Developer Magic™: ProDev™ WorkShop Overview

Document Number 007-2582-003

CONTRIBUTORS

Written and illustrated by John C. Stearns
Production by Laura Cooper
Engineering contributions by Lia Adams, Jim Ambras, Trevor Bechtel, Wes Embry, Alan Foster, Christine Hanna, David Henke, Marty Itzkowitz, Mahadevan Iyer, Lisa Kvarda, Stuart Liroff, Song Liang, Allan McNaughton, Michey Mehta, Sudhir Mohan, Ashok Mouli, Anil Pal, Andrew Palay, Tom Quiggle, Kim Rachmeler, Jack Repenning, Paul Sanville, Ravi Shankar, John Templeton, Michele Chambers Turner, Shankar Unni, Mike Yang, Jun Yu, and Doug Young.

Developer Magic™: ProDev™ WorkShop Overview
Document Number 007-2582-003

Contents

List of Figures

List of Tables

About This Guide

This manual is an introduction and overview of ProDev WorkShop, Release 2.5.1 It contains the following:

- The body of the manual, “ProDev WorkShop Overview,” describes the major tools in the ProDev WorkShop toolkit. It provides a user model for each tool, highlights some major features, and provides pointers to the user guides where you can get detailed information on the tools.
- Appendix A, “Using Graphical Views,” describes the features and operation of graphical views in the ProDev WorkShop toolkit.
- Appendix B, “Customizing ProDev WorkShop Tools,” describes features and resources available for customizing the look and operation of ProDev WorkShop tools.
- A glossary of commonly used terms in the ProDev WorkShop toolkit.

ProDev WorkShop Overview

Welcome to ProDev WorkShop, a major part of the Developer Magic™ software development environment. ProDev WorkShop is a software toolset for the development of C, C++, Ada, and Fortran applications. These powerful, highly visual tools help you understand your program's structure and operation so that you can diagnose very difficult, traditionally time-consuming problems in a short amount of time. With them, you can develop applications for the entire Silicon Graphics® product line, from Indy™ to POWER Onyx™ workstations.

Note: In the past, the software development environment was called CASEVision™; that name has been replaced by Developer Magic. In addition to ProDev WorkShop, the Developer Magic environment includes ProMPF—a special module for multi-process Fortran programming—and IDO (IRIX Development Option)—the base compiler and libraries. Some of the documentation may still use the CASEVision name; those documents will be updated soon.

The ProDev WorkShop toolset provides:

- **Comprehensive control over the debugging process**—You can set simple breakpoints with the click of a mouse button or define complex conditions for your traps. ProDev WorkShop's fast data watch points with kernel support are especially adept at tracking memory corruption problems.
- **Visual debugging environment for examining data in your active program**—ProDev WorkShop provides convenient, graphical views of variables, expressions, large arrays, and data structures. If you prefer a tty-style interface, you can always dump values directly using WorkShop's Debugger command line.
- **Powerful static analysis for understanding your program**—You can view the structure of your program and relationships such as call trees, function lists, class hierarchies, and file dependencies. And you can get this information whether or not the program can be compiled.

- **The ability to collect performance and coverage information during test runs**—ProDev WorkShop's Performance Analyzer lets you see where your program spends its time and pinpoint performance bugs, including those due to memory problems. The Tester tool shows you which source lines and basic blocks are covered in your tests.
- **Convenient recompiling from within the ProDev WorkShop environment**—WorkShop's standard build tools let you view file dependencies and compiler requirements and fix compile errors conveniently.
- **Quick recompiles for simple changes**—The Fix and Continue tool lets you make simple changes without having to go through a major recompile and relinking, dramatically reducing the number of edit-compile-debug cycles.
- **Ability to analyze structures and relationships in C++ and Ada code**—The Browser provides global graphical and textual views of relationships between language-specific entities, including inheritance, containment, and interactions.
- **Specialized debugging for X/Motif® applications**—The X/Motif Analyzer lets you solve the special problems in X/Motif application development. You can look at object data, set breakpoints at the object or X protocol level, trace X and widget events, and tune performance.
- **Rapid application development**—The RapidApp tool lets you create graphical interfaces for C++ applications quickly and easily. RapidApp lets you build graphical interfaces by dragging and dropping interface elements (based on IRIX IM (X/Motif) widgets and IRIS ViewKit™-style components) onto a template window.

This overview gives you a broad exposure to the ProDev WorkShop toolset as well as pointers to the documentation for getting detailed information. The overview is organized as follows:

- "Using the ProDev WorkShop Debugger"
- "Navigating Through Code With the Static Analyzer and Browser"
- "Pinpointing Performance Problems With the Performance Analyzer"
- "Determining the Thoroughness of Test Coverage With Tester"
- "Recompiling Within the ProDev WorkShop Environment With Build Manager"

- "Making Quick Changes With Fix and Continue"
- "Debugging X/Motif Programs"
- "Building Application Interfaces With RapidApp"

In addition to the ProDev WorkShop tools, you can separately purchase:

- **Developer Magic Pro MPF**—a visual code parallelization tool used with the Power Fortran Accelerator™ to help balance parallel loops in Fortran applications
- **Developer Magic ClearCase**™—a toolset for version control, configuration management, and process control for software organizations

  Note: If you use ClearCase, SCCS, or RCS, you can check source files directly into or out of ProDev WorkShop and MegaDev.
- **Developer Magic Tracker**—an application builder for creating change control and change tracking systems. It can be integrated with ClearCase.

Using the ProDev WorkShop Debugger

The Debugger is a UNIX® source-level debugging tool that provides special windows (views) for displaying program data and execution state as the program executes. The Debugger lets you set various types of breakpoints and watch points where you can conveniently view data such as variables, expressions, structures, large arrays, call stacks, and machine-level values. The WorkShop Debugger goes far beyond the capabilities of *dbx*. It includes fast data watchpoints and other types of traps; graphical views for displaying local variables, source-level expressions, array variables, and data structures; and debugging at the machine level.

Debugger User Model

All WorkShop activities can be accessed from the Main View window, which is illustrated in Figure 1.

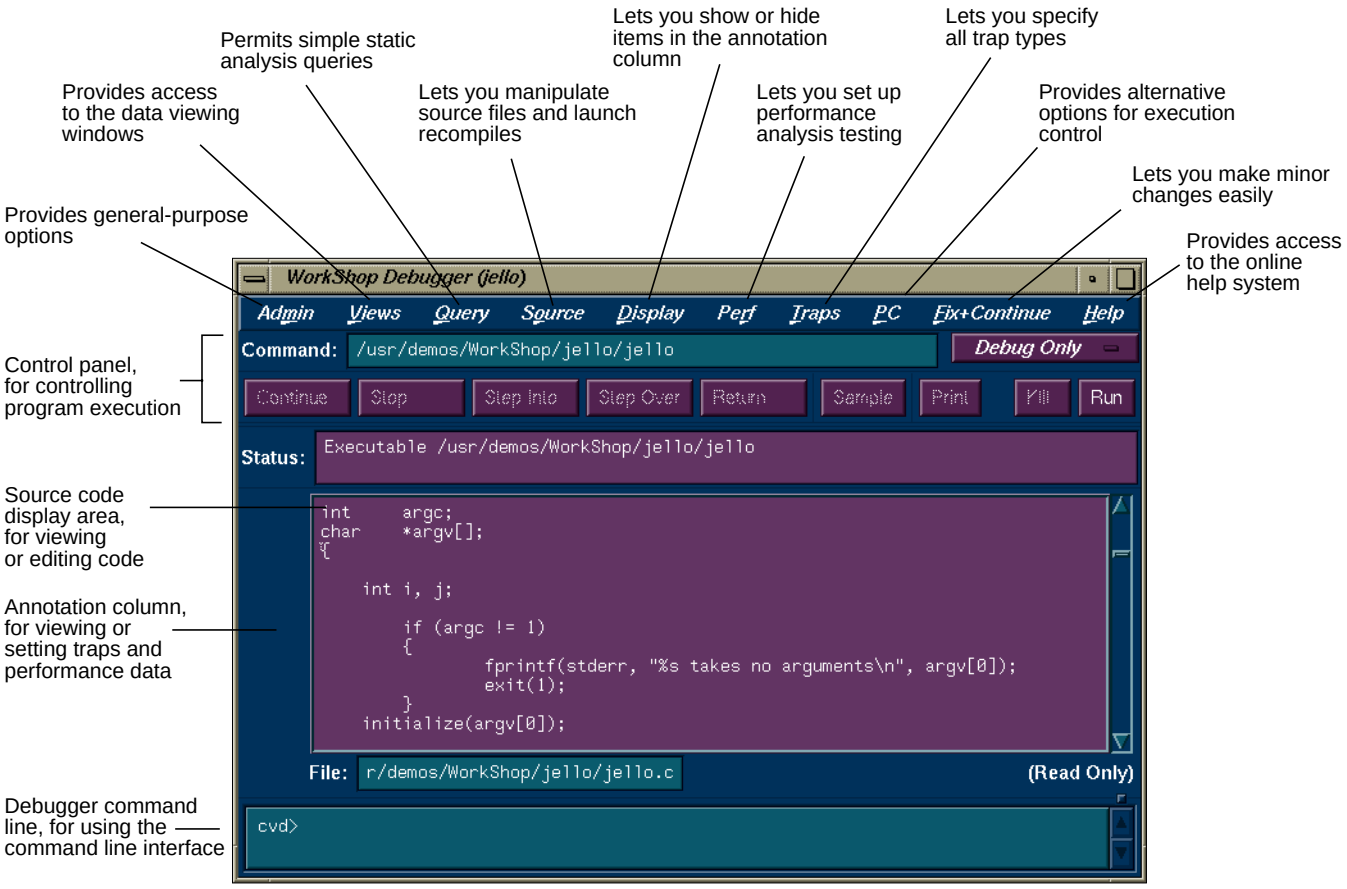


Figure 1 Major Areas of the Main View Window

The basic model for using the Debugger is to:

1. Invoke the Debugger by typing:

   `cvd [-pid` *pid*`] [-host` *host*`] [`*executable* `[`*corefile*`]] [&]`

   The **`-pid`** option lets you attach the Debugger to a running process. You can use this to determine why a live process is in an infinite loop or is otherwise hung.

The argument *executable* is the name of the executable file for the process you want to run. It is optional; you can invoke the Debugger first and specify the executable later.

The *corefile* option lets you invoke the Debugger and specify a core file (with its executable) to try to determine why a program crashed.

The **`-host`** option lets you specify a remote host on which the target executable will be run; the Debugger runs locally. This option is useful if:

- you don't want the Debugger windows to interfere with the application you are debugging.
- you are supporting an application remotely.
- you don't want to use the Debugger on the target system for another reason.

2. Set stop traps, that is, breakpoints, in the source code.

   Simple traps are set by clicking the left mouse button in the annotation column to the left of the source code display or by using the Traps menu. More complex traps, including watch points, can be set and managed from the Trap Manager, Signal Panel, and Syscall Panel, which can be accessed from the Views menu. You can also set traps by typing them at the Debugger command line in Main View. You can stop a process at any time by clicking the *Stop* button in the Main View control area.

3. Start the program by clicking the *Run* button in Main View.

4. When the process stops at a trap or other stopping point of interest, you can examine the data in the Debugger view windows (accessed from the Views menu).

   You can display view windows at any time; they update automatically each time the program stops. Figure 2 shows four typical Debugger views and indicates how you access them from the Views menu.

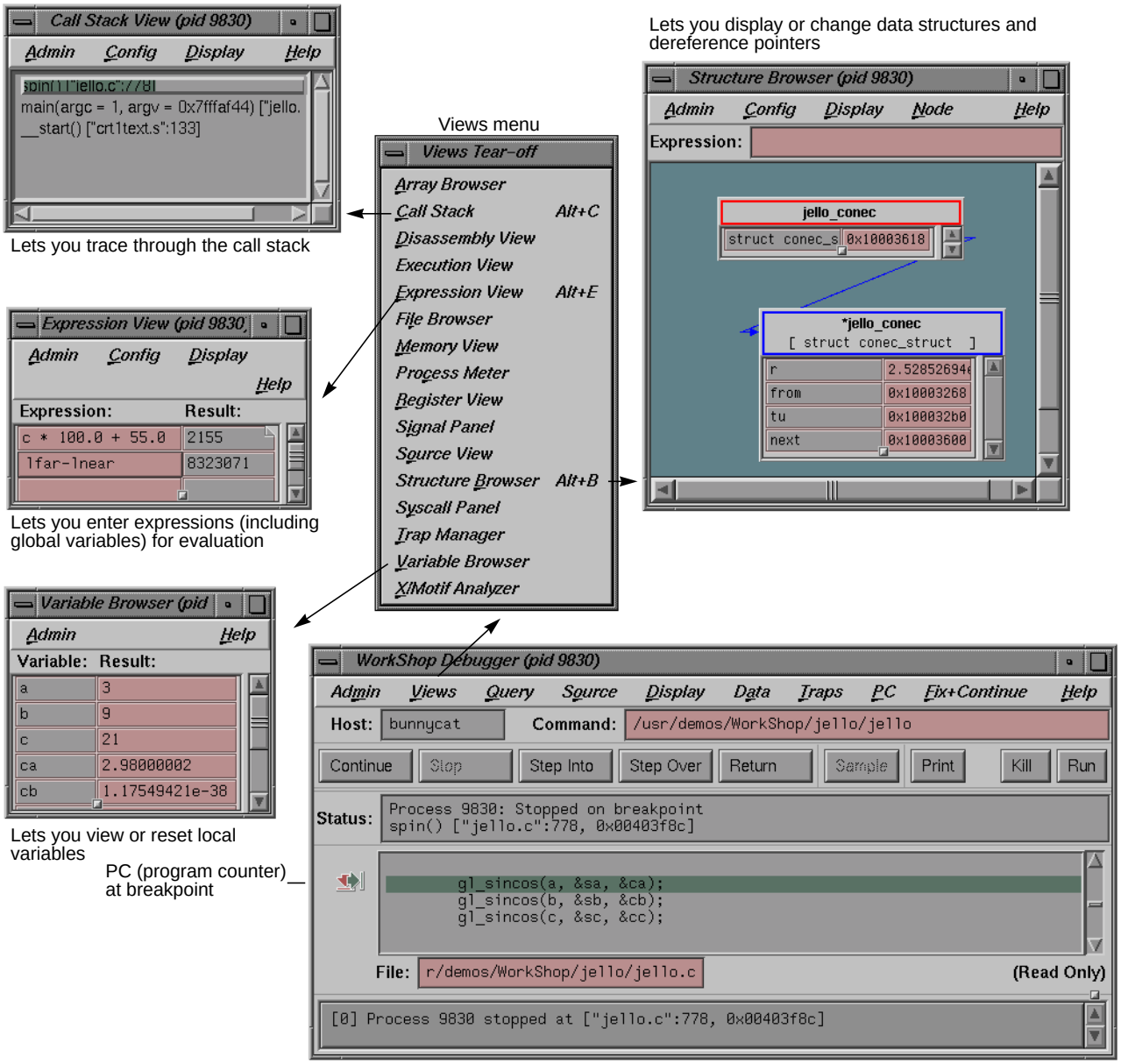


Figure 2 Typical Debugger Views Accessible at a Breakpoint

Figure 3 shows the Array Visualizer, a powerful view for examining data in arrays of up to 100 x 100 elements. You can look for problem areas in a 3D rendering of the array, click on the area of interest, and view the numerical values in a spreadsheet format. In Figure 3, the hue option has been set so that the values appear in a color spectrum from blue (lowest) to red (highest) with out-of-range anomalies appearing in gray. Note the high point coming out of the 3D image; it demonstrates how anomalies in large arrays stand out.

If you need to debug your program at the machine level, you can use Register View, Disassembly View, and Memory View, as shown in Figure 4. These are accessed from the Views menu in the Debugger Main View as well.

5. Use the control panel options in Main View to continue execution (see Figure 1).

   From any breakpoint, you have these options:

   - The *Continue* button runs the program until the next breakpoint.
   - The "Continue To" selection in the PC menu proceeds to a specified source line. Placing the cursor in a line specifies it.
   - The "Jump To" selection in the PC menu goes to a specified line (by the cursor), skipping over any intermediate code.
   - The *Step Into* button continues execution by one step or a number specified by holding down the right mouse button over the *Step Into* button and selecting the number from the dialog box. The process then continues the specified number of source lines and enters any called functions.
   - The *Step Over* button similarly proceeds a specified number of lines but executes intermediate functions without stepping into them.
   - The *Return* button executes the remaining instructions in a function and stops on return from that function.

