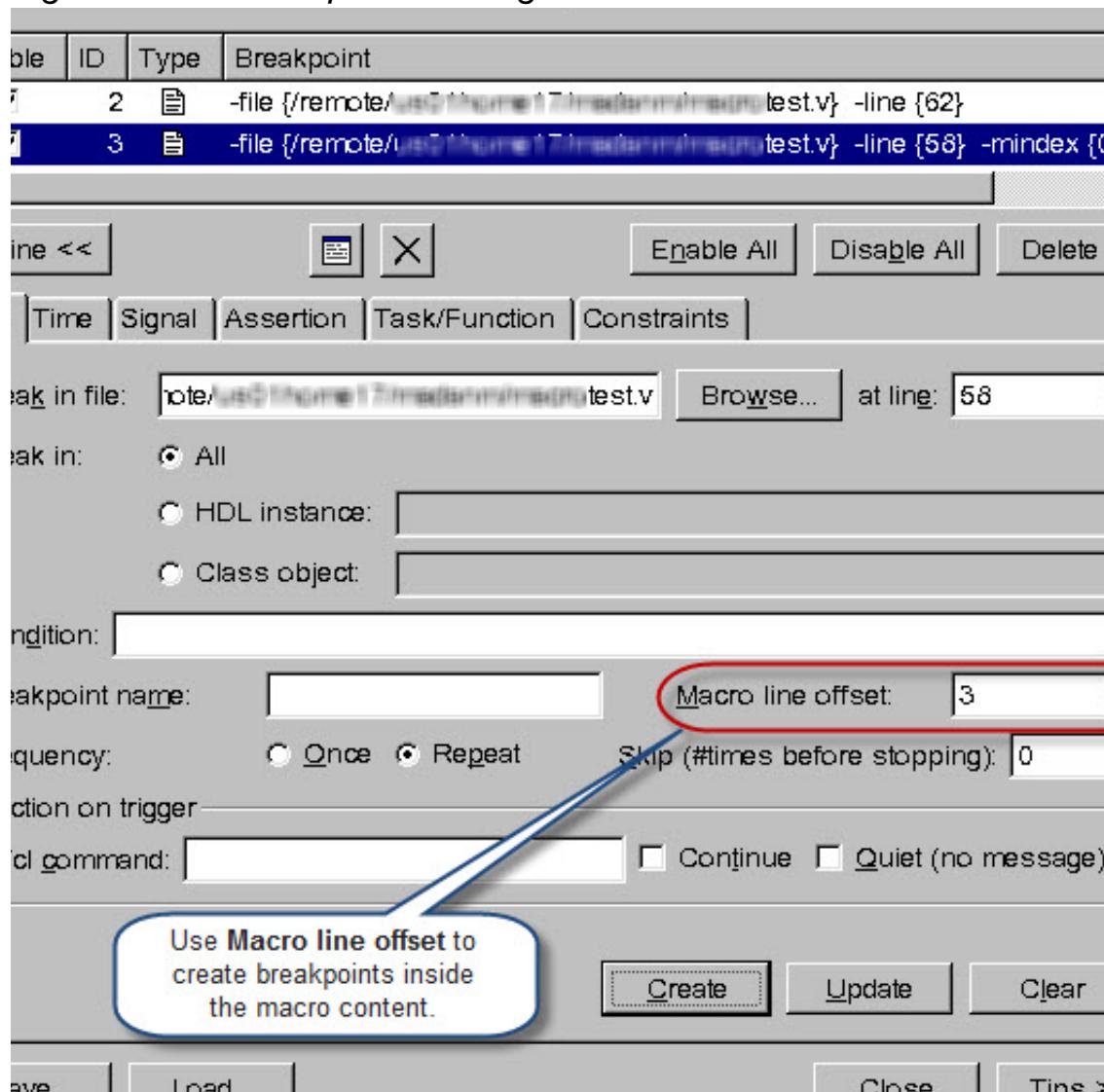
Right-click on a breakpoint, then select **Delete Breakpoint** or **Delete All Breakpoints**.

Creating Breakpoints in the Macro Content Using Breakpoints Dialog

You can use the Breakpoints dialog box to create and show breakpoints inside the macro content of the specified file.

1. Select a line in the line attribute area, right-click, and select **Properties**. The Breakpoints dialog box appears (see [Figure 15-8 on page 19](#)).

Figure 15-8 Breakpoints Dialog Box



2. Select the **Line** tab and fill out the following fields:

- **Break in file** — Enter the file name or browse to the file where you want to create the breakpoint.
- **at line** — Enter the line number for the breakpoint.

- **Macro line offset** — Choose a line offset inside the macro to add a macro breakpoint. If you choose 0 as the macro offset, the breakpoint is set on the line where the macro is used, but not inside the macro.

Note:

The Macro line offset field is enabled only if the specified file is open in a source window and there is macro usage at the specified line.

Setting Breakpoint in the Macro Content Using DVE Tcl Command

You can use the following DVE Tcl command to set a breakpoint inside your macro content:

```
gui_sim_watch -file <file> -line <line> -macroline <offset>
```

where,

- <file> — File in which the macro is used.
- <line> — Line number in which the macro is used.
- <offset> — Line number in the macro content.

Stepping In and Out of Macros

DVE allows you to step through the macro content line-by-line.

[Table 15-1](#) explains the three buttons that help you step through the macro content.

Table 15-1 Buttons or Icons in DVE for Stepping Into and Out of Macros

Icon	Icon Name	Description
OR  	Step Next	<p>When the simulation stops at a line that includes macro usage, you can use this icon to step the current active line into the macro content. The current active line steps into the macro content, only if:</p> <ul style="list-style-type: none"> •The current execution file is opened and there is a macro instance in the current execution line •The macro content is expanded <p>For example, when the simulation stops at line 57, as shown in Figure 15-9, and the macro is expanded, then click the “step” or “next” icon to move the current active line into the macro content, as shown in Figure 15-10.</p>
	Step out	<p>When the current active line is in the macro content, you can use this icon to step the current active line out of the current expanded macro content. This causes the simulator to execute until the end of the macro (an end statement) is reached, and then stop at the next executable line of code outside the macro.</p> <p>For example, when the current active line is inside the macro content, as shown in Figure 15-10, then click this icon to step the current active line out of the current expanded macro content, as shown in Figure 15-11.</p> <p>Note: This icon is enabled only when the simulator steps inside the macro content.</p>

Figure 15-9 Current Active Line at a Line which Includes Macro Usage

A screenshot of a debugger interface showing a code editor with the following content:

```

57  `MY_ALWAYS
1    always
2      begin
3        wait ((clk==1'b1) && (data==8'bz))
4          begin
5            ->dbevt;
6          end
7        wait (clk==1'b0);
8      end
58  `MY_ALWAYS_P(clk_m, data_m, dbevet_m)

```

The line 57, which contains the macro call ``MY_ALWAYS`, is highlighted with a red dot indicating it is the current active line. A callout bubble points to the line 58, labeled "Expanded macro".

Figure 15-10 Stepping into the Macro Content

A screenshot of a debugger interface showing a code editor with the following content:

```

56  event dbevt;
● 57  `MY_ALWAYS
1    always
2      begin
3        wait ((clk==1'b1) && (data==8'bz))
4          begin
5            ->dbevt;
6          end
7        wait (clk==1'b0);
8      end
58  `MY_ALWAYS_P(clk_m, data_m, dbevet_m)
59  assign data1 = `y;

```

The line 57, which contains the macro call ``MY_ALWAYS`, is highlighted with a red dot indicating it is the current active line. A callout bubble points to the line 58, labeled "Current active line inside the macro content".

Figure 15-11 Stepping out of the Macro Content

A screenshot of a debugger interface showing a code editor with the following content:

```

56  event dbevt;
● 57  `MY_ALWAYS
1    always
2      begin
3        wait ((clk==1'b1) && (data==8'bz))
4          begin
5            ->dbevt;
6          end
7        wait (clk==1'b0);
8      end
58  `MY_ALWAYS_P(clk_m, data_m, dbevet_m)
59  assign data1 = `y;
60  assign `MYDATA = 10;
61  assign accum = data + 10;

```

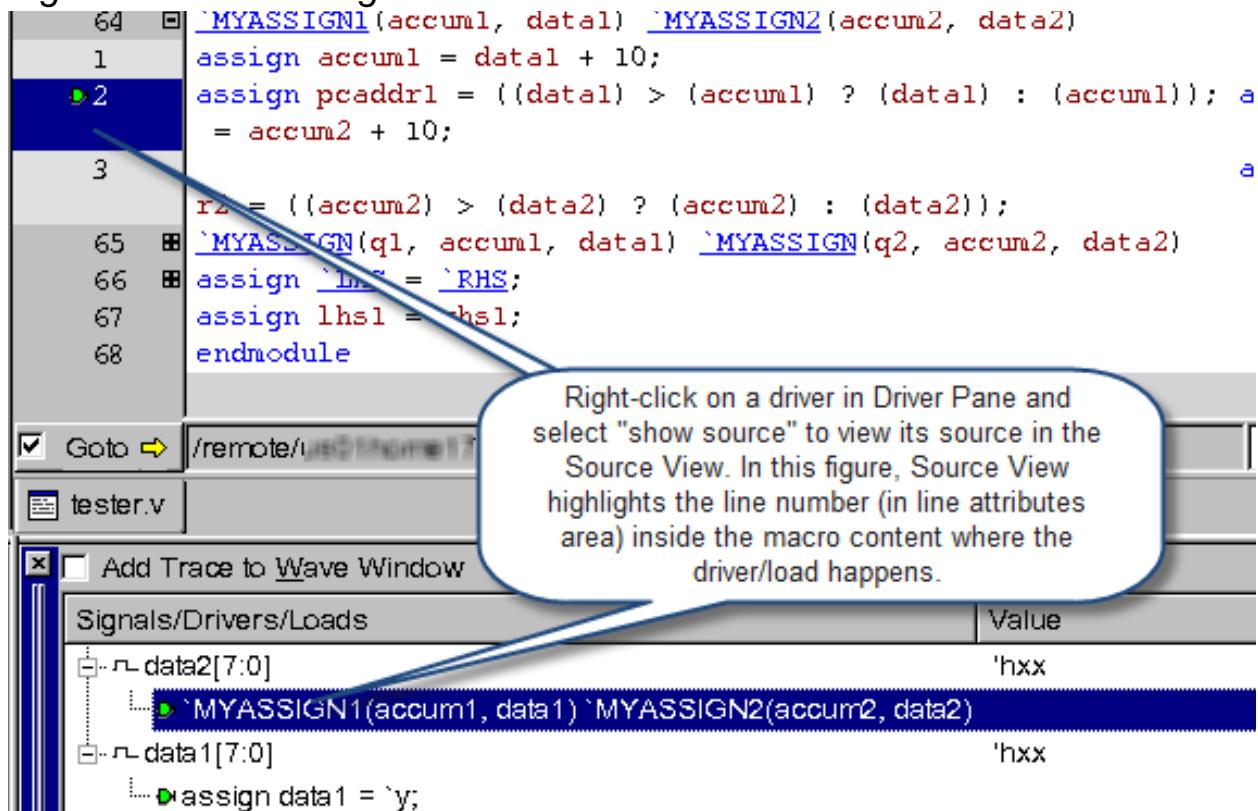
The line 58, which contains the macro call ``MY_ALWAYS_P`, is highlighted with a red dot indicating it is the current active line. A callout bubble points to the line 59, labeled "Current active line outside the macro content".

Tracing Drivers and Loads Inside Macro Content

DVE allows you to trace drivers and loads of signals inside the macro content. Do the following to trace drivers and loads inside the macro content:

- Right-click on the desired driver or load in the Driver Pane and select **show source**. The driver is highlighted in the Source view, as shown in [Figure 15-12](#).

Figure 15-12 Tracing Drivers and Loads Inside the Macro Content



Macro Expansion Location

DVE expands the macro from the end line of the macro usage, as shown in [Figure 15-13](#).

Figure 15-13 Macro Expansion Location

```
80
81     `MY_ALWAYS_P(clk_m,
82                     data_m,
83                    dbevet_m)
84
1     always
2     begin
3         wait (`CHECKCLK(clk_m) && `CHECKDATA(data_m))
4             begin
5                 ->dbevet_m;
6             end
7             wait (clk==1'b0);
8         end
84
```

DVE now expands the macro from the end line of macro usage.

Nested Macro Support

DVE supports viewing macro definitions for nested macros, as shown in [Figure 15-14](#).

Figure 15-14 Viewing Macro Definition for Nested Macro

```
90
91  #MYASSIGN1(accum1, data1) #MYASSIGN2(accum2, data2)
    'hxx      'hxx          'hxx      'hxx
28
29 `define MYASSIGN1(x, y) \
30     assign x = y + 10; \
31     assign pcaddr1 = `MY_Max(y, x);
44
45 `define LHS lhs
46
47 `define MY_Max(a,b) ((a) > (b) ? (a) : (b))
48
49 module cpu;
```

For example, consider macro "MYASSIGN1" defined at line 91. Click this macro to view its definition at line 29.

You can now view the definition for macro defined in the macro definition. For example, you can see that macro "MY_Max" is used in the definition of "MYASSIGN1". Click this macro to view its definition at line 47.

Macro Debugging Limitations

- Macro debugging is not supported in UCLI.
- If a macro (used in the same place) has multiple definitions, only the last definition parsed by the compiler is supported. [Figure 15-15](#) illustrates this limitation.

Figure 15-15 Macro Usage Multiple Definitions

The figure consists of three vertically stacked windows from a logic simulation tool (DVE). Each window shows a portion of a Verilog module and its source file path.

