ProDev™ WorkShop: Overview

007–2582–006

Cover design by Sarah Bolles, Sarah Bolles Design, and Dany Galgani, SGI Technical Publications.

Record of Revision

Version	Description
1.0	March 1995 Original Printing.
2.7	June 1998 Revised to reflect changes for the ProDev WorkShop 2.7 release.
2.8	April 1999 Revised to reflect changes for the ProDev WorkShop 2.8 release.
006	November 2001 Revised to reflect changes for the ProDev WorkShop 2.9.1 release.

Contents

Figures

Tables

About This Guide

This publication provides an overview to the ProDev WorkShop tool kit products, release 2.9.1, running on IRIX systems. It describes the different products within the tool kit, and provides pointers to other books which describe the products in more detail.

This release of the tool kit requires the following software levels:

- IRIX 6.2 or higher
- IRIX 6.4 or higher (for pthread support)
- MIPSpro 7.2 or higher
- SpeedShop 1.3.2 or higher

Related Publications

The following documents contain additional information that may be helpful:

- *ProDev Workshop: Debugger User's Guide*
- *ProDev Workshop: Performance Analyzer User's Guide*
- *ProDev Workshop: Static Analyzer User's Guide*
- *ProDev Workshop: Tester User's Guide*
- *ProDev WorkShop: ProMP User's Guide*
- *MIPSpro Fortran 77 Programmer's Guide*
- *MIPSpro Fortran 77 Language Reference Manual*
- *MIPSpro Fortran 90 Commands and Directives Reference Manual*

The *Guide to SGI Compilers and Compiling Tools* provides an overview of all documentation for SGI compilers and the different performance and analysis tools for those compilers.

Obtaining Publications

To obtain SGI documentation, go to the SGI Technical Publications Library at:

`http://techpubs.sgi.com.`

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.
GUI	This font denotes the names of graphical user interface (GUI) elements such as windows, screens, dialog boxes, menus, toolbars, icons, buttons, boxes, fields, and lists.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact us in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library World Wide Web page:

  http://techpubs.sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Pkwy., M/S 535
  Mountain View, California 94043–1351

- Send a fax to the attention of "Technical Publications" at +1 650 932 0801.

We value your comments and will respond to them promptly.

ProDev WorkShop Overview

Welcome to the ProDev WorkShop group of tools, a major part of the ProDev software development environment. ProDev WorkShop is a software toolset for the development of C, C++, Fortran 77, Fortran 90, and Ada applications. These powerful, highly visual tools help you understand your program's structure and operation so that you can diagnose difficult, traditionally time-consuming problems in a short amount of time. With these tools, you can develop applications for the entire SGI product line, from Indy workstations to Origin2000 supercomputers.

The ProDev WorkShop toolset has the following functionality:

- **Comprehensive control over the debugging process**—You can set simple breakpoints with the click of a mouse button or define complex conditions for your breakpoints. 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**—The ProDev WorkShop Debugger 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**—The ProDev WorkShop 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+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.

Note: In addition to the ProDev WorkShop tools, you can separately purchase ProDev WorkShop ProMP, which is a visual code parallelization tool used with the Power Fortran Accelerator to help you balance parallel loops in Fortran applications.

This document gives you an overview of the ProDev WorkShop tools as well as pointers to the documentation for getting detailed information. The book is organized as follows:

Using the Debugger

The WorkShop Debugger is a UNIX source-level debugging tool that provides special windows (views) for displaying program data and execution states 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 Debugger goes far beyond the capabilities of `dbx`. It includes fast data watchpoints and other types of breakpoints; 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 2-1.

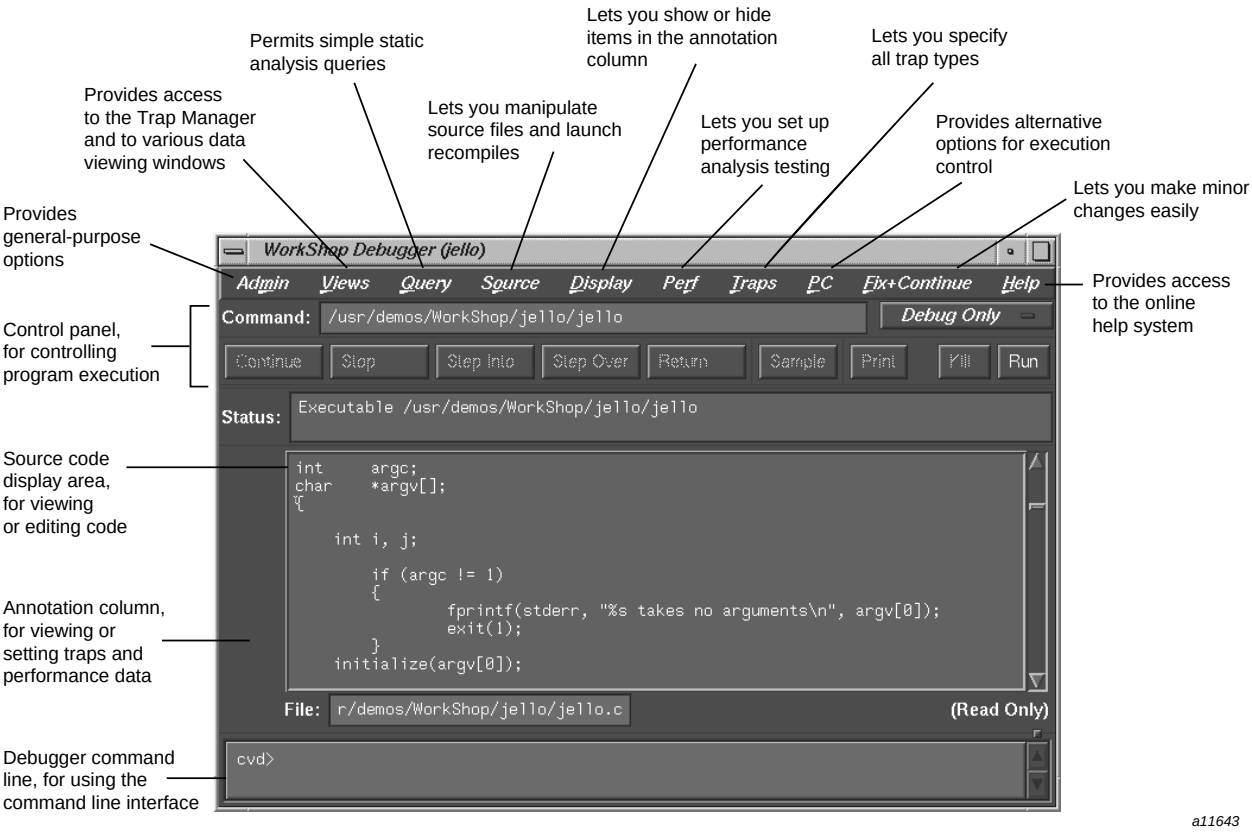

Figure 2-1 Major Areas of the Main View Window

The basic steps for debugging an application are as follows:

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 `-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 any of the following criteria apply to your application:

- You do not want Debugger windows to interfere with the application you are debugging.
- You are supporting an application remotely.
- You do not want to use the Debugger on the target system for another reason.

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 later specify the name of the executable file.

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

2. Set breakpoints in the source code.

   You can set a simple breakpoints 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 can be set and managed from the **Trap Manager** window, the **Signal Panel**, and the **Syscall Panel**, all of which can be accessed from the **Views** menu. You can also set breakpoints by typing them at the Debugger command line in the Main View window. You can stop a process at any time by clicking the **Stop** button in the Main View.

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

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

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

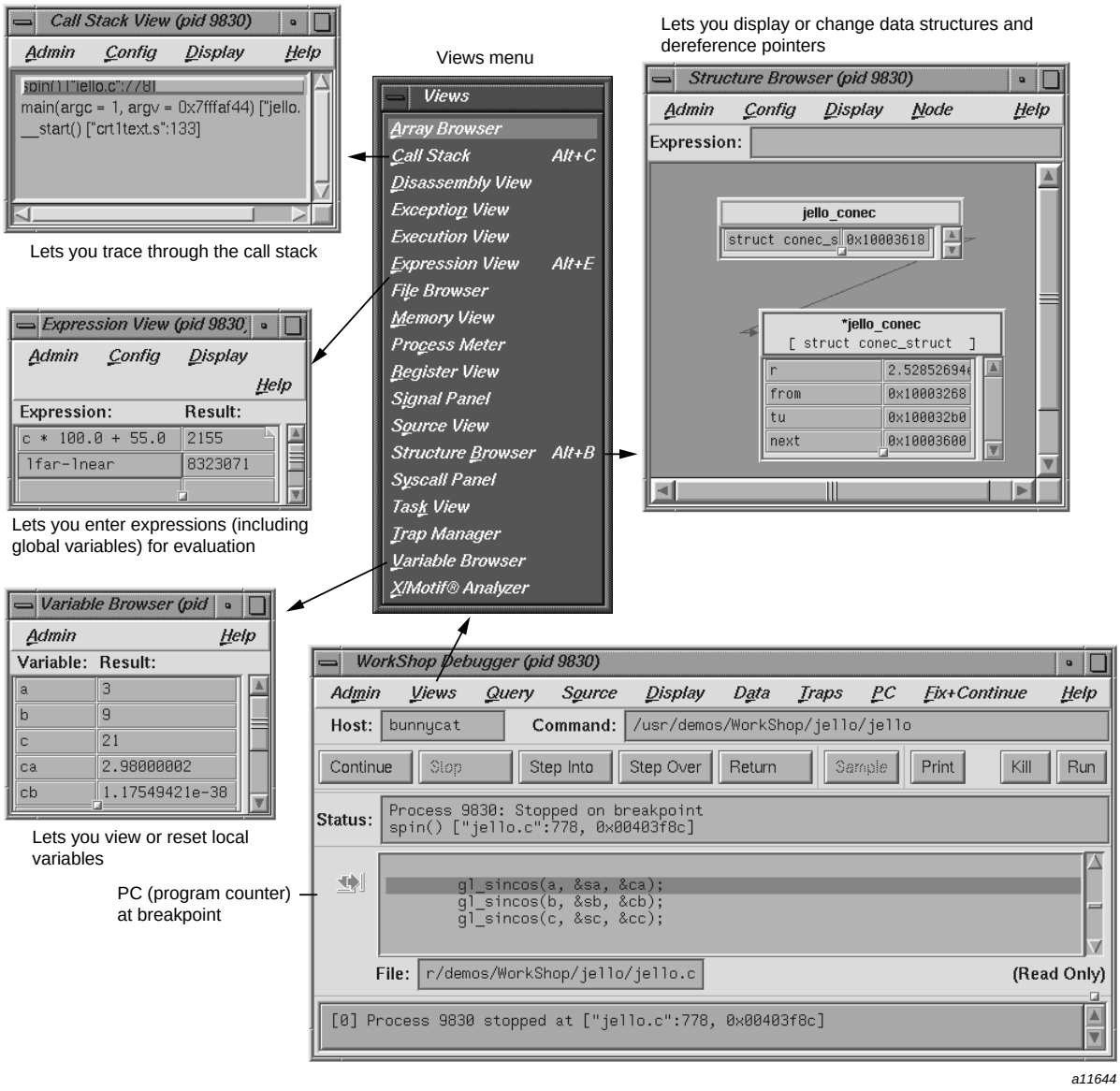

Figure 2-2 Typical Debugger Views Accessible at a Breakpoint

Figure 2-3, page 8, shows the **Array Visualizer** window, 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. Note the high point coming out of the 3D image; it demonstrates how anomalies in large arrays stand out.