6. Check out the source code that needs to be fixed.

   If you find a bug and are using an integrated source control program such as ClearCase, RCS, or SCCS, you can check out the source code from Main View (or Source View, an alternate editing window).

   Choose "Check Out" from the Versioning submenu in the Source menu.

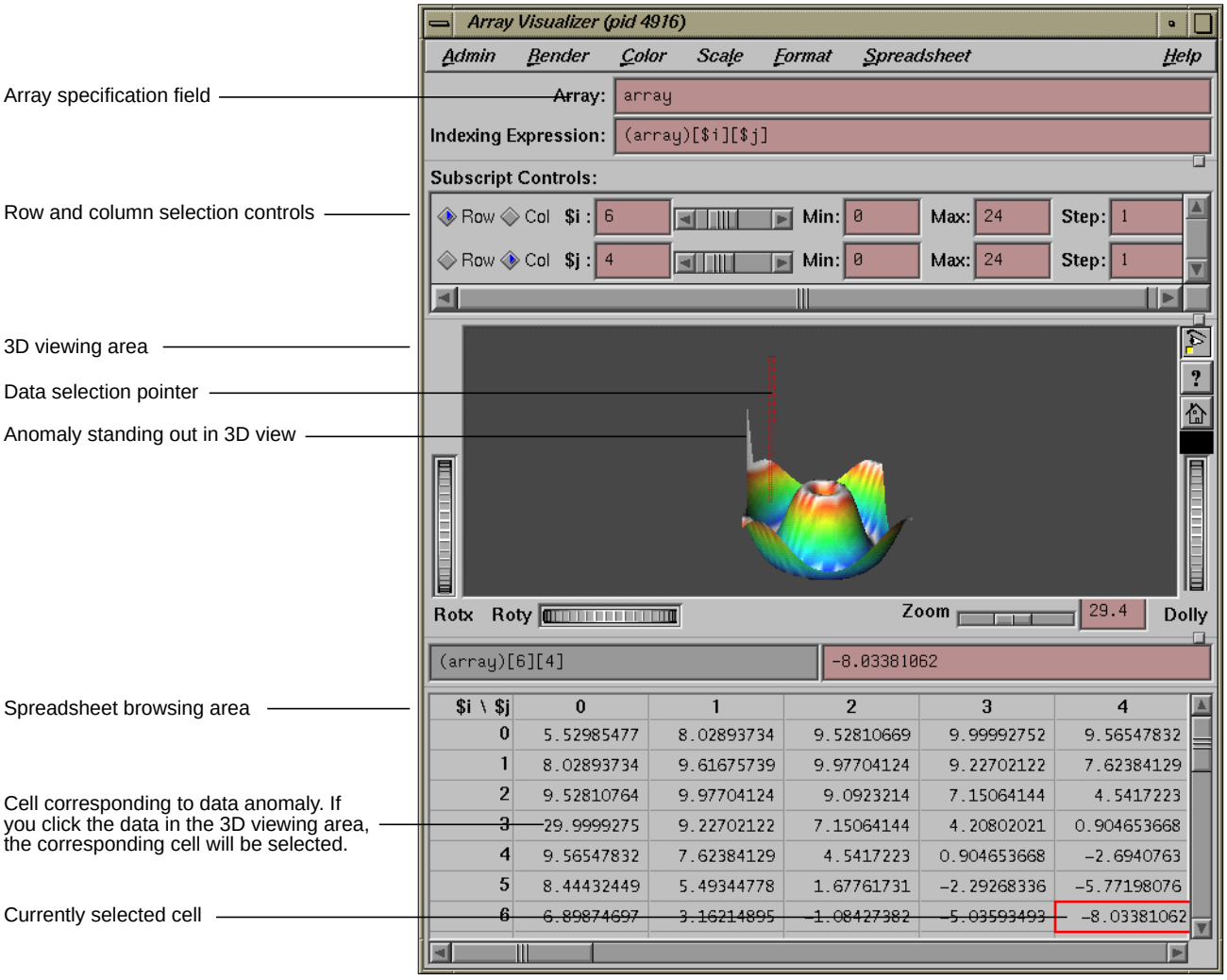


Figure 3 Array Visualizer

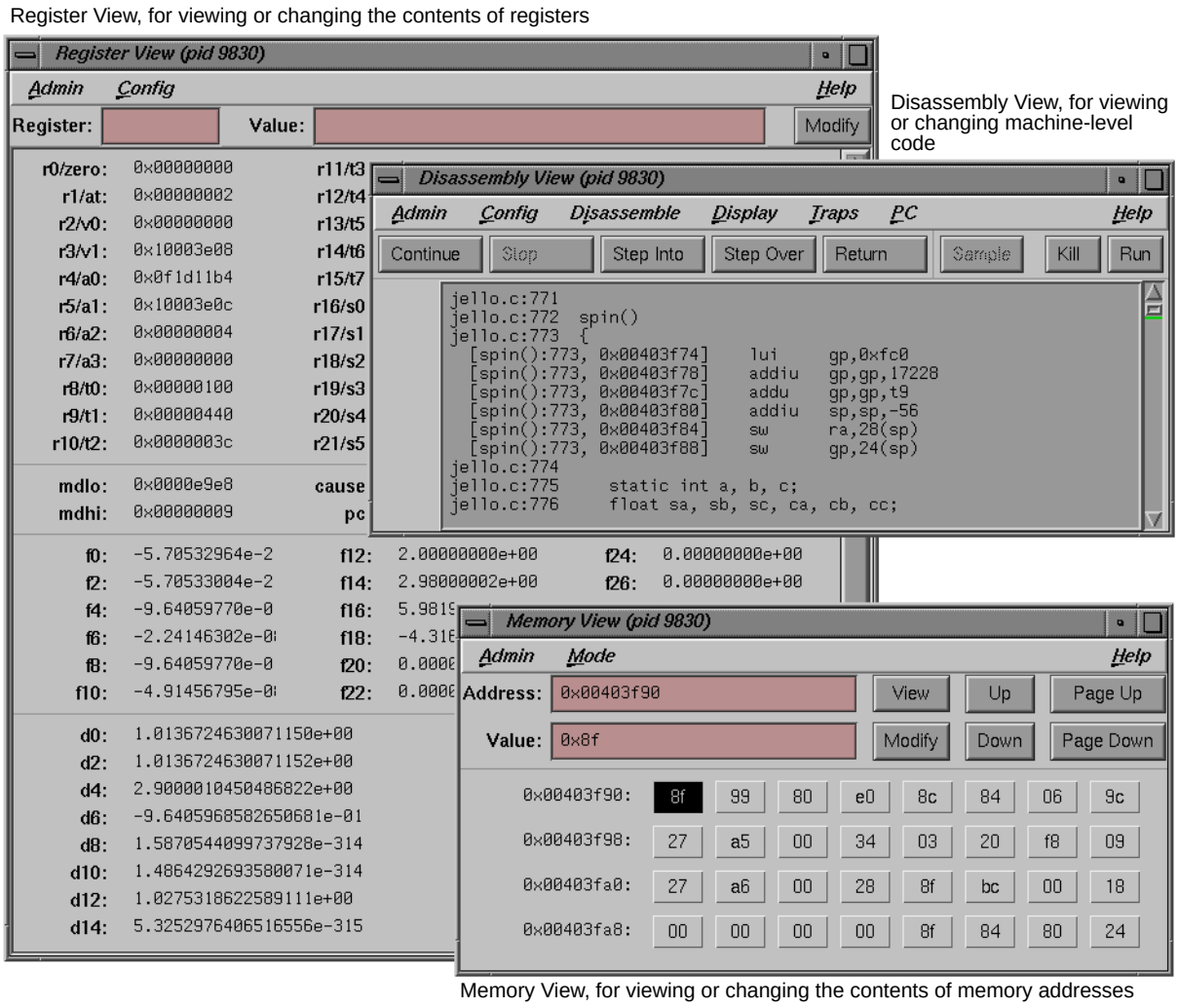


Figure 4 Machine-Level Debugger Views

7. Fix any problems in your code using the source code display area in Main View, Source View, or the editor of your choice.

   Both Main View and Source View let you do simple editing and annotate the code with trap indicators. Source View also lets you display test data from the Performance Analyzer and Tester in the annotation column. If you prefer to view source code in a text editor other than Source View, add the line

   `*editorCommand:` *editor*

   to your *.Xdefaults* file, where *editor* is the command for the editor you wish to use.

8. Recompile using Build Manager.

   Build Manager has two windows: Build View and Build Analyzer. Build View lets you compile, view compile error lists, and access the offending code in Source View or an editor of your choice. Build Analyzer lets you view build dependencies and recompilation requirements, and access source files. Build View uses the UNIX *make* facility as its default build software. Although Build Analyzer determines dependencies using *make*, you can substitute the build software of your choice, any *make* that runs on Silicon Graphics platforms.

Where to Find Debugger Information

To find out more about the Debugger, refer to Table 1.

Table 1 Where to Find Debugger Information in the *Developer Magic: Debugger User's Guide*

Topic	See ...
General Debugger information	Chapter 1, "Getting Started with the WorkShop Debugger"
Debugger tutorial	Chapter 3, "A Short Debugger Tutorial"
Debugger interaction with source files	Chapter 2, "Managing Source Files"
Managing windows while performing multiple tasks	"Project Session Management Windows" on page 224
Comprehensive trap information	Chapter 4, "Setting Traps"

Table 1 (continued) Where to Find Debugger Information in the *Developer Magic: Debugger User's Guide*

Topic	See ...
Controlling execution in a process (stepping, jumping, etc.)	Chapter 5, "Controlling Process Execution"
Examining Debugger data in general at the source level	Chapter 6, "Examining Debugger Data"
Tracing through the call stack	"Tracing Through Call Stack View" on page 85
Entering expressions to be evaluated at stopping points	"Evaluating Expressions" on page 88
Viewing or changing the values of variables	"Variable Browser" on page 260
Examining data in arrays using the 3D or spreadsheet format	"Array Browser" on page 232
Determining the data structures of variables	"Structure Browser" on page 249
Using the Debugger command line	"Debugger Command Line" on page 289
Examining debugger data at the machine level	"Machine-level Debugging Windows" on page 262
Using the debugger to trap memory allocation problems	Chapter 8, "Detecting Heap Corruption"
Debugging multiprocess programs	"Multiple Process Debugging Windows" on page 273

Navigating Through Code With the Static Analyzer and Browser

The ProDev WorkShop Static Analyzer is a source code analysis and navigation tool for analyzing source code written in C, C++, Fortran, or Ada (with purchase of ProDev Ada **only**). (The Browser has additional features for Ada and C++ and is described in "Browser User Model" on page 16.) The Static Analyzer shows you the code's structure (graphically or in text format) including function calls, definitions of variables, file dependencies, macro locations, class hierarchies, file dependencies, and other structural details for understanding your code. You can also make specific queries, such as showing where a function is used. You can even analyze programs that don't compile, a particularly nice feature for those porting code.

The Static Analyzer works by reading through source code files that you specify and creating a database of program elements such as functions, files, classes, methods, packages, and their relationships. The main Static Analyzer window with a typical call graph is illustrated in Figure 5.

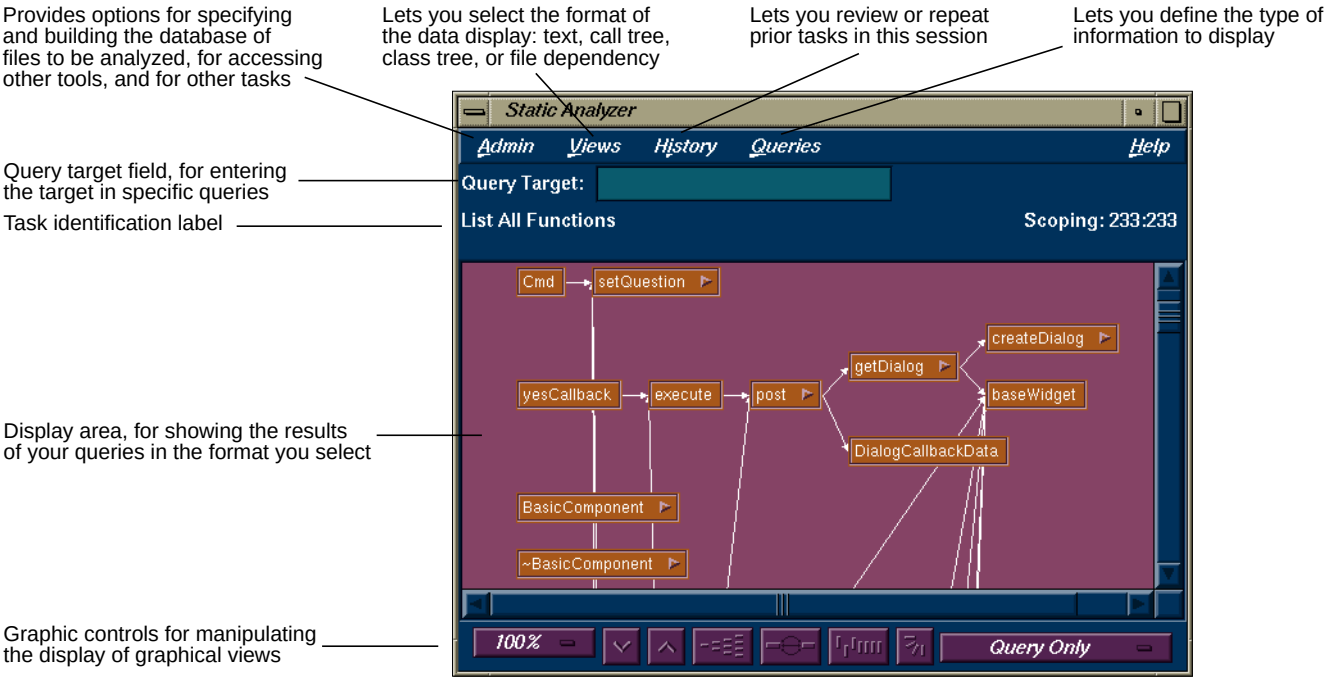


Figure 5 Main Static Analyzer Window

Static Analyzer User Model

Follow these steps for using the Static Analyzer:

1. Invoke the Static Analyzer, either by typing **`cvstatic`** or by selecting "Static Analyzer" from the Launch submenu in any ProDev WorkShop Admin menu (preferably from the directory where your source is located).

2. Decide which files are to be analyzed.

   You designate which files are to be analyzed in a special file called a *fileset*. A fileset is a regular ASCII file with a format of one entry per line, each line separated from the next by a carriage return. The entries can be regular expressions, filenames, or included directories preceded by the designator `-I`.

   To specify a fileset, you can

   - create the fileset manually using a text editor
   - use the Fileset Editor, which is accessed from the Admin menu in the Static Analyzer window
   - let the Static Analyzer create the fileset automatically at startup by defaulting to the files in the current directory that match the expression `*.[cCfF]`
   - let the Static Analyzer create the fileset automatically at startup from the command line by typing **`cvstatic`** with the **`-executable`** flag and designating an executable
   - use the compiler to create a fileset (and database) by adding the **`-sa,<`***dbdirectory***`>`** option to your makefile

   Note: Many programs are so big that a query covering the entire scope is useless due to its size and complexity. There are two ways to keep the scope of your analysis at a manageable size: (1) Limit the number of files to be analyzed or (2) avoid queries that begin with "List All ...".

3. Decide how you are going to build the database.

   Before you can specify a fileset, you must decide how you are going to build the database. You can choose to create the database in *scanner* mode (the default), which is fast but not sensitive to any specific programming language, or in *parser* mode, which uses the compiler and is slower but more thorough. Use scanner mode for large programs or

for programs that do not compile. Scanner mode is particularly suited to porting situations. Parser mode is better when you have code that compiles and you need to determine language-specific relationships, particularly in Fortran, Ada, and C++.

4. Build the database.

5. Perform your queries.

   Queries are selected from the Queries menu in the Static Analyzer. They fall into 13 categories, as shown in Figure 6. Remember that the "List All ..." queries can produce overwhelming results for large programs.

6. View (and save) the results.

   The Static Analyzer has four ways of presenting data, which are selected from the Views menu:

   - "Text View" displays query results in a text format. In addition to listing the queried items, it indicates the source filename and line number, and includes the actual source line.
   - "Call Tree View" applies to function queries. It presents the data in a graphical format with *nodes* (rectangles) representing functions and *arcs* (arrows) representing calls to functions.
   - "Class Tree View" applies to C++ class queries. It presents a class inheritance tree with nodes representing classes and arcs representing parent-child class relationships.
   - "File Dependency View" applies to file queries. It presents a graph, with nodes representing files and arcs representing include relationships.