- Top Window:** Shows the code for module `chl()`. It includes a macro definition `'define m1 10`, a call to `'include "incl.svh"`, and an `endmodule` statement. A callout bubble points to the `'include` line with the text: "For example, in this figure, macro "m1" is defined in both "t1.v" and "t2.v", and it is used in include file "inc1.svh". The status bar shows the file path: /remote/.../t1.v.
- Middle Window:** Shows the code for module `ch2()`. It includes a macro definition `'define m1 20`, a call to `'include "incl.svh"`, and an `endmodule` statement. A callout bubble points to the `'include` line with the text: "If you view the source for the include file "incl.svh", then the macro "m1" will always be expanded to "20" (as shown in below figure), the one defined in "t2.v" (which is the last definition of "m1" parsed by compiler)". The status bar shows the file path: /remote/.../t2.v.
- Bottom Window:** Shows the code for an `initial` block. It contains two `$display` statements: `$display ("%m m1", `m1);` and `$display ("%m m1", 20);`. A callout bubble points to the first `$display` statement with the text: "If you view the source for the include file "incl.svh", then the macro "m1" will always be expanded to "20" (as shown in below figure), the one defined in "t2.v" (which is the last definition of "m1" parsed by compiler)". The status bar shows the file path: /remote/.../incl.svh.

16

DVE Interactive Rewind

You can create multiple simulation snapshots using the DVE "Checkpoint" feature during an interactive debug session. In the same debug session, you can go back to any of those previous snapshots, by using the DVE "Rewind" feature and do 'What if' analysis.

When you create multiple checkpoints, say at times "t1, t2, t3, ...tn", and you want to rewind from your current simulation time to a previous simulation time say t2, then all the checkpoints that follows t2 (t3, t4 etc.) gets deleted. This is intentional, because when you go back to history using the rewind operation, you are given an option to force/release the signal values and continue with a different simulation path until you get the desired results. This is called as "What if" analysis. This way, you need not restart your simulation from time zero and you save time.

Following are the advantages of the Checkpoint and Rewind feature:

- Checkpoint directly saves multiple simulation states and you can rewind to any of those saved states using "Rewind".
- Checkpoint and Rewind are done by the tool.
- More user friendly, and very quick in performance.
- Lists all the checkpoints, within a session, with respective simulation time.

Interactive Rewind Vs Save and Restore

Interactive rewind seems similar to Save and Restore operation. Even though there are similarities, there are also differences.

Similarities between Save/Restore and Checkpoint/Rewind

- You can save a snapshot at a particular simulation time, when the simulator is in a "Stop" State.
- You can go back to the previously saved state.
- You can remove the intermediate saved data. In Save-Restore, you delete the saved data. In Checkpoint/Rewind, you need to issue the `checkpoint -kill` or `-join` commands.

Differences between Save/Restore and Checkpoint/Rewind:

Save/Restore	Checkpoint/Rewind
Persistent across different simv runs.	Not persistent across simv runs. As soon as simv quits, all the checkpoint data is lost.
Doesn't describe saved state.	Describes various checkpoint state using the checkpoint -list command. You can also see the list of checkpoints in the tooltip.
Save/Restore operation is slow.	Faster than Save/Restore for the same simulation run.
Not supported in SystemC	Supported for SystemC designs.

Usage Model

In this section, you will see how interactive rewind works in DVE.

Example

top.v

```
module top;
    reg clock,count;
    leaf_entity m(.x(clock));
    initial
        begin
            clock <=1'b0;
            count <= count+1;
            #1 clock <=1'b1;
            count <= count+1;
            #2;
        end
endmodule
```

leaf.v

```
module leaf_entity(input x);
    reg clock;
    int counter = 0;
```

```
reg dummy; // leave it at 'X'

initial begin
    clock = 0;
    forever begin
        #1 clock = ~clock;
        counter++;
        if (counter==100)
            $finish();
    end
end

endmodule
```

Compile the `top.v` and `leaf.v` examples:

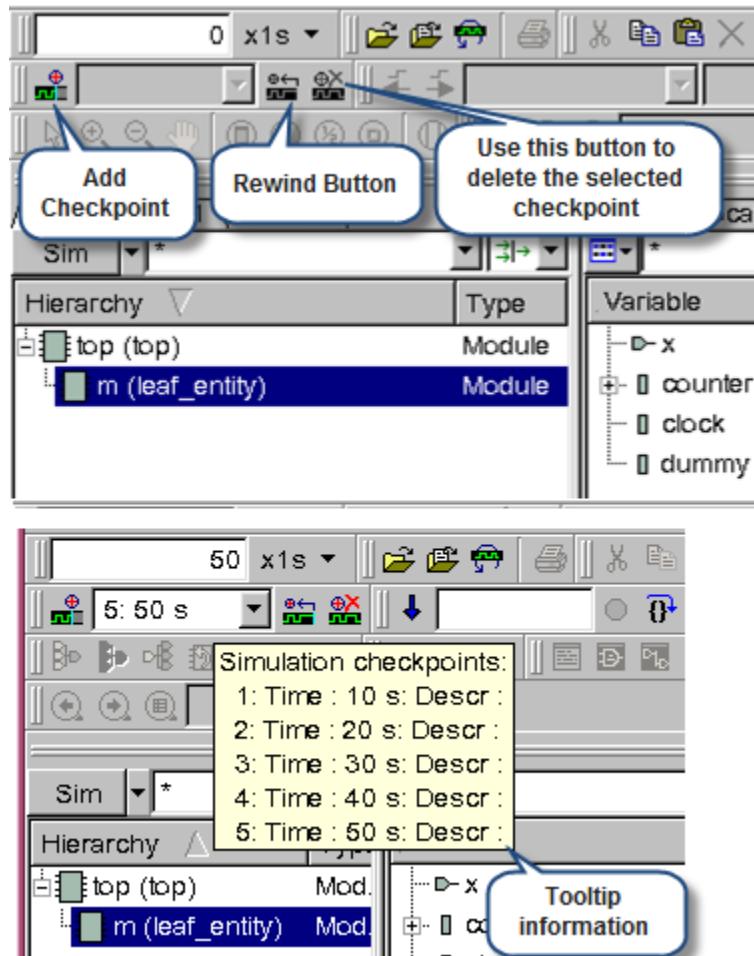
```
% vcs -nc -debug_all -sverilog leaf.v top.v
```

Invoke the DVE GUI:

```
% simv -gui&
```

To create simulation checkpoint in DVE, click the **Add Checkpoint**

icon.  on the toolbar.

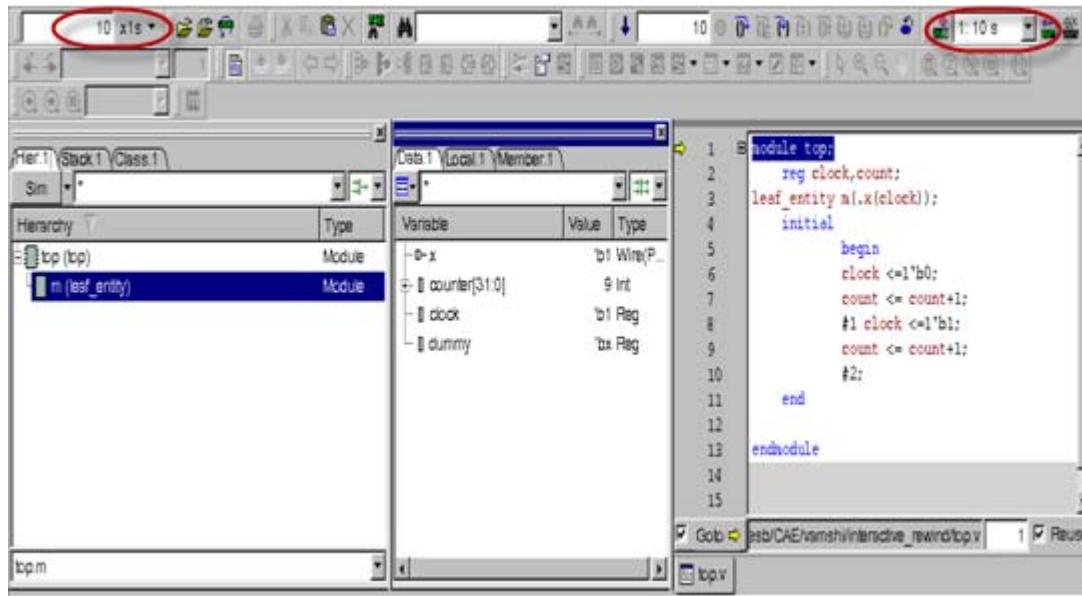


To rewind or go back to the checkpoint, select a simulation time available in the **Simulation Checkpoint** box and click the **Rewind to the selected checkpoint** icon  on the toolbar.

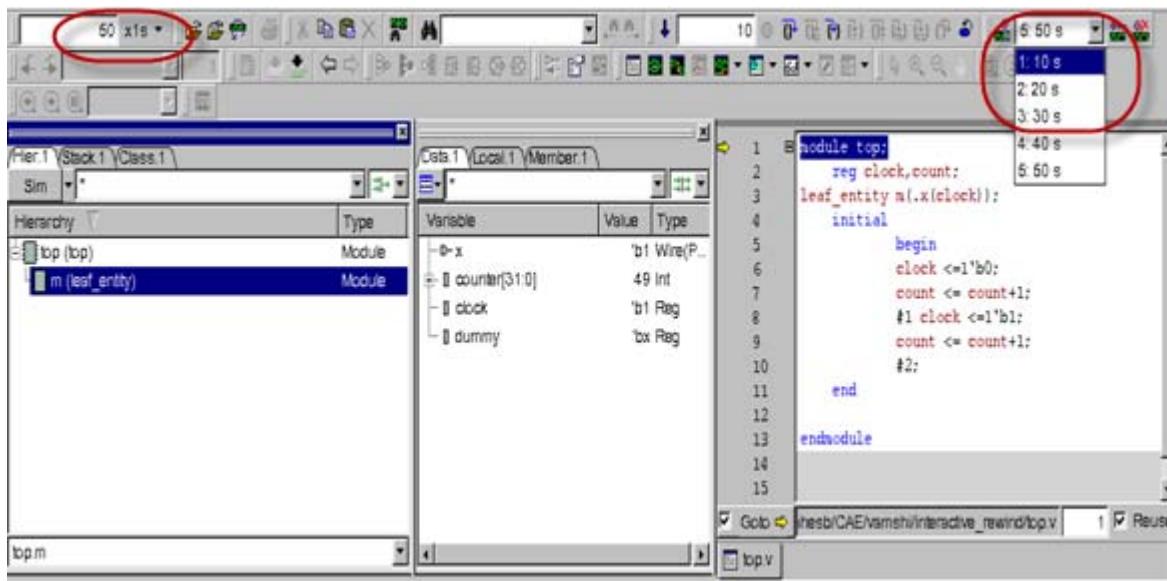
To delete a checkpoint, select a simulation time available in the **Simulation Checkpoint** box and click the **Checkpoint kill** icon  on the toolbar. You cannot delete the first checkpoint.

The following illustrations show how to create checkpoints and rewind to a previous checkpoint in DVE:

1. Create a checkpoint at 10 ns.



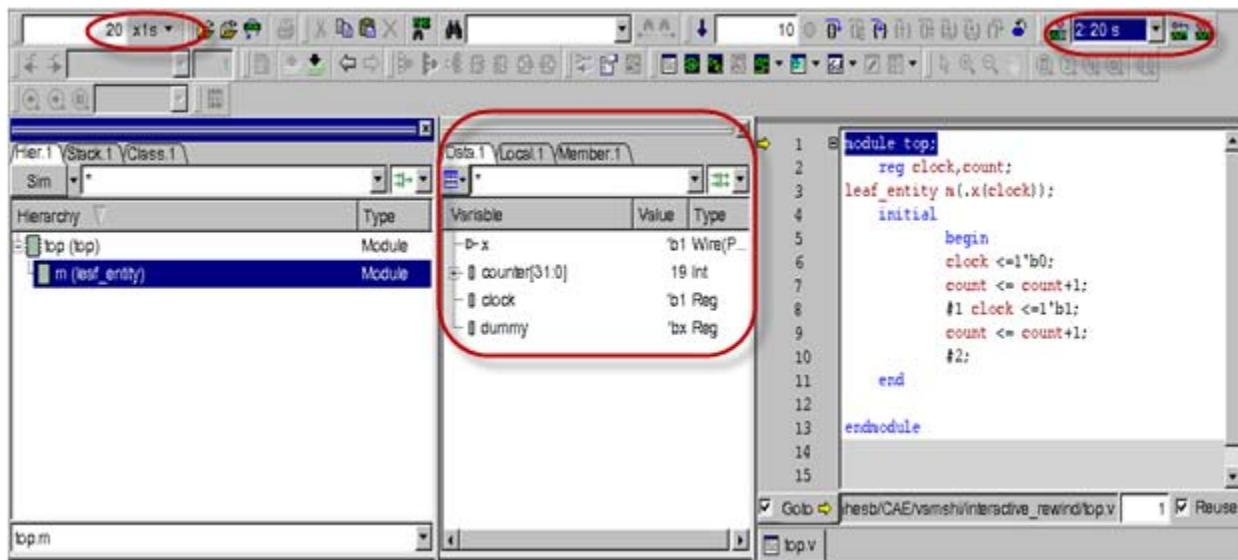
2. Create several checkpoints as shown in the drop-down menu.
The following illustration show the current simulation time is 50 ns.