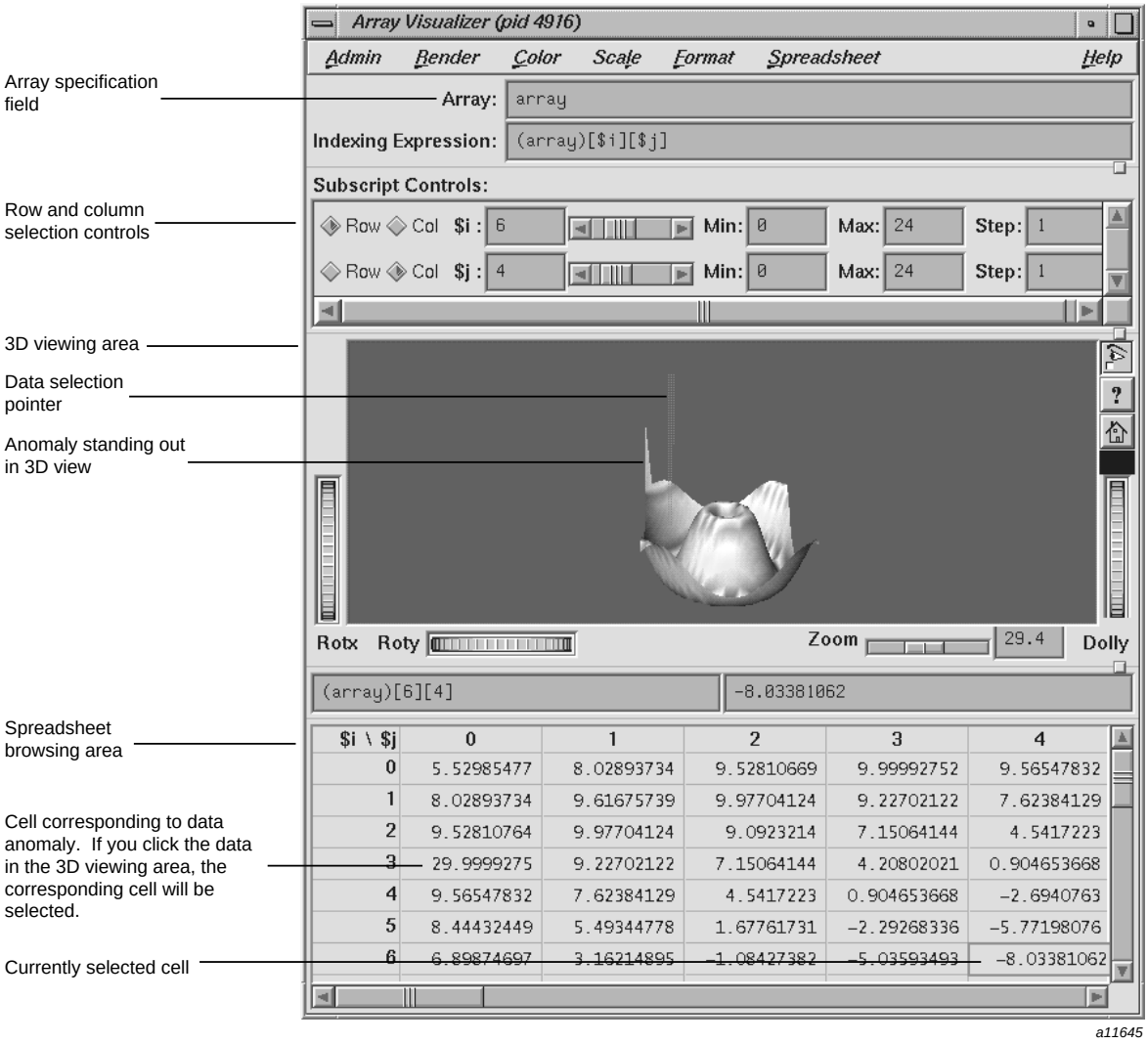

Figure 2-3 Array Visualizer Window

Note: For an explanation of all controls and menus on this window, refer to the *ProDev Workshop: Debugger User's Guide.*

If you need to debug your application at the machine level, you can use **Register View**, **Disassembly View**, or **Memory View**, as shown in Figure 2-4. These views are accessed from the **Views** menu in the Debugger Main View.

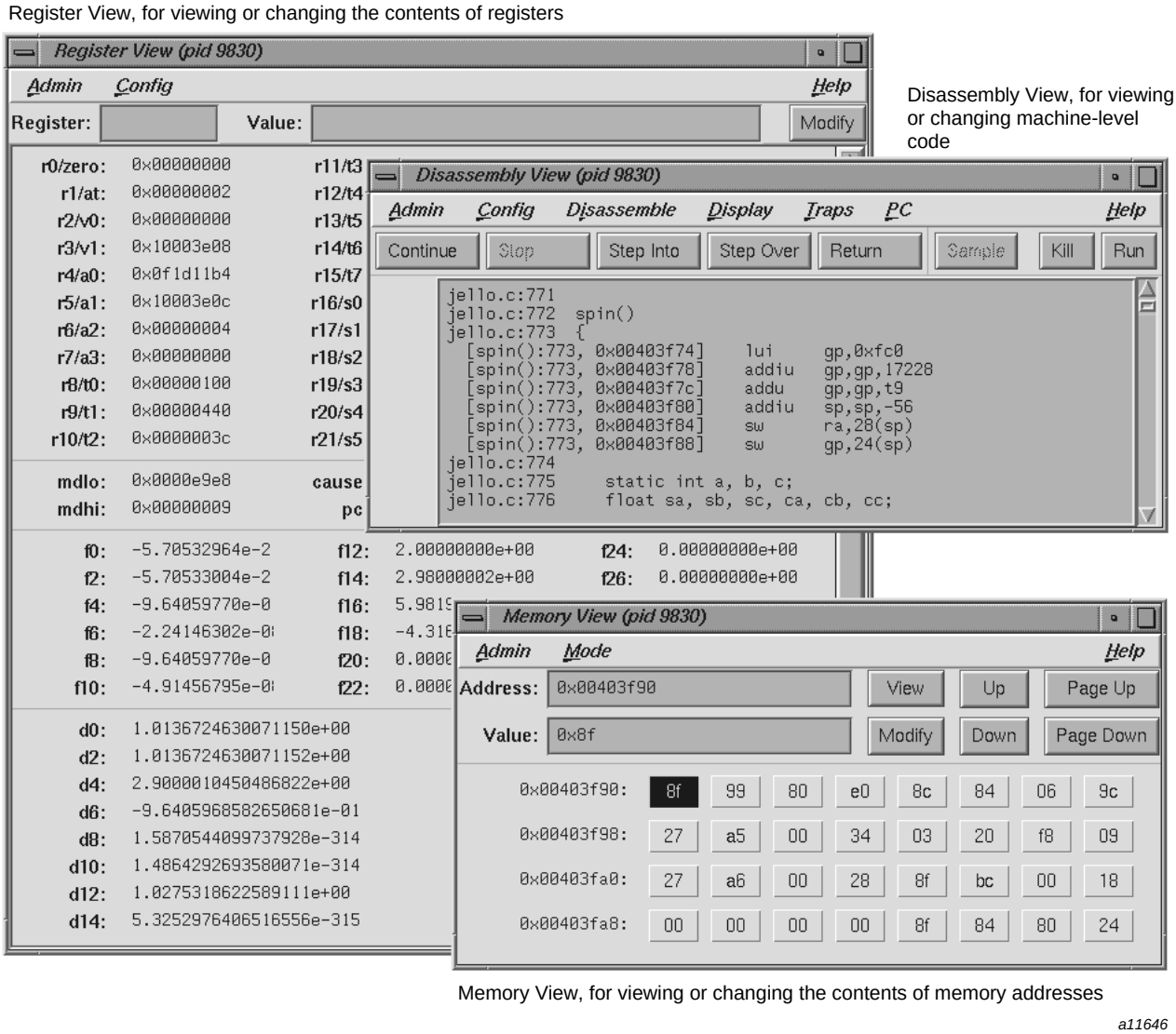

Figure 2-4 Machine-Level Debugger Views

5. Use the control panel options in the Main View to continue execution (see Figure 2-1, page 4).

   From any breakpoint or trap, you have the following options:

   - Click the **Continue** button to run the program until the next breakpoint.
   - Choose the **Continue To** selection in the **PC** menu to proceed to a specified source line. (Placing the cursor on a specific source line also specifies the line.)
   - Choose the **Jump To** selection in the **PC** menu to go to the line specified by the cursor, skipping over any intermediate code.
   - Click the **Step Into** button to continue execution by one line or by a specified number of lines. To specify a number, hold down the right mouse button while the cursor is positioned over the **Step Into** button and select a number from the resulting dialog box. The process then continues the specified number of source lines and enters any called functions.
   - Click the **Step Over** button to continue execution by one line or by a specified number of lines as with the **Step Into** button. The process then continues the specified number of source lines but does not enter any called functions.
   - Click the **Return** button to execute the remaining instructions in a function and stop upon 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, you can check out the source code from the Main View (or the **Source View** window, an alternate editing window).

   Choose **Check Out** from the **Versioning** submenu of the **Source** menu.