   If you want to save a query in a graphical view, you can save a PostScript® version by selecting "Save Query..." from the Admin menu and print it out at your leisure.

7. Access the source code.

   Double-clicking any node in a graph or item in Text View brings up the Source View window containing the corresponding source code. Double-clicking any arc (arrow) displays Source View with the corresponding call site or file inclusion.

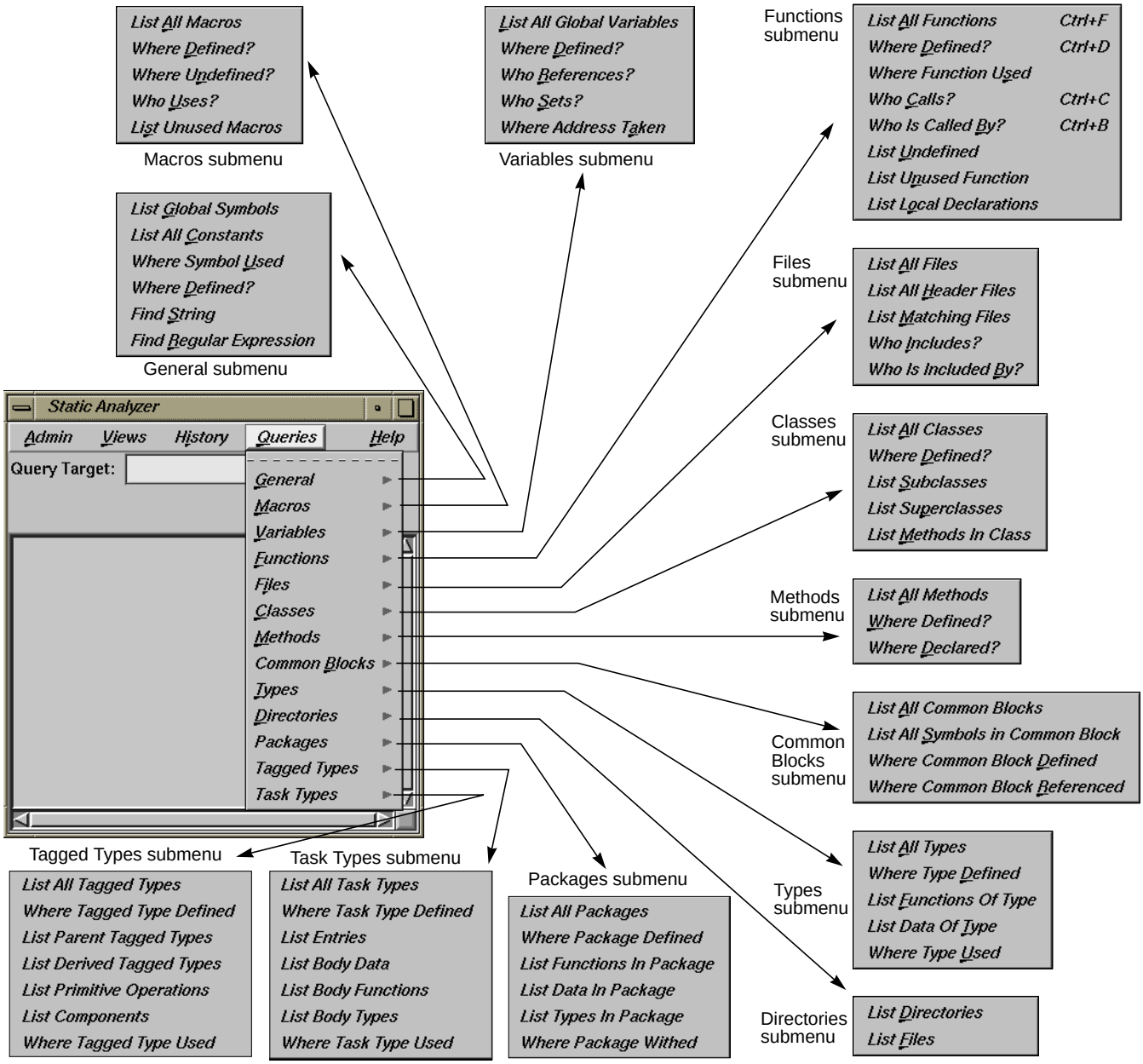


Figure 6 Static Analyzer Queries Menu with Submenus

Where to Find Static Analyzer Information

To find out more about the Static Analyzer, refer to Table 2.

Table 2 Where to Find Static Analyzer Information in the *Developer Magic: Static Analyzer and Browser User's Guide*

Topic	See ...
General Static Analyzer description	Chapter 1, "Introduction to the WorkShop Static Analyzer"
Static Analyzer tutorial	Chapter 2, "A Sample Session With the Static Analyzer"
Specifying a fileset	"Fileset Specifications" on page 32
Building a database using scanner mode	"Scanner Mode" on page 41
Building a database using parser mode	"Parser Mode" on page 42
Performing queries	Chapter 4, "Static Analyzer: Queries"
Static Analyzer viewing formats	Chapter 5, "Static Analyzer: Views"
Strategies for analyzing large programs	Chapter 6, "Static Analyzer: Working on Large Programming Projects"

Browser User Model

The Browser user model is similar to the Static Analyzer user model. After you build the database (which must be done in parser mode), you access the Browser by selecting "Browser" from the Static Analyzer Admin menu.

The Browser lets you display different sets of information including relationships about C++ classes and members, Ada packages, tagged types, tasks, and their members through these three views:

- Browser View—displays member and related information in an expandable, hierarchical outline format with the members of the current class, package, tagged type, or task in the left pane and related elements on the right (see Figure 7). Clicking the diamond-shaped icons next to the headings in the list hides or displays the associated information.

Like the Static Analyzer, you have numerous queries available through the Query menu. In addition, if you select an item in either of the Browser View lists and hold down the right mouse button, you can access the Queries menu specific to that type of item, that is, methods, data members, classes, and so on.

Figure 7 Browser View Window and Query Menus with C++ Data

You can create man page templates for classes, packages, tasks, or tagged types by selecting “Generate man pages...” from the Browser View Admin menu. You simply specify one or more elements, click the *Generate* button, and the Browser fills in the man page template for you. Similarly you can create web pages by selecting “Generate web pages...” from the Browser View Admin menu. See Figure 8.

- Class Graph—displays the hierarchy for the current subject in the Browser View window with nodes as subjects and arcs as relationships. Class Graph can show four types of relationships: inheritance, containment, interaction, and friends. You can display all subjects, limit the scope to those derived from the current subject, or get a butterfly view showing the immediate base and derived subjects of the current one.
- Call Graph—displays the calling relationships of methods, virtual methods, or functions selected from Browser View with options for customizing the display of the graph.

To find out more about the Browser, refer to Table 3.

Table 3 Where to Find Browser Information in the *Developer Magic: Static Analyzer and Browser User's Guide*

Topic	See ...
General Browser description	Chapter 7, “Getting Started with the Browser”
C++ Browser tutorial	Chapter 8, “Using the Browser for C++: A Sample Session”
Ada Browser tutorial	Chapter 9, “Using the Browser for Ada: A Sample Session”
Detailed reference information	Chapter 10, “The Browser Reference”
Browser View window	“Browser View Window” on page 147
Class Graph window	“Class Graph Window” on page 173
Call Graph window	“Call Graph Window” on page 176
Generating man pages	“Man Page Generation” on page 157
Generating web pages	“Web Page Generation” on page 159

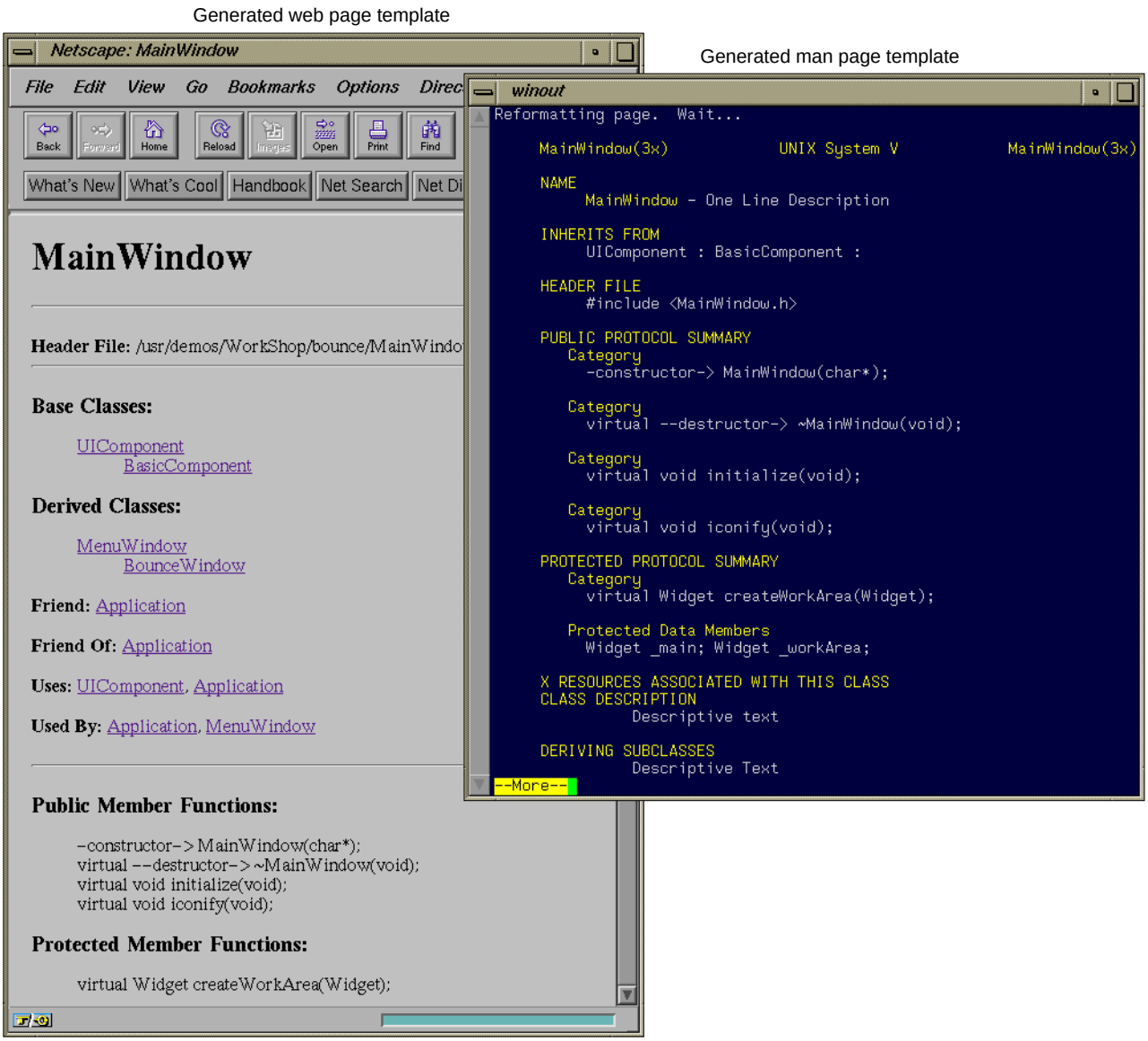


Figure 8 Generated Man and Web Page Templates

Pinpointing Performance Problems With the Performance Analyzer

The ProDev WorkShop Performance Analyzer helps you understand how your program performs so that you can correct any problems. In performance analysis, you run experiments to capture performance data and see how long each phase or part of your program takes to run. You can then determine if the performance of the phase is slowed down by the CPU, I/O activity, memory, or a bug and attempt to speed it up.

A menu of predefined tasks is provided to help you set up your experiments. With the Performance Analyzer views, you can conveniently analyze the data. These views show CPU utilization and process resource usage (such as context switches, page faults, and working set size), I/O activity, and memory usage (to capture such problems as memory leaks, bad allocations, and heap corruption).

The Performance Analyzer has three general techniques for collecting performance data:

- Counting—It can count the exact number of times each function and/or basic block has been executed. This requires *instrumenting* the program, that is, inserting code into the executable to collect counts.
- Profiling—It can periodically examine and record the program's PC (program counter), call stack, and resource consumption.
- Tracing—It can trace events that affect performance, such as *reads* and *writes*, system calls, page faults, floating point exceptions, and *mallocs*, *reallocs*, and *frees*.

Performance Analyzer User Model

1. Set up a general experiment to determine areas for improvement in your program.

   To set up a performance experiment, select a task from the Select Task submenu in the Perf menu in the Debugger Main View. The task menu lets you select predefined experiment tasks (see Figure 9). At this point, you probably haven't formed a hypothesis yet about where the performance problems lie. If this is the case, select the "Determine bottlenecks, identify phases" task. This is useful for determining the general problem areas within the program.

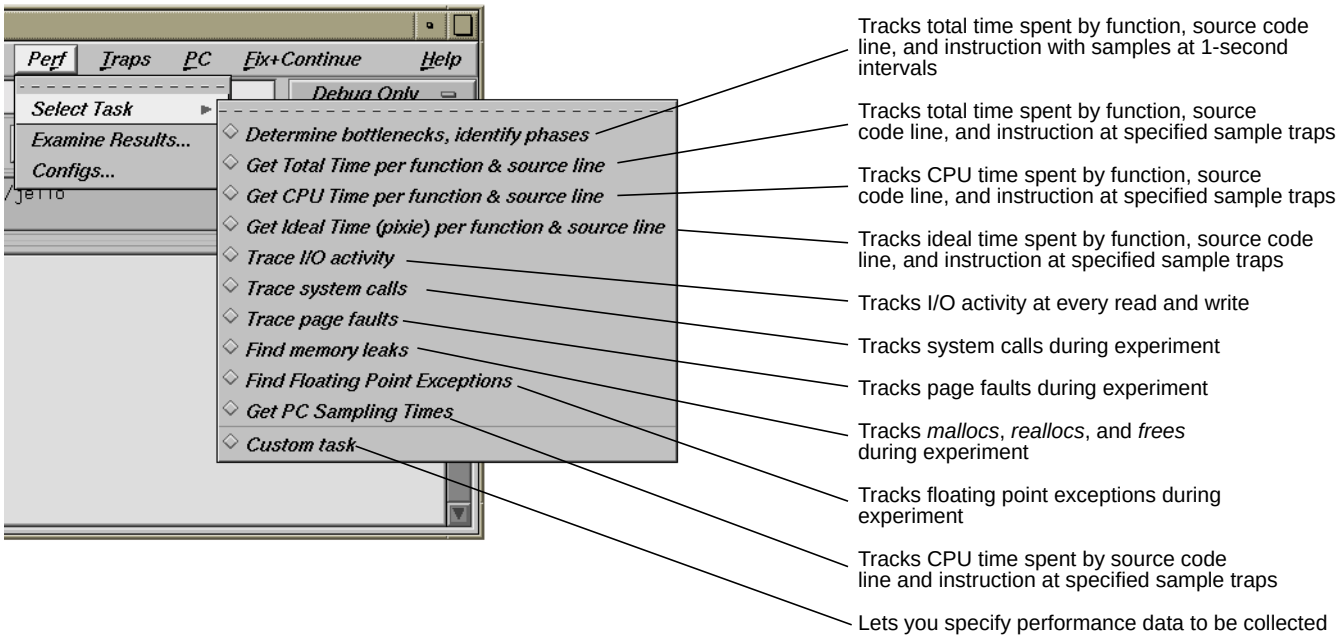


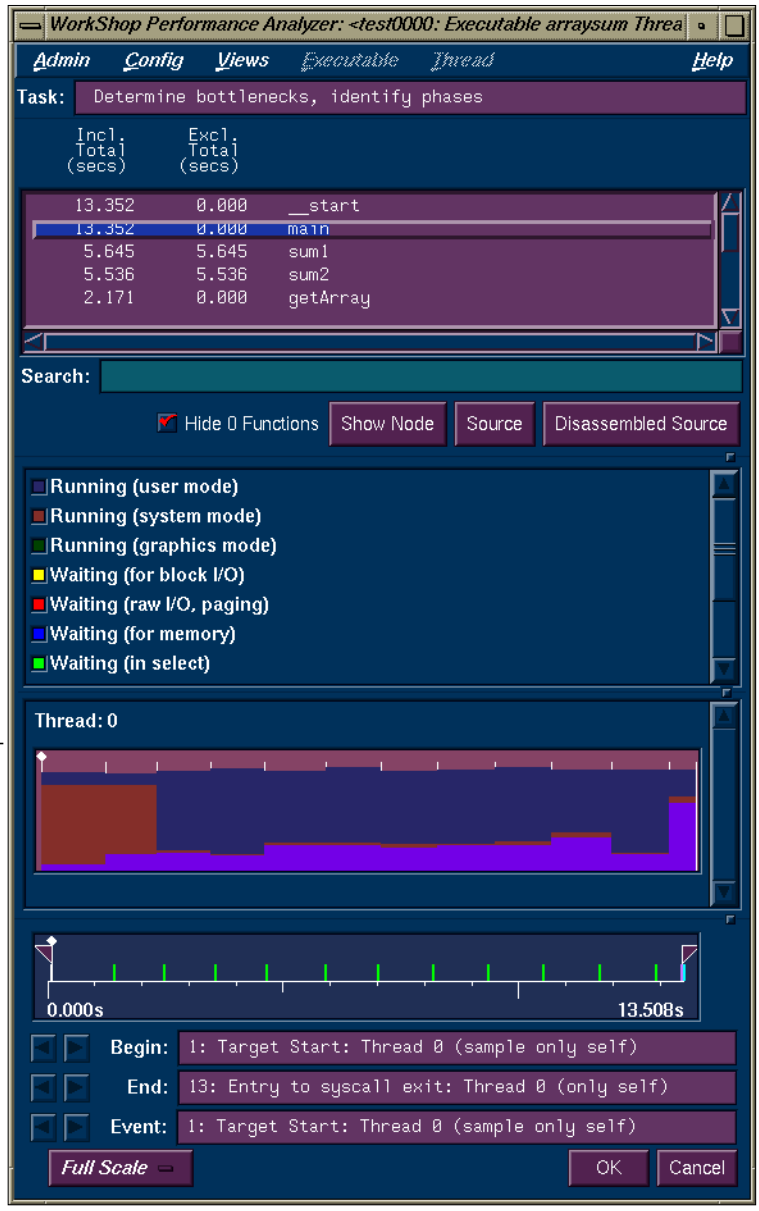
Figure 9 Debugger Main View With Perf Task Menu Displayed

2. Start the program by clicking the *Run* button in Main View.

   This runs the experiment and collects the performance data, which is written to a directory *test0000* (or a name of your choice); *test0000* is the identification for your experiment.