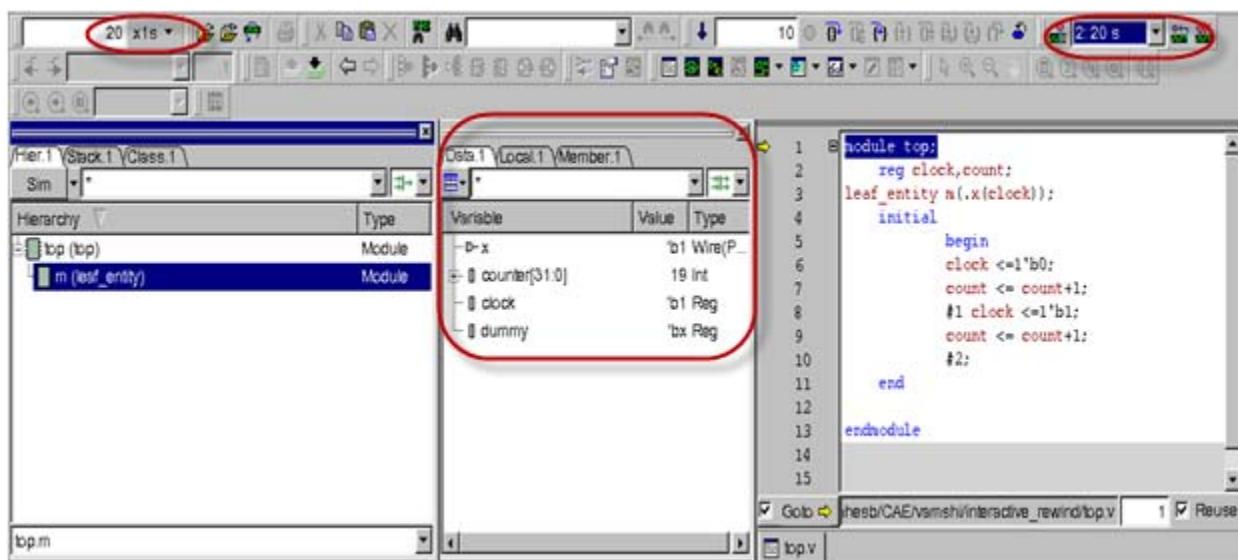
- Click the Rewind button to rewind to a previous checkpoint, say 20 ns.

The following illustrations show the comparison of variables in the Data pane and in the Source view when the simulation checkpoint was created at 20 ns and the status of the same after the rewind from 50ns to 20 ns.

Checkpoint created at 20ns



After the rewind at 20ns



You can also perform the following tasks from the DVE command prompt:

- Change the checkpoint depth using the `config checkpointdepth` command in the DVE Console pane.
- Delete a particular Session State in DVE using the command `checkpoint -kill <Checkpoint ID>`. Thus, a particular checkpoint ID is deleted and the remaining checkpoints are available with the same checkpoint ID.

Note:

Interactive rewind works with all debug options like `-debug`, `-debug_pp`, `-debug_all`. For more information about these debug switches, see the VCS User Guide in the VCS Online Documentation.

Limitations

- Not supported with FSDB dumping and with Specman GUI, Vera GUI, C debugger (cbug), Parallel VPD, Parallel VHDL, and Parallel VCS.
- Rewind operation doesn't rewind the Tcl files that you have created.
- If your PLI contains SOCKET connections or Multi-threads, and spanned processes, then those will not be rolled back along with the Checkpoint rewind.
- Interactive rewind is not supported for analog designs.

A

Menu Bar and Toolbar Reference

This chapter describes the menu bar, toolbar, command-line, preferences, command shortcuts, and GUI customization options. It includes the following topics:

- “[Menu Bar Options](#)” on page 2
- “[Editing Preferences](#)” on page 26
- “[Toolbar Reference](#)” on page 41
- “[Using the Context-Sensitive Menu](#)” on page 55
- “[Keyboard Shortcuts](#)” on page 67
- “[Using the Command Line](#)” on page 72

Menu Bar Options

This section provides an overview of the following TopLevel window menus and describe how to create user-defined menus:

- “File Menu”
- “Edit Menu”
- “View Menu”
- “Simulator Menu”
- “Signal Menu”
- “Scope Menu”
- “Trace Menu”
- “Window Menu”
- “Help Menu”
- “Testbench Debugger Menu Options”
- “User-Defined Menu”

File Menu

The following options comprise the **File** menu

Options	Description
Open Database...	Displays the Open Database dialog box, which enables you to select and open simulation database (VCD or VPD) files for post-processing.
Close Database...	Displays the Close Database dialog box, which enables you to close an open simulation database (VPD) file.

Load Waveform Updates	Loads the waveforms updates.
Reload Databases	Reloads the open databases for post-processing.
Open File	Displays the Open Source File dialog box, which enables you to select and display a source file in the Source view.
Close File	Closes the source file displayed in the active Source view or window.
Save Values	Saves values according to the selection: Tabular List — For List view Event Based List — For List view Memory Contents — For Memory view
Execute Tcl Script...	Displays the Execute Tcl Script dialog box, which enables you to select and source a Tcl script.
Load Session...	Displays the Load Session Dialog, which enables you to load a saved session.
Save Session...	Displays the Save Session dialog box, which enables you to save the current session.
Save Current View	Saves the current view without database information to session file.
Load Last Auto-Session	Loads the previously saved session.
Print	Prints to printer or file the contents of an active wave, list, or Schematic view.
Recent Databases	Displays a list of recently opened databases to choose from.
Recent Tcl Scripts	Displays a list of recently run scripts to choose from.
Recent Sessions	Displays a list of recently opened sessions to choose from.
Close View/Pane	Closes the currently active view or pane but the TopLevel window will still be open.
Close Window	Closes the currently active TopLevel window.
Exit	Exits DVE.

Edit Menu

The following options comprise the **Edit** menu:

Options	Description
Cut/Copy/Paste/Paste From/Delete	Copy works on any text. If the copy function can determine the text to be an object, copy will copy the object. Otherwise it will copy the selected text. Copied text can be pasted in any widget that supports text, for example an editor or the DVE command line. Object copies work in widgets, such as DVE panes, which support DVE objects that sort DVE objects such as any DVE panes. Note:Cut and Delete work only on DVE objects and are limited to some windows, such as the Wave, List, and Memory views. Paste From generates a list of clipboard objects copied from.
Expand By Levels	Contains the following submenu: All – Expands all items for all levels under the currently selected item. This option can take a long time if executed at the root level of a huge design. 2 – Expands 2 levels of child items from currently selected item. 3 – Expands 3 levels of child items from currently selected item. 4 – Expands 4 levels of child items from currently selected item. 5 – Expands 5 levels of child items from currently selected item.
Expand All	Expands all child items under all parents regardless of what item is selected.
Collapse Parent	Collapses to the parent of the currently selected item. If no item is selected, no action is taken.
Collapse All	Collapses all children to the top most parent regardless of what item is selected.

Synchronize Selection	Selection is not global. You can have different items selected in different panes at any time. Synchronize Selection allows you to sync up all panes to one selection.
	For example, if you have a signal selected in the Wave window but its parent scope is not shown in the Hierarchy window, clicking on Synchronize Selection will cause the parent scope of the signal to be highlighted in the Hierarchy window and also that signal will be highlighted in the Data pane and any other pane where it exists. This functionality is particularly useful in the Schematic window.
Select by Levels	Contains the following submenu: All – Selects all items for all levels under the currently selected item. Note: this can take a long time if executed at the root level of a huge design. 2 – Selects 2 levels of child items from currently selected item. 3 – Selects 3 levels of child items from currently selected item. 4 – Selects 4 levels of child items from currently selected item. 5 – Selects 5 levels of child items from currently selected item.
Select All	Selects all objects in pane or window.
Find...	Displays the Find dialog box.
Find Next	Active if any text exists in Find dialog box or find menu line edit. If clicked, finds next occurrence of the text in the active pane.
Find Previous	Similar to Find Next but finds previous occurrence of text.
Go To Address...	Displays a dialog box in which you can enter an address. This menu option is active if the Memory view is open and populated.
Search for Signals/ Instances...	Displays the Search for Signals dialog box. Use this to find any object that exists in the opened and current database. If the object is not loaded, this dialog box will attempt to load it.

Create Marker	Creates a marker in the Wave view. This is only active if a Wave pane exists in the TopLevel window for which the Edit menu is activated. If a Wave pane exists, clicking this menu option puts you in create marker mode. A white hashed marker is created and it follows the mouse in the Wave pane. The marker will be placed on the next mouse-click.
Markers...	Displays the Marker dialog box (See " Cursors and Markers " on page 54).
Go To Marker	Provides a list of markers that you have created in the current Wave view. You can select any marker to view the marker in the center of the Wave view. If no markers are present in the current window, the submenu will be empty.
Delete Marker	Provides a list of markers that you have created in the current Wave view. You can select any marker to delete it. If no markers exist, the submenu will be empty.
Move Marker	Provides a list of markers that you have created in the current Wave view. You can select any marker, then click the desired location in the Wave pane to move the marker to that location. If no markers are present in the current window, the submenu will be empty.
Set Reference Marker	Sets a currently selected marker as the reference marker for displaying values in relation to other markers.
Show Marker Values	Displays Absolute, Adjacent, or Relative values for signals at a selected marker. See " Cursors and Markers " on page 54.
Preferences...	Displays the Application Preferences dialog box (See " Editing Preferences " on page 24).
Undo Signal Group Operation	Undo the last signal group relative operation.
Redo Signal Group Operation	Redo the last signal group relative operation.

View Menu

The following options comprise the **View** menu:

Options	Description
Selection Tool	Used for schematic views. You can use this option to select objects. Ctrl-click adds objects to the selected set. Click and drag creates a box. Everything inside the box will be selected.
Zoom In Tool	Used for Schematic views. Enlarges the view of the selected object.
Zoom Out Tool	Used for Schematic views. Enables you to reduce the magnification level of an active object.
Pan Tool	Used for schematic views. Allows you to interactively move the center of the object without changing the scale. When you select this tool, the cursor changes to a hand. Click the toolbar selection arrow or press escape key to change the cursor back to normal.

Options	Description
Zoom>	<p>Contains a submenu for all zoom operations. These menu items are only applicable for Schematic and Wave views.</p> <p>Zoom Full – Fits all viewable objects into the current view.</p> <p>Zoom In – Enlarges the object.</p> <p>Zoom Out – Reduces the magnification level of the selected object.</p> <p>Zoom Fit Selection – Changes view to center on the selected set of schematic objects and so all selected objects are visible in the current pane.</p> <p>Zoom Fit Highlight – Same as Zoom Fit Selection but for highlighted schematic objects. This option is available if there are highlighted objects.</p> <p>Pan To Selection – Moves the view so the selected set of schematic objects is centered and viewable in the current pane. It does not change the zoom.</p> <p>Pan To Highlight – Same as Pan To Selected but for highlighted schematic objects.</p> <p>Zoom to Cursors – If the two Wave view cursors are present, this puts the area between the cursors in the view with each cursor at opposite sides of the view.</p> <p>Zoom to Time Range... – Displays a dialog box for entering a time range, then zooms to that time range.</p> <p>Back in Zoom and Pan History – Iterates through saved zoomed or panned views for current pane. When you change a zoom or a pan in a view, DVE stores the previous view so you can retrieve it.</p> <p>Forward in Zoom and Pan History – If you have gone backward in zoom/pan history, this menu item provides an easy way to go to the next view. Clicking this item will eventually get to the current view.</p> <p>Named Zoom and Pan Settings... – Displays a dialog box that allows you to choose from any views that you saved with a name.</p>
Set Time Scale...	Opens the Set Time Scale dialog box that allows you to change the time unit.
List Window Time Range...	Opens the Set Time Range dialog box that allows you to change the start time and end time in the List view. This option is available only from the List view menu bar.

Options	Description
Delta Cycle >	<p>This is available only if delta cycle information exists in the database. (Also see Capture Delta Cycle Values in Simulator Menu.)</p>
	<p>Expand Time – Expands at the simulation time (C1) time to show the delta cycles within that time.</p>
	<p>Collapse Time – Collapses the expanded delta cycle display at C1 time.</p>
	<p>Collapse All – Collapses any expanded delta cycle displays regardless of where C1 is.</p>
Associate With	<p>Associates a signal group in waveform or signal group pane with a specified database, so that when a signal from a different database is being added, the signal with the same hierarchical name of the specified database is actually added. Associate with "Any Database" to remove this setting.</p>
Go to Beginning	<p>Goes to the start of all simulation data (usually 0). This moves the cursor and changes the view in the Wave view to show the beginning of simulation time.</p>
Go to End	<p>Same as above but for the end of simulation data.</p>
Go to Time	<p>Displays the Go To Time dialog box that allows you to change the simulation time in the debugger. The Wave view shows the new simulation time.</p>
Link C1 to Sim Time	<p>Sets the debugger time with the current interactive simulation time. This option is available only in an interactive debug session.</p>
Move C1 to Sim Time	<p>Synchronizes the debugger time with the current interactive simulation time. This option is available only in an interactive debug session.</p>
Use Global Time (C1)	<p>Keeps simulation data in current TopLevel with the global C1 marker.</p>
Increase Row Height	<p>Increases the height of all traces in the Wave view.</p>
Decrease Row Height	<p>Decreases the height of all traces in the Wave view.</p>
Set Row Height to Default	<p>Resets the height of all traces in the Wave view to the default.</p>

Options	Description
Watch	<p>Provides some operations for the Watch pane:</p> <p>Add New Page - Adds a new page in the Watch pane.</p> <p>Delete Current Page - Deletes the currently selected page from the Watch view.</p> <p>Rename Current Page - Assigns a new name to the current page.</p> <p>Edit Variable - Lets you edit the variable name.</p>
Toolbars	<p>Contains the following submenu. You can select or clear the following check boxes to view or hide them as desired.</p> <ul style="list-style-type: none"> Edit File Scope Trace Window Signal Simulator Time Operations Zoom Zoom and Pan History

Simulator Menu

The following options comprise the **Simulator** menu.

Options	Description
Setup...	Displays the Simulation Setup dialog box to allow modification of default simulation runtime settings. Note, this does not allow you to control simulator compile-time settings. See “ Starting an Interactive Session from the DVE GUI ” on page 7.
Rebuild and Start	Rebuilds the simulator by executing the VCS Makefile, then starts the simulation.
Start/Continue	Runs the simulation until a breakpoint is hit, the simulation finishes, or for the duration specified in the Set Continue Time dialog box.
Stop	Stops a running simulation (same as CTRL+C in UCLI mode).
Step	Moves the simulation forward by stepping one line of code, irrespective of the language of the code. This is the same as the UCLI Step command.
Next	For VHDL, Verilog, and TB code, next steps over tasks and functions.
Next in CBug	Advances to the next line in CBug code.
Step In Active Thread	Stops at the next executable line in the current active thread.
Next in Active Thread	Advances over to the next executable line in the current thread only. This will step over tasks, functions, etc.
Step In Testbench	For Native TestBench (NTB) OpenVera and SystemVerilog testbenches, stops at the next executable line in the testbench.
Step Out	Steps to the next executable line outside of the current function or a task.
Restart	Stops the currently running simulation and restarts it with the current simulation setup. This retains all open windows and GUI setups. If the simulation is not running, this option will start it.
Show stack	Displays C-language, SystemVerilog and NativeTestBench stacks in Console pane.

