CASEVision™/WorkShop User's Guide

About This Guide

This manual is a user's guide for CASEVision/WorkShop. It is contained in the following chapters:

- **Chapter 1, "Using the Build Manager"**
- **Chapter 2, "Introduction to the WorkShop Static Analyzer"**
- **Chapter 3, "A Sample Session with the Static Analyzer"**
- **Chapter 4, "Static Analyzer: Creating a Fileset and Generating a Database"**
- **Chapter 5, "Static Analyzer: Queries"**
- **Chapter 6, "Static Analyzer: Views"**
- **Chapter 7, "Static Analyzer: Working on Large Programming Projects"**
- **Chapter 8, "Getting Started with the WorkShop Debugger"**
- **Chapter 9, "A Short Debugger Tutorial"**
- **Chapter 10, "Debugger: Managing Source Files"**
- **Chapter 11, "Debugger: CASEVision Project Session Management"**
- **Chapter 12, "Debugger: Setting Traps in WorkShop"**
- **Chapter 13, "Debugger: Controlling Process Execution"**
- **Chapter 14, "Examining Debugger Data"**
- **Chapter 15, "Machine–level Debugging"**
- **Chapter 16, "Debugger: Detecting Heap Corruption"**
- **Chapter 17, "Multiple Process Debugging"**
- **Chapter 18, "Introduction to the Performance Analyzer"**
- **Chapter 19, "Performance Analyzer Tutorial"**
- **Chapter 20, "Setting Up Performance Analysis Experiments"**
- **Chapter 21, "Performance Analyzer Reference"**
- **Chapter 22, "Using Tester"**
- **Chapter 23, "Tester Command Line Interface Tutorial"**
- **Chapter 24, "Tester Command Line Reference"**
- **Chapter 25, "Tester Graphical User Interface Tutorial"**
- **Chapter 26, "Tester Graphical User Interface Reference"**

Appendix A, "Graphical Views in the CASEVision Environment" provides a generic description of how to use graphical views, which are common to many CASEVision tools.

Appendix B, "Customizing WorkShop Tools" describes how you can customize the operation and appearance of WorkShop tools.

Please refer to the table of contents for further descriptions of each chapter.

Chapter 1

Using the Build Manager

WorkShop lets you compile software without leaving the WorkShop environment. Thus, you can look for problems using the WorkShop analysis tools (Static Analyzer, Debugger, and Performance Analyzer), make changes to the source, suspend your testing, and run a compile. WorkShop provides two tools to help you compile:

- **"Build View"**—for compiling, viewing compile error lists, and accessing the code containing the errors in Source View (the CASEVision editor) or an editor of your choice. Build View helps you find files containing compile errors so that you can quickly fix them, recompile, and resume testing.
- **"Build Analyzer"**—for viewing build dependencies and recompilation requirements and accessing source files.

Build View uses the UNIX® *make* facility as its default build software. Although Build Analyzer determines dependencies using *make*, you can plug in the build software of your choice.

Build View

You can access Build View from the WorkShop analysis tools, from a command line (by typing `cvmake`), or from Build Analyzer (see next section).

To access Build View from WorkShop, select "Recompile" from the Source menu in the Main View window in the Debugger or from the File menu in Source View (for more information on Main View and Source View, refer to **Chapter 8, "Getting Started with the WorkShop Debugger"**). Selecting "Recompile" detaches the current executable from the WorkShop analysis tools and displays Build View. You can edit the *Directory* and *Target(s)* fields as needed and click *Build* to compile. If the source compiles successfully, the new executable is reattached when you reenter the WorkShop analysis tools.

The Build View window is shown in **Figure 1-1** with its Admin menu.