3. Analyze the results in the Performance Analyzer window and the Usage View (Graphs) window.

After the experiment has finished, you can display the results in the Performance Analyzer window by selecting "Performance Analyzer" from the Launch submenu in any ProDev WorkShop Admin menu or by typing **`cvperf -exp`** *experimentname*. The results from a typical performance analysis experiment appear in Figure 10, the main Performance Analyzer window, and Figure 11, which shows a subset of the graphs in the Usage Views (Graphs) window. You should be able to determine where the phases of execution occur so that you can set *sample traps* between them. Sample traps collect performance data at specified times and events in the experiment.



Function list area displays functions with their performance data from the experiment, time spent in the function including its called functions, and time spent in the function excluding calls

Usage chart area indicates general resource usage during the experiment

Time line area indicates where samples were taken. The calipers let you limit the scope of the analysis to a portion of the experiment. Double-clicking a sample point displays the call stack that occurred there.

Caliper and sample point selector controls

Figure 10 Performance Analyzer Main Window

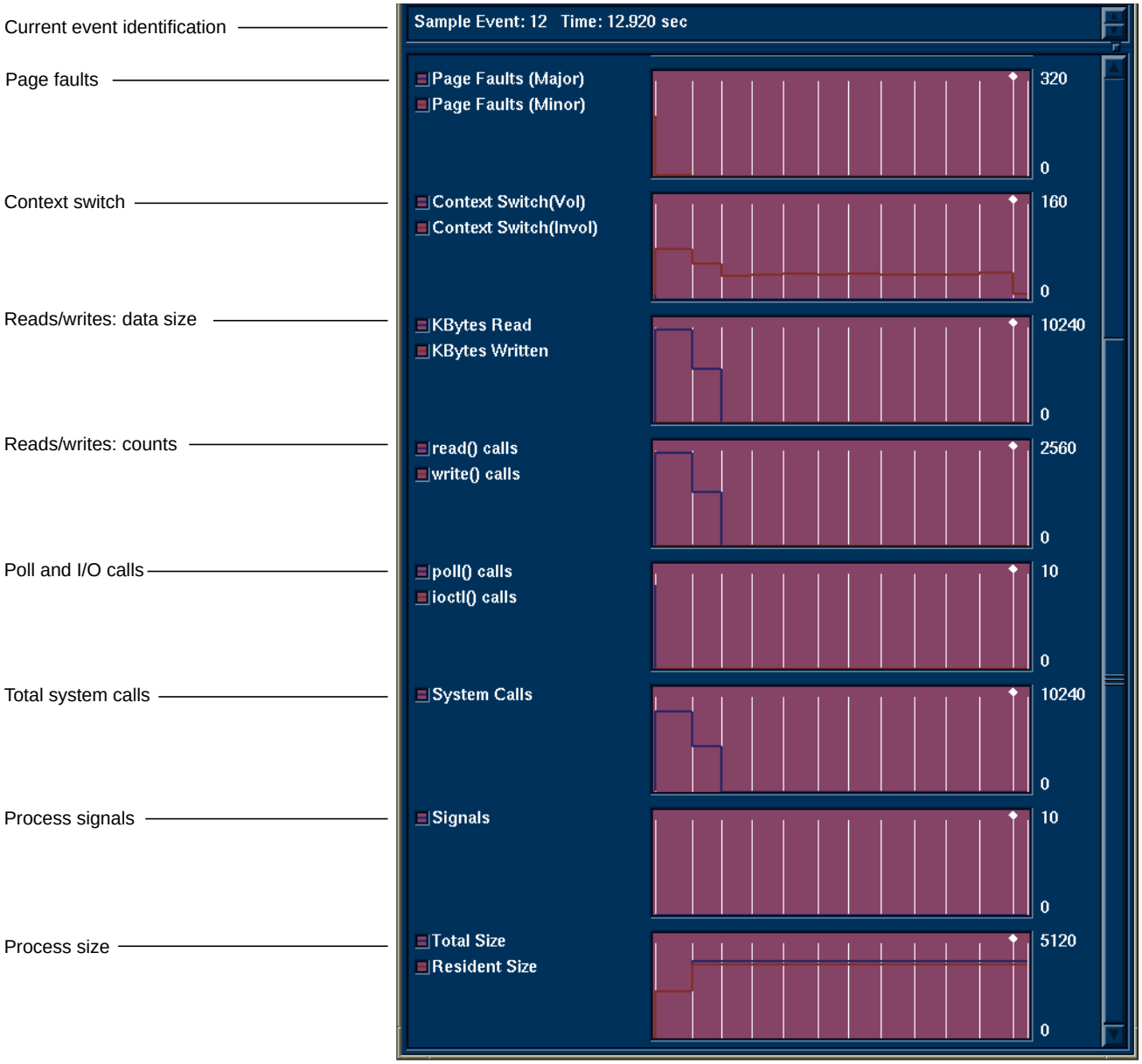


Figure 11 Usage View (Graphs) Window: Lower Graphs

4. Set sample traps at the start and end of each phase.

   Setting sample traps between phases isolates the data to be analyzed on a phase-by-phase basis. Sample traps are set by selecting “Sample”, “Sample at Function Entry”, or “Sample at Function Exit” from the Set Trap submenu in the Traps menu in the Debugger Main View or through the Traps Manager.

5. Select your next experiment from the Task Menu in the Performance Panel and run it by clicking the *Run* button in the Main View window.

   You need to form a hypothesis about the performance problem and select an appropriate task (see Figure 9) for your next experiment. There are trade-offs in selecting tasks—experiments can collect huge amounts of data and may perturb the results in some cases.

6. Analyze the results using the Performance Analyzer main window, its views, or Source View with performance data annotations displayed.

   A typical Performance Analyzer view, Malloc Error View, is shown in Figure 12. The Performance Analyzer provides results in the windows listed in Table 4.

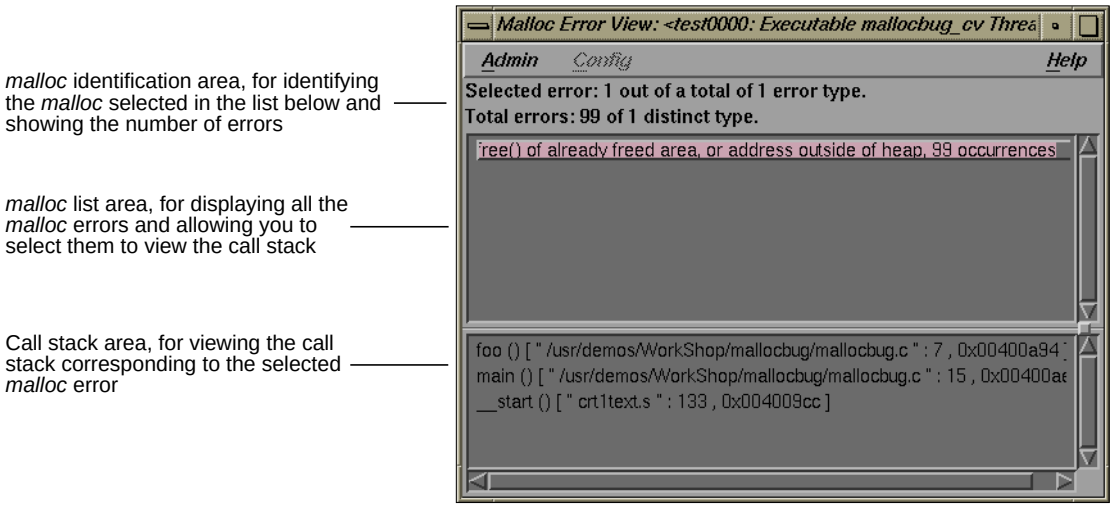


Figure 12 Malloc Error View

Table 4 Performance Analyzer Views and Data

Performance Analyzer Window	Data Provided
Performance Analyzer main window	Function list with performance data, usage chart showing general resource usage over time, and time line for setting scope on data
Call Stack View	Call stack recorded when selected event occurred
Usage View (Graphs)	Specific resource usage over time, shown as graphs
Usage View (Numerical)	Specific resource usage for selected (by caliper) time interval, shown as numerical values
Call Graph View	A graph showing functions that were called during the time interval, annotated by the performance data collected
I/O View	A graph showing I/O activity over time during the time interval
Malloc View	A list of all *mallocs*, their sizes and number of occurrences, and, if selected, their corresponding call stack within the selected time interval
Malloc Error View	A list of *malloc* errors, their number of occurrences, and if selected, their corresponding call stack within the time interval
Leak View	A list of specific leaks, their sizes and number of occurrences, and if selected, their corresponding call stack within the time interval
Heap View	A generalized view of heap memory within the time interval
Source View	The ProDev WorkShop text editor window showing source code annotated by performance data collected
Working Set View	The instruction coverage of dynamic shared objects (DSOs) that make up the executable, showing instructions, functions, and pages that were not used within the time interval
Cord Analyzer	The Cord Analyzer is not actually part of the Performance Analyzer, but it works with data from Performance Analyzer experiments. It lets you try out different ordering of functions to see the effect on performance.

Where to Find Performance Analyzer Information

To find out more about the Performance Analyzer, refer to Table 5.

Table 5 Where to Find Performance Analyzer Information in the *Developer Magic: Performance Analyzer and Tester User's Guide*

Topic	See ...
General Performance Analyzer information	Chapter 1, "Introduction to the Performance Analyzer"
Performance analysis theory	"Sources of Performance Problems" on page 43
General Performance Analyzer tutorial	Chapter 2, "Performance Analyzer Tutorial"
Memory leak tutorial	"Memory Experiment Tutorial" on page 148
Setting up performance analysis experiments including task selection	Chapter 3, "Setting Up Performance Analysis Experiments" for details and "Selecting Performance Tasks" on page 94 for a summary
Setting sample traps	Chapter 4, "Setting Traps" in the *ProDev WorkShop Debugger User's Guide*
Performance Analyzer main window	"The Performance Analyzer Main Window" on page 106
Usage View (Graphs) window	"Usage View (Graphs)" on page 119
Watching an experiment without collecting data in the Process Meter	"Process Meter" on page 125
Usage View (Numerical) window	"Usage View (Numerical)" on page 125
Tracing I/O calls using the I/O View window	"I/O View" on page 128
Call Graph View window	"Call Graph View" on page 129
Finding memory problems	"Analyzing Memory Problems" on page 138
Specifying performance annotations for Source View and Call Graph View	"Config Menu" on page 114
Call Stack View window	"Call Stack" on page 151
Improving working set behavior	"Analyzing Working Sets" on page 152

Determining the Thoroughness of Test Coverage With Tester

Tester is a software quality assurance toolset for measuring dynamic coverage over a set of tests. It tracks the execution of functions, individual source lines, arcs, blocks, and branches.

Tester User Model

This section describes the user model for designing a single test. After you have your instrumentation file and test directories set up, you can automate your testing and create larger test sets. Tester has both a command line interface (see Table 6) and a graphical user interface (see Figure 13).

1. Plan your test.
2. Create (or reuse) an instrumentation file.

   The instrumentation file defines the coverage data you wish to collect in this test.
3. Apply the instrument file to the target executable(s).

   This creates a special executable for testing purposes that collects data as it runs.
4. Create a test directory to collect the data files.
5. Run the instrumented version of the executable to collect the coverage data.
6. Analyze the results.

   Tester produces a wide variety of column-based reports. Most are available in both interfaces: command line and graphical. The reports can show source and assembly line coverage; coverage of functions; arc coverage, that is, coverage of function calls; call graphs indicating caller and callee functions and their counts; basic block counts; count information for assembly language branches; summaries of overall coverage; and argument tracing.

Table 6 Tester Command Line Interface Summary

Command Category	Command Name	Description
general	cvcov cattest	Describes the test details for a test, test set, or test group
	cvcov lsinstr	Displays the instrumentation information for a particular test
	cvcov lstest	Lists the test directories in the current working directory
	cvcov mktest	Creates a test directory
	cvcov rmtest	Removes tests and test sets
	cvcov runinstr	Adds code to the target executable to enable you to capture coverage data, according to the criteria you specify
	cvcov runtest	Runs a test or a set of tests
coverage analysis	cvcov lssum	Shows the overall coverage based on the user-defined weighted average over function, line, block, branch, and arc coverage
	cvcov lsfun	Lists coverage information for the specified functions in the program that was tested
	cvcov lsblock	Displays a list of blocks for one or more functions and the count information associated with each block
	cvcov lsbranch	Lists coverage information for branches in the program, including the line number at which the branch occurs
	cvcov lsarc	Shows arc coverage, that is, the number of arcs taken out of the total possible arcs
	cvcov lscall	Lists the call graph for the executable with counts for each function
	cvcov lsline	Lists the coverage for native source lines
	cvcov lssource	Displays the source annotated with line counts
	cvcov lstrace	Shows the argument tracing information
	cvcov diff	Shows the difference in coverage for different versions of the same program
test set	cvcov mktset	Makes a test set
	cvcov addtest	Adds a test or test set to a test set or test group
	cvcov deltest	Removes a test or test set from a test set or test group

Table 6 (continued) Tester Command Line Interface Summary

Command Category	Command Name	Description
	cvcov optimize	Selects the minimum set of tests that give the same coverage or meet the given coverage criteria as the given set
test group	cvcov mktgroup	Creates a test group that can contain other tests or test groups; targets are either the target libraries or DSOs