Move up Stack	Steps up the current stack.
Move down Stack	Steps down the current stack.
Breakpoints...	Displays the Breakpoints dialog box that allows you to view, create, edit, enable, disable and delete breakpoints. See “ Setting Breakpoints in Interactive Simulation ” on page 18.
Run to Cursor	Runs and stops in the selection position if possible.
Save State...	Brings up a File Browser dialog box that allows you to save the current state of the simulator as a file.
Restore State...	Brings up a File Browser dialog box that allows you to restore a saved simulation state.
Add Checkpoint	Adds a simulation checkpoint.
Rewind to Checkpoint	Rewinds to the selected checkpoint.
Delete Checkpoint	Deletes the selected checkpoint.
Terminate	Kills a running simulation.
Force Value	<p>Force to 0 - Forces the values of the selected signal/variable to 0.</p> <p>Force to 1 - Forces the values of the selected signal/variable to 1.</p> <p>Force to x - Forces the values of the selected signal/variable to x.</p> <p>Force Release - Release the forces of the selected signal/variable.</p> <p>Set Force.. - Open the force dialog.</p>
Dump Full Hierarchy	Dumps the hierarchy of the entire design into a vpd file without triggering the dump of values.
Add Dump...	Brings up the Dump Values dialog box to specify scopes or signals to dump value change information starting at the current time. See “ Dumping Signal Values ” on page 9.
Dump	Turns on value change dumping at the current time for any selected scope or signal in the active pane.
Capture Delta Cycle Values	Toggles to turn on/off delta cycle dumping starting at the current time. Note that this substantially increases the VPD file size. You should try to limit the time span for dumping delta cycle values.

Continue For Time...	Displays the Continue for Time dialog box where you enter a time. The time specifies the duration for which the simulation runs if no breakpoints are hit. For example, if set to 10, the simulation runs ten times when you click the Continue toolbar button.
Periodic Waveform Update Interval...	Opens the Value Update Interval dialog box. You can choose time interval to enable periodic waveform update. This allows you to see waveforms dynamically as the simulator runs. The smaller the interval the worse the performance.
C/C++ Debugging	Enable - Enables debugging of C, C++, and SystemC source code. Allows to step in C/C++ code, set breakpoint etc. Show External Functions - Shows user-defined external functions (PLI, DPI, Direct C).

Signal Menu

The following options comprise the Signal menu:

Options	Description
Display Signal Group	Contains the following submenu that allows you to group the signals in the Wave view. New Signal Group - Creates a new signal group. All – Turns on visibility for all existing signal groups. All the signal groups are listed. You can choose to display or hide the particular signal group by selecting or clearing the respective check box.
Add to Waves	Contains the following submenu: New Wave view - Adds the selected signal to a new Wave view. Recent (New View) - Adds the selected signal to recently created Wave view. If none exists, creates a new Wave view. Create New Group - Creates a new group in the Wave view and adds the selected signal under the newly created group.
Add to Lists	Same as Add to Waves but for the List view.
Add to Groups	New Group – Creates a new signal group. The name will be Group<n>, where n is one more than the highest number of existing signal group. The new signal group is created at the top of the signal list. The existing signal groups are also displayed. You can add signals to the existing signal groups.
Add to Watches	Adds signal to the Watch pane.
Show Memory	Displays signal in the Memory view if the selected signal is a memory or MDA. If there is no current Memory view, DVE creates one according to the target policy.
Show Back Trace Schematic	Displays the Back Trace Schematic for the selected object.
Set Insertion Bar	Sets the insertion bar above the selected signal in the Wave or List view. This menu item is not available in other windows. If more than one signal is selected, DVE places the insertion bar above the signal closest to the top of the signal list.
Insert Divider	Inserts a blank divider row in the waveform display.
Show Definition	Locates the definition of interface or modport ports in the Hierarchy pane.

Sort Signals	Displays signals by declaration, ascendingly, or descendingly.
Set Bus...	Displays the Bus/Expression dialog box for managing the bus and expression creation/deletion. See “ Building Buses and Setting Expressions ” on page 5-29.
Set Expressions...	Displays the Bus/Expression dialog box for managing the bus and expression creation/deletion. See “ Building Buses and Setting Expressions ” on page 5-29.
Set Search Constraints	<p>Searches for signals per the search constraint. When the constraint is matched, the C1 cursor moves to that time location. Searching works only on the selected set of signals. If no signals are selected, it will search all the signals in the current view.</p> <p>Following are search constraints:</p> <ul style="list-style-type: none"> Any Edge – (Default) Searches for signals with any edge. Rising – Searches for signals with rising edge. Falling – Searches for signals and points to the falling edges. Failure – Stops on next or previous assertion failure. This options is available for assertion signals. Success – Stops on next or previous assertion success. This options is available only if the signal is an assertion. Match – Searches for the next or previous match. Mismatch – Stops on next or previous assertion mismatch. This option is available only if the signal is an assertion. X Value – Searches for any signal that contains x value. Signal Value... – Opens the Value Search dialog box that allows you to enter a specific value as the constraint. If the value is found, the search will stop and the cursor C1 is positioned at that value. Values must match the radix that is currently selected for the signal.
Search Backwards	Searches backwards in time.
Search Forward	Searches forward in time.
Highlight X Values	Toggles on/off highlighting X values for the singly selected signal. Useful at high zoom levels.
Compare...	Displays the Signal Compare dialog box where you can select waveforms to compare. See “ Comparing Signals, Scopes, and Groups ” on page 5-26.
Show Compare Info	Shows the result of the last signal comparison. This option is available only if a signal comparison has been performed.
Analog Overlay	Overlays selected signals in Wave view in a overlay signal group and displays the waveform in analog style.

Unoverlay	Unoverlays the overlaid signals.
Shift Time...	Displays the Shift Time dialog box to specify the parameters to shift a signal in time.
Set Radix	<p>Provides options for radix changes of the selected signal. Changing radix on a signal is global and will change the radix wherever the signal is displayed in DVE.</p> <p>User Defined > Allows you to specify and edit user-defined radices. See “Managing User-Defined Radices” on page 5-12.</p> <p>The other radices available are:</p> <ul style="list-style-type: none"> Enumerated Type ASCII Binary Octal Decimal Hexadecimal Unsigned Signed magnitude One’s Complement Two’s Complement Strength Default
Transaction Filter	Sets transaction filter.
Default Properties	Applies default properties to the selected signal. Clear the check box to allow the selected item to have its own radix and signal property.
Properties...	Displays the Signal Properties dialog box that allows you to change the signal properties.

Scope Menu

The following options comprise the Scope menu:

Options	Description
Show Source	Shows the source of the object selected. If multiple objects are selected, shows the source of the first object in the selected set. If the Use checkbox is checked, the Source view used is the current open Source view. If no Source view exists, DVE creates a new Source view according to the target policy.
Show Schematic	Same as above except that it shows the object in a Design Schematic pane.
Show Path Schematic	Same as above except that it shows the object in a Path Schematic pane.
Note: The following menu items affect the currently active Source and Schematic views.	
Show in Class Browser	Opens Class and Member pane; locates the currently selected class or member.
Show Full Hierarchy	Displays the full hierarchy in the Hierarchy pane.
Move Up	Moves up one level of hierarchy to the parent.
	Moves the selection in the Source view up one level of hierarchy from the scope of the currently selected line. If the current line is the top of the hierarchy or no line is selected, this item is not available.
Move Down	Moves down one level to the selected scope.
	Moves the selection to the start of the definition of the currently selected object. Note that in the Source view, this only works if the object itself is selected. It does not work if the entire line is selected and more than one token is selected on that line.
Back	Moves to the previous view of source information in the current Source view. DVE maintains a history of Source view views, so that it is easy to go back to a previous view of the source. This is useful for large source files and reduces the need for scrolling.
Forward	Same as above but moves forward in the source view history if you have previously gone backwards.

Show	Allows navigation to certain types of relative source lines based on the selected object type. The Source view only shows definition. This menu contains the following submenu: Definition – Shows the definition of the selected object. Current Scope – Changes the selection to the first line of the current scope. Assertion – Changes the Source view to the definition of the selected assertion if the object is an instance of an assertion. Unit Binding – Changes the Source view to the location where the assertion is bound to a module with a bind statement for OVA assertion instances. Entity – Changes the Source view to the architecture's entity if the current selection is in a VHDL architecture. Architecture – Changes the Source view to the entity's architecture if the current selection is in a VHDL entity. Macro - Shows the value and information of the selected macro in a separate Source view. The Source view contains all the queried macros in it. Macro Definition - Shows the definition of the selected macro in the Source view. In Class Browser - Shows class in the Class browser.
Edit Source	Displays a text editor based on your Editor source preference setting (vi editor is default). DVE preloads the editor with the source file that is in the currently active DVE Source view and positions it on the same selected line. If no file is open in the Source view or a different kind of DVE pane is active, this menu option is not available.
Edit Parent	Same as above except that the source of the parent instance is preloaded in the text editor.
Expand Path	Expands fanin or fanout of the current path to one additional level from the selected object.
Add Fanin/Fanout...	Displays the Fanin/Fanout dialog box in which you specify fanin/fanout parameters for the path schematic. This menu option is available for Path Schematic only.
Annotate Values	Allows you to toggle signal annotation on and off for the current scope in Source/Data/Schematic/Path Schematic views. Annotation allows you to see values within the context of the display. For example in the Source view, the annotation will be below the source text for variables, while in the schematic, the annotation will be on pins or nets.

Properties	Opens the Properties dialog box that shows all available properties for the currently selected schematic or path schematic object. This menu option is available for Schematic and Path Schematic only.
------------	---

Trace Menu

The following options comprises the **Trace** menu:

Options	Description
Trace Assertion	Traces the assertion and displays the results in the current or a new Wave view automatically. This menu option is active only if an assertion is selected in the currently active pane and certain conditions are met (libassertiondebug.so must be available). If a specific assertion attempt is selected, that attempt will be traced. If you select no specific attempt, DVE traces the first attempt. The assertion trace gives detailed debugging information for a specific assertion attempt, so you can easily point to the expression in the assertion that failed.
Assertion Attempts...	Displays the Assertion Attempts dialog box. If the selected object in the currently active pane is an assertion, the dialog box will be populated with all attempt information for that assertion. If the selected object is not an assertion, the dialog box is empty.
Trace Drivers	Finds the driver of the object, shows the driver in the drivers/loads pane and shows the driver in the Source view of the current window. If no window is present, DVE creates one. If a drivers pane already exists, the new trace information is added to it.
	This menu option will be active if the object selected in the currently active pane is a variable or signal. For this capability to work, the design must be compiled with one of the -debug options with complete simv.daidir directory.
Trace Loads	Finds loads of the signal.
Drivers/Loads	Allows you to navigate and manage drivers/loads. See Tracing Drivers and Loads .

Follow Signal	Displays a list of instances of a selected signal in the Source view.
Highlight	<p>Allows you to highlight and manage highlighted objects in all panes.</p> <p>Recent Color - Highlights any currently selected object in the current view with the recently used color.</p> <p>Clear Selected – Removes all highlight color from the selected object.</p> <p>Clear All – Removes all highlights regardless of whether the object is selected or not.</p> <p>Clear by Color - Removes the highlights from the object. (The color is selected in the submenu.)</p>
Spot Signal Path	Follows the selected signal through the design in path schematic.
Stop Signal Spotting	Stops following the signal path.
Back Trace	Starts back tracing on selected signal.
Back Trace Options	Displays the Back Trace Options dialog box.
Hide/Show Wave View	Toggles display of the Back Trace Wave view.
Trace Transaction	Sets context for the transaction message.
Show Relations	Opens the Transaction Relations dialog box.

Window Menu

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

Options	Description
New	Opens a new instance of one of the following window selections according to current target criteria: Source View Schematic View Path Schematic View Wave View List View Memory View
Set This Frame Target For	Sets the currently active frame as the target for one of the following selections: Source View Schematic View Path Schematic View Wave View List View Memory View
Panes	Displays the selected pane in the active TopLevel window. Console Hierarchy Data Signal Groups DriverLoad Stack Local Watch Assertion
New Top Level Frame	Opens a new TopLevel frame displaying one of the following selections: Empty Assertion + Hierarchy Hierarchy + Data Hierarchy + Data + Console Console
Load Default Layout	Returns the currently active TopLevel frame to the default layout.

Load Layout	Loads a layout session. Reset Layout - Resets the layout to the initial layout. From File - Loads a pre-saved layout session file.
Save Current Layout	Saves the current DVE layout. To default - Saves the current DVE layout as <code>~/.synopsys_dve_default_layout.tcl</code> . When DVE restarts, it uses this default layout instead of the last layout when you have exited DVE. To File - Saves the current DVE layout to a layout session file.
Maximize View	Maximizes the selected view to cover the entire layout.
Arrange	Arranges all the non-docked panes in current toplevel window. Cascade - Display windows in cascade format. Tile - Display windows in tile format. Vertical - Display windows in vertical tile format. Horizontal - Display windows in horizontal tile format.
Dock in New Row	Positions the currently active window or pane in a new row of the current TopLevel frame according to one of the following selections: Left Right Top Bottom
Dock in New Column	Positions the currently active window or pane in a new column of the current TopLevel frame according to one of the following selections: Left Right Top Bottom
Undock	Undocks the selected window from the TopLevel window.
Move To	Moves to new Top level window.
Set Top Level Title	Sets the title of the top level window.
Current Window List	Lists the current TopLevel windows.

Help Menu

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

Options	Description
Help Contents	Opens the DVE User Guide in the VCS Online Documentation. Note: Firefox is the default browser for viewing the product online help documentation. To change the default browser setting, you must set the environment variable BROWSER before starting DVE. For example, to view the online help documentation in Mozilla browser, set BROWSER to mozilla, as shown below: % setenv BROWSER mozilla
Help Search	Opens the Search tab of the VCS Online Documentation.
A Quick Start Verilog Example	Loads an example design.
A Quick Start Mixed Example	Loads an example design when VCS MX is running.
Tutorial for Mixed Example	Loads an example design for VCS MX.
About	Displays DVE version and copyright information.