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

   Both the Main View and **Source View** allow you to annotate and do simple edits on your code. **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 following line to your `.Xdefaults` file:

   `*editorCommand:` *editor*

   The *editor* option is the command for the editor you want to use.

8. Recompile using the Build Manager.

   The Build Manager has two windows: **Build View** and **Build Analyzer**. **Build View** lets you compile, view compile error lists, and access the problem 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, that is, any `make` facility that runs on SGI platforms.

The following table details where to find more information about the Debugger in the *ProDev Workshop: Debugger User's Guide*.

Table 2-1 Debugger Information Details

Topic	See
General Debugger information	Chapter 1, "WorkShop Debugger Overview"
Basic Debugger Usage	Chapter 2, "Basic Debugger Usage
Debugger interaction with source files	Chapter 3, "Selecting Source Files
Debugger tutorial	Chapter 4, "Tutorial: The `jello` Program"
Comprehensive information about setting breakpoints and traps	Chapter 5, "Setting Traps"
Controlling execution in a process (stepping, jumping, etc.)	Chapter 6, "Controlling Program Execution"
Examining Debugger data in general at the source level	Chapter 7, "Viewing Program Data"
Tracing through the call stack	Chapter 7, "Viewing Program Data"
Entering expressions to be evaluated at stopping points	Chapter 7, "Viewing Program Data "
Fix+Continue	Chapter 8, "Debugging with Fix+Continue"
Using the debugger to trap memory allocation problems	Chapter 9, "Detecting Heap Corruption"
Debugging multiprocesses	Chapter 10, "Multiple Process Debugging"

Topic	See
X/Motif Analyzer	Chapter 11, "X/Motif Analyzer"
Viewing or changing the values of variables	Appendix A, "**Variable Browser**" subsection
Examining data in arrays using the 3D or spreadsheet format	Appendix A, "**Array Browser**" subsection
Determining the data structures of variables	Appendix A, "**Structure Browser**" subsection
Using the Debugger command line	Appendix A, "Debugger Command Line" subsection
Examining debugger data at the machine level	Appendix A, "Machine-level Debugging Windows" subsection

Navigating Through Code

The Static Analyzer is a source code analysis and navigation tool for analyzing source code written in C, C++, Fortran 77, or Ada. The Browser has additional features for Ada and C++, which are described in "Browser User Model", page 18.

Note: Support for Fortran 90 is limited to the MIPSPro Fortran 90 compiler, version 7.1, and newer.

The Static Analyzer shows you code 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 helpful 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 3-1, page 14.

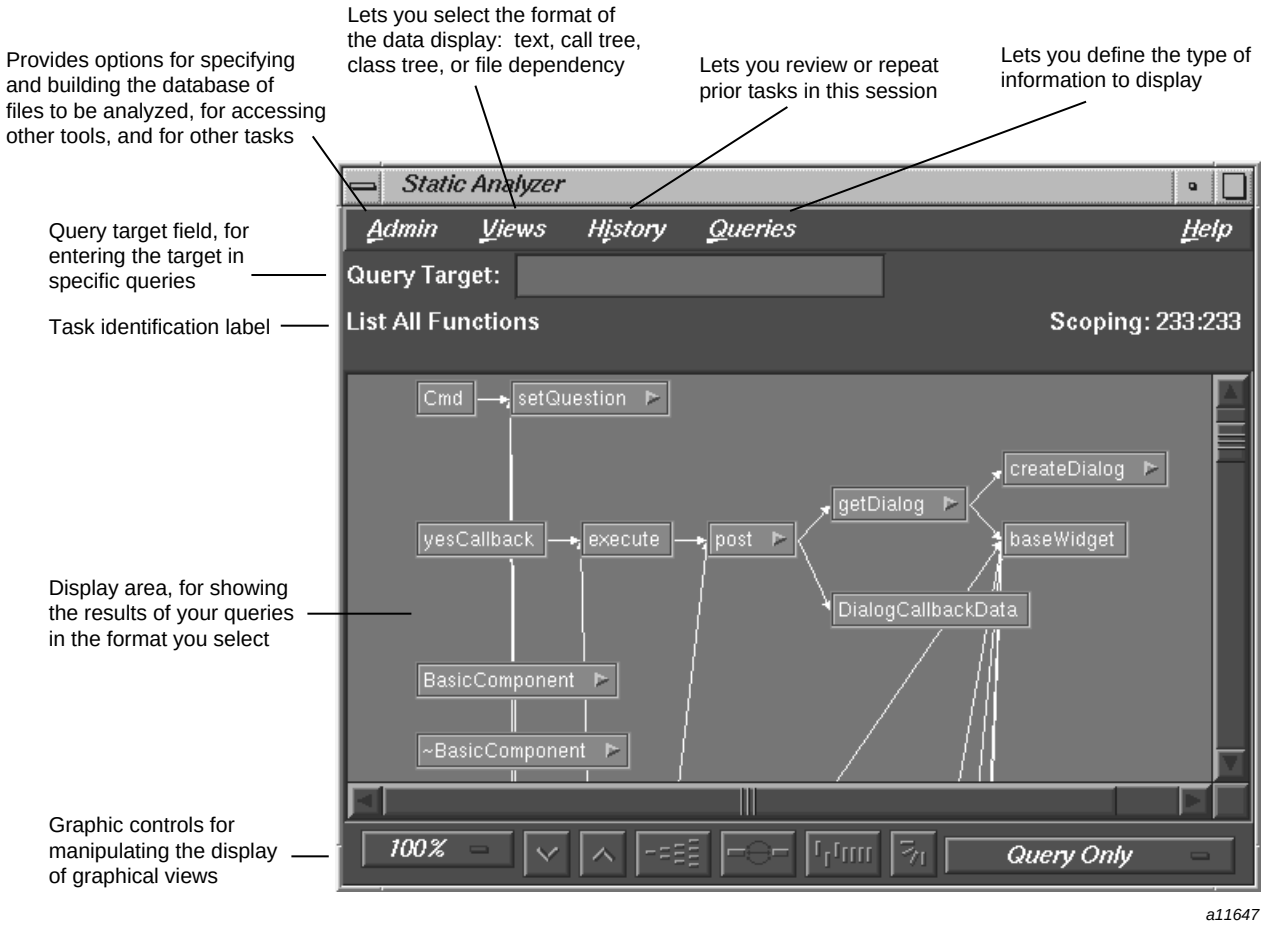

Figure 3-1 Main **Static Analyzer** Window

Static Analyzer User Model

The following steps outline basic static analysis:

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 code is located).

2. Decide which files to analyze.

   You designate which files to analyze 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. Entries can be regular expressions, filenames, or included directories preceded by the designator `-I`.

   To specify a fileset, you can use any of the following methods:

   - 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 `*.[c|C|f|F]`.
   - Let the Static Analyzer create the fileset automatically at startup from the command line by typing **`cvstatic`** with the **`-executable`** flag and designating the name of an executable file.

   Many programs are so large that a query covering the entire scope is not helpful because of the size and complexity of the results. There are two ways to keep the scope of your analysis at a manageable size:

   - Limit the number of files to be analyzed.
   - 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.

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 3-2, page 17.

6. View and save the results.

   The Static Analyzer can present data in several ways that are selected from the **Views** menu. The following displays are available:

   - **Text View** displays query results in a text format. In addition to listing the queried items, it indicates the source file name 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 later.

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 the **Source View** window with the corresponding call site or file inclusion.

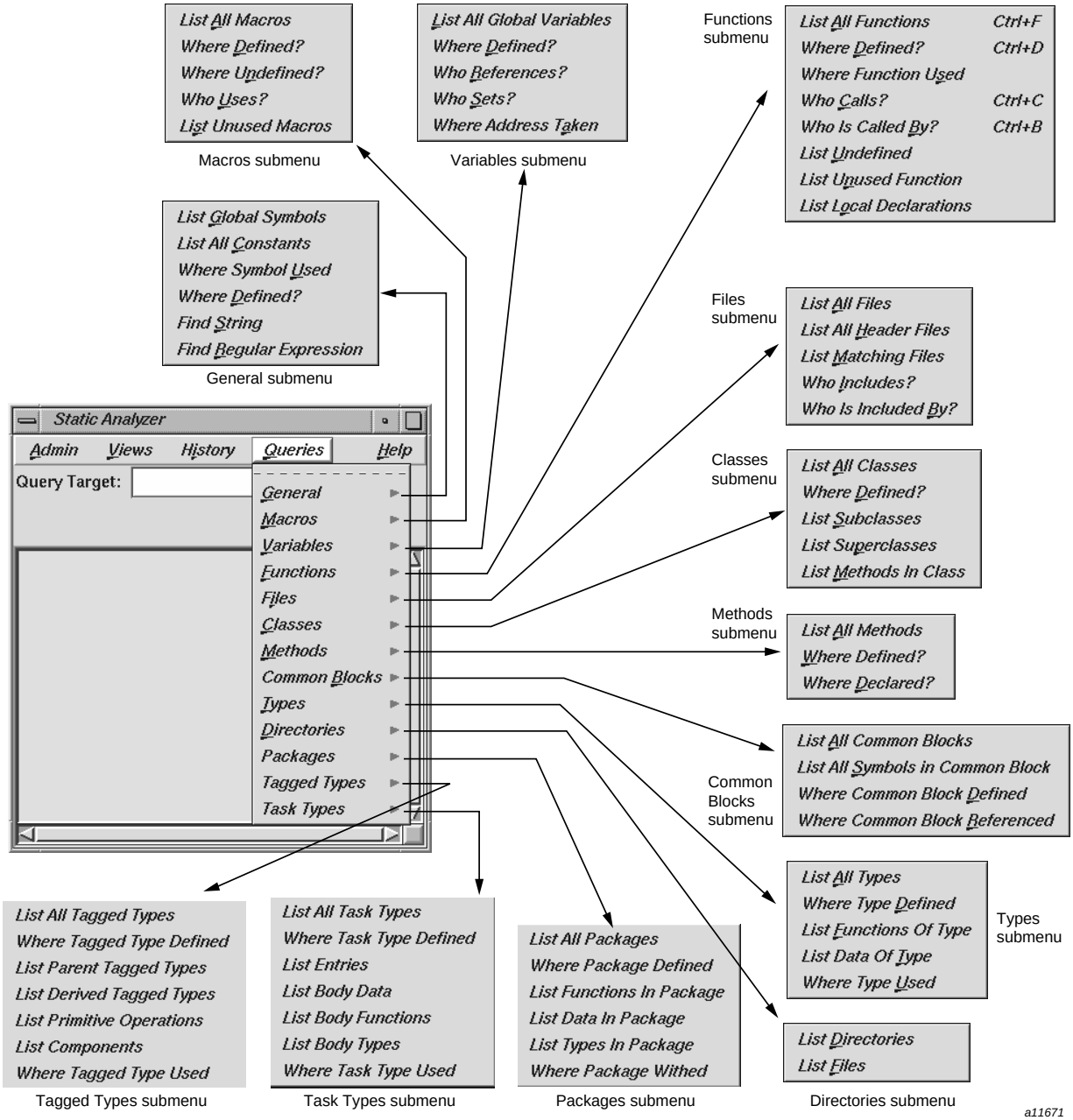

Figure 3-2 Static Analyzer **Queries** Menu with Submenus

The following table details where to find more information about the Static Analyzer in the *ProDev Workshop: Static Analyzer User's Guide*.