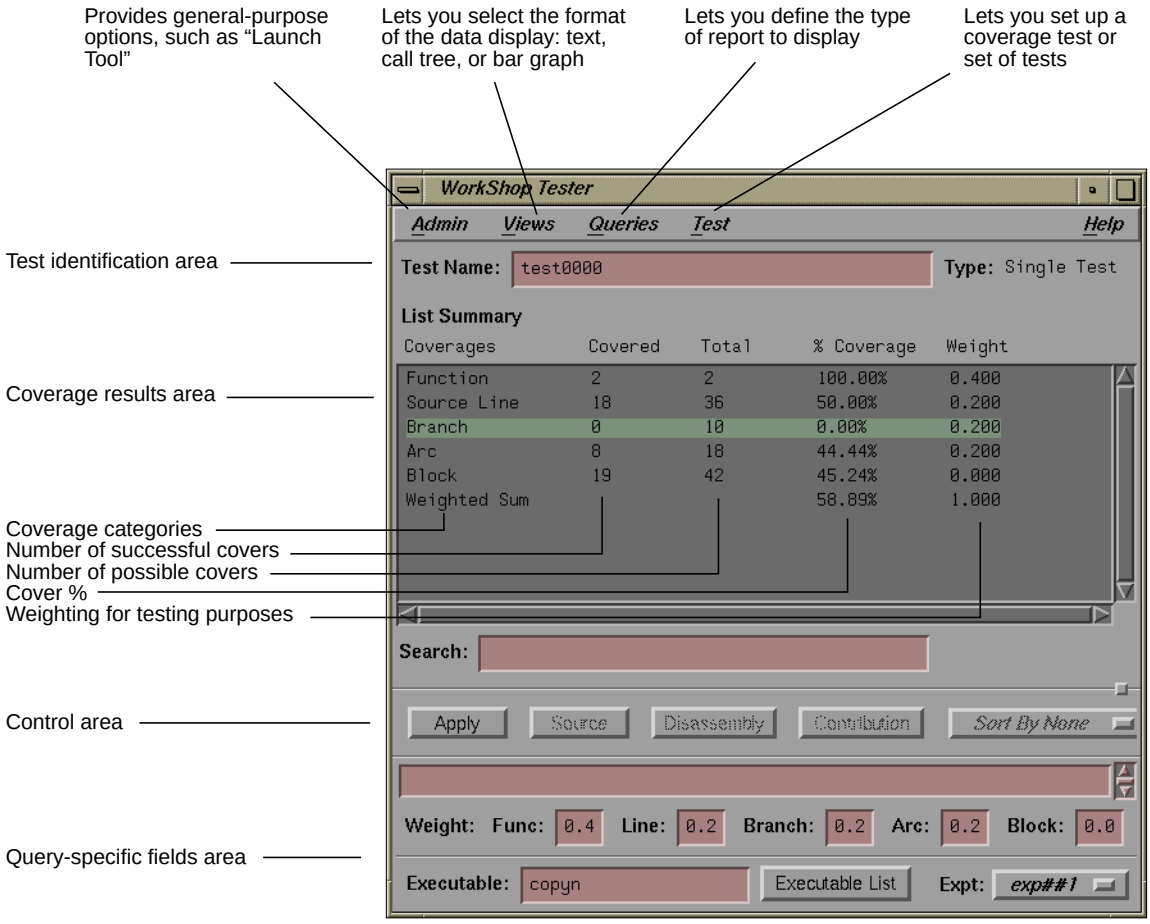


Figure 13 Major Areas of the Main Tester Window

Where to Finder Tester Information

To find out more information about Tester, refer to Table 7.

Table 7 Where to Find Tester Information in the *Developer Magic: Performance Analyzer and Tester User's Guide*

Topic	See ...
General Tester information	"Tester Overview" on page 435
Automated testing	"Automated Testing" on page 446
Command line interface tutorial	Chapter 23, "Tester Command Line Interface Tutorial"
Graphical user interface tutorial	Chapter 25, "Tester Graphical User Interface Tutorial"
Command line interface details	Chapter 24, "Tester Command Line Reference"
Graphical user interface details	Chapter 26, "Tester Graphical User Interface Reference"

Recompiling Within the ProDev WorkShop Environment With Build Manager

The Build Manager lets you view file dependencies and compiler requirements, fix compile errors conveniently, and compile software without leaving the WorkShop environment. It provides two views:

- Build View—for compiling, viewing compile error lists, and accessing the code containing the errors in Source View (the ProDev WorkShop editor) or an editor of your choice.
- Build Analyzer—for viewing build dependencies and recompilation requirements and accessing source files.

For more information on Build Manager, see Appendix B, "Using the Build Manager," in the *Developer Magic:Debugger User's Guide*.

Making Quick Changes With Fix and Continue

Fix and Continue is part of the Developer Magic MegaDev module. The Fix and Continue feature lets you make minor changes to your code from within WorkShop without having to recompile and link the entire system. You issue Fix and Continue commands in the Debugger Main View window, either by selecting them from the Fix+Continue menu or typing them in directly in the Debugger command line area.

With Fix and Continue, you can edit a function, parse the new function, and continue execution of the program being debugged. Fix and Continue enables you to speed up your development cycle significantly. For example, a program that takes 5 minutes to rebuild through a conventional compile might take 45 seconds using Fix and Continue.

Fix and Continue lets you:

- Redefine existing function definitions
- Disable, reenable, save, and delete redefinitions
- Set breakpoints in and single-step within redefined code
- View the status of changes
- Examine differences between original and redefined functions

Figure 14 shows you the WorkShop Main View during a Fix and Continue session and explains how to use the Fix and Continue menu.

Fix and Continue User Model

1. Invoke the Debugger as you normally would by typing:

   `cvd` **`[-pid`** *pid*`]` **`[-host`** *host*`]` [*executable* [*corefile*]] **`[&]`**

   See “Debugger User Model” on page 3.

2. Navigate to the function to be changed.

   You can get to the function numerous ways, by selecting “Search...” from the Source menu, typing **`func`** *functionname* at the Debugger command line, or simply scrolling to the location. If you did not use **`func`** *functionname*, you need to place the cursor inside the function.

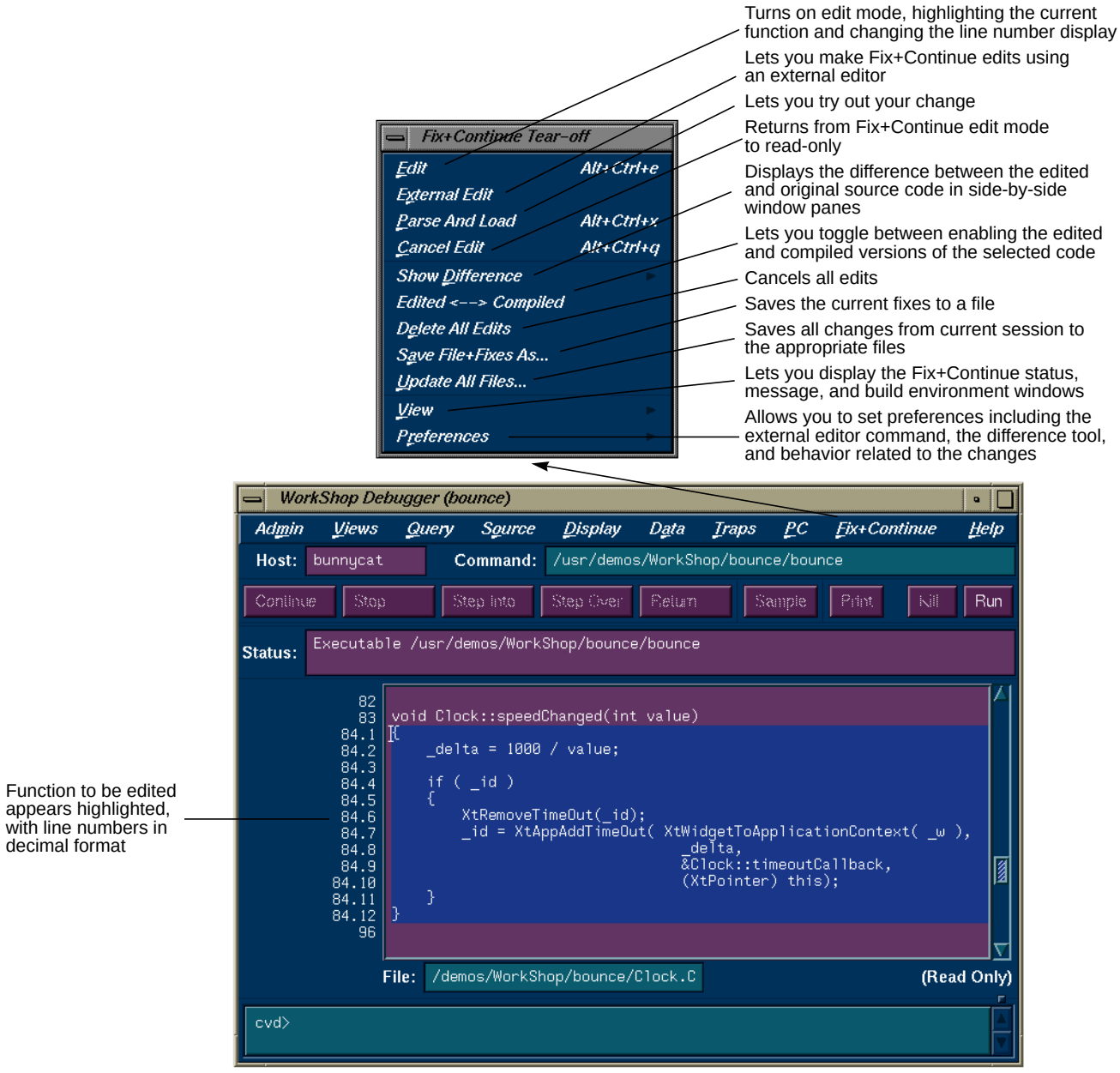


Figure 14 Using Fix+Continue

3. Select “Edit” from the Fix+Continue menu.

   This turns on edit mode, highlighting the function source code. If line numbers are displayed, those in the selected function appear with a two-part number separated by a decimal point. The left part represents the starting line number of the function in the source file before selecting “Edit”. The right part renumbers the source within the function to make it easier to keep track of added new lines.

4. Make your changes to the source code.

   You can do this directly in Main View or you can use a preferred editor by selecting “External Edit” from the Fix+Continue menu.

5. Try out your changes.

   Selecting “Parse And Load” adds your changes to the executable you are debugging. The changed function will get executed the next time it is invoked. If you stopped in the edited function, the Debugger will let you continue from the corresponding line in the new function, barring certain restrictions.

6. If the changes are satisfactory, save them for later compiling.

   “Save File+Fixes As...” saves the fixes in your current file. “Update All Files...” saves all fixes in your current session.

At any point, you can make comparisons with your old code. “Show Difference” displays the old and new source code in a side-by-side format. “Edited<-->Compiled” lets you toggle between the old and new executables making it easy to verify or demonstrate your bug fix.

Where to Find Fix and Continue Information

To find out more information about Fix and Continue, refer to Table 8.

Table 8 Where to Find Fix and Continue Information in the *Developer Magic: Debugger User's Guide*

Topic	See ...
General information and tutorial	Chapter 7, "Debugging with Fix+Continue: A Tutorial"
Detailed command information	"Fix+Continue Windows" on page 277

Debugging X/Motif Programs

The X/Motif Analyzer provides special debugging support for X/Motif applications and is available from the WorkShop Views menu. The X/Motif Analyzer operates in a number of modes (referred to as *examiners*) for examining different types of X/Motif objects. The X/Motif Analyzer provides information unavailable through conventional debuggers. It also lets you set widget-level breakpoints and collect X event history.

When you first invoke the X/Motif Analyzer, it comes up in its Widget Examiner mode. You can switch to the other examiners through the Examiner menu or by clicking the tabs at the bottom of the window (See Figure 15).

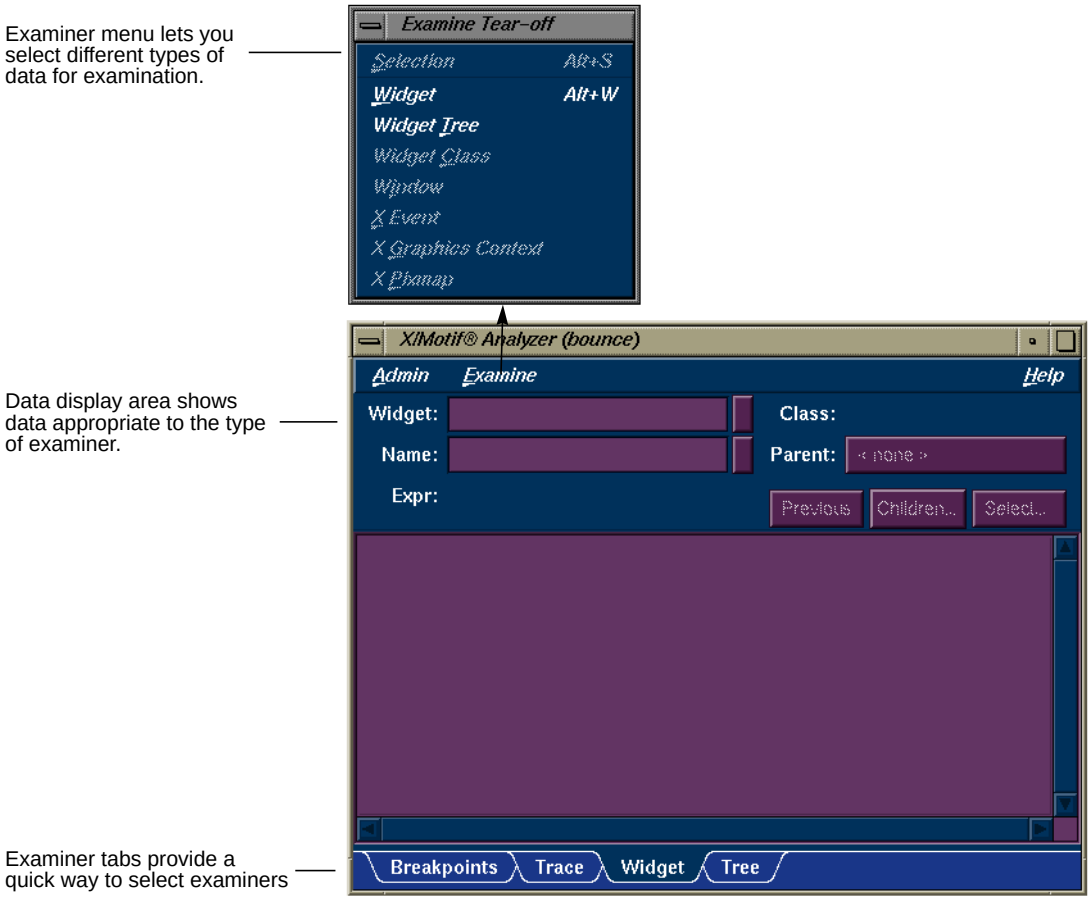


Figure 15 The X/Motif Analyzer Window

Features of the X/Motif Analyzer

The X/Motif Analyzer provides the following types of examiners:

- Widget examiner—identifies a widget's ID, name, class, and parent, and displays its definitions.
- Widget tree examiner—displays the widget hierarchy (see Figure 16). The widgets can be displayed by name, class, or ID by selecting from

the widget display menu. Double-clicking a widget node switches to the widget examiner and displays the data for the selected widget.