Testbench Debugger Menu Options

A few menu options specific to debugging your testbench programs are available. These options are added to different menus in the main DVE GUI such as View menu, Simulator menu, and the Window menu.

The following section explains the new options in different menus.

View Menu

The View menu allows you to customize your viewing options in the Watch pane. It has a new menu option, Watch. Click **View > Watch** and select the relevant option, as described in the following table:

Options	Description
Add New Page	Adds a new tab to your Watch pane.
Delete Current Page	Deletes the Watch tab from the Watch pane.
Rename Current Page	Renames your Watch tab.
Edit Variable	Allows you to cut, copy, and paste the variable names in the Watch tab.

Signal Menu

The Signal menu allows you to add signals to the Watch tab. To add a signal for monitoring, select **Signal > Add to Watches > Watch**.

Simulator Menu

The Simulator menu allows you to run and control the simulation. Use the following commands to control the simulation:

Options	Description
Step In Testbench	Stops at the next executable line in the testbench for Native Testbench (NTB), OpenVera, and SystemVerilog testbenches.
Step Out	Steps to the next executable line outside of the current function or task.
Step in ActiveThread	Stops at the next executable line in the current active thread.
Next in Active Thread	Advances the simulation stepping over tasks and functions in the current active thread.

You can also use the ucli -thread command.

Window Menu

The Window menu allows you to select or clear the following panes to debug your testbench programs. To enable the panes, select **Window > Pane** and then select the relevant option, as described in the following table:

Options	Description
Stack	Stack pane to your DVE top-level window.
Local	Local pane to your DVE top-level window.
Watch	Watch pane to your DVE top-level window.

User-Defined Menu

You can create your own menu in DVE using the following command:

```
gui_add_menu -menu "myMenu > MenuTcl" -tcl_cmd myExec -icon  
$::env(DVE_AUXX_GUI)/images/toolbars/grn_dot.xpm  
proc myExec {} { source myTCL.tcl }
```

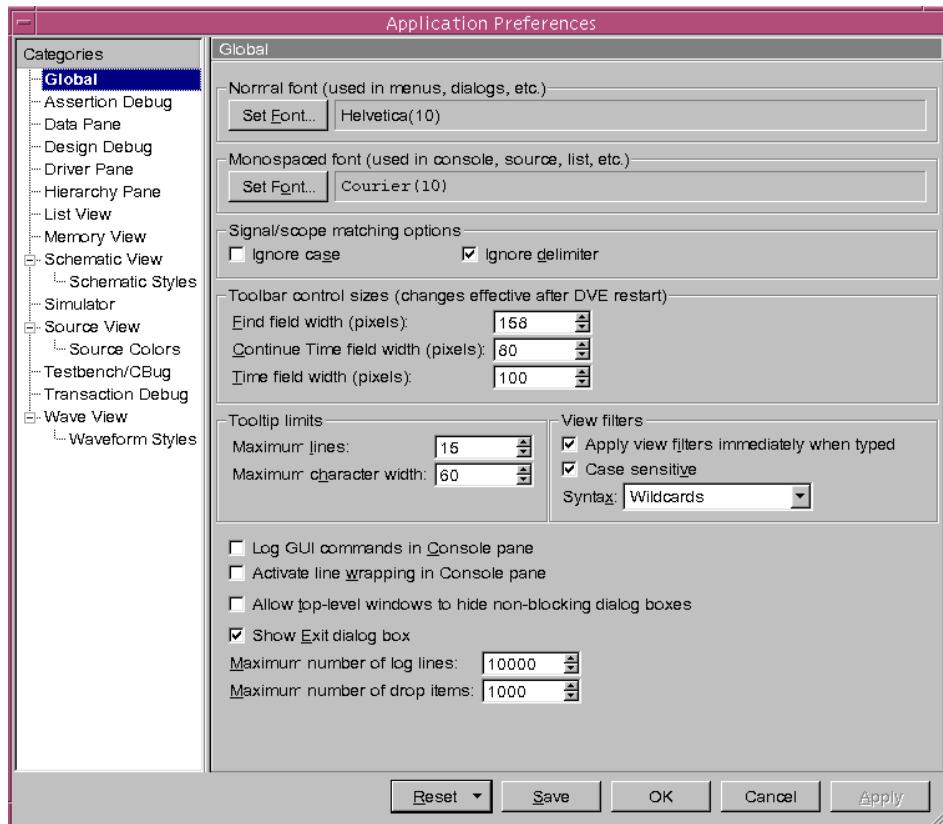
This command creates a new menu item (user-defined) called myExec in the menu myMenu and executes the Tcl script myTCL.tcl when selected. The icon displayed next to this menu item is specified by the -icon argument.

You cannot add a new icon to an existing menu item (even if it is copied to another menu).

For example, **File > Exit** is added to the menu myMenu (which becomes **myMenu > Exit**). Since **File > Exit** has no icon associated with it, **myMenu** will also not have any icon, even if you add it explicitly. All the menu add commands have to be in the `~/.synopsys_dve_usersetup.tcl` file.

Editing Preferences

DVE creates the '`.synopsys_dve_prefs.tcl`' file, which stores user preference information. The Application Preferences dialog box contains the following categories and options.



Global Options

Options	Description
Set Font	Allows you to set the font of dialog box, menu commands, and font in Source view, Console pane, List view etc.
Signal/scope matching options	Ignore case - Ignores cases while searching for signals or scopes. Ignore delimiter - Ignores delimiter characters while searching for signals or scopes, such as comma character.
Toolbar control sizes	Find field width - Specifies the width of the Find field. Continue time field width - Specifies the width of the Continue Time field. Time field width - Specifies the width of the Time field.
Tooltip limits	Maximum lines - Specifies the maximum lines of tooltip to be shown. Maximum Character widths - Specifies the maximum width of characters in the tooltip.
View filters	Apply view filters immediately when typed - Allows the text filters to work while the text is still being typed. If it is off, you need to press the Enter key for the text filter to take effect. Case sensitive - Specifies of the filter should be case sensitive. Syntax - Specifies the syntax for filters, regular expression, wildcards, or simple strings.
Log GUI commands in Console pane	Records GUI commands also in the Console pane with the UCLI commands.
Activate line wrapping in Console pane	Displays the log commands with line wrapping enabled, which lets you view the commands in next line when the line is full, such that each line fits in the viewable window.
Allow top-level windows to hide non-blocking dialog boxes	Allows non-modal dialog box to go behind its parent window, so that the parent window can be fully shown without closing the dialog box.
Show Exit dialog box	Displays the Exit dialog box when you exit DVE, if selected.
Maximum number of log lines	Specifies maximum number of log lines to be displayed.

Options	Description
Maximum number of drop items	Specifies maximum number of drop items to be displayed.

Assertion Debug Options

Options	Description
Automatically open Assertion pane	Opens the Assertion view, if DVE detects that the database contains assertion data.
Dump all scopes with assertions when Assertion pane is opened	Dumps all the scopes when the Assertion view is opened, so that all assertions can have values in the VPD after that.
Docked Assertion pane	Docks the Assertion view in the top-level window, instead of showing it as a pane like the Source view.
Show warning when libassertdebug.so is missing	Shows a warning message if libassertdebug.so is not available for assertion debugging.
Show cover and sequence mismatches in Wave view	Decides whether the cover and sequence mismatches will be shown in the Wave view.
Label assertion attempts with start time	Decides whether the assertion attempt's start time will be labelled always, never, or only with failures.

Data Pane Options

Options	Description
Initial state	Specifies in which state you want to display the Data pane, Detail mode or List mode.
Options for detail mode	Specifies the display options for the Detail mode. Shows grid, Value column, value annotation, adjusts width of Name, Value, and Type column if selected. Sort signal on loading — Sorts the signals after loading in to the Data pane. Show signal name by level — Shows the name of the signals in waveform with levels of scopes. Show horizontal scroll bar — Displays the horizontal scroll bar per pane or per column as per your selection.
Options for list mode	Specifies the maximum number of characters for signal name and adds selected signal to Wave/List views in the order of display or selection.

Design Debug Options

Options	Description
Initial display time units	Sets the time unit of DVE according to first opened database or a specified time unit.
Always use units	Uses the selected time unit.
Signal value display	Sets the display style of value changes.
Maximum number of elements of array	Sets the biggest array that can be added into Wave or List views. For bigger arrays, Memory view can be used to view the values.
Limited number of elements of array for annotate values	Sets the number of elements of array that value annotation will show.
Use the simulation as design debug library in interactive	Allows DVE to use simv in interactive mode as design debug library to get connection data.

Options	Description
Use separate thread for loading value changes	Uses a separate thread for loading the value changes.
Pre-load design debug library	Allows DVE to load the static design debug library in advance in post-process mode, or when this preference option is not selected.
Use design debug library for design hierarchy in post process	Allows you to view the complete hierarchy in a partially dumped VPD.
Use design debug library for signal search	Allows DVE to search for objects in the design debug library instead of the vpd.
Treat modules defined within 'cellddefines as blackbox (library) cells	Hides the internal schematic for cellddefined modules.
Treat modules defined within 'uselib or -y/-v as blackbox (library) cells	Hides the internal schematic for uselib or -y/-v modules.
Sync the current scope and the GUI	Syncs the simulator's current scope or active scope with DVE GUI widgets. Shows the source code of the current scope.
Use a highlight color scheme for ports	Allows you to select the highlight color for In, Out, and Inout ports.

Driver Pane Options

Options	Description
Display full path names for signals	Displays full path of the signals, if selected.
Driver displays scope instance rather than source line	Shows the scope of the driver instead of the source and line in the Driver pane.
Hide duplicate drivers/ loads that point to the same source line	Hides the duplicate drivers/loads that point to the same source line, if selected.
Driver value display radix	Specifies the radix for driver value, Hexadecimal, Octal, or Binary.
Active statement driver detection	Turns on active statement driver tracing for tracing drivers.
Show both active and inactive drivers	Displays both the active and inactive drivers.

Hierarchy Pane Options

Options	Description
Display options	Displays grid lines, scope navigator, and Type column in the Hierarchy pane, if selected. Adjusts Name and Type column width.
Show horizontal scroll bar	Allows the horizontal scrollbar to be shown per column or use one for the entire pane.
Initial states of filters	Sets the initial status of type filters for Hierarchy pane. You need to restart DVE for this option to take effect.
Maximum number of child items to sort when expanding a scope	Sets the maximum number of child items that Hierarchy pane can sort.

List View Options

Options	Description
Show grid	Show signal names in grid form, and justify text preferences for the signal pane. These settings are similar to data pane preferences.
Truncate signal name for number of characters to keep	Sets the maximum number of characters shown for each signal in the List view.
Use Insertion Bar	Shows the red insertion bar in the List View.
Bring List view to the front when adding signals	Brings the List view in the front when adding objects to the existing List view, which is hidden behind other top-level windows.
Show signal name by level	Shows the name of the signals in waveform with levels of scopes.
Minimum spacing between value columns	Sets the minimum number of characters between two value columns.
Show database name for signals	Displays the database name for signals using either the design designator or the name of the VPD file, as per your selection.

Memory View Options

Options	Description
Change the text alignment of memory table to left	Aligns the text of Memory table to the left.
Selection color in the memory table	Specifies the color for the Memory table.

Schematic View Options

Options	Description
Maximum number of cells in schematic	Specifies the maximum number of cells to be displayed in the Schematic view.
Text size scheme	Specifies the font size in schematics.
Automatic zoom to selection in design schematic	Moves and zooms the schematic automatically, so that the new selection from searching can be shown in the visible range.
Expand path connections for any connected direction	Expands path in path schematic to get more connection which doesn't contribute to the expanded item but you might want to know.
Disable tooltip in Schematic view	Disables the tooltip in the Schematic view, if selected.
Enable buffer (buf) and inverter (not) compression	Enables the buffer and inverter compression in the Schematic view.
Enable zooming in Schematic view with mouse wheel	Enables zooming when you roll the mouse.
Ignore read-only Verilog system calls	Ignores the read-only Verilog system calls, for example \$monitor or \$display, when selected.
Show Value Annotation	Displays the annotated values of the signals.
Maximum length of net's name in schematic	Specifies the maximum length of net's name in the Schematic view.
Hover color	Specifies the color on mouse hover.
Path schematic setting	Specifies the path schematic settings. Scroll/zoom to fit the new path and the starting point - If selected, the path schematic view is zoomed or scrolled so that the newly created path and its starting point is fit in the window view. Select the new path and starting point - Selects the newly created path and its starting point. Highlight newly expanded paths in color - Highlights the newly expanded paths in the chosen color.

Options	Description
Schematic Styles	Specifies visibility and color for port/pin and other schematic objects. Pin symbol - Specifies the pin symbol that you want to use, box with direction or a simple line.

Simulator Options

Options	Description
Show Simulation Setup dialog box for Simulator > Rebuild and Start	Displays the Simulation Setup dialog box when you select the Rebuild and Start option from the Simulator menu, when this check box is selected.
Show Simulation Setup dialog box for Simulator > Restart	Displays the Simulation Setup dialog box when you select the Restart option from the Simulator menu, when this check box is selected.
Set simulator executable in Simulation Setup dialog box to simv/scsim by default	Selects simv/scsim as the simulator executable by default in the Simulator Executable field in the Simulation Setup dialog box, when this check box is selected.
Allow using GUI when executing 'run' command in the script	Allows you to use the DVE GUI when you are running the simulation, when this check box is selected.
Seek next can advance simulation	Advances the simulation when you select the Seek Next option, when this check box is selected.

Source View Options

Options	Description
Automatically load source for top module	Loads source code for the top module in the Source view when you open DVE.
Show value annotation	Enables source code annotation, if selected. You can view the value of the simulation run at a given time directly in the source code.
Activate line wrapping	Lets you view the commands in next line when the line is full, such that each line fits in the viewable window.
Show line numbers	Shows number of each line of the source code.
Reuse first Source view	Displays the source code in the existing Source view.