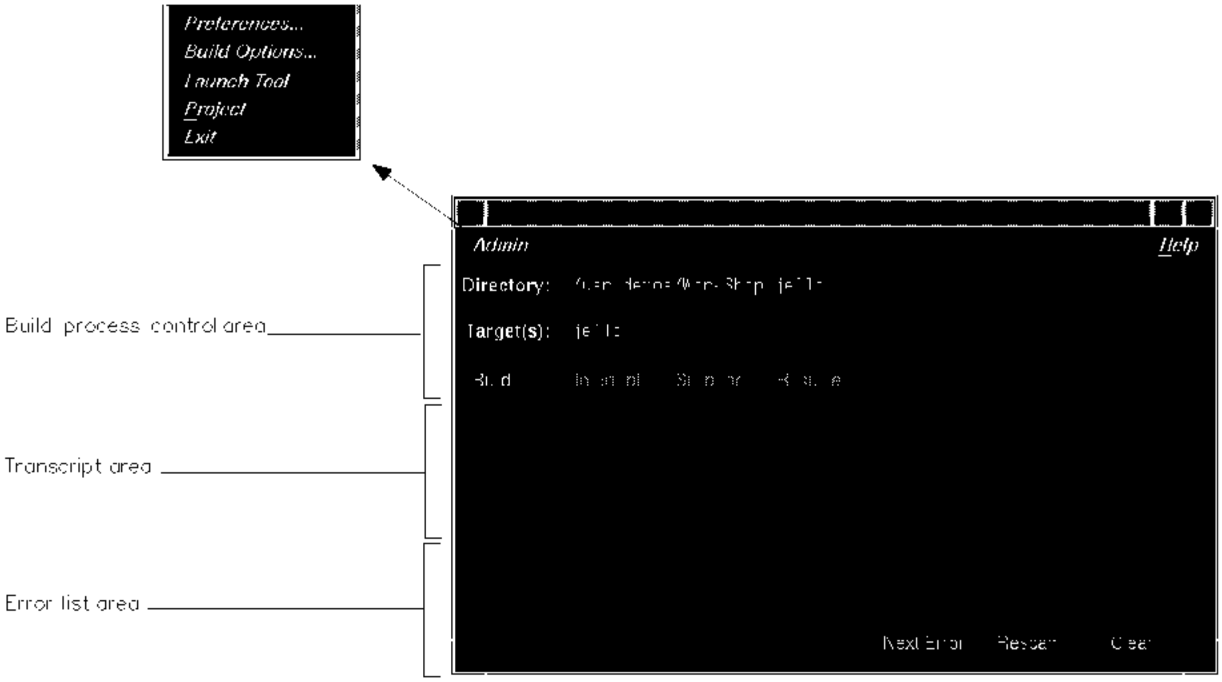


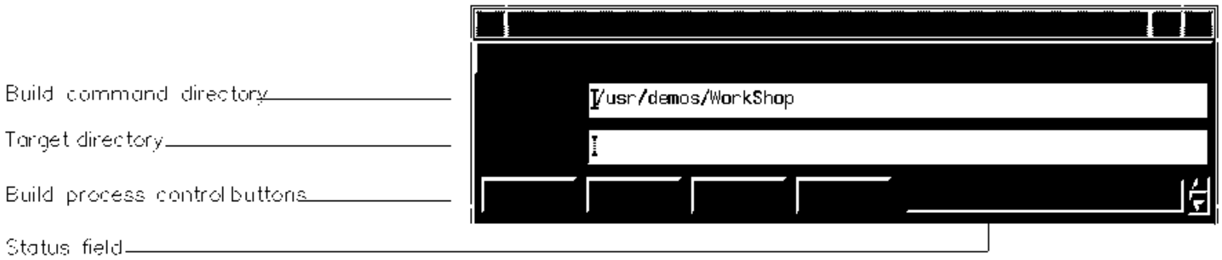
Figure 1-1 Build View Window with Admin Menu Displayed

The Build View window has three major areas:

- **"Build Process Control Area"**
- **"Transcript Area"**
- **"Error List Area"**

Build Process Control Area

The *build process control area* lets you run or stop the build and view the status. See **Figure 1-2**.



/usr/demos/WorkShop

Figure 1−2 Build Process Control Area in Build View Window

The directory in which the build will run displays in the *Directory* field at the top of the area. The current directory displays by default. You can specify the build using *make*, *smake*, *pmake*, *clearmake*, or any other builder and any flags or options that the builder understands (see **"Build View Preferences"** and **"Build Options"**). The target to be built is specified in the *Target(s)* field.

The *build process control buttons* let you control the build process. The buttons are:

Build

runs (or reruns) a build.