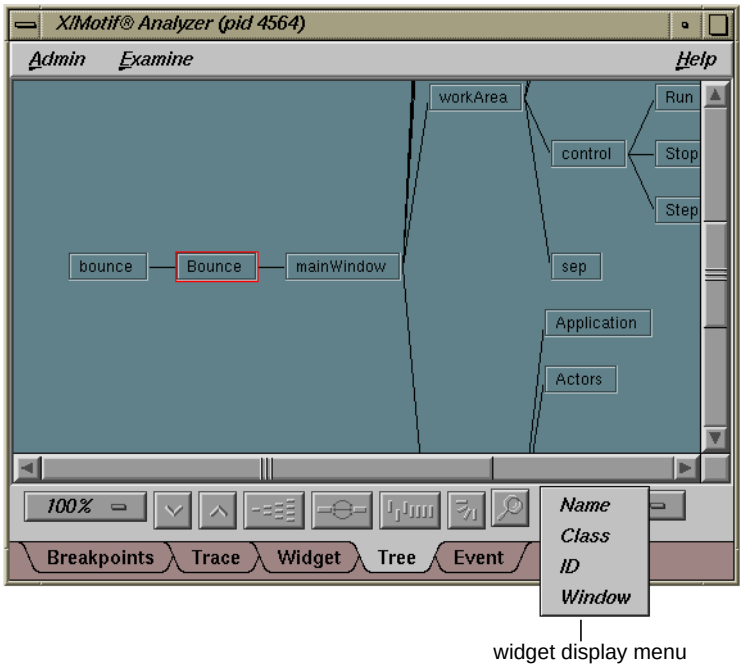


Figure 16 X/Motif Analyzer Widget Tree Examiner

- Breakpoints examiner—lets you set breakpoints at the widget and widget class level. You can set breakpoints at
  - callback functions
  - widget events
  - resource changes caused
  - timeout callback functions
  - input callback functions
  - widget state changes
  - X events
  - X requests

- Trace examiner—lets you trace the execution of your application and collect the following types of data:
  - X Server Events
  - X Server Requests
  - widget event dispatch information
  - widget resource changes
  - widget state changes
  - Xt callbacks

  Figure 17 is a typical example of the trace examiner. The events appear in a list. Double-clicking an event displays its details.
- Callback examiner—comes up automatically when the process stops in a callback. It displays
  - the callstack frame for the callback function
  - widget information
  - the callback data structure
- Window examiner—identifies the window, its parent and any children, and displays window attribute information
- Event examiner—displays the event structure for a given XEvent pointer
- Graphics context (GC) examiner—displays the X graphics context attributes for a given GC pointer
- Pixmap examiner—displays the basic attributes of an X pixmap, including size and depth, and can provide an ASCII display of small pixmaps, using the units digit of the pixel values
- widget class examiner—displays the widget class attributes for a given widget class pointer

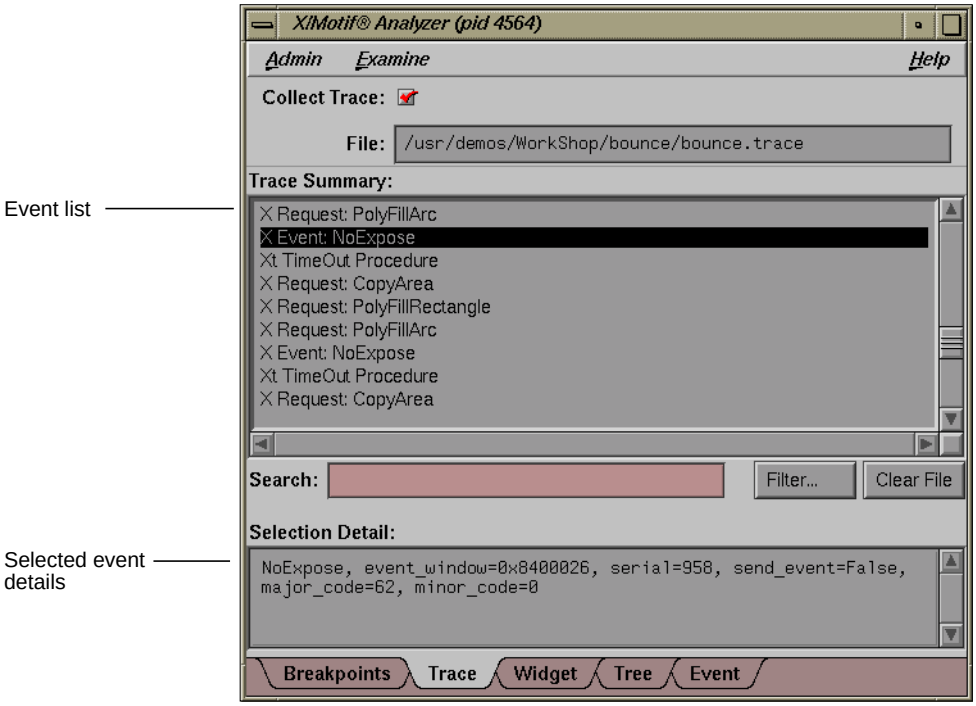


Figure 17 X/Motif Analyzer Trace Examiner

Where to Find X/Motif Analyzer Information

To find out more information about the X/Motif Analyzer, refer to Table 9.

Table 9 Where to Find X/Motif Analyzer information in the *Developer Magic: Debugger User's Guide*

Topic	See ...
General information and tutorial	Chapter 10, "Using the X/Motif Analyzer: A Tutorial"
Detailed reference information	"X/Motif Analyzer Windows" on page 193
Setting breakpoints to capture widget-level information	"Breakpoints Examiner" on page 197
Tracing widget-level data through the execution of a program	"Trace Examiner" on page 214
Getting information on a specified widget	"Widget Examiner" on page 216
Displaying a graph of the widget hierarchy	"Tree Examiner" on page 217
Getting information on a specified callback	"Callback Examiner" on page 219
Getting information on a specified window	"Window Examiner" on page 219
Getting information on a specified X event	"Event Examiner" on page 220
Getting information on a specified graphics context	"Graphics Context Examiner" on page 221
Getting information on a specified pixmap	"Pixmap Examiner" on page 222

Building Application Interfaces With RapidApp

RapidApp is a simple, interactive tool for creating application interfaces. It's integrated with the other WorkShop tools to provide a complete environment for developing object-oriented applications quickly and easily. RapidApp generates C++ code, with interface classes based on the IRIS ViewKit toolkit and IRIS IM (the Silicon Graphics version of X/Motif). RapidApp also includes predefined interface components that allow you to conveniently use other Developer Magic libraries such as OpenGL™ and Open Inventor™. Applications produced by RapidApp are automatically integrated into the Indigo Magic Desktop environment, letting you take advantage of Silicon Graphics' interface and desktop technology.

Working with RapidApp is similar to using a drawing tool such as Showcase™. A typical RapidApp window is shown in Figure 18. RapidApp lets you create interface elements by clicking icons representing widgets or components in the palette area, positioning them in a template window, and setting their resources in the editing area.

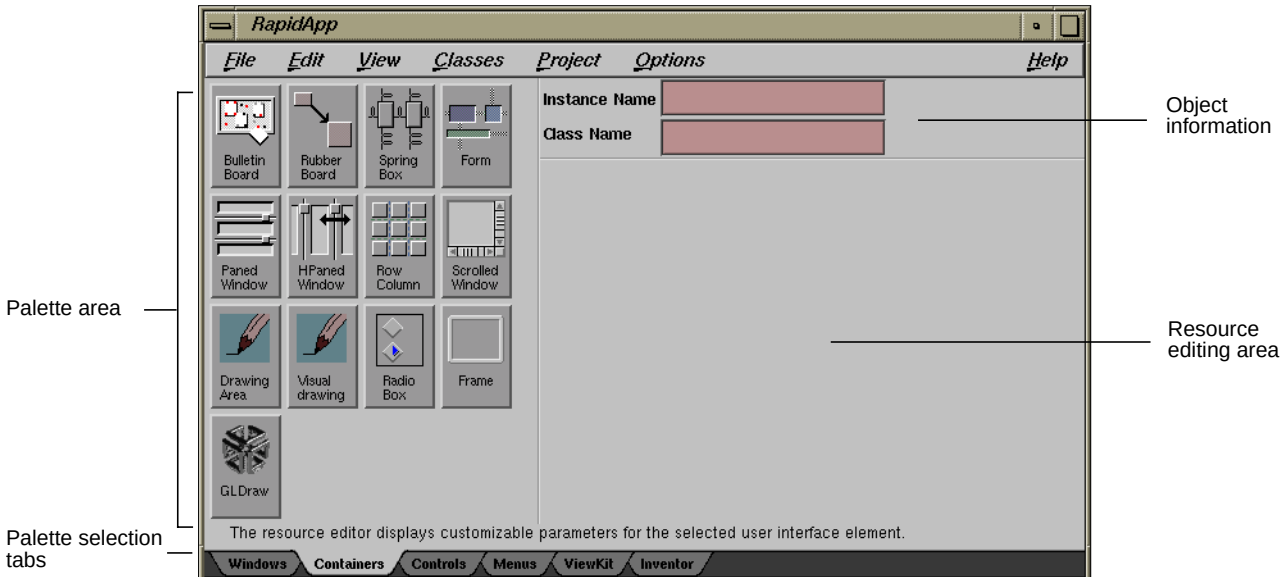


Figure 18 RapidApp Window Displaying Container Palette

RapidApp User Model

RapidApp users should be familiar with IRIX IM, IRIS ViewKit, and C++ programming. Here's the basic user model:

1. Invoke RapidApp by typing **`rapidapp`** in the directory in which you wish to build your application.

   The RapidApp window is displayed as shown in Figure 18. There are six palettes of icon widgets available. The number of palettes and icons available will increase over time as new, useful widgets are developed. The palettes and icons are:

   - Container palette—provides container widgets, that is, widgets that can hold other widgets
   - Controls palette—provides miscellaneous widgets, typically for controls, fields, and so on.
   - Windows palette—provides simple or special-purpose windows and window-oriented controls, such as menu bars and pulldown menus
   - ViewKit palette—provides ViewKit components, that is, prepackaged assemblies of widgets from the ViewKit libraries
   - Inventor palette—provides viewers, editors, and drawing areas compatible with IRIS Inventor™

   The process is then one of selecting containers, populating them with widgets, and assembling them into elements of your user interface.

2. Select a container widget.

   A rubber-band box appears, representing the initial default size of the widget. Use the mouse to drag it to a working area on your desktop (or inside another container). After you've positioned the new container widget, you can adjust its size by dragging the corners.

3. Edit the widget's resources.

   Customize the widget for your application. RapidApp changes the resource editing area according to the type of widget you are working with. It displays text fields for string resources, radio buttons for Booleans, and menus for resources with multiple values. Figure 19

illustrates the creation of a drawing area container widget. The drawing area icon has been selected from the container palette, the new widget has been placed, and the resource editing area has changed accordingly.

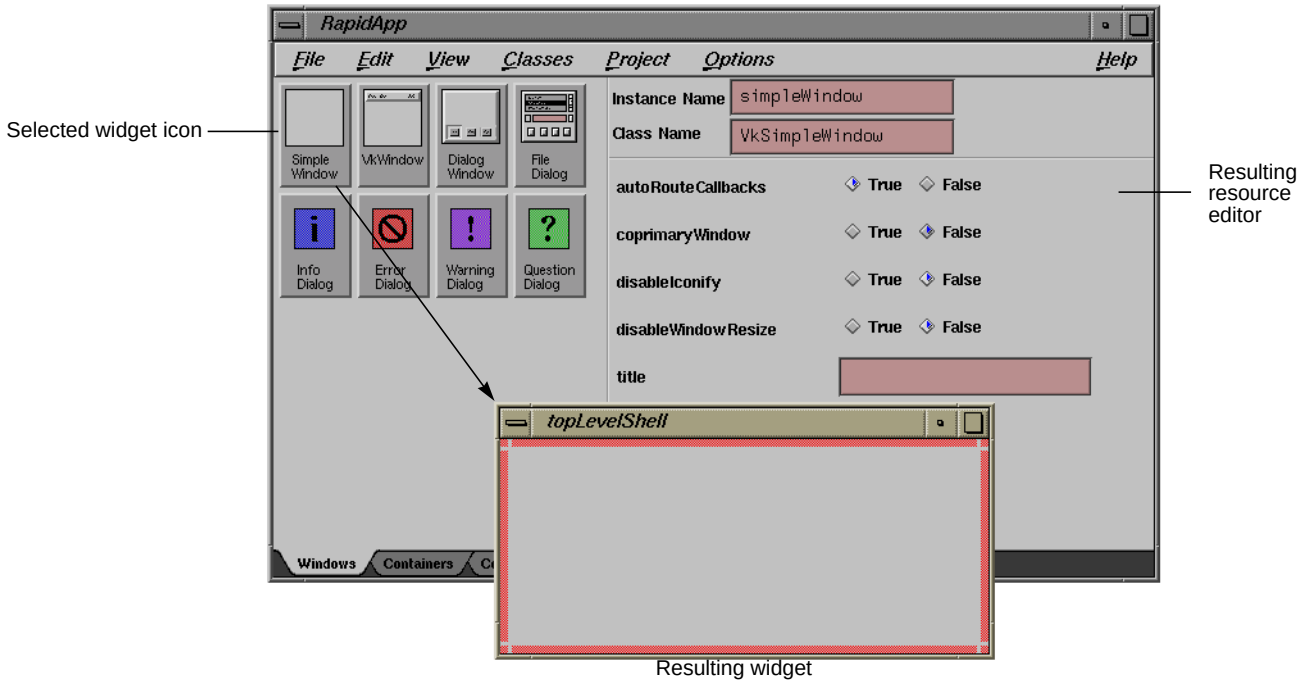


Figure 19 Creating a Widget

4. Select “Play Mode” from the View menu.

   This lets you try out the interface design. When you are through trying it out, go back to working on the interface by selecting “Build Mode” from the View menu.

5. Perform any further edits on the widget.

6. Repeat steps 2-5 until your window (or application) is complete.

7. Select “Generate C++” from the Project menu.

   This produces the source code (including Makefile) necessary to implement the interface you have designed. It also displays the Builder information window, a shell that displays RapidApp status messages.

8. Select “Edit File ...” from the Project menu to make any necessary adjustments to the source code.

   A file selection dialog box displays showing the contents of the directory containing the generated source files. When you choose a file, it will appear in your default editor.

9. Select “Build Application” from the Project menu to compile the new program.

   The WorkShop Build View displays and starts a compile going and lets you view any compile errors (see “Recompiling Within the ProDev WorkShop Environment With Build Manager” on page 30).

10. Use the other ProDev WorkShop and MegaDev tools, if necessary, to fix any coding problems.

    RapidApp is fully integrated with the rest of the Developer Magic environment so that the full range of tools and libraries are at your disposal for completing your application.

Where to Find RapidApp Information

To find out more information about RapidApp, refer to Table 10.

Table 10 Where to Find RapidApp Information in the *Developer Magic: Application Builder User’s Guide*

Topic	See ...
Understanding the RapidApp window	“The RapidApp Interface” in Chapter 1
Using RapidApp	“Basic Interaction Techniques” in Chapter 1 and Chapter 3, “Building Interfaces With RapidApp”
General tutorial	“Example: A Calculator” in Chapter 1
Inventor tutorial	Chapter 4, “Example Programs”
Windows	“Choosing and Using Windows” in Chapter 3
Containers	“Using Containers” in Chapter 3

Table 10 (continued) Where to Find RapidApp Information in the *Developer Magic: Application Builder User's Guide*

Topic	See ...
Generating software code	Chapter 2, "Creating Applications With RapidApp"
Applying the other ProDev tools to RapidApp applications	"Debugging and Interactively Adding Functionality" in Chapter 2
Detailed reference information	Appendix B, "RapidApp Reference"

Appendix A

Using Graphical Views

Many tools in ProDev WorkShop and related products provide graphical views. The graphical view is a useful device for depicting relationships. This appendix covers these topics:

- "General Graphical View Characteristics"
- "Manipulating the Display"
- "Filtering Nodes and Arcs"

General Graphical View Characteristics

The purpose of a graphical view is to provide an overview that shows relationships between entities and a means of accessing the detail information. In a graphical view, entities are shown as rectangles (or *nodes*) and relationships as connecting arrows (or *arcs*). When entities represent source code, double-clicking a node will bring up Source View with the corresponding code available for editing.

A typical graphical view appears in Figure A-1. Graphical views have a display area with a row of controls underneath it. If the graph is larger than the viewing area, scroll bars will be enabled.

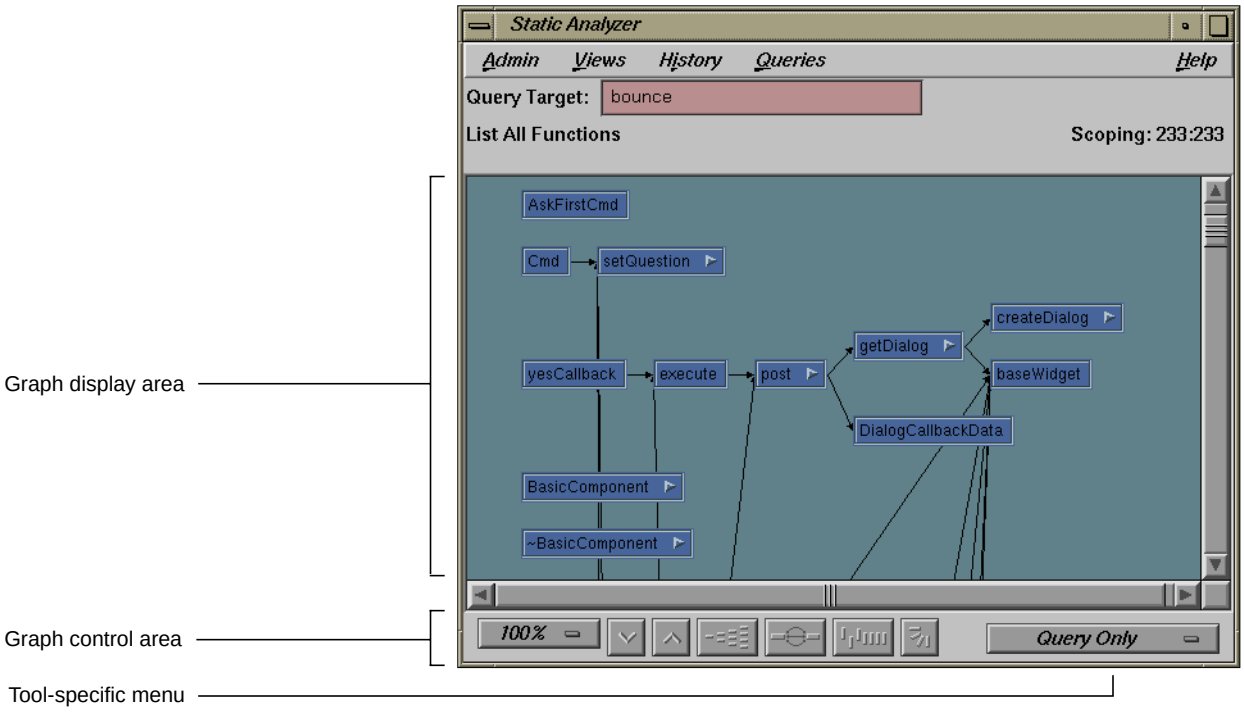


Figure A-1 Typical Graphical View

Since an overwhelming amount of information can be displayed in a graphical view, a number of methods are provided for simplifying the display. They fall into two categories: those that manipulate the display without changing the current contents and those that let you filter nodes and arcs from the display.