Options	Description
Use dialog to warn about outdated source files	Displays a dialog box that warns that the source files are outdated.
Display encrypted content in Source view	Shows the encrypted source. By default, DVE hides the encrypted source so that the junk characters are not displayed.
Enable 'include file expansion in source code	Enables 'include file expansion in source code
Tab width	Defines the number of spaces for each tab key ('\t') in the Source view.
Editor	Lets you select the editor type to view the source code.
Reload source when changed on disk	Allows DVE to detect the file change and automatically reloads the source if "Always" is set.
Source Window Colors	Specifies colors to display Source view components by clicking the drop-down arrows and selecting a color. You can also select the background color of the active and inactive scope.

Testbench/CBug Options

Options	Description
Watch array elements display limit	Sets the maximum number of elements that the GUI can display for a large array when it is expanded in the Local pane and Watch pane.
Stack depth display limit	Sets the maximum levels of stacks for Cbug.
Enable testbench debugging for interactive design	Enables the Testbench Debugger panes in the DVE main window. This will disable "step in testbench" and the programs will not be dumped and shown in the Hierarchy pane. However, you can still open the testbench source files manually and set line breakpoints, but you cannot see the testbench signals.
Share Hierarchy and Stack panes in dockable pane	Opens the Hier pane and the Stack pane in a dockable window. Clearing this check box opens the pane in a separate window.
Share Data and Local panes in dockable pane	Opens the Local pane and the Data pane in a dockable window. Clearing this check box opens the pane in a separate window.
Sync the current frame and the GUI (Stack pane)	Syncs the current frame or active frame with DVE GUI widgets. Shows the source code of the current frame.
SystemC sc_fifo dumping	Contains the following options: Dump details of tokens stored in FIFO - Dumps the tokens details in first in first out order. Dump names of processes waiting for read/write - Dumps the processes name that are waiting for either read or write. Maximal number of tokens - Sets the tokens limit.
CBug: Store source file/line info for all the SystemC instances and processes in VPD (performance impact)	Stores information of the source file/line for all instances and processes of SystemC in the VPD file.

Options	Description
CBug: Store SystemC class member variables which are not derived from SystemC base classes in VPD (performance impact)	Saves the class member variables of SystemC, that are not derived from the SystemC base classes, in the VPD file.

Transaction Debug Options.

Options	Description
Transaction pane settings	Contains the following two options: Automatically open Transaction pane — Opens the Transaction pane automatically when you start DVE. Docked Transaction pane — Displays the Transaction pane in the same TopLevel window, if selected.
Stream list settings	Contains options to show grid, left justify Name and Scope columns.
Waveform settings	Contains the following options: Zero-time message box width — Displays the message box in the width selected. Show relations — Displays the relations specified in the \$vcplusmsglog statement. Show values — Displays the values of the \$vcplusmsglog and \$vcplusmsglog objects in the Wave view. Maximum value lines — Displays the values upto the lines selected. Show call stack — Displays the call stack's head, tail, both, or nothing in the Wave view and in the Tooltip. Maximum call stack — Displays the call stack upto the maximum levels chosen.

Wave View Options

Options	Description
Display settings	Specifies the waveform display settings.
Use Insertion Bar in the Signal pane for the location to insert the signal(s)	Inserts the signal in the Signal pane wherever the insertion bar is placed.

Options	Description
Keep original signals when creating overlay signal	Keeps the original signal from waveform after they are overlaid.
Ignore case when sorting signals	Ignores cases while sorting signals.
Disable dragging to change the height for digital waveforms	Disables changing the height of waveform signals in digital drawing style, if selected.
Enable tracing on double-click	Starts tracing the signal when you double-click on a signal.
Enable zoom mouse gestures with the left mouse button	<p>Allows zoom action using left mouse button when the mouse is dragged in different direction.</p> <p>Drag from left to right or from right to left means zoom by selected range.</p> <p>Drag towards down-left direction means last zoom, drag towards down-left direction means next zoom in history,</p> <p>Drag upwards or downwards means zoom full, drag towards up-right direction means zoom in 2x, while drag towards down-right direction means zoom out 2x.</p>
Enable zoom mouse gestures with the middle mouse button	Same as above, but uses the middle mouse button.
Enable vertical scrolling synchronization for linked views	Allows you to scroll linked views vertically, if selected.
Bring Wave view in the front when adding signals	Brings the Wave view in the front when you add signals.
Show signal name by level	Shows the name of the signals in waveform with levels of scopes.
Default wave row height	Defines the row height of the wave.
Defaultminimumpixels to trigger stroke	Specifies the number of pixels needed to trigger a zoom operation by mouse gestures except for zoom range (left to right or right to left).
Bus value	Displays bus value as either LSB or MSB
Waveform value font	Specifies the font to be used for waveform value.

Options	Description
Show database name for signals	Displays the database name for the signal by either using the design designator or the name of the VPD file.
Waveform Styles	Allows you to select the style scheme and color for various data types.
	Analog values changes within one pixel - Chooses values to plot.

Toolbar Reference

This section describes all toolbar text fields, menus, and icons. You can drag and drop toolbars into any location in a TopLevel DVE window toolbar using the toolbar handles.

To toggle the display of toolbars on and off, select **Edit > Toolbars**, then select the desired toolbar.

File

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

Icon	Description
	Displays the Open Database or Open File dialog box, depending on the DVE window displayed, and enables you to select and open a VPD file.
	Loads waveform updates

Open Database or File

Load Waveform Updates



Print

Prints to printer or file the contents of an active wave, list, or Schematic view.

Edit

The following items comprise the **Edit** toolbar:

Icon	Description
	<p>Cut, Copy, Paste, Delete</p> <p>Copy works on any text. If the copy function can determine the text to be an object, copy will copy the object. Otherwise it will copy the selected text. Copied text can be pasted in any widget that supports text, for example an editor or the DVE command line.</p> <p>Object copies work in widgets, such as DVE panes, which support DVE objects that sort DVE objects such as any DVE panes.</p> <p>Cut and Delete works only on DVE objects and some windows and are limited to some windows, such as the Wave, List, and Memory views.</p>
	<p>Search for Signals/Instances</p> <p>Displays the Search for Signals dialog box. Use this to find any object that exists in the opened and current database. If the object is not loaded, this dialog box will attempt to load it.</p> <p>Selects string to search for, then press Enter to search.</p>
	<p>Find</p>
	<p>Find Previous/Next</p> <p>Active if any text exists in the Find dialog box or the Find menu text box. If clicked, finds the previous or next occurrence of the text in the active pane.</p>

Zoom/Zoom and Pan History

Toolbar Command	View Menu Command	Action
	Selection Tool	Prepares the cursor for selecting objects (the default cursor).
	Zoom In Tool	Prepares the cursor for zooming in. The cursor becomes a magnifying glass. Drag a bounding box around the area to enlarge.
	Zoom Out Tool	Prepares the cursor for zooming out. The cursor becomes a magnifying glass. Drag a small box to zoom out by a large amount, or a large box to zoom out by a small amount.
	Pan Tool	Prepares the cursor for panning the window view. The cursor becomes a hand shape. Point and drag to pan the view.
	Zoom Full	Zooms out to display entire design.
	Zoom In 2X	Zooms in 2x.
	Zoom Out 2X	Zooms out 2x.
	Named zoom and pan setting	Zooms to your settings.
	Zooms to selection	Zooms to area selected with the Selection Tool.
	Go back in zoom and pan history	Goes back to the last zoom setting.
	Go forward in zoom and pan history	Goes forward in the zoom history.
	Zoom to Cursors	Zooms to cursors C1 and C2.

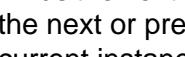
Scope

The following items comprise the **Scope** toolbar:

Icon	Description
	Displays the currently active scope signal values at the current time in the Source view.
	Moves the selection in the Source view up one level of hierarchy from the scope of the currently selected line. If the current line is the top of the hierarchy or no line is selected, this item is not available.
	Moves the selection to the start of the definition of the currently selected object. Note that in the Source view, this only works if the object itself is selected. It does not work if the entire line is selected and more than one token is selected on that line.
	Back or forward arrow moves to the previous view of source information in the current source view or forward in the source view history if you have previously gone backwards.
	DVE maintains a history of Source view, so that it is easy to go back to a previous view of the source. This is useful for large source files and reduces the need for scrolling.

Trace

The following items comprise the **Trace** toolbar:

Icon	Description
	This menu item is active if the object selected in the currently active pane is a variable or signal. For this capability to work, the design must be compiled with one of the -debug options and an mdb library must exist.
	When clicked, this finds the driver of the object, shows the driver in the Drivers/Loads pane and shows the driver in the Source view of the current window. If no window is present, DVE creates one. If a drivers pane already exists, the new trace information is added to it.
	Finds the next or previous driver or load or the next or previous driver or load in the current instance respectively.

Window

The following items comprise the **Window** toolbar:

Icon	Description
	Opens a new Source view and display source for the selected object.
	Shows the schematic pane.



Opens a new Path Schematic pane.

Show Path Schematic

Show Back Trace

Displays the back trace schematic for the selected signal.



Schematic



Opens a new Wave pane or display a previously opened pane.

Show Wave



Opens a new List pane or display a previously opened pane.

Show List



Opens a new Memory pane.

Show Memory

Adds signal to the Watch pane.



Add to Watches

Opens the selected pane



Panes

Back Trace

The following items comprise the **Back Trace** toolbar:

Icon	Description
	Displays the Back Trace Wave view.
	Toggle display of the Back Trace Wave view
	Opens the Back Trace Options dialog box.
	Show Back Trace Options
	Starts back tracing the selected signal.
	Start Back Trace on selected signal

Interactive Rewind

Interactive Rewind is an LCA feature. For more information, see the VCS Online Documentation.

The following items comprise the **Interactive Rewind** toolbar:

Icon	Description
	Adds a simulation checkpoint.
	Rewinds to the selected checkpoint.
	Deletes the selected checkpoint.

Signal

The following items comprise the **Signal** toolbar:

Icon	Description
------	-------------



Search Backward/Forward/ Select Search Criteria

Backward or forward arrow launches search in time for the constraint selected in the listbox.

- **Any Edge** – (Default) Search stops and positions C1 cursor on the next or previous edge found.
- **Rising** – Search stops and positions C1 cursor on the next or previous rising edge only.
- **Falling** – Search stops and positions C1 cursor on the next or previous falling edge only.
- **Failure** – Available only if the signal is an assertion; stops on next or previous assertion failure.
- **Success** – Available only if the signal is an assertion; stops on next or previous assertion success.
- **Vacuous** – Available only if the signal is an assertion; stops on the next or previous vacuous success.
- **Signal Value ...** – Displays a small dialog box that allows you to enter a specific value as the constraint. If the value is found, the search will stop and position C1 where the signal takes on the entered value. Values must match the radix that is currently selected for the signal.

Set the number of matched values to seek for one search backward/forward operation by clicking the search icon.

Set number of seeks

Simulator

The following items comprise the **Simulator** toolbar:

Icon	Description
	Runs the simulation until a breakpoint is hit, the simulation finishes, or for the duration specified in the Set Continue Time dialog box or toolbar time entries.
	Runs the simulation for the specified time, then stops.
	This icon is active when the simulation is running. Click to stop the simulation.
	For VHDL, Verilog, and TB code, next steps over tasks and functions.
	Moves the simulation forward by stepping one line of code, irrespective of the language of the code. This is the same as the UCLI Step command.
	Steps to the executable line in the active thread.
	Steps to the next executable line in the active thread.
Next in Active Thread	



Steps in C code

Next in CBUG Code



For Native TestBench (NTB) OpenVera and SystemVerilog testbenches, stops at the next executable line in the testbench.

Step In Any Testbench Thread



Steps to the next executable line outside of the current function or a task.

Step Out



Stops the currently running simulation and restarts it with the current simulation setup. This retains all open windows and GUI setups. If the simulation is not running it is started.

Restart

Time Operations

The following items comprise the **Time Operations** toolbar:

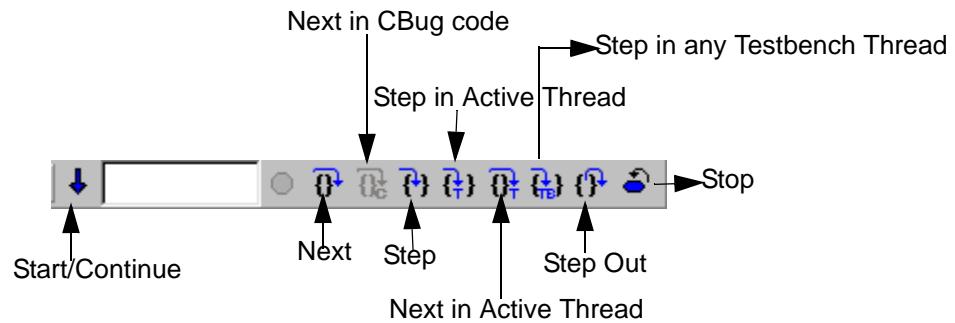
Icon	Description
	Displays current time of the C1 cursor. Set the current time by entering a new time in this field.
Sets current time and display time unit	Displays the time units for displaying simulation data. Select View > Set Time Scale to set time units and precision.

Grid

The following items comprise the **Grid** toolbar:

Icon	Description
	Opens the Grid Properties dialog box.
Setting Grid in Wave view	

Testbench GUI Simulator Toolbar Options



For more information on these icons description, see the section [“Simulator”](#)

Note:

Move the cursor over the icon to display the ToolTip.

Customizing the DVE Toolbar

For customizing the DVE toolbar, you need to use Tcl commands. You can create toolbars only for items that exist in one of the DVE menus. For example in the **Edit** toolbar, **Copy** and **Cut** are toolbar items.

Note:

For repeated usage, integrate the Tcl commands in your
~/ .synopsys_dve_usersetup.tcl file.

Adding a New Toolbar

To add a new toolbar, use the `gui_create_toolbar` command.

```
gui_create_toolbar -name myName -title myTitle
```

The argument passed to `-name` should be used to add items to the new toolbar. Note that the toolbar is not visible until you add an item to it.