Table 3-1 Static Analyzer Information Details

Topic	See ...
General Static Analyzer description	Chapter 1, "Introduction to the WorkShop Static Analyzer"
Static Analyzer tutorial	Chapter 2, "Tutorials for the Static Analyzer"
Creating filesets and a database	Chapter 3, "Creating a Fileset and Generating a Database"
Performing queries	Chapter 4, "Queries"
Static Analyzer viewing formats	Chapter 5, "Views"
Strategies for analyzing large programs	

Browser User Model

The Browser user model is similar to the Static Analyzer user model. After building the database (which must be done in parser mode), 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 the following 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 3-3, page 19). Clicking the diamond-shaped icons next to the headings in the list hides or displays the associated information.

  As in the Static Analyzer, numerous queries are 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.

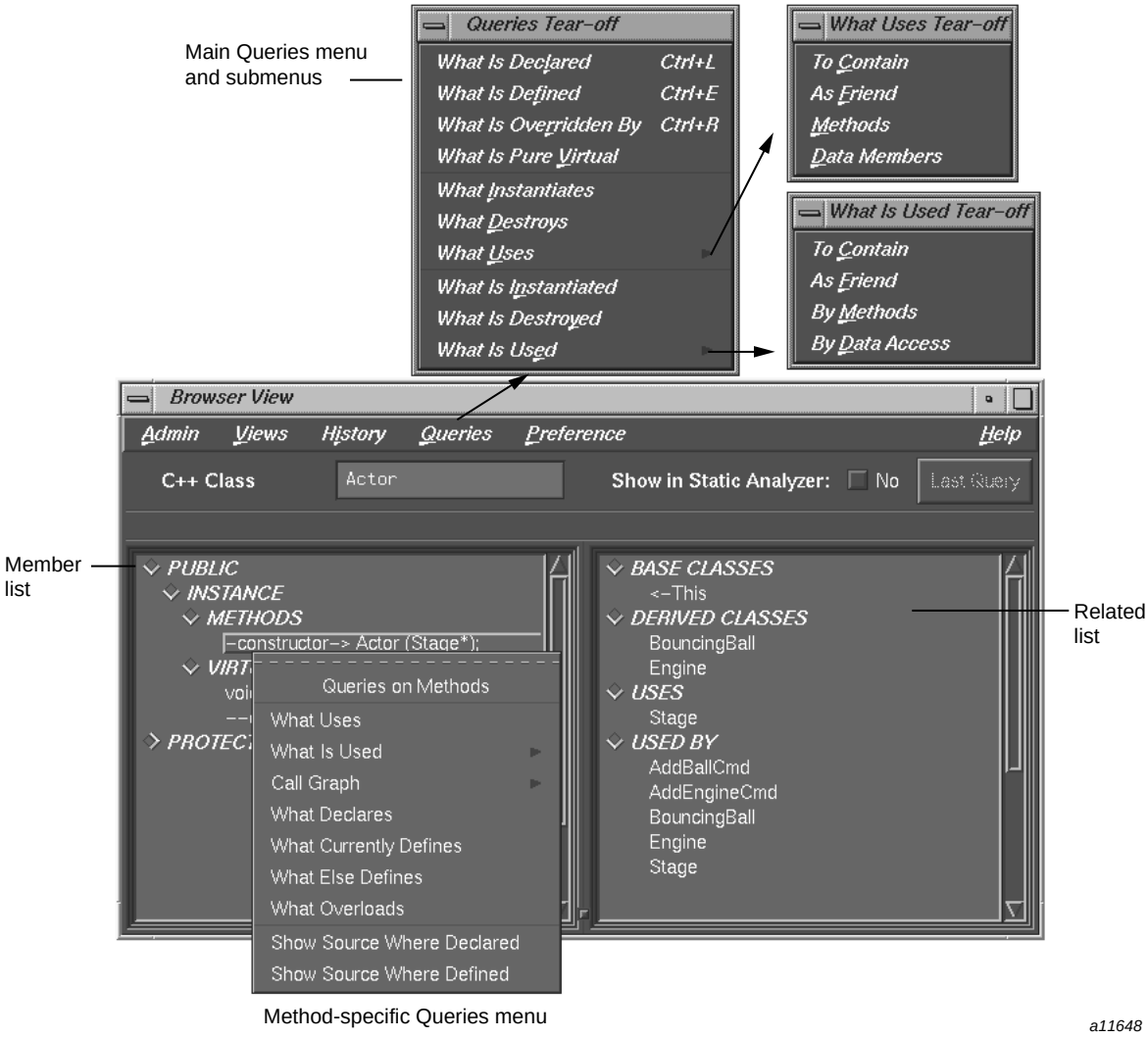

Figure 3-3 Browser View with **Query** Menus with C++ Data

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.

- **Graph Views** window—displays the hierarchy for the subject currently dislayed in the **Browser View** window with nodes as subjects and arcs as relationships. This window 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** window—displays the calling relationships of methods, virtual methods, or functions selected from **Browser View** with options for customizing the display of the graph.

The following table details where to find more information about the Browser in the *ProDev Workshop: Static Analyzer User's Guide*.

Table 3-2 Browser Information Details

Topic	See ...
General Browser description	Chapter 7, "Getting Started with the Browser"
C++ Browser tutorial	Chapter 8, "Browser Tutorial for C++"
Ada Browser tutorial	Chapter 9, "Browser Tutorial for Ada"
Detailed reference information	Chapter 10, "The Browser Reference"

Pinpointing Performance Problems

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 you can 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: counts the exact number of times each function or basic block has been executed. This requires *instrumenting* the program, that is, inserting code into the executable file to collect counts.
- Profiling: periodically examines and records a program's program counter (PC), call stack, and resource consumption.
- Tracing: traces events that affect performance, such as reads and writes, MPI calls, system calls, page faults, floating-point exceptions, and mallocs, reallocs, and frees.

Performance Analyzer User Model

The Performance Analyzer can record a number of different performance experiments, each of which provides one or more measures of code performance.

1. To set up a performance experiment, select a task from the **Select Task** submenu on the **Perf** menu in the Debugger Main View. The **Select Task** menu lets you select among several predefined experiment tasks. If you have not formed an opinion of where performance problems lie, select either the **Profiling/PC Sampling** task or the **User Time/Callstack Sampling** task. They are useful for locating general problem areas within a program.

2. Start the program by clicking the **Run** button in Main View.
3. After the experiment has finished running, 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 the following:

   ```
   % cvperf -exp experimentname
   ```

   Results from a typical performance analysis experiment appear in Figure 4-1, page 23, the main **Performance Analyzer** window, and Figure 4-2, page 24, which shows a subset of the graphs in the **Usage Views (Graphs)** window. From the graphs, you should be able to determine where execution phases occur so that you can set traps between them to sample performance data and events at specified times and events during the experiment.