Manipulating the Display

This section covers those methods that change the display without altering the contents.

Graph Control Area

All graphical views have a control area containing a row of graph controls as shown in Figure A-2.

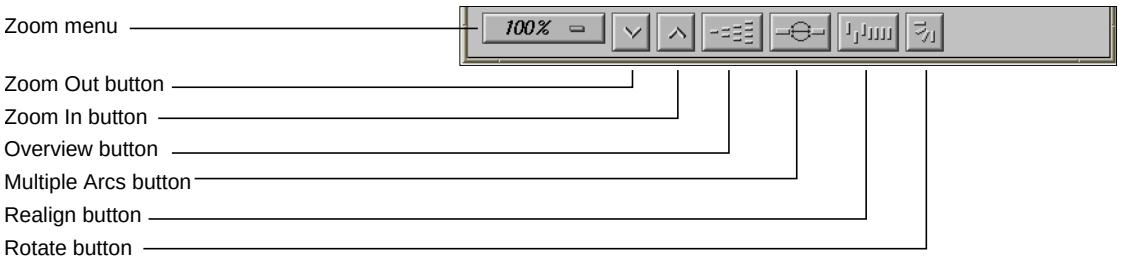


Figure A-2 Graph Display Controls

Note: In some cases, the *Multiple Arcs* button may be disabled. This is appropriate where there can only be one arc between nodes.

These graphical view controls are:

Zoom menu

shows the current scale of the graph. If clicked on, a pop-up menu appears displaying other available scales. The scaling range is between 15% and 300% of the normal (100%) size.

Zoom Out button

resets the scale of the graph to the next available smaller size in the range.

Zoom In button

resets the scale of the graph to the next available larger size in the range.

Note: If you reposition the nodes by dragging and then use one of the *Zoom* buttons, the configuration will return to the initial position.

Overview button

invokes the overview pop-up display, which shows a scaled-down representation of the graph. The nodes appear in the analogous places on the overview pop-up, and a white outline may be used to position the main graph relative to the pop-up. Alternatively, the main graph may be repositioned with its scroll bars. See the following section.

Multiple Arcs button

toggles between single and multiple arc mode. Multiple arc mode is extremely useful for the "List Arcs" query, because it indicates graphically how many of the paths between two functions were actually used.

Realign button

redraws the graph, restoring the positions of any nodes that were repositioned.

Rotate button

flips the orientation of the graph between horizontal (calling nodes at the left) and vertical (calling nodes at the top).

Note: If you reposition the nodes by dragging and then change orientation, the nodes will return to the initial positioning relative to each other.

Overview Window

The Overview window lets you view the entire graph at a reduced scale. To display the Overview window, you click the *overview* button (see Figure A-3).

Figure A-4 shows a typical Overview window with the resulting graph. The Overview window has a movable viewport that lets you select the portion of the graph displayed in the main window. Special nodes and arcs are highlighted for easy detection.

Figure A-3 Admin Menu in the Overview Window

The Overview window has an Admin menu (see Figure A-3) with these three selections:

"Scale to Fit"
: scales the graph to match the aspect ratio of the overview window.

"Show Arcs"
: displays or hides the arcs between the nodes.

"Close"
: closes the Overview window.

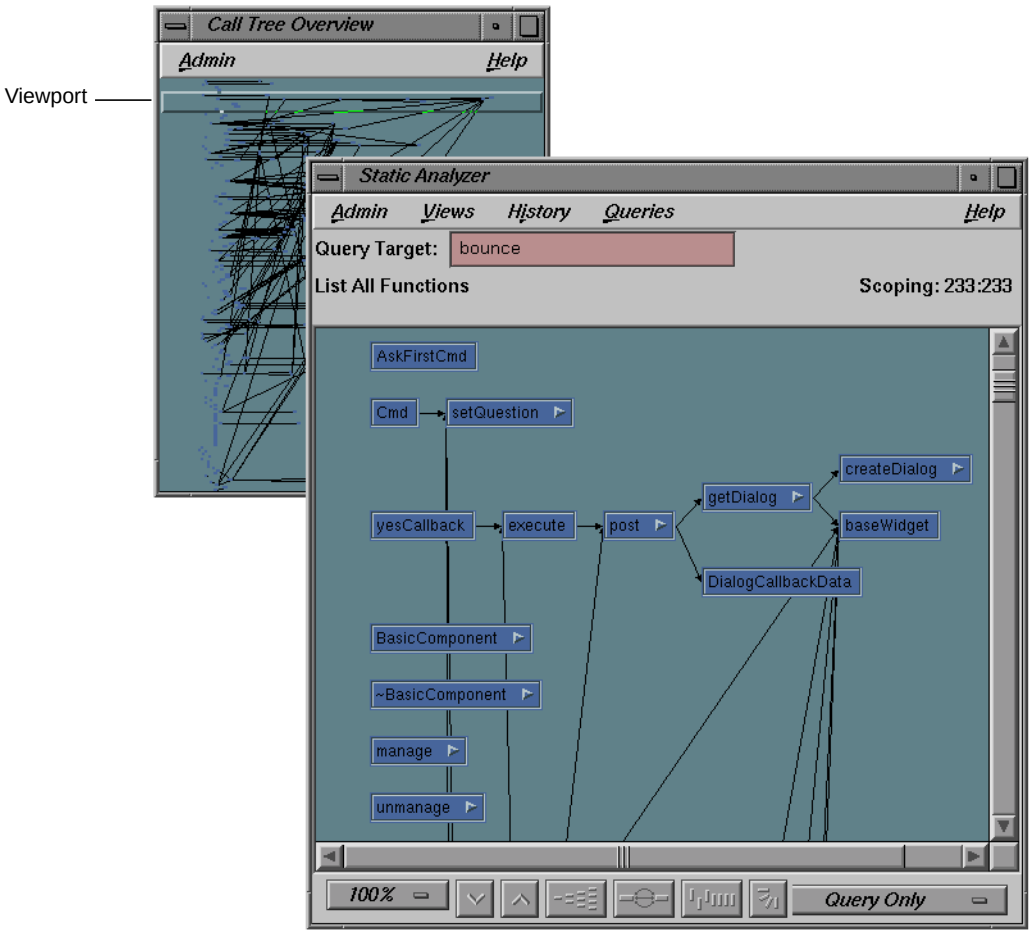


Figure A-4 Overview Window with Resulting Graph

Using the Mouse in a Graph

You can move an individual node in a graph by dragging it with the middle mouse button. This can help reveal obscured arc annotations.

You can select multiple nodes by dragging a selection rectangle around them. You can Ctrl-click to add a single node to the group. Shift-clicking a

node adds it to the group along with all the nodes that it calls. Once you have selected a group of nodes, you can move them as a group with the middle mouse button or perform other operations on them.

Selecting Nodes from outside the Graph

Often you can specify a node from a text view, search field, or dialog box and it will be highlighted in the graph.

Filtering Nodes and Arcs

Another approach to simplifying a graph is to reduce the number of nodes and arcs. Different tools have different filtering options. All graphs have two types of node menus (accessed by holding the right mouse button) for filtering nodes: the Node menu and the Selected Nodes menu. Both menus are shown in Figure A-5.

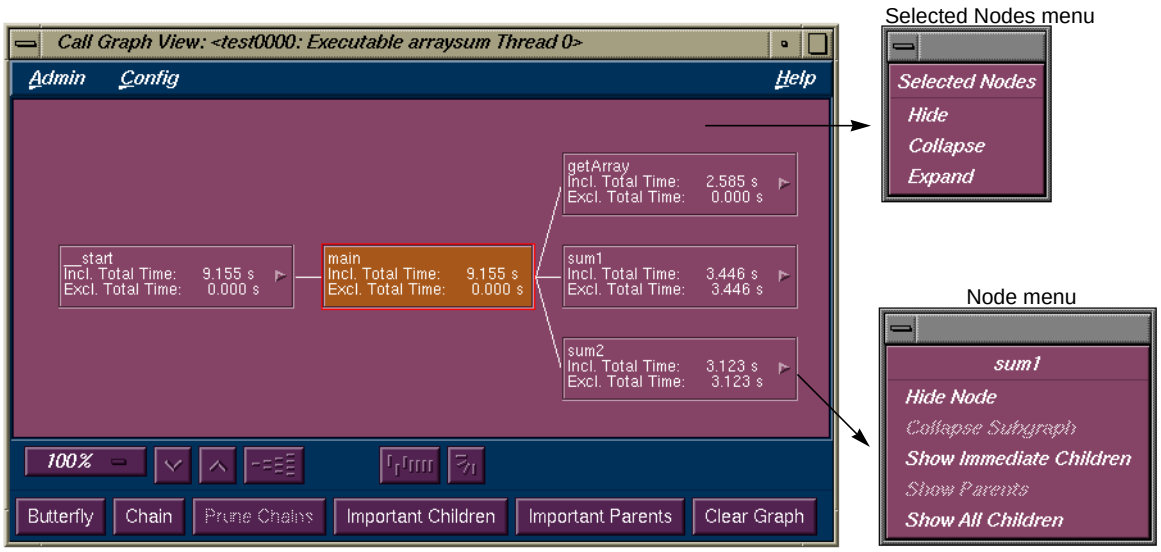


Figure A-5 Node Pop-up Menus

Node Menu

The Node menu lets you filter a single node. It is displayed by holding the right mouse button down while the cursor is over the node. The name of the selected node appears at the top of the menu. The Node menu selections are:

"Hide Node"
: removes the selected node from the graph display.

"Collapse Subgraph"
: removes the nodes called by the selected node (and subsequently called nodes) from the graph display.

"Show Immediate Children"
: displays the functions called by the selected node.

"Show Parents"
: displays all the functions that call the selected node.

"Show All Children"
: displays all the functions (descendants) called by the selected node.

Selected Nodes Menu

The Selected Nodes menu lets you filter multiple nodes. You can select multiple nodes by dragging a selection rectangle around them. You can also Shift-click a node and it will be selected along with all the nodes that it calls. Holding down the right mouse button anywhere in the graph displays the Selected Nodes menu. The Selected Nodes menu items are:

"Hide"
: removes the selected nodes from the graph display.

"Collapse"
: removes the nodes called by the selected nodes (and descendant nodes) from the graph display.

"Expand"
: displays all the functions (descendants) called by the selected node.

Appendix B

Customizing ProDev WorkShop Tools

If the configuration of a window or view does not meet your particular needs, you may be able to adjust the graphical user interface accordingly. This appendix discusses how to make such changes.

- "Customizing Within ProDev WorkShop"
- "Changing X Resources"

Customizing Within ProDev WorkShop

If you want to change the appearance of the ProDev WorkShop windows, we recommend that you start with the menus provided for that purpose:

- WorkShop Main View: Display menu
- Array Browser: Color, Scale, Format, and Spreadsheet menus
- Call Stack View: Config and Display menus
- Disassembly View: Config, Disassemble, and Display menus
- Expression View: Config, Display, Language popup, and Format popup menus
- Memory View: Mode menu
- Process Meter: Charts and Scale menu
- Register View: Config menu
- Source View: Display menu
- Structure Browser: Config, Display, Node, and Format popup menus
- Trap Manager: Config and Display menus
- Variable Browser: Language popup, and Format popup menus
- Build Analyzer: Filter, Selected Node popup and Node popup menus and graphic controls
- Build View: Admin menu—"Preferences..." and "Build Options..."
- Performance Analyzer: Config menus in all views, and Selected Node popup and Node popup menus and graphic controls in graphical views
- Static Analyzer: Views menu, and Selected Node popup and Node popup menus and graphic controls in graphical views
- Tester: Views menu, and Selected Node popup and Node popup menus and graphic controls in graphical views

Changing X Resources

While there are hundreds of X resources that can be changed, we recommend that you avoid modifying recources if at all possible. However, in some cases, there may be no way within WorkShop to make the desired change. Here are some X resources for the Debugger and its views that you may find useful:

**autoStringFormat*
: if true, sets default format for **char* results as strings in Expression View, the Variable Browser, and the Structure Browser; otherwise the default format will be the hex address.

**varBrowser*maxSymSize*
: lets you set the maximum number of variables that can be displayed by the Variable Browser. The current default is 25.

**expressionView*maxNumOfExpr*
: lets you set the maximum number of expressions that can be read from a file by Expression View. The current default is 25.

*cvmain*sourceView*nameText.columns*
: sets the length of the *File* field in the WorkShop Main View window. The default is 30 characters.

*Cvmain*disableLicenseWarnings* and **disableLicenseWarnings*
: disable the license warning messagse that display when you start *cvd* and the other tools.

The following resources apply to Source View:

**tabWidth*
: sets the number of spaces for tabs in Source View.

**sourceView*textEdit.scrollHorizontal*
: if true, displays a horizontal scroll bar in Source View.

**sourceView*nameText.columns*
: sets the length of the *File* field in Source View. The default is 30 characters.

**svComponent*lineNumbersVisible*
: displays source line numbers by default.

The following resource applies to the Build Analyzer:

**buildCommand*
: is used by the Build Analyzer to determine which program to use to build with (*make*, *smake*, *clearmake*, etc.). The default value is *make*.

To change these resources, you need to set the desired value in your *.Xdefaults* file, re-run *xrdb* if you use it, and then restart your application so that the resource gets picked up.

Glossary

anti-leak

See bad free.

arc

A relation between two entities in a program depicted graphically as lines between rectangles (nodes). For example, arcs can represent function calls, file dependency, or inheritance.

Array Browser

A Debugger view that displays the values of an array in a spreadsheet format and can also depict them graphically in a 3D rendering.

bad free

A problem that occurs when a program frees a malloced piece of memory that it had already freed (also referred to as an anti-leak condition or double free).

Bar Graph View

A display mode of Tester that shows a summary of coverage information in a bar graph.

basic block

A block of machine-level instructions used as a metric in Performance Analyzer and Tester experiments. A basic block is the largest set of consecutive machine instructions that can be formed with no branches into or out of them.

boundary overrun

A problem that occurs when a program writes beyond a specified region, for example overwriting the end of an array or a malloced structure.

boundary underrun

A problem that occurs when a program writes in front of a specified region, for example writing ahead of the first element in an array or a malloced structure.

breakpoint

See trap

Browser (Static Analyzer)

A facility within the Static Analyzer for viewing structural and relationship information in C++ or Ada programs. It provides three views: Browser View for displaying member and class information; Class Graph for displaying inheritance, containment, interaction, and friend relationships in the hierarchy; and Call Graph for displaying the calling relationships of methods, virtual methods, and functions.

Build Analyzer

A view that displays a graph of program files (source and object) indicating build dependencies and provides access to the source files.

Build Manager

A tool for recompiling programs within WorkShop. The Build Manager has two windows: Build Analyzer and Build View.

Build View

A view that lets you run compiles. In addition, Build View displays compile errors and provides access to the code containing the errors.

calipers

See time line

call graph

A generic term for views used in several tools (Static Analyzer, C++ Browser, Performance Analyzer, and Tester) that display a graph of the calling hierarchy of functions. Double-clicking a function in a call graph causes the Source View window to be displayed showing the function's source code.