Adding Items to a Toolbar

To add items to a toolbar, use the `gui_add_toolbar_item` command.

```
gui_add_toolbar_item -toolbar myName -item "Edit > Find..."
```

Ensure to retain the trailing dots in the toolbar item name if the corresponding menu item has it.

Deleting an Existing Toolbar

Determine the toolbar name by placing the mouse over the toolbar handle. The toolbar can be deleted from the GUI or from the command-line.

To delete a toolbar from the GUI, perform any of the following steps:

- Right-click on the corresponding toolbar handle and select **Hide**.
- Right-click on any toolbar handle or empty toolbar area and clear the check box corresponding to the toolbar.

To delete a toolbar from the command-line, use the following command:

```
gui_delete_toolbar -name "<toolbar_name>"  
gui_delete_toolbar -name "&Edit"
```

Deleting an Item in a Toolbar

To delete a single item from a toolbar

1. Determine the toolbar name and the name of the item to delete.
2. Run the `gui_delete_toolbar_item` command.

```
gui_delete_toolbar_item -toolbar Edit -menu "Edit > Copy"
```

Ensure to retain the trailing dots in the toolbar item name if the corresponding menu item has it. Toolbar items can only be deleted from the command-line.

Using the Context-Sensitive Menu

The following sections explain the commands and their descriptions available in the CSM of each of the DVE panes and views. In any pane/view, right-click to display the CSM, then select the desired command.

Hierarchy Pane CSM

Command	Description
Copy	Copies the selected object or string to the clipboard buffer.
Scope Navigator	Toggles On/Off the scope navigator toolbar
Show Source	Displays source code in the Source view for the selected object.
Show Schematic	Displays a design schematic for the selected object.
Show Path Schematic	Displays a path schematic for the selected object.
Add To Waves	Displays the selected signal or signals in the Wave view.
Add To Lists	Displays the selected signal or signals in the List view.
Add To Groups	Adds objects to a signal group (new or existing per your selection) in the Signal pane.
Add To Watches	Adds objects to the last active Watch pane.
Show in Class Browser	Opens the Class and Member panes and locates the currently selected class or member.
Show Full Hierarchy	Shows the full hierarchy in the Hierarchy pane.
Move Up	Moves up one level of hierarchy to the parent.
Move Down	Moves down one level to the selected scope.
Expand By Levels	Allows expansion by multiple levels with a single action.
Expand All	Expands the entire hierarchy at once. There may be a delay getting the hierarchy from the simulation when working interactively.
Collapse Parent	Collapses the parent scope.
Collapse All	Collapses all expanded scopes.
Select By Levels	Allows you to select scopes by levels. You can select more than one level at a time.
Select All	Selects all that are visible in the hierarchy (does not implicitly expand).
Add Dump	Dumps the values of signals and scopes into the VPD file.

Command	Description
Dump	Dumps the values of signals and scopes recursively into the VPD file.
Show External Functions	Shows user-defined external (PLI, DPI, DirectC) functions.

Data Pane CSM

Command	Description
Copy	Copies the selected object or string to the clipboard buffer.
Show Source	Displays source code in the Source view for the selected object.
Show Schematic	Displays a design schematic for the selected object.
Show Path Schematic	Displays a path schematic for the selected object.
Show Back Trace Schematic	Displays a back trace schematic of the selected object.
Show Memory	Displays the contents of the memory variable in the Memory view.
Add To Waves	Displays the selected signal or signals in the Wave view.
Add To Lists	Displays the selected signal or signals in the List view.
Add To Groups	Adds objects to a signal group (new or existing per your selection) in the Signal pane.
Add To Watches	Adds objects to the last active Watch pane.
Show Definition	Locates the definition of interface or Modport Port in the Hierarchy pane.
Expand All	Expands the entire hierarchy at once. There may be a delay getting the hierarchy from the simulation when working interactively.
Collapse All	Collapses all expanded scopes.
Select All	Selects all that are visible in the hierarchy (does not implicitly expand).
Show Value Annotation	Annotates signal values for selected items.

Command	Description
Set Radix	Sets the notation of the selected signals to the selected radix.
Set Bus	Collects a group of signals for display as if they were a bus.
Set Expression	Opens the Bus/Expression dialog box.
Highlight Item	Highlights the currently selected objects in the chosen color.
Delete All Breakpoints	Deletes all breakpoints.
Set Breakpoint	Sets breakpoint on the selected signal.
Add Dump	Dumps the values of signals and scopes into the VPD file.
Dump	Dumps the values of signals and scopes recursively into the VPD file.
Force Value	<p>Force to 0 - Forces the values of the selected signal/variable to 0.</p> <p>Force to 1 - Forces the values of the selected signal/variable to 1.</p> <p>Force to x - Forces the values of the selected signal/variable to x.</p> <p>Force Release - Release the forces of the selected signal/variable.</p> <p>Set Force.. - Open the force dialog.</p>

Source View CSM

Command	Description
Copy	Copies the selected object or string to the clipboard buffer.
Follow Signal	Follows the selected signal in the source code.
Move Up	Moves up one level of hierarchy to the parent level.
Move Down	Moves down one level to definition of the selected scope or cell.
Show	Shows the selected object's definition, current scope, assertion, unit binding, entity, architecture, macro, or macro definition.
Show Schematic	Displays a design schematic for the selected object.
Show Path Schematic	Displays a path schematic for the selected object.
Show Back Trace Schematic	Displays the back trace schematic of the selected object.
Show Memory	Displays the contents of the memory variable in the Memory view.
Add To Waves	Displays the selected signal or signals in the Wave view.
Add To Lists	Displays the selected signal or signals in the List view.
Add To Groups	Adds objects to a signal group (new or existing per your selection) in the Signal pane.
Add To Watches	Adds objects to the last active Watch pane.
Expand All	Expands the entire hierarchy at once. There may be a delay getting the hierarchy from the simulation when working interactively.
Collapse All	Collapses all expanded scopes.
Edit Source	Opens the source code in the editor you select in the Application Preferences dialog box.
Edit Parent	Edits the source file of the parents instance.
Annotate Values	Annotates the signal values for selected scope.
Set Radix	Sets the notation of the selected signals to the selected radix.
Trace Drivers	Traces drivers for the selected signal.

Command	Description
Trace Loads	Traces loads on the selected signal.
Drivers/Loads	Traces/deletes drivers or loads per the selection of submenu.
Highlight item	Highlights the currently selected objects in the chosen color.
Delete All Breakpoints	Deletes all breakpoints.
Set Breakpoint	Sets breakpoint on the selected signal.
Run to Cursor	Runs and stops where the cursor is positioned.
Back Trace	Start Back Trace on selected signal.
Force Value	<p>Force to 0 - Forces the values of the selected signal/variable to 0.</p> <p>Force to 1 - Forces the values of the selected signal/variable to 1.</p> <p>Force to x - Forces the values of the selected signal/variable to x.</p> <p>Force Release - Release the forces of the selected signal/variable.</p> <p>Set Force.. - Open the force dialog.</p>
Add Dump	Dumps the values of signals and scopes into the VPD file.
Dump	Dumps the values of signals and scopes recursively into the VPD file.

Schematic View CSM

Command	Action
Copy	Copies selected object or string to the clipboard.
Paste	Pastes objects or string from the clipboard
Show Source	Displays the source code for the selected object.
Show Path Schematic	Shows path schematic for the selected object.
Show Back Trace Schematic	Displays the back trace schematic of the selected object.
Add to Waves	Adds signals to the Wave view.
Add to Lists	Displays the selected signal or signals in the List view.
Add to Groups	Adds objects to a signal group (new or existing per your selection) in the Signal pane.
Move Up	Moves up one level of hierarchy to the parent level.
Move Down	Moves down one level to definition of the selected scope or cell.
Back	Moves back in list of scopes or schematics.
Forward	Moves forward in list of scopes or schematics.
Selection Tool	Changes mouse cursor to an arrow to select objects.
Zoom In Tool	Changes mouse cursor to a zoom-in tool. You need to select the object to view a magnified image.
Zoom Out Tool	Changes mouse cursor to a zoom-out tool. You need to select the object to view a small image.
Pan Tool	Changes the mouse cursor to a hand used to pan the object view in both dimensions.
Zoom	Zooms in/out/full per the selection.
Annotate Values	Annotates the signal values for selected scope.
Set Radix	Sets the notation of the selected signals to the selected radix.
Highlight Item	Highlights the currently selected objects in the chosen color.
Trace Drivers	Shows drivers of the selected signal.
Trace Loads	Shows loads of the selected signal.
Drivers/Loads	Shows the drivers/loads in the next, previous instance or in the current instance, deletes all or the selected driver/load per your selection.

Trace X	Highlights all signals/cell, which could cause the selected signal to be of X value.
Add Dump . . .	Dumps the values of signals and scopes into the VPD file.
Dump	Dumps the values of signals and scopes recursively into the VPD file.
Force Value	<p>Force to 0 - Forces the values of the selected signal/variable to 0.</p> <p>Force to 1 - Forces the values of the selected signal/variable to 1.</p> <p>Force to x - Forces the values of the selected signal/variable to x.</p> <p>Force Release - Release the forces of the selected signal/variable.</p> <p>Set Force.. - Open the force dialog.</p>

Wave View CSM

Command	Action
Cut	Cuts (copies and removes) the selected object into the clipboard.
Copy	Copies the selected object into the clipboard buffer.
Paste	Pastes or inserts the copied object from the clipboard.
Paste From	Inserts the object from the clipboard but gets the data from the specified database.
Delete	Removes the selected object.
Set Search Constraint	Searches for signals per the selected criteria/constraint in the sub-menu, any edge, rising, falling, failure, success, match, mismatch, signal with X value, or any other specific value etc.
Signal Value	Opens the Value Search dialog box and searches for specified signal value.
Search Backward	Searches for previous match based on search constraint settings.
Search Forward	Searches for next match based on search constraint settings.

Zoom	Zooms objects as per the selection in the sub-menu.
Create Marker	Creates a marker where you click in the Wave view.
Markers	Opens the Markers dialog box.
Go To Marker	Goes to the selected marker.
Delete Marker	Deletes the selected marker.
Move Marker	Moves the marker to the selected point.
Expand Time	Expands the data cycle data at the time.
Collapse Time	Collapses the data cycle data at time.
Collapse All	Collapses the expanded delta cycle data.

Signal Pane CSM

Only those options are explained, which are not included in the description of “[Signal Menu](#)” on page 14.

Command	Action
Cut	Cuts (copies and removes) the selected object into the clipboard.
Copy	Copies the selected object into the clipboard buffer.
Past	Pastes or inserts the copied object from the clipboard.
Paste From	Inserts the object from the clipboard but gets the data from the specified database.
Delete	Removes the selected object.
Rename	Renames the selected signal group.
Sort In Group	Sorts all signals in this group in ascending, descending, or declaration order.
Associate With	Associates signal group with database.
Highlight Item	Highlights the currently selected objects in the selected color.
Trace Drivers	Traces drivers for the selected signal.
Trace Loads	Traces loads for the selected signal.

Set Draw Style Scheme Sets a draw style for the selected signal.

List View CSM

The options in the List view CSM are similar to the ones available in the CSM of the Signal pane.

Driver Pane CSM

Command	Action
Trace Drivers	Shows drivers of the selected signal.
Trace Loads	Shows loads of the selected signal.
Show Bit Driver Info	Shows bit driver information.
Show Source	Removes all information from the driver pane.
Show Schematic	Shows schematic for the selected object.
Show Path Schematic	Shows path schematic of the selected object.
Show Back Trace Schematic	Shows the back trace schematic of the selected object.
Add To Waves	Adds the trace information to the Wave view.
Add To Lists	Adds the signals to the List view.
Add To Groups	Adds the signal in the selected group or creates a new group.
Highlight Item	Advances the simulation to the specified time.
Add Dump	Dumps the value of signals and scopes into the VPD file.
Dump	Dumps the value of signals and scopes recursively into the VPD file.
Force Value	Force to 0 - Forces the values of the selected signal/variable to 0. Force to 1 - Forces the values of the selected signal/variable to 1. Force to x - Forces the values of the selected signal/variable to x. Force Release - Release the forces of the selected signal/variable. Set Force.. - Open the force dialog.
Copy	Copies the selected object into the clipboard.

Delete	Clears drivers/loads in this window.
Delete All	Clears drivers/loads in all windows.
Synchronize Source Window	Toggle button On (checked) means Source view tracks selection from the driver pane. Selecting signal highlights signal port or declaration in Source view. Selecting driving statement highlights that statement.
Synchronize Path Schematic Window	Toggle button On (checked) means Schematic view tracks selection from the driver pane. Selecting signal highlights the signal. Selecting driving statement shows design with scope selected.

Watch Pane CSM

The options in the CSM of Watch pane is similar to other windows and has been already explained in the previous sections.

Memory View CSM

Command	Action
Go to address	Goes to the specified address in the Memory view.
Add to Waves	Adds the signals to the Wave view.
Add to Lists	Adds the signals to the List view.
Add to Groups	Adds the signal in the selected group or creates a new group.
Set Radix	Sets the notation of the selected signal to the chosen radix.
Properties	Opens the Memory Properties dialog box where you can modify the property to display a signal in the Wave view.

Assertion Pane CSM

Command	Action
Copy	Copies selected object to the clipboard buffer.

Trace Assertion	Traces the selected assertion in the Wave view.
Assertion Attempts	Displays attempts for selected assertion.
Show Source	Displays source code of the selected object.
Add to Waves	Adds the signals to the Wave view.
Add to Lists	Adds the signals to the List view.
Add to Groups	Adds the signal in the selected group or creates a new group.
Synchronize Selection	Synchronizes selection with other views.
Delete All Breakpoints	Deletes all breakpoints.
Set Breakpoint	Sets breakpoint on the selected signal.

Keyboard Shortcuts

You can create Hotkey/Shortcut key in DVE by having the command in the `~/.synopsys_dve_usersetup.tcl` file or executing the same at the DVE command-line as follows. The Tcl file is stored in your VCS Home directory.

```
gui_set_hotkey -menu "Simulator->Setup..." -hot_key "F5" -replace
```

The `-replace` switch overrides an already existing hotkey. For more information on setting hotkeys, type 'help `gui_set_hotkey`' at the DVE command-line.