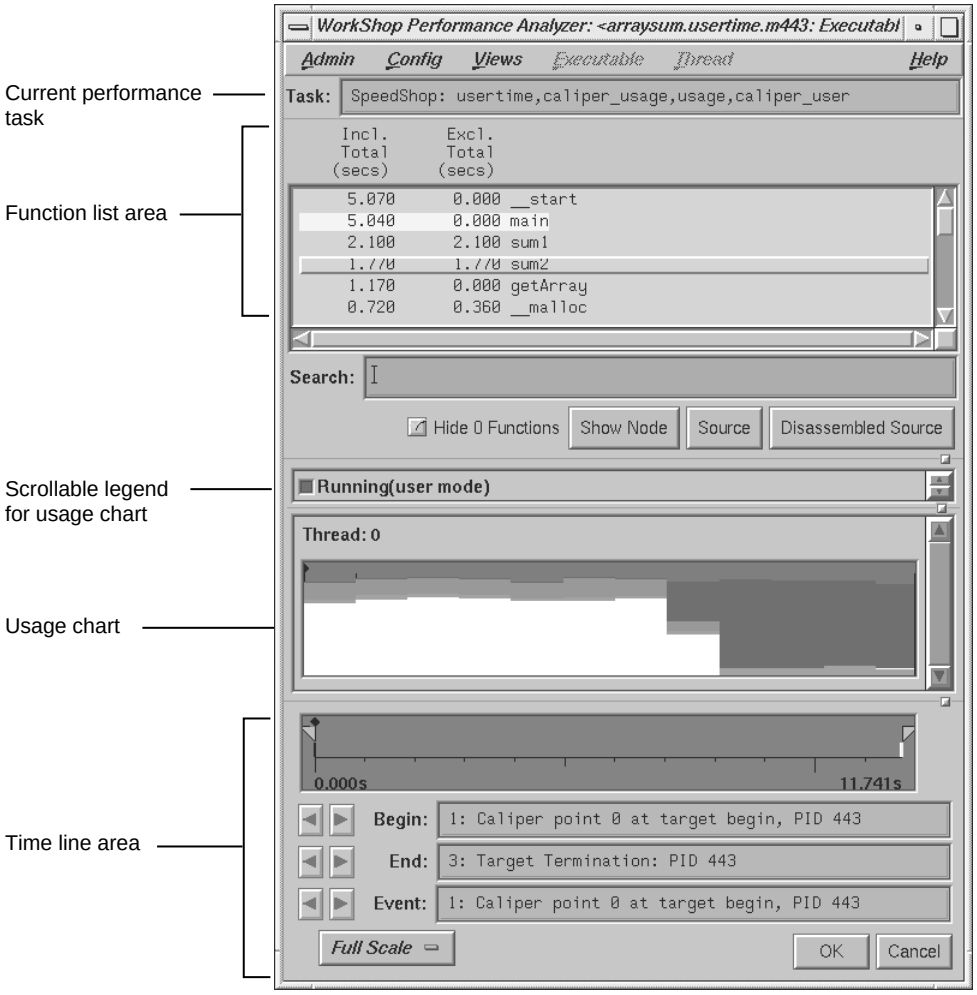

Figure 4-1 Performance Analyzer Main View

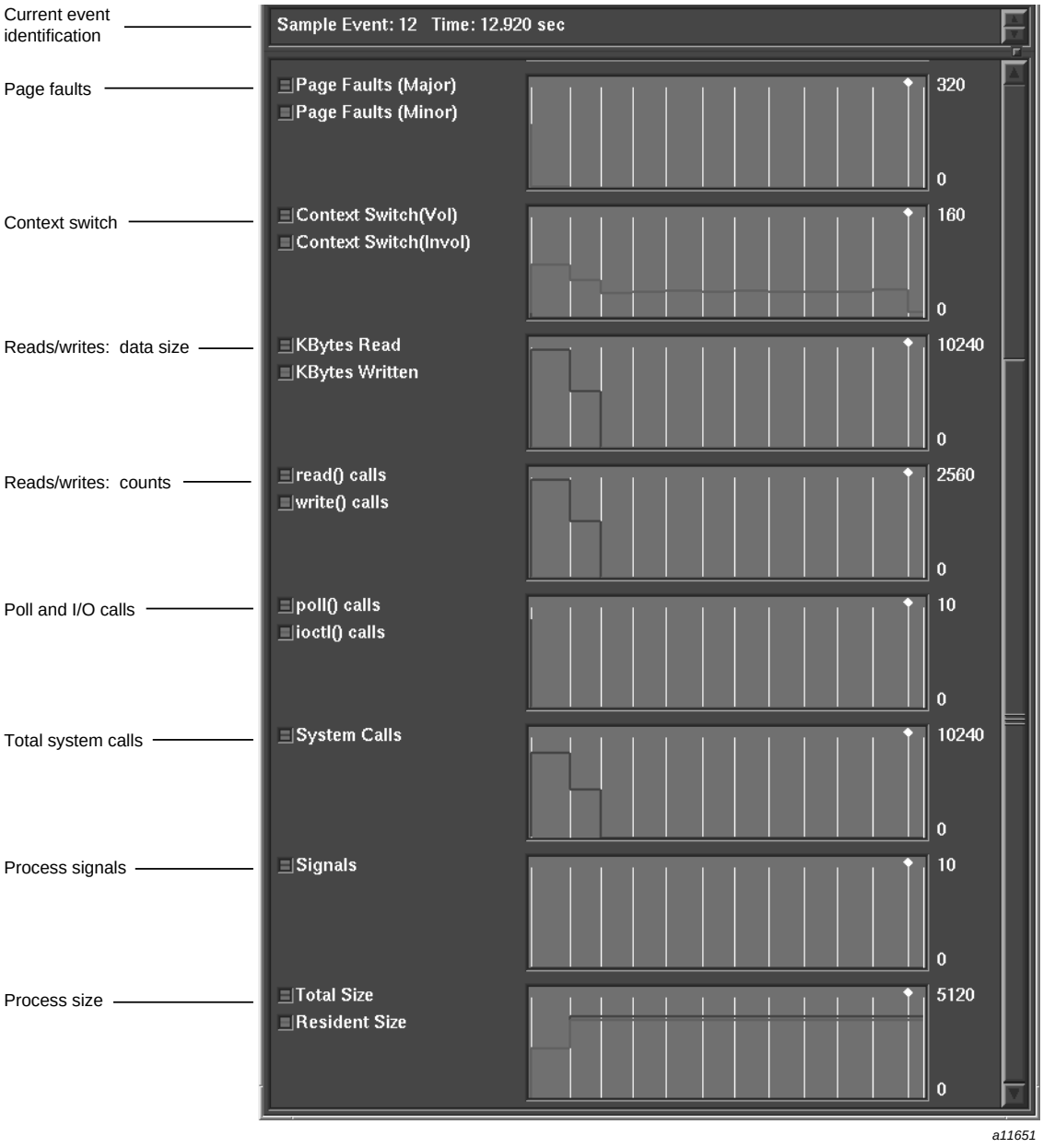

Figure 4-2 Usage View (Graphs) Window: Lower Graphs

4. Setting traps to sample data between execution phases isolates the data to be analyzed on a phase-by-phase basis. To set a sample trap, select **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** window.
5. Select your next experiment from the **Task** menu in the **Performance Pane** and run it by clicking the **Run** button in Main View.

   At this point you need to form a hypothesis about the source of the performance problem and select an appropriate task from the **Select Task** menu for your next experiment.
6. When the results of the second experiment are returned to you, you can analyze the results by using the Main View, any of its views, or **Source View** with performance data annotations displayed.

   The Performance Analyzer provides results in the windows listed in Table 4-1.

Table 4-1 Performance Analyzer Views and Data

Performance Analyzer Window	Data Provided
Performance Analyzer Main View	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 mallocs 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

Performance Analyzer Window	Data Provided
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, which shows instructions, functions, and pages that were not used within the time interval
Butterfly View	The callers and callees of designated functions.
MPI Stats View (Graphs)	A display of various MPI information in the form of graphs.
MPI Stats View (Numerical)	A display of various MPI information in the form of text.
Cord Analyzer	The Cord Analyzer is not actually part of the Performance Analyzer, but it works with data from Performance Analyzer experiments. It allows you to arrange functions in defferent orders to determine the effect on performance.

The following table details where to find more information about the Performance Analyzer in the *ProDev Workshop: Performance Analyzer User's Guide*.

Table 4-2 Performance Analyzer Details

Topic	See
General Performance Analyzer information	Chapter 1, ''Introduction to the Performance Analyzer''
General tutorial	Chapter 2, ''Performance Analyzer Tutorial''
Setting up experiments	Chapter 3, ''Setting up Performance Analysis Experiments'' for details and Chapter 4, ''Selecting Performance Tasks'' heading for a summary
Setting sample traps	Chapter 3, ''Setting Sample Traps'' subsection
Main View	Chapter 4, ''The Performance Analyzer Main Window'' subsection
Usage View (Graphs)window	Chapter 4, ''The **Usage View (Graphs)** Window'' subsection
Watching an experiment using**Process Meter**	Chapter 4, ''The **Process Meter** Window'' subsection

Topic	See
Tracing I/O calls using the **I/O View**window	Chapter 4, ''The **I/O View** Window'' subsection
Call Graph Viewwindow	Chapter 4, ''The **Call Graph View** Window'' subsection
Finding memory problems	Chapter 4, ''Analyzing Memory Problems'' subsection
Call Stack View window	Chapter 4, ''The Call Stack Window''subsection

Testing, Recompiling, and Making Quick Changes

This chapter discusses three tools available with the WorkShop toolset:

- "Determining the Thoroughness of Test Coverage with Tester", page 29, which describes the Tester quality assurance tool.
- "Recompiling with Build Manager", page 33, describes the tool that lets you recompile programs without leaving the WorkShop environment.
- "Making Quick Changes with Fix+Continue", page 33, describes how to make minor changes to your code without recompiling and linking.

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 your test directories set up, you can automate your testing and create larger test sets. Tester has both a command line interface (see Table 5-1, page 30) and a graphical user interface (see Figure 5-1, page 32). Typical steps for running Tester are shown below:

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

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

   This creates a special executable file that is used for testing purposes. This file 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 reports. Most are available through both the command line and the graphical user interfaces. The reports show the following types of information:

   - Arc coverage, that is, coverage of function calls
   - Argument tracing
   - Basic block counts
   - Call graphs indicating caller and callee functions and their counts
   - Count information for assembly language branches
   - Function coverage
   - Source and assembly line coverage
   - Summaries of overall coverage

Table 5-1 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

Command Category	Command Name	Description
	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 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
	cvcov optimize	Selects the minimum set of tests that give the same coverage or meets 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

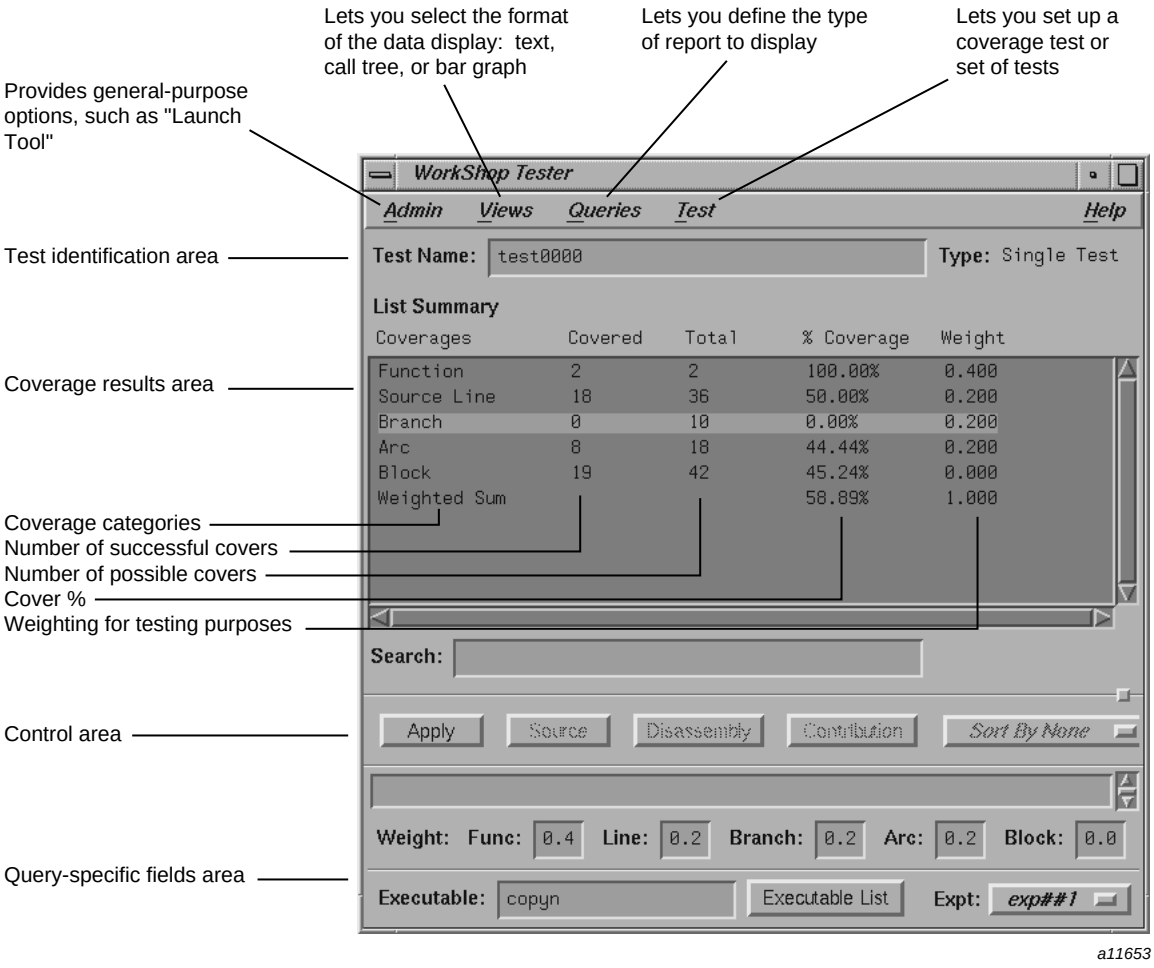

Figure 5-1 Major Areas of the Tester Window

The following table details where to find more information about Tester in the *ProDev Workshop: Tester User's Guide*.