Call Graph

A display mode of the C++ Browser that shows methods and their calls. *See*

also call graph and C++ Browser.

Call Graph View

A display mode of the Performance Analyzer that shows functions, their calls, and associated performance data. *See also* call graph and C++ Browser.

Call Stack

A view that displays the call stack at the current context. In the Debugger this means where the process is stopped; in the Performance Analyzer this means sample traps and other events where data was written out to disk. Each frame in the Call Stack window can show the function; argument names, values, and types; the function's source file and line number; and the PC (program counter). Double-clicking a frame in the Call Stack causes the Source View window to be displayed showing the corresponding source code.

Call Tree View (Static Analyzer version)

A display mode of the Static Analyzer that displays the results of function queries as a call graph. *See also* call graph and Static Analyzer

Call Tree View (Tester version)

A display mode of Tester that displays function coverage information in a call graph. *See also* Tester

Call View

A display mode of the C++ Browser for displaying member and class information. *See also* C++ Browser

Class Graph

A display mode of the C++ Browser for displaying inheritance, containment, interaction, and friend relationships in the class hierarchy.

Class Tree View

A display mode of the Static Analyzer that displays the results of class queries as a class hierarchy. *See also* Static Analyzer

ClearCase™

A tool in the Developer Magic™ environment for performing configuration management and version control.

command line (Debugger)

A field in the Debugger Main View that lets you enter a set of commands similar to *dbx* commands.

cord

A system command used to rearrange procedures in an executable to reduce paging and achieve better instruction cache mapping. The Cord Analyzer and Working Set View let you analyze the effectiveness of an arrangement and try out new arrangements to improve efficiency.

Cord Analyzer

A tool that lets you analyze the paging efficiency of your executable's working sets, that is, the executable code brought into memory during a particular phase or operation. It also calculates an optimized ordering and lets you try out different working set configurations to reduce paging problems. The Cord Analyzer works with the Working Set View, a part of the Performance Analyzer. *See also* cord, working set, and Working Set View

counts

The number of times a piece of code (function, line, instruction, or basic block) was executed as listed by Tester or the Performance Analyzer.

coverage

A term used in Tester. Coverage means a test has exercised a particular unit of source code, such as functions, individual source lines, arcs, blocks, or branches. In the case of branches, coverage means the branch has been executed under both true and false conditions.

CPU-bound

A performance analysis term for a condition in which a process spends its time in the CPU and is limited by CPU speed and availability.

CPU time

A performance analysis metric approximating the time spent in the CPU. CPU time is calculated by multiplying the number of times a PC appears in the profile of a function, source line, or instruction by 10 ms.

cvcord

The name of the Cord Analyzer executable. *See also* Cord Analyzer

cvcov

The name of the Tester command line interface executable. *See also* Tester

cvd

The name of the Debugger executable. *cvd* has options for attaching the Debugger to a running process (**-pid**), examining core files (executable), and running from a remote host (**-host**). *See also* Debugger

cvperf

The name of the executable that calls the Performance Analyzer. *cvperf* has an option (**-exp**) for designating the name of the experiment directory. *See also* Performance Analyzer

cvspeed

The name of the executable that brings up the Performance Panel, a window for setting up Performance Analyzer experiments. *See also* Performance Panel

cvstatic

The name of the executable that calls the Static Analyzer. *See also* Static Analyzer

cvxcov

The name of the executable that calls the graphical interface of Tester. *See also* Tester

cycle count

The specified number of times to hit a breakpoint before stopping the process, it defaults to 1. The cycle count for any trap can be set through the Trap Manager view in the Debugger.

DCC

A native C++ compiler that allows you to use dynamic classes (also known as Delta C++). *See also* the DCC(1) reference page for more information

Debugger

A tool in ProDev WorkShop for analyzing general software problems using a live process. The Debugger lets you stop the process at specific locations in the code by setting breakpoints (referred to as *traps*) or by clicking the Stop

button. At each trap, you can display special windows called *views*, for examining data. *See also cvd*

Disassembly View

A view that lets you see the program's machine-level code. The Debugger version shows you the code; the Performance Analyzer version additionally displays performance data for each line.

double free

See bad free

DSO (dynamic shared object)

An ELF (Executable and Linking Format) format object file, similar in structure to an executable program but with no **main**. It has a shared component, consisting of shared text and read-only data; a private component, consisting of data and the GOT (Global Offset Table); several sections that hold information necessary to load and link the object; and a liblist, the list of other shared objects referenced by this object. Most of the libraries supplied by SGI are available as dynamic shared objects.

erroneous free

A problem that occurs when a program calls **free()** on addresses that were not returned by malloc, such as static, global, or automatic variables, or other invalid expressions.

event

An action that takes place during a process, such as a function call, signal, or a form of user interaction. The Performance Analyzer uses event tracing in experiments to help you correlate measurements to points in the process where events occurred.

exclusive performance data

Performance Analyzer data collected for a function without including the data for any functions it calls. *See also* inclusive performance data

Execution View

A Debugger view that serves as a simple shell to provide access outside of WorkShop. It's typically used to set environment variables, inspect error messages, and conduct I/O with the program being debugged.

experiment

The model for using the Performance Analyzer and Tester. The steps in creating an experiment are (1) creating a directory to hold the results, (2) instrumenting the executable (instrumentation is recompiling with special libraries for collecting data), (3) running the instrumented executable as a test, and (4) analyzing the results using the views in the tools. The first two steps are done automatically when you use the Performance Panel and select a performance task (performance experiments only). The term *experiment* can also refer to the actual data itself that was saved.

Expression View

A Debugger view that lets you specify one or more expressions to be evaluated whenever the process stops or the callstack context is changed. Expression View lets you save sets of expressions for subsequent reuse, specify the language of the expression (Ada, Fortran, C, or C++), and specify the format for the resulting values.

File Dependency View

A display mode of the Static Analyzer that displays the results of queries in a graph indicating file dependency relationships. *See also* Static Analyzer

Fileset Editor

A window for specifying a fileset, that is, the set of files to be used in creating a database for Static Analyzer queries. The Fileset Editor also lets you specify whether a file is to be analyzed using scanner mode or parser mode. *See also* parser mode, scanner mode, and Static Analyzer

fine-grained usage

A technique in performance analysis that captures resource usage data between sample traps.

Fix + Continue

A feature in the Debugger that lets you make source level changes and continue debugging without having to perform a full compile and relinking.

floating point exception

A problem that occurs when a program cannot complete a numerical calculation due to division by zero, overflow, underflow, inexact result, or invalid operand. Floating point exceptions can be captured by the

Performance Analyzer and can also be identified in the Array Browser.

freed memory

Freed memory is memory that was originally malloced and has been returned for general use by calling free(). Accessing freed memory is a problem that occurs when a program attempts to read or write this memory, possibly corrupting the free list maintained by malloc.

function list

A generic type of view used in several tools (Static Analyzer, Performance Analyzer, Tester, and Cord Analyzer) to list functions and related information, such as location, experiment data, and executable code size. Double-clicking a function displays its source code in Source View.

GLDebug

A graphical software tool for debugging application programs that use the IRIS Graphics Library (GL). GLdebug locates programming errors in executables when GL calls are used incorrectly. GLDebug is not part of WorkShop but is accessible from the Admin menu in Main View.

heap corruption

A memory problem that may be due to boundary overrun or underrun, accessing uninitialized memory, accessing freed memory, freeing a memory location twice, or attempting to free a memory location erroneously. *See also* malloc debugging library

Heap View

A Performance Analyzer view that displays a map of memory indicating how blocks of memory were used in the time interval set by the time line calipers.

ideal time

A performance analysis metric that assumes that each instruction takes one cycle of the particular machine's time. It's then useful to compare the ideal time with the actual time in an experiment.

inclusive performance data

Performance Analyzer data collected for a function where the total includes data for all of the called functions. *See also* exclusive performance data

instrumentation

See experiment

I/O-bound

A performance analysis term for a condition in which a process has to wait for I/O to complete and may be limited by disk access speeds or memory caching.

I/O View

A Performance Analyzer view that displays a chart devoted to I/O system calls. I/O View can identify up to 10 files involved in I/O.

IRIS IM™

A user interface toolkit on Silicon Graphics® systems based on X/Motif®.

IRIS IM Analyzer

A Debugger view for debugging X/Motif applications. The IRIS IM Analyzer lets you look at object data, set breakpoints at the object or X protocol level, trace X and widget events, and tune performance.

IRIS ViewKit™

A Developer Magic toolkit that provides predefined widgets and classes for building applications.

Leak View

A Performance Analyzer view that displays each memory leak that occurred in your experiment, its size, the number of times the leak occurred at that location during the experiment, and the call stack corresponding to the selected leak.

library search path

A path you may need to specify when debugging executables or core files to indicate which DSOs (dynamic shared objects) are required for debugging. *See also* DSO

Main View

The main window of the Debugger. The MainView provides access to other tools and views, process controls, a source code display, and a command line for entering a set of commands similar to dbx. You can also add custom

buttons to Main View using the command line.

Malloc Error View

A Performance Analyzer view that displays each malloc error (leaks and bad frees) that occurred in an experiment, the number of times the malloc occurred (a count is kept of mallocs with identical call stacks), and the call stack corresponding to the selected malloc error.

malloc debugging library

A special library (libmalloc_cv.a) for detecting heap corruption problems. Relinking your executable with the malloc library sets up mechanisms for trapping memory problems.

Malloc View

A Performance Analyzer view that displays each malloc (whether or not it caused a problem) that occurred in your experiment, its size, the number of times the malloc occurred (a count is kept of mallocs with identical call stacks), and the call stack corresponding to the selected malloc.

MegaDev

The package name for a set of advanced Developer Magic tools for the development of C and C++ applications.

Memory-bound

A performance analysis term for a condition in which a process continuously needs to swap out pages of memory.

memory leak

A problem when a program dynamically allocates memory and fails to deallocate that memory when it is through with the space.

Memory View

A Debugger view that lets you see or change the contents of memory locations.

Multiprocess View

A Debugger view that lets you manage the debugging of a multiprocess executable. For example, you can set traps in individual processes or across groups of processes.

NCC

A native C++ compiler that uses the same compiler as DCC, but doesn't allow you to use dynamic classes.

node

The rectangles in graphical views. A node may represent a function, class, or file depending on the type of graph.

Overview window

A window in graphical views that displays the current graph at a reduced scale and lets you navigate to different parts of the graph.

palette

The portion of the RapidApp window that provides user interface elements for creating graphical interfaces. *See also* RapidApp

parser mode

A method of extracting Static Analyzer data from source files. Parser mode uses the compiler to build the Static Analyzer database. It is language-specific and very thorough; as a result, it is slower than scanner mode. *See also* scanner mode and Static Analyzer

Path Remapping

A dialog box that lets you set mappings to redirect filenames used in building your executable to their actual locations in the filesystem.

PC (program counter)

The current line in a stopped process, indicated by a right-pointing arrow with a highlight in the source code display areas and by a highlighted frame in the Call Stack views.

Performance Analyzer

A tool in ProDev WorkShop for measuring the performance of an application. To use the tool, you select one of the predefined analysis tasks, run an experiment, and examine the results in one of the Performance Analyzer views. *See also cvperf*

Performance Panel

A window for setting up Performance Analyzer experiments. The panel

displays toggles and fields for specifying data to be captured. As a convenience, you can select performance tasks (such as “Determine bottlenecks...” or “Find memory leaks”) from a menu that specifies the data automatically. *See also cvspeed*

performance task

See Performance Panel

phase

A performance analysis term for a period in an experiment covering a single activity. In a phase, there is one limiting resource that controls the speed of execution.

pollpoint sampling

A technique in performance analysis that captures performance data, such as resource usage or event tracing, at regular intervals.

Process Meter

A view that monitors the resource usage of a running process without saving the data. *See also* Performance Analyzer and Performance Panel

ProDev WorkShop

The package name for the core WorkShop tools.

profile

A record of a program’s PC (program counter), call stack, and resource consumption over time, used in performance analysis.

Project View

A Debugger view for managing ProDev WorkShop and MegaDev tools operating on a common target.

query

The term for a search through a Static Analyzer database to locate elements in your program. Queries are similar to the IRIX grep command but provide a more specific search. For example, you can perform a query to find where a method is defined. *See also* Static Analyzer

RapidApp

A tool in the Developer Magic environment for creating graphical interfaces quickly and easily. RapidApp lets you drag and drop user interface elements (for example, IRIS IM widgets, IRIS ViewKit components, Inventor components, and so on) onto a template window to create the interface.

Register View

A Debugger view that lets you see or change the contents of the machine registers.

Results Filter

A dialog box that lets you limit the scope of Static Analyzer queries. *See also* query and Static Analyzer

sample trap

Similar to a stop trap except that instead of stopping the process, performance data is written out to disk and the process continues running. *See also* trap

sampling

In performance analysis, the capture of performance data, such as resource usage or event tracing, at points in an experiment so that a graph of usage over time can be created.

scanner mode

A method of extracting Static Analyzer data from source files. Scanner mode is fast but not language-specific so that the source code need not be compilable. Results may have minor inaccuracies. *See also* parser mode and Static Analyzer

Signal Panel

A dialog box for specifying signals to trap.

Smart Build

An option to the compiler where only those files that must be recompiled are recompiled.

Source View

A window for viewing or editing source code. Source View is an alternative

editing window to Main View. If you have conducted Performance Analyzer or Tester experiments, you can view the results in the column to the left of the source code display area.

stack

See Call Stack

Static Analyzer

A tool in ProDev WorkShop for viewing the structure of a program at different levels and locating where elements of the program are used or defined. The Static Analyzer works by extracting structure and location information from files that you specify and storing the information in a database for subsequent analysis. You can view the analysis as a text list or graphically. *See also cvstatic*, Call Tree View, Class Tree View, File Dependency View, and Text View

stop trap

A breakpoint. *See also* trap

Structure Browser

A Debugger view that graphically displays data structures including data values and pointer relationships.

Syscall Panel

A dialog box for specifying system calls to trap. You can designate whether to trap the system calls at the entry or exit from the call.

test group

A grouping of experiments in Tester used to test a common DSO (dynamic shared object).

test set

A group of experiments in Tester used to test a common executable.

Tester

A tool in ProDev WorkShop for measuring dynamic coverage over a set of tests. It tracks the execution of functions, individual source lines, arcs, blocks, and branches. Tester has both a command line and a graphical interface.

Text View (Static Analyzer version)

A display mode of the Static Analyzer that displays the results of queries as a scrollable text list. *See also* Static Analyzer

Text View (Tester version)

A display mode of Tester that displays function coverage information in a report form. *See also* Tester

time line

A feature in the main Performance Analyzer window that shows where events occurred in an experiment and provides calipers for controlling the scope of analysis for the Performance Analyzer views.

tracing

A record of a specified type of event (such as reads and writes, system calls, page faults, floating point exceptions, and mallocs, reallocs, and frees) over time, used in performance analysis.

trap

A mechanism for trapping data at specified points and conditions in a live process. Also referred to as a breakpoint. There are two types of traps: stop traps are used in debugging to halt a process, and sample traps are used in performance analysis to collect data without halting the process. *See also* watchpoint

Trap Manager

A window for managing traps. It lets you set simple or conditional traps, browse (or modify) a list of traps, and save or load a set of traps.

uninitialized memory

Memory that is allocated but not assigned any specific contents. Accessing uninitialized memory is a problem that occurs when a program attempts to read memory that has not yet been initialized with valid information.

Usage View (Graphical)

A Performance Analyzer view that contains charts indicating resource usage and the occurrence of events, corresponding to time intervals set by the time line calipers.

Usage View (Textual)

A Performance Analyzer view that displays the actual resource usage values corresponding to time intervals set by the time line calipers.

Variable Browser

A Debugger view that displays the local variables valid in the current context and their values (or addresses). The Variable Browser also lets you view the previous value at the breakpoint. You can enter a new value directly if you wish.

view

A window that lets you analyze data.

ViewKit

See IRIS ViewKit

watchpoint

A trap that fires when a specified variable or address is read, written, or executed.

working set

The set of executable pages, functions, and instructions brought into memory during a particular phase or operation. *See also* Working Set View

Working Set View

A Performance Analyzer view that lets you measure the coverage of the dynamic shared objects (DSOs) that make up your executable. It indicates instructions, functions, and pages that were not used in a particular phase or operation in an experiment. Working Set View works with the Cord Analyzer. *See also* working set and Cord Analyzer

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2582-003.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389