Note: If you have modified any files in Source View, you will be prompted to save the new version prior to the compile.

Interrupt

terminates a build.

Suspend

stops a build temporarily.

Resume

restarts a suspended build.

The *status field* is to the right of the build process control buttons. It indicates the progress of the build.

Transcript Area

The *transcript area* displays the verbatim output from the build. The vertical scroll bar lets you go through the list; the horizontal scroll bar lets you see long messages obscured from view. A sash between the compile transcript area and the error list area lets you adjust the lengths of the lists displayed. See **Figure 1−3** .

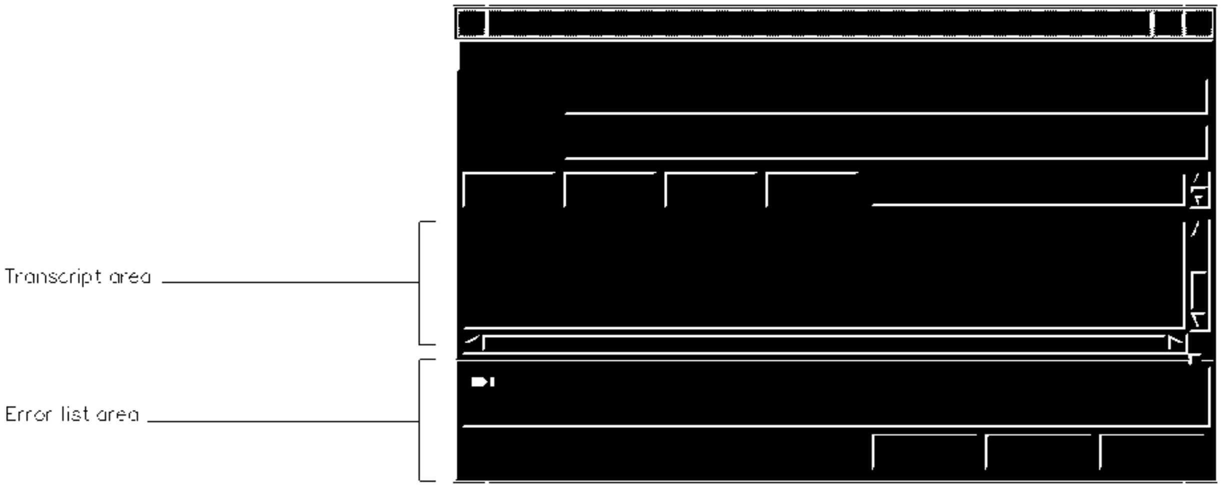


Figure 1−3 Build View Window with Typical Data

Error List Area

The *error list area* consists of the error list display and three control buttons. The buttons are:

Next Error

brings up the default editor scrolled to the next error location. This button is below the error list display.

Rescan

refreshes the error list display.

Clear

clears the error list display area.

The error list area displays compile errors (see **Figure 1−3**). The errors are annotated according to their severity level (fatal has a solid icon and the warning icon is hollow). Double−clicking the text portion of an error brings up the default editor scrolled to the error location and displays a check mark to help you keep track of where you are in the error list. Check marks also display when you click the *Next Error* button.

Build View Admin Menu

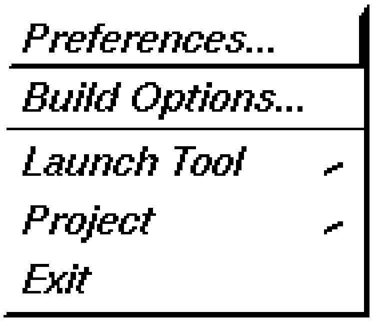


Figure 1−4 Admin Menuin Build View Window

The Admin menu in Build View (see **Figure 1−4**) has two selections in addition to the standard WorkShop entries:

- **"Build View Preferences"**
- **"Build Options"**

For more information on "Launch Tool" and "Project"menu selections, refer to **Chapter 8, "Getting Started with the WorkShop Debugger"** .

Build View Preferences

The "Preferences..." selection brings up the dialog box shown in **Figure 1–5** . The options are:

Maker Program field
lets you enter the program you use to build your executable.