You can map hotkeys for a tcl script as well. For example, to show hierarchical signal path in the Signal view in DVE, run the following commands:

```
% gui_set_hotkey -tcl "gui_set_pref_value -category {wave_name_column} -key {fullname} -value {All}" -hot_key "Ctrl+p"
```

OR

```
% gui_set_hotkey -tcl_cmd "gui_set_pref_value -category  
{wave_name_column} -key {fullname} -value {All}" -hot_key  
"Ctrl+p"
```

To create a shortcut key binding, add a line such as the following:

```
::snpsMenu::add_hotkey_binding -menu "Edit > Find..." -  
hot_key "Shift+F"
```

This sets Shift+F to be the keyboard shortcut for the Find dialog box. Use the syntax **Menu > MenuItem** to select the menu item you want to set the key binding for.

File Command Shortcuts

Commands	Shortcut Keys
Open Database	Ctrl+O
Close Window	Ctrl+W
Load Waveform Updates	Ctrl+U

Edit Command Shortcuts

Commands	Shortcut Keys
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Delete	DEL
Synchronize Selection	Ctrl+Y
Select All	Ctrl+A
Find	Ctrl+F3
Find Next	F3
Find Prev	Shift+F3
Go to Address	Ctrl+G

Search for Signals/
Instances Ctrl+H

View Command Shortcuts

Commands	Shortcut Keys
Selection Tool	ESC
Zoom In Tool	=
Zoom Out Tool	-
Zoom > Zoom Full	F
Zoom > Zoom In	+
Zoom > Zoom Out	O
Zoom > Zoom Fit Selection	Ctrl+T
Zoom > Zoom Ft Highlight	Ctrl+Shift+T
Zoom > Zoom To Cursors	Ctrl+Q

Simulator Command Shortcuts

Commands	Shortcut Keys
Start/Continue	F5
Step/Next > Step	F11
Step/Next > Next	F10
Step/Next > Next In CBug	Ctrl+F10
Step/Next > Step In Active Thread	F12
Step/Next > Next in Active Thread	Ctrl + F9
Step/Next > Step In testbench	Ctrl+F11
Step/Next > Step Out	F9
Restart	Ctrl+F5

Signal Command Shortcuts

Commands	Shortcut Keys
Add To Waves > Recent Wave	Ctrl+4
Add To Lists > Recent	Ctrl+5
Add To Watches > Recent	Ctrl+6
Show Memory	Ctrl+7
Search Backward	Shift+F4 OR <
Search Forward	F4 OR >

Scope Command Shortcuts

Commands	Shortcut Keys
Show Source	Ctrl+1
Show Schematic	Ctrl+2
Show Path Schematic	Ctrl+3
Move Up	F8
Move Down	Shift+F8

Trace Command Shortcuts

Commands	Shortcut Keys
Trace Drivers	Ctrl+D
Trace Loads	Ctrl+L
Highlight Item > Recent Color	Ctrl+E
Highlight Item > Clear Selected	Ctrl+Shift+M
Highlight Item > Clear All	Ctrl+M

Help Command Shortcuts

Commands	Shortcut Keys
Help Contents	F1

Window Command Shortcuts

Commands	Shortcut Keys
Switch Top Level	Ctrl + S

Tcl GUI Commands Shortcuts

Commands	Shortcut Keys
Scroll Current View Left	Left
Scroll Current View Up	Up
Scroll Current View Right	Right
Scroll Current View Down	Down
Toggle Scope Navigator	Ctrl+R

Using the Command Line

Use the command line to enter DVE and Unified Command Line Interface (UCLI) commands. The table below describes some of these commands.

Note:

For more information on the list of DVE Tcl commands or DVE GUI commands, type `help <command_name>` in the DVE Console pane.

Commands marked with an asterisk (*) are UCLI commands.

Command	Description
<code>open_db</code>	Opens a database file.
<code>close_db</code>	Closes a database file.
<code>open_file</code>	Opens file.
<code>exit</code>	Exits application.
Design Query:	
<code>show*</code>	Displays design information for a scope or nested identifier.
<code>drivers*</code>	Obtains driver information for a signal/variable.
<code>loads*</code>	Obtains load information for a signal/variable.
<code>fanin</code>	Extracts the fanin cone of the specified signal(s).
<code>search</code>	Locates design objects whose names match the name specification you provide.
Simulator:	
<code>open_sim</code>	Setup Simulator executable and arguments.
<code>start*</code>	Starts tool execution.
<code>step*</code>	Advances the tool one statement.
<code>next*</code>	Advances the tool stepping over tasks and functions.
<code>run*</code>	Advances the tool and stop.
<code>finish*</code>	Allows the tool to finish then return control back to UCLI.

restart* Restarts tool execution, and keeps the setting in the last run.

Breakpoints:

stop Adds or displays stop breakpoints.

Navigation:

scope* Gets or changes the current scope.

thread* Displays thread information or move the current thread.

stack* Displays thread information or move the call stack.

listing* Displays source text.

add_schem Shows scope in Schematic view.

add_source Shows signal/scope in Source view.

Signal/Variable/Expression:

get* Obtains the value of a signal/variable.

force* Forces or deposit a value on a signal/variable.

release* Releases a variable from the value assigned using 'force'.

call* Executes a system task or function within the tool.

sexpr* Evaluates an expression in the tool.

vbus* Creates, deletes, or display a virtual object.

add_group Adds signals to Group.

add_list Adds signals to the List view.

add_mem Adds memory to the Memory view.

add_pathschem Shows path schematic for signal(s)/ scope(s).

add_watch Adds signals to Watch view.

add_wave Adds signals to the Wave view.

delete_group Deletes signals from given group.

delete_list Deletes signals from given list view.

delete_watch Deletes signals from global watch view.

delete_wave Deletes signals from given wave view.

compare Compares signal/scopes.

view Opens, closes or lists view.

Signal Value and Memory:

dump* Creates/manipulates/closes dump value change file information.

memory* Loads/writes memory type values from/to files.

add_mem	Adds memory to the Memory view.
Session Management:	
save*	Saves simulation state into a file.
restore*	Restores simulation state saved in a file.
save_session	Saves session.
open_session	Opens session.
Help Routines:	
help	Lists basic commands. Use -all for listing all commands.
alias*	Creates an alias for a command.
unalias*	Removes one or more aliases.
config*	Displays/sets current settings for configuration variables.
Macro Control:	
do*	Evaluates a macro script.
onbreak*	Specifies script to run when a macro hits a stop-point.
onerror*	Specifies script to run when a macro encounters an error.
resume*	Resumes execution of a macro file.
pause*	Pauses execution of a macro file.
abort*	Aborts evaluation of a macro file.
status*	Displays the macro file stack.
Misc.:	
ace*	Evaluates analog simulator command.
cbug*	Debugs support for C, C++ and SystemC source files
coverage*	Evaluates coverage command(s).
power	Powers measure.
quit*	Exits application.
senv*	Displays one or all synopsys::env array elements.
setenv*	Sets the value of a system environment variable.
sn*	Displays the Specman prompt when used without arguments, and executes e code commands when they are entered as optional arguments.
tcheck*	Disables/enables timing check upon a specified instance/port at runtime.
virtual	Creates, deletes or displays a virtual object.

Index

Symbols

.size() 11-71
+vpi+1+assertions 10-7, 10-8
\$sdf_annotation 9-40
\$vcdbllog 11-63
\$VCS_HOME 10-11

A

About DVE(Help menu selection) A-23
Add Fanin/Fanout A-18
Add to Lists A-14, A-62
Add to Waves A-14, A-62
Annotate Values A-18
-assert dump_sequences 9-39
Assertion Attempts A-19
Assertion Failure Summary Pane 10-2
Assertion failures 10-2
Assertion Unit 3-4
Assertion Window 10-1
automatic step-through
 Systemc 13-60

B

Back A-17

Beginning (menu selection) A-9
binary radix 5-5
bits, displaying in waveform 5-7
Breakpoints A-12
buffer
 compressing 7-14
Building Buses 5-31
Bus Builder 5-31
buses, building 5-31

C

C1 cursor 5-6
CBug 13-57
-class 14-9
Close Database (File menu selection)
 reference A-2
Close File (File menu selection)
 reference A-3
Close Window (File menu selection)
 reference A-3
closing a VPD database 1-22
column headings
 rearranging 3-6
Commands Supported by the C Debugger 13-5
Compare A-15
compile-time options 1-3

-condition 11-55
Configuring CBug 13-35
constraint_mode 14-24
context-sensitive menus, using A-56
Continue A-11
coverage databases 1-5
cursor C1 5-54
cursor C2 5-54
cursors, inserting 5-54

D

database
 closing 1-22
 opening 1-11
–debug_all 14-3
debug_all, option 1-4
debug_pp, option 1-3
debug, option 1-3
debugging options 1-3
default symbol 7-42
Display Signal Groups A-14
Dock
 Window menu selection A-22
dock and undock windows 2-13
drag zooming 5-62
Dump Values A-12, A-63
duplicate signals, displaying 5-24
DVE 10-8, 13-57
 exiting 1-22
DVE Help (Help menu selection) A-23

E

Edit Bus A-15
Edit Parent A-18
Edit Source A-18
Edit User-Defined Radices A-16
End (View>Go To menu selection)

reference A-9
Execute Tcl Script (File menu selection)
 reference A-3
Exit (File menu selection)
 reference A-3
exitingDVE 1-22
Expand Path A-18

F

filee, required 1-4
Find (Edit menu selection)
 reference A-5
flip-flop schematic symbols 7-43
force signal values 3-19
force values 3-19
Forward A-17

G

Go To (View menu selection)
 reference A-9
gui, option 1-6

H

hexadecimal radix 5-5
hyperlink 4-6
hyperlink include file 4-6

I

icons
 Verilog Named Begin 3-4
 Verilog Named Fork 3-4
 Verilog Task 3-4
immediate assertion support 10-7
include file as hyperlink 4-6
inserting new markers 5-57
integers, storing 5-7

interactive options 1-6
interval between cursors 5-54
inverter
 compressing 7-14

L

List Window A-21
 panes 6-2
 save format 6-6
 set markers 6-3
 set signal properties 6-4
 using 6-1
 view simulation data 6-3
Load Session (File menu selection)
 reference A-3
loading a VPD database 1-11
lower timescale 5-41

M

Main Window
 example 2-3
 using 2-1
Markers dialog box 5-55, 5-58
markers, inserting 5-54
Menu Bar
 reference A-2
 using 2-20
Move Down to Definition A-17
Move Up to Parent A-17

N

Name column (signal pane) 5-6
new markers, inserting 5-57

O

-object_id 11-54
offset 15-20

Open Database
 Toolbar icon A-41
Open Database (File menu selection)
 reference A-2
Open Database dialog box 1-11
Open File (File menu selection)
 reference A-3
opening
 database 1-11
OVA library 1-4

P

path schematic
 delete objects, select objects 7-23
PLI 10-8
post-processing, options 1-5

Q

quitting DVE 1-22

R

radix
 binary 5-5
 hexadecimal 5-5
rand_mode 14-24
rearranging column headings 3-6
Reload Database A-3
rename signals in wave view 5-8
Required Files 1-4
Restore State A-12
RTL schematic 7-35
Run A-11
Run Example (Help menu selection) A-23

S

save preferences 7-21

Save Session (File menu selection)
 reference A-3

Save State A-12

scalar signals 5-5

schematic views 7-2

Scopes
 example 3-3

script, running 1-14

Search Backward A-15

Search Forward A-15

searching in the Waveform pane 5-64

Set Expression A-15

Set Precision text field A-52

Set Radix A-16, A-62

Set Search Constant A-15

set signal properties 6-4

Set Time text field A-52

Setup A-11

Shift Time A-16

Show A-18

Show Comparison Info A-15

show In class browser 14-28

show In solver view 14-31

Show Memory A-14

show -object 11-49

Show Path Schematic A-17

show -randomize 14-24

Show Relation 14-28

Show Schematic A-17

Show Source A-17

show value annotation 7-21

Signal Menu A-14

Signal Properties A-16

signal renaming 5-8

signals
 scalar 5-5
 vector 5-5

Simple logic Schematic Symbols 7-44

Source Pane
 Toolbar icon 1-31, 2-9, 2-10, A-46, A-47

Standard Template Library 13-60

std randomize 14-24

Step A-11

steptotblib 11-61

step-out feature
 using 13-59

STL 13-60

Stop A-11
 stop at port 9-15

stop -file -line -skip 14-5

stop -in -object_id 11-54

stop -solver 14-5

stop -solver -once 14-5

stop -solver -serial 14-5

support for immediate assertion 10-7

Supported Platforms for Debugging with CBug
 13-43

sva_vpi+1assertions.tcl 10-11

SystemC
 automatic step-throug 13-60

SystemVerilog 10-7

T

Terminate A-12

time data type 5-7

Time... (View>Go To menu selection)
 reference A-9

Tips button 5-58

TLM 13-57

TLM-2.0 13-57

Toolbar
 using 2-20

Trace Drivers A-19, A-62

Trace Loads A-19, A-62

Trace Value Change 9-13

U

UCLI 10-8, 13-57, 14-4
ucli, option 1-6
Undock A-22
undock windows 2-13
upper timescale 5-41
Using Breakpoints with CDebugger 13-16
Using the C, C++, and SystemC Debugger
13-1, 14-1

visualization of driving signals 9-9

VPD file 1-4
closing 1-22
loading 1-11
VPI 9-39, 10-8
vpi_control 10-7
vpi_handle_by_name 10-7
vpi_iterate 10-7
vpi_register_assertion_cb 10-8

V

value annotation in schematic view 7-21
Value column (Signal pane) 5-6
value transitions 5-40
vector signals 5-5
Verilog Named Begin icon 3-4
Verilog Named Fork icon 3-4
Verilog Task icon 3-4
View menu, reference A-7

W

Waveform pane
cursors 5-54
Waveform pane, using 5-1
Window Menu
reference A-21

Z

zooming
by dragging 5-62