Table 5-2 Tester Information Details

Topic	See ...
Command line interface tutorial	Chapter 2, "Tester Command Line Interface Tutorial"
Graphical user interface tutorial	Chapter 4, "Tester Graphical User Interface Tutorial"
Command line interface details	Chapter 3, "Tester Command Line Reference"
Graphical user interface details	Chapter 5, "Tester Graphical User Interface Reference"

Recompiling with Build Manager

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

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

For more information about the Build Manager, see Appendix B, "Using the Build Manager," in the *ProDev Workshop: Debugger User's Guide*.

Making Quick Changes with Fix+Continue

The Fix+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+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.

Fix+Continue enables you to speed up your development cycle. This is because you are no longer required to rebuild a program each time you make changes to the code,

but can instead replace bad code and simply continue with program execution in a fraction of the time. Fix+Continue lets you perform the following functions:

- Redefine existing function definitions
- Disable, re-enable, 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

Fix+Continue User Model

The basic model steps for using the Fix+Continue feature are as follows:

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

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

2. Find the function that you want to change.

   You can find a function in numerous ways. You can select **Search...** from the **Source** menu, type **func** *functionname* at the Debugger command line, or scroll to the location.

3. Select **Edit** from the **Fix+Continue** menu.

   This turns on edit mode and highlights the source code of the selected function. If line numbers are displayed, those in the selected function appear with a two-part number separated by a decimal point. The left part of the number represents the starting line number of the function in the source file before you selected **Edit**. The right part of the number is the renumbered source line within the function. This numbering scheme makes it easy to keep track of new lines that have been added.

4. Make your changes to the source code.

   You can do this directly in the Main View or you can use an editor of your choice by selecting **External Edit** from the **Fix+Continue** menu.

5. Try your changes.

   Selecting **Parse And Load** adds your changes to the executable file that you are debugging. The changed function will be 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 current fixes in the current file. **Update All Files...** saves all fixes in the current session.

At any point, you can compare the new code to the 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.

The following table details where to find more information about Fix+Continue in the *ProDev Workshop: Debugger User's Guide*.

Table 5-3 Fix+Continue Information Details

Topic	See ...
General information and tutorial	Chapter 8, "Debugging with Fix+Continue"
Detailed command information	Appendix A, "Fix+Continue Menu" subsection

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 widget examiner mode. You can switch to other examiners through the **Examiner** menu or by clicking the tabs at the bottom of the window (See Figure 6-1).

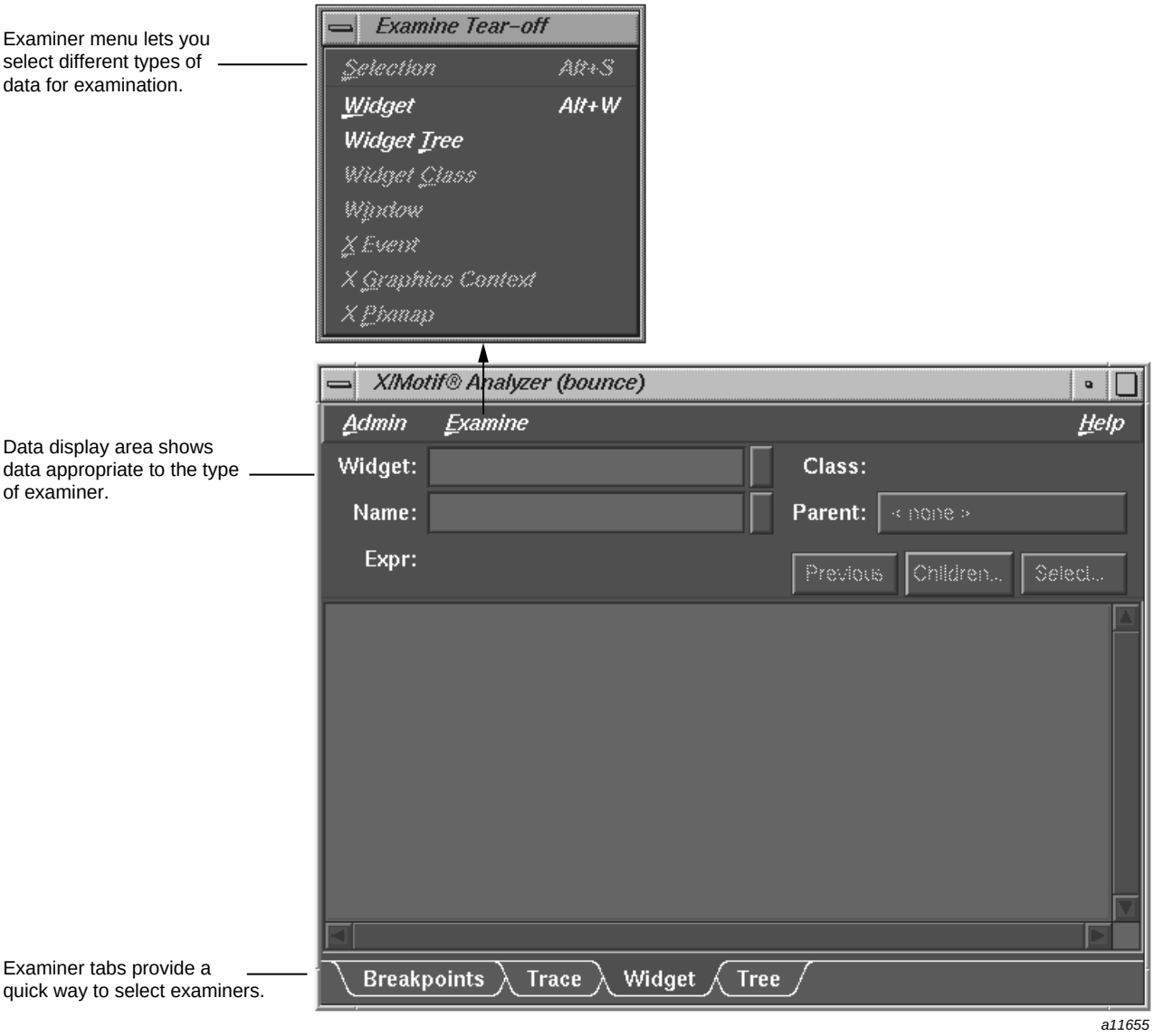

Figure 6-1 The **X/Motif Analyzer** Window

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 6-2). You can display widgets by name, class, or ID by selecting from the widget display menu. Double-clicking a widget node switches to the widget examiner and displays data for the selected widget.

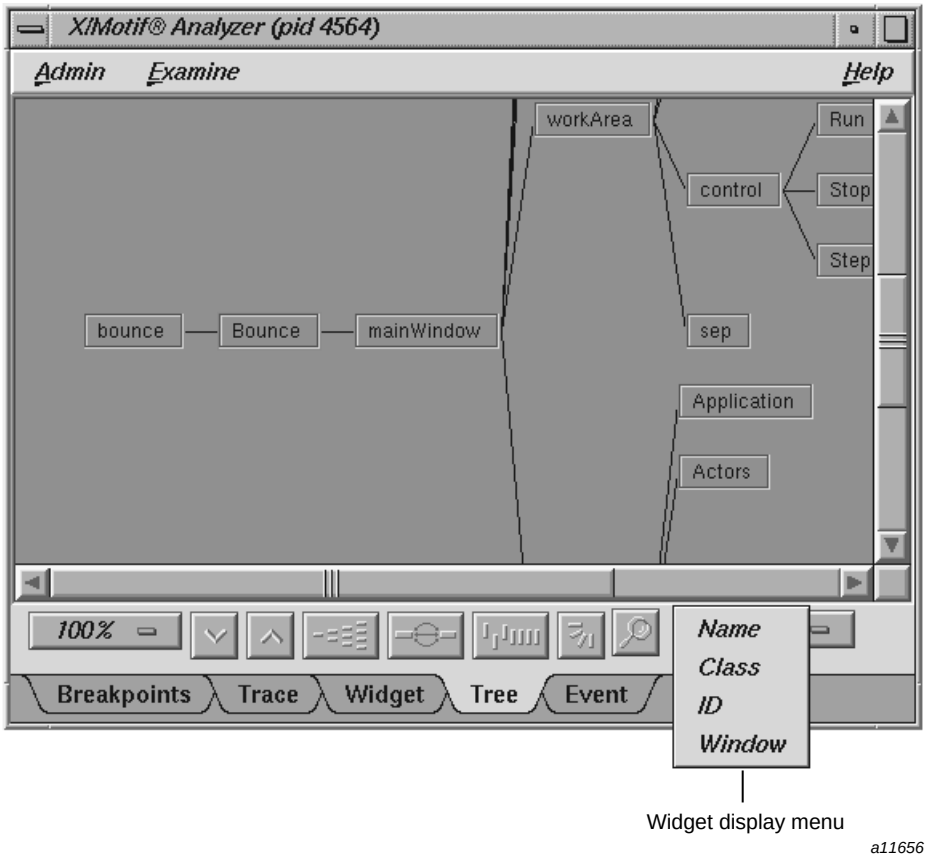

Figure 6-2 X/Motif Analyzer Widget Tree Examiner

- Breakpoints examiner—lets you set breakpoints at the widget and widget–class level. You can set breakpoints at the following levels:
  - Callback functions
  - Widget events
  - Resource changes
  - 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
  - X callbacks

  Figure 6-3, page 42, 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 a process stops in a callback. It displays the following information:
  - Callstack frame for the callback function
  - Widget information
  - 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 X event 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.

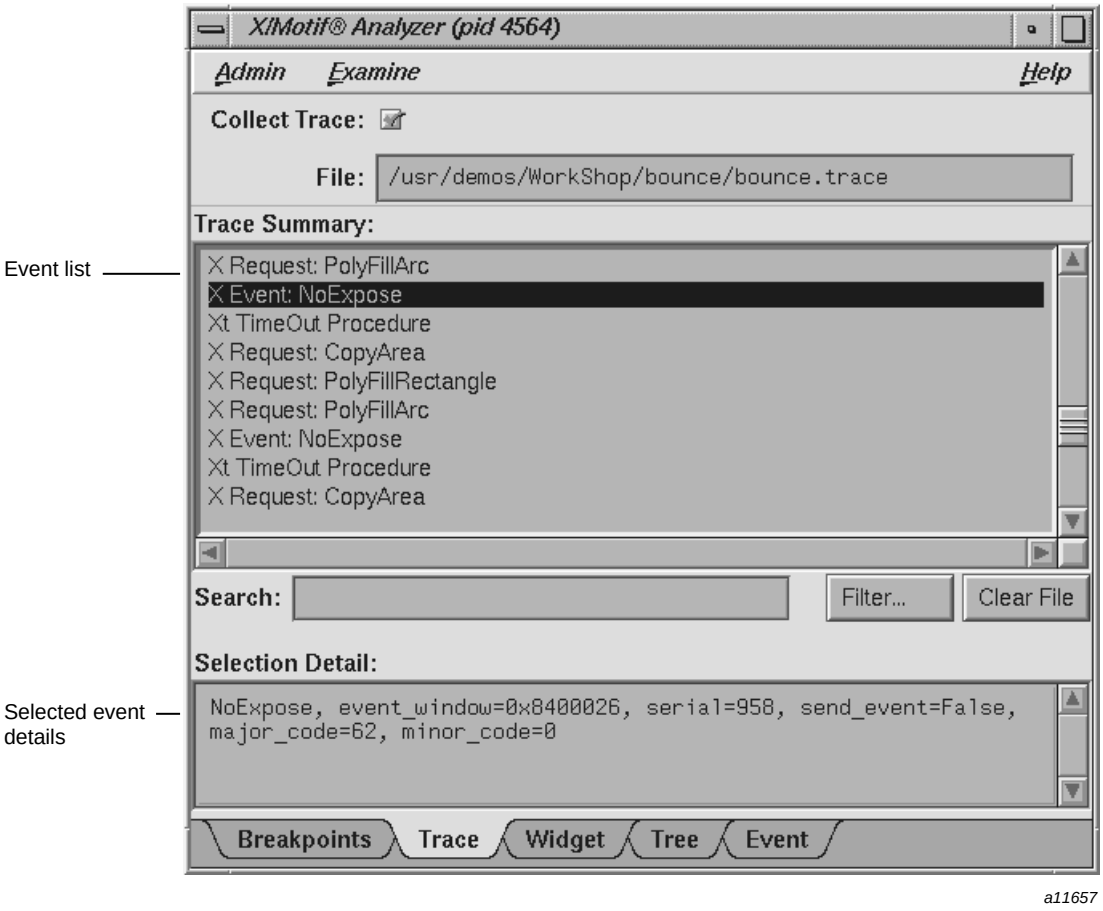

Figure 6-3 X/Motif Analyzer with Trace Examiner Data

The following table details where to find more information about the X/Motif Analyzer in the *ProDev Workshop: Debugger User's Guide*.

Table 6-1 X/Motif Analyzer Information Details

Topic	See ...
General information and tutorial	Chapter 11, “Using the X/Motif Analyzer: X/Motif Analyzer”
Detailed reference information	Appendix A, "X/Motif Analyzer Windows” subsection
Setting breakpoints to capture widget-level information	Appendix A; "Breakpoints Examiner” subsection
Tracing widget-level data through the execution of a program	Appendix A, "Trace Examiner” subsection
Getting information on a specified widget	Appendix A, "Widget Examiner” subsection
Displaying a graph of the widget hierarchy	Appendix A, "Tree Examiner” subsection
Getting information on a specified callback	Appendix A, "Callback Examiner” subsection
Getting information on a specified window	Appendix A, "Window Examiner” subsection
Getting information on a specified X event	Appendix A, "Event Examiner” subsection
Getting information on a specified graphics context	Appendix A, "Graphics Context Examiner” subsection
Getting information on a specified pixmap	Appendix, "Pixmap Examiner” subsection

Appendix A

Using Graphical Views

Many tools in the ProDev WorkShop toolkit 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

Graphical views provide data overviews that show the relationships between entities. 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 brings up **Source View**, which displays the corresponding code in a format that is 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 are be enabled.

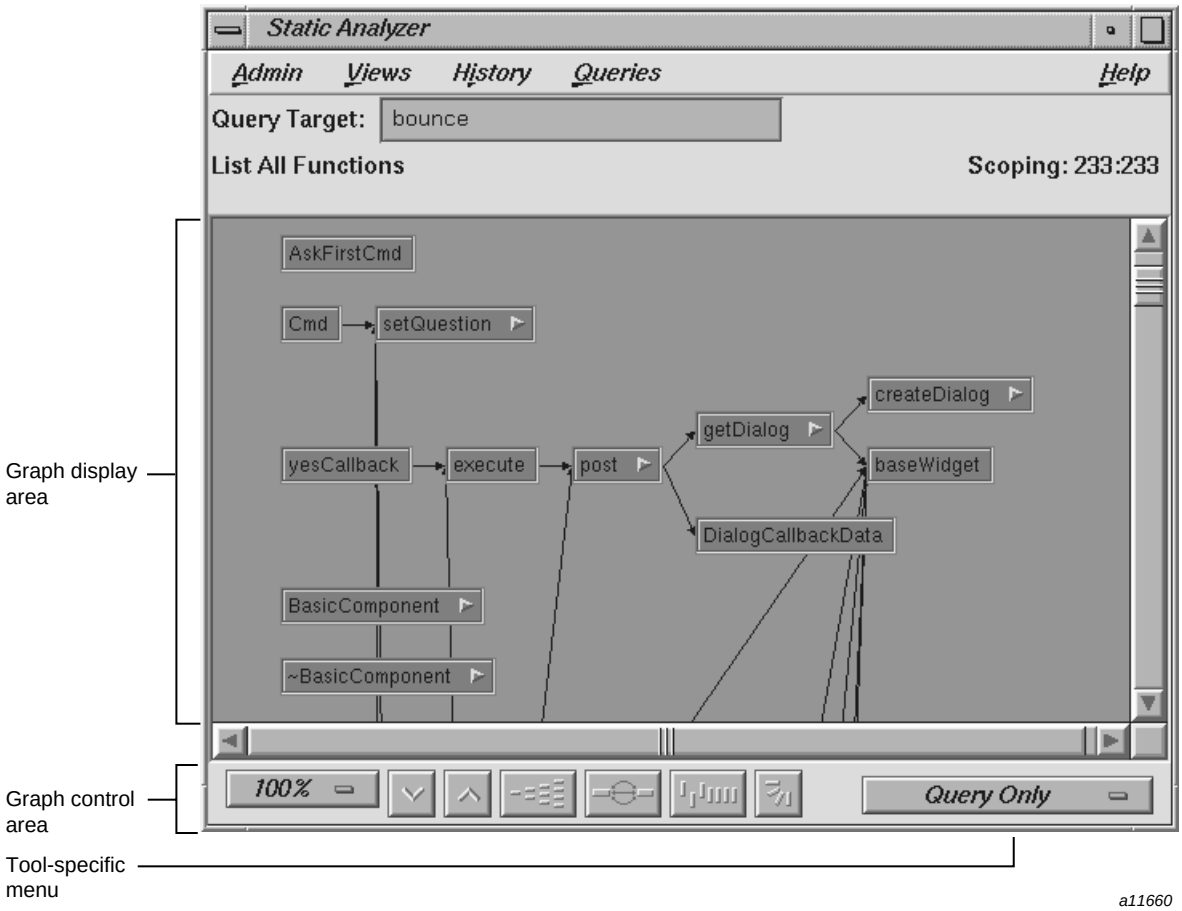

Figure A-1 Typical Graphical View

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 the 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.

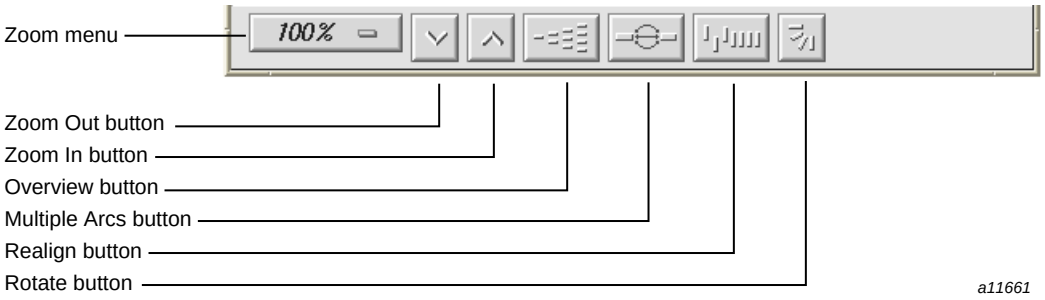

Figure A-2 Graph Display Controls

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 as follows:

- 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 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 for more details.
- Multiple Arcs button: toggles between single and multiple arc mode. Multiple arc mode is extremely useful for the **List Arcs** query, because it graphically indicates 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, click the **overview** button.

Figure A-3 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.

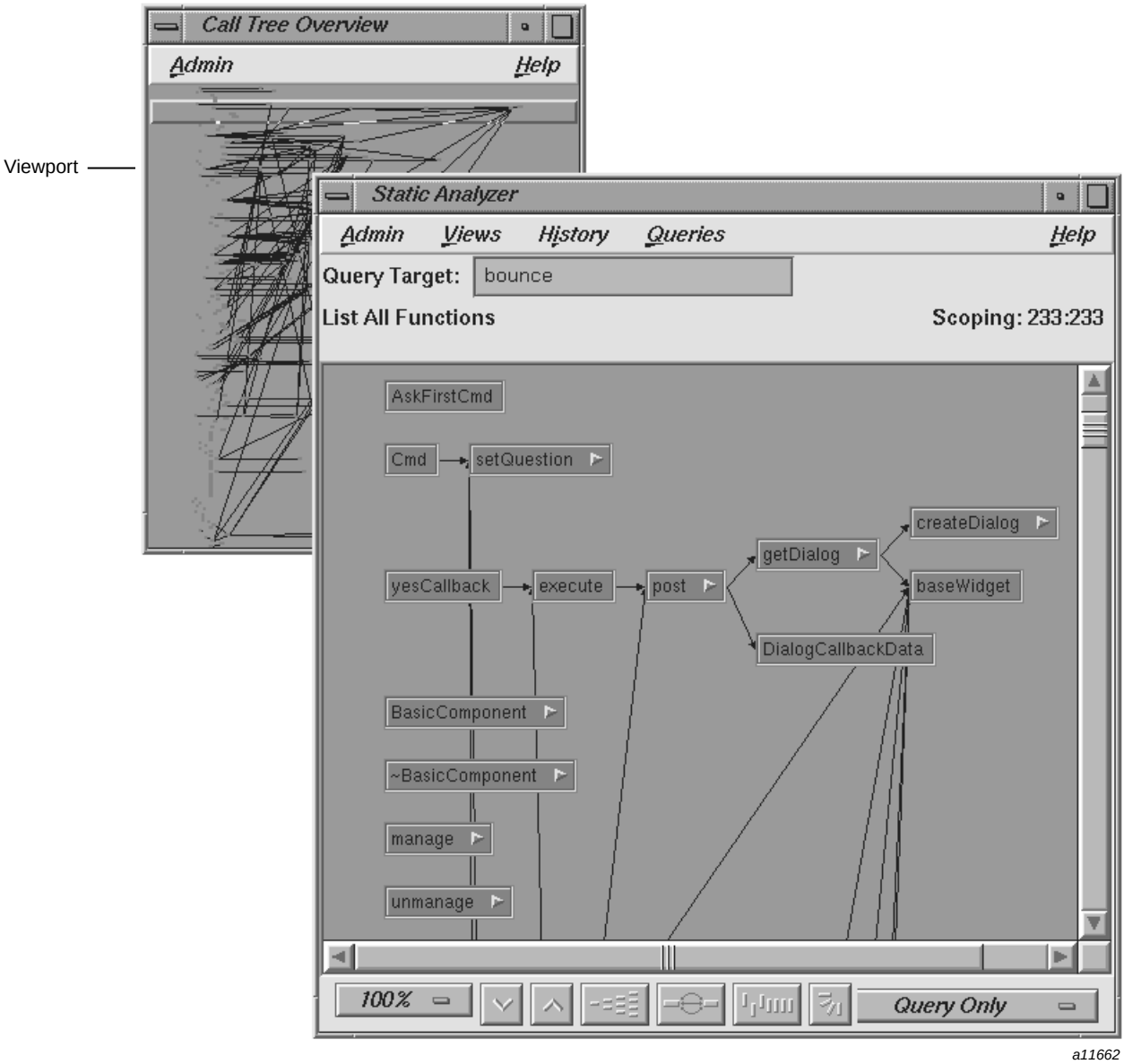

Figure A-3 Overview Window with Resulting Graph

The **Overview** window has an **Admin** menu with the following 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.

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 way to simplify 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-4.

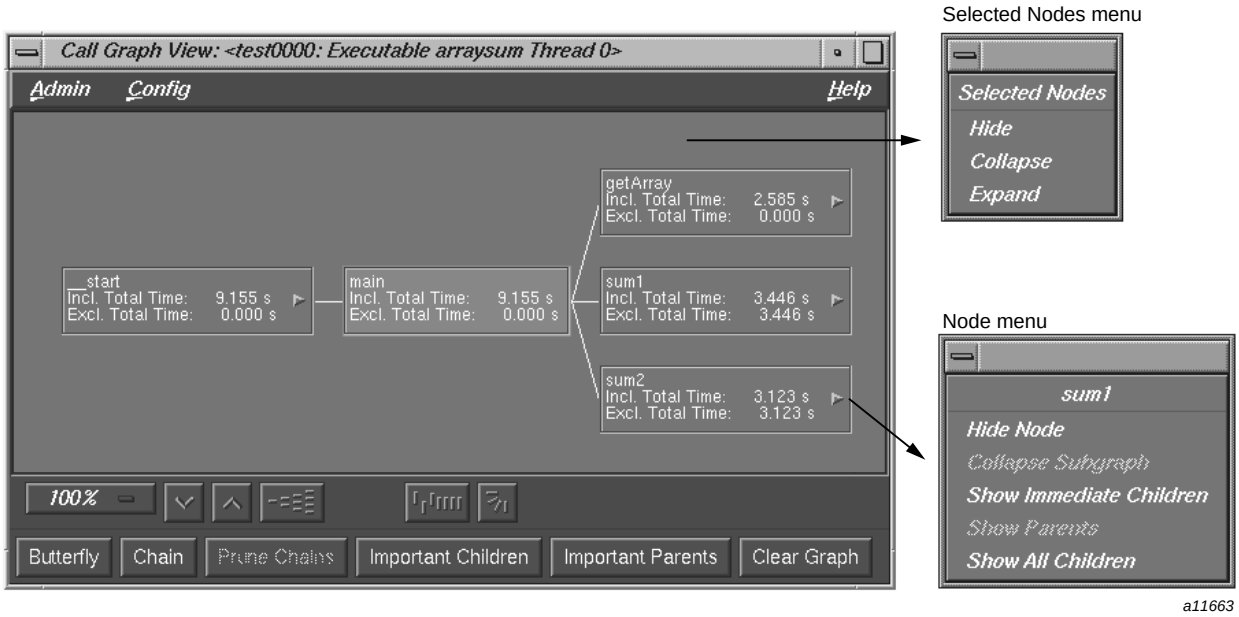

Figure A-4 Node Pop-up Menus

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 contains the following selections:

- **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 has the following selections:

- **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 the ProDev WorkShop Toolkit

If you want to change the appearance of the ProDev WorkShop windows, we recommend that you start with the following menus that are 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** and **Format** popup menus
- Build Analyzer: **Filter**, **Selected Node** and **Node** popup menus, and the graphic controls

- Build View: **Preferences...** and **Build Options...** selections from the **Admin** menu
- Performance Analyzer: **Config** menus in all views, **Selected Node** and **Node** popup menus, and graphic controls in graphical views
- Static Analyzer: **Views** menu, **Selected Node** 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 Window System Resources

While there are hundreds of X Window System resources that you can change but we recommend that you avoid modifying these recources if at all possible. In some cases, there may be no way within WorkShop to make the desired change. If you must modify resources, the following X Window System resources for the Debugger and its views may be useful:

`*autoStringFormat`

If set to 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 hexadecimal address.

`cvmain*sourceView*nameText.columns`

Sets the length of the File field in the Main View. The default is 30 characters.

`Cvmain*disableLicenseWarnings` and `*disableLicenseWarnings`

Disables the license warning message that displays when you start the Debugger and the other tools.

`*editorCommand`

If you prefer to view source code in a text editor rather than in **Source View**, lets you specify a text editor. The default is the `vi` editor.

`*expressionView*maxNumOfExpr`

Lets you set the maximum number of expressions that can be read from a file by Expression View. The default is 25.

`*varBrowser*maxSymSize`

Lets you set the maximum number of variables that can be displayed by the Variable Browser. The default is 25.

The following resources apply to **Source View**:

`*svComponent*lineNumbersVisible`

Displays source line numbers by default.

`*sourceView*nameText.columns`

Sets the length of the **File** field in **Source View**. The default is 30 characters.

`*sourceView*textEdit.scrollHorizontal`

If set to true, displays a horizontal scroll bar in **Source View**.

`*tabWidth`

Sets the number of spaces for tabs in **Source View**.

The following resource applies to **Build View**:

`*buildCommand`	Is used to determine which program to used with `make`(1), `smake`(1), `clearmake`(1), and so forth.). The default value is `make`(1).
`*runBuild`	Specifies whether `cvmake`(1) begins its build immediately upon being launched. The default value is true.

To change these resources, you need to set the desired value in your `.Xdefaults` file, and rerun the `xrdb`(1) command, if you use it. 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 (breakpoint) and watchpoint (data-breakpoint).

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 tool 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 Performance Analyzer view that shows functions, their calls, and associated performance data. See also call graph and C++ Browser.

Call Stack View

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** view 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** view causes the **Source View** window to be displayed showing the corresponding source code.

Call Tree View (Static Analyzer version)

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

Call Tree View (Tester version)

A Tester view that displays function coverage information in a call graph. See also Tester.

Call View

A C++ Browser view for displaying member and class information. See also C++ Browser.

Class Graph

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

Class Tree View

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

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 file 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 file. `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 file 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 file that brings up the Performance Panel, a window for setting up Performance Analyzer experiments. See also Performance Panel.

cvstatic

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

cvxcov

The name of the executable file 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 one. The cycle count for any trap can be set through the **Trap Manager** view in the Debugger.

Debugger

A tool in the ProDev WorkShop toolkit used 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 examine data by displaying special windows called views. 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 Silicon Graphics 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 the WorkShop environment. It is 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 77, Fortran 90, C, or C++), and specify the format for the resulting values.

File Dependency View

A Static Analyzer view 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 is 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 ten 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.

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.

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 the ProDev WorkShop toolkit used 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`(1).

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 the ProDev WorkShop toolkit 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`(1) command but provide a more specific search. For example, you can perform a query to find where a method is defined. See also Static Analyze

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 compliable. 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 the ProDev WorkShop toolkit used 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`(1), **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 file.

Tester

A tool in the ProDev WorkShop toolkit used 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 Static Analyzer view that displays the results of queries as a scrollable text list. See also Static Analyzer.

Text View (Tester version)

A Tester view 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 to allow the debugger to get control at specified points and conditions in a live process. More commonly referred to as a breakpoint (either a code breakpoint or a data-breakpoint [watchpoint]).

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 while halting the process only briefly (and continuing execution automatically). 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 report 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

Commonly referred to as a data-breakpoint. A trap that fires when a specified variable or address is read or written.

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.

Index