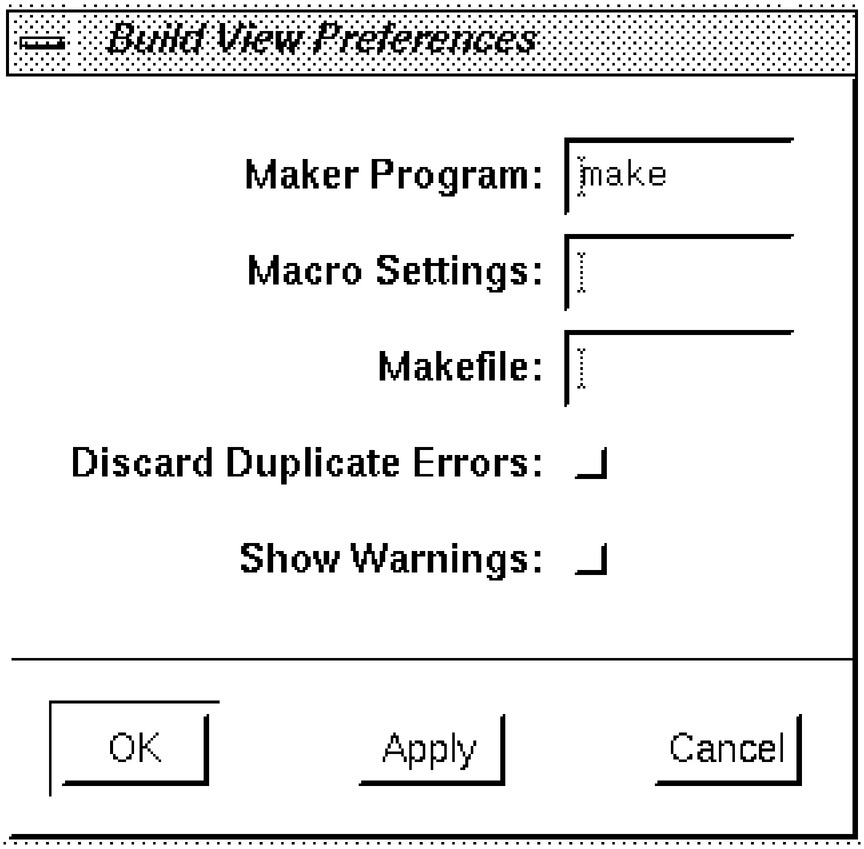
Macro Settings field
lets you enter build macros, such as

`CFLAGS=-g.`

Makefile field
lets you enter the name of a makefile if you do not wish to use the default.

Discard Duplicate Errors button
eliminates subsequent duplicates of errors in the error list area.

Show Warnings button
toggles the option to display warnings in the list.



Build View Preferences

Maker Program: make

Macro Settings:

Makefile:

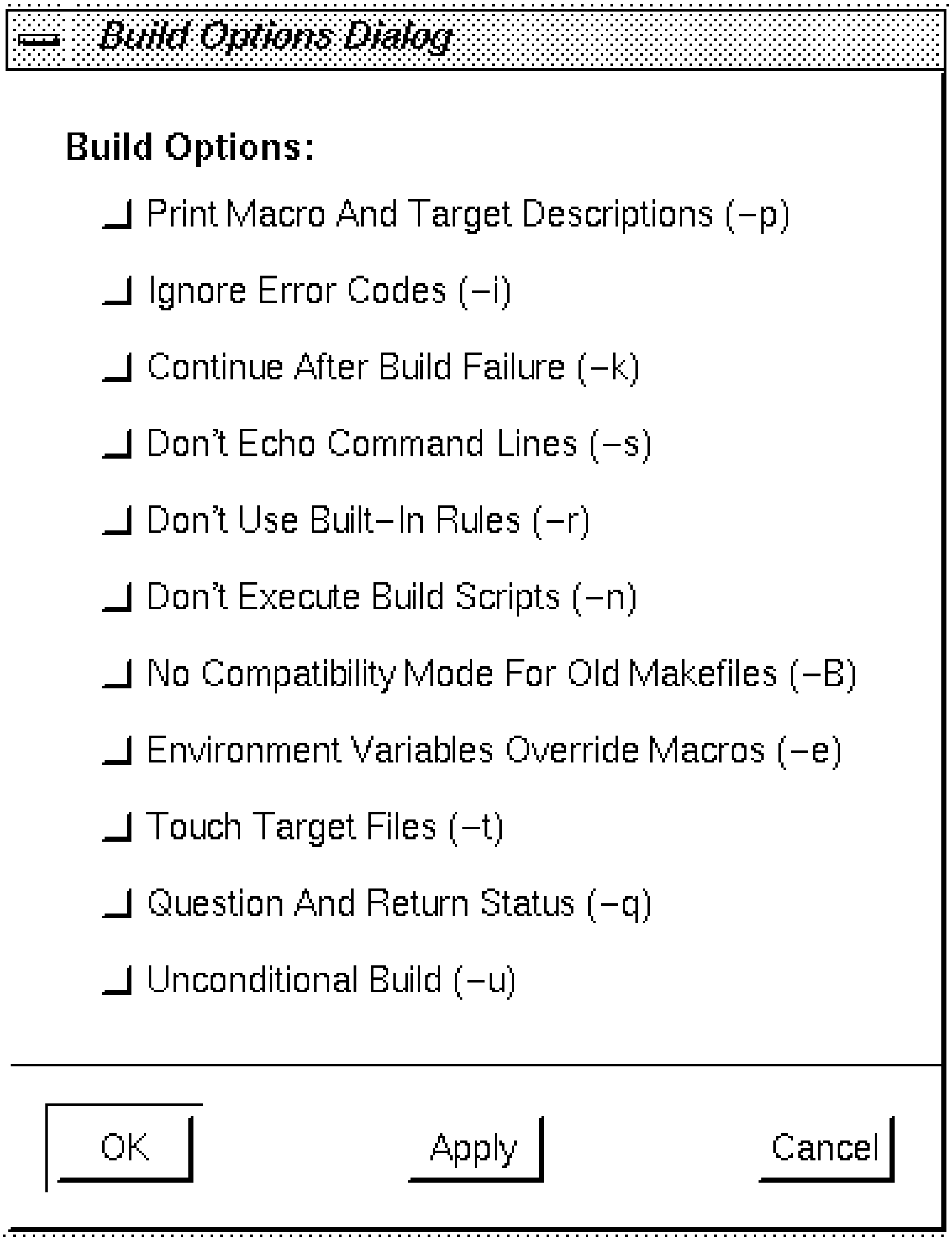
Discard Duplicate Errors: ☐

Show Warnings: ☐

Figure 1–5 Build View Preferences Dialog Box

Build Options

The Build Options dialog box lets you add the options shown in **Figure 1–6** to your *make* command.



Build Options:

- ☐ Print Macro And Target Descriptions (–p)
- ☐ Ignore Error Codes (–i)
- ☐ Continue After Build Failure (–k)
- ☐ Don't Echo Command Lines (–s)
- ☐ Don't Use Built–In Rules (–r)
- ☐ Don't Execute Build Scripts (–n)
- ☐ No Compatibility Mode For Old Makefiles (–B)
- ☐ Environment Variables Override Macros (–e)
- ☐ Touch Target Files (–t)
- ☐ Question And Return Status (–q)
- ☐ Unconditional Build (–u)

Figure 1–6 Build Options Dialog Box

Using Build View

The steps in running a compile using Build View are:

1. Bring up Build View.
2. Edit the *Targets* and *Directory* fields as required.
3. Specify your preference regarding duplicate errors and warnings using the Admin menu (optional).
4. Click *Build* to start the build. All compile information displays in the transcript area; errors are grouped in a list below.
5. Click *Interrupt* to terminate or *Suspend* for a temporary stop, if you want to stop the build. The

Resume button will restart a suspended build.

6. Double–click an error to bring up your preferred editor with the appropriate source code. A check mark indicates that an error has been accessed.

   Note: The default editor is determined by the *editorCommand* resource in the *app–defaults* file. The value of this resource defaults to `wsh -c vi +%d,` which means run *vi* in a *wsh* window and scroll to the current line. If the editor lets you specify a starting line, enter **`%d`** in the resource to indicate the new line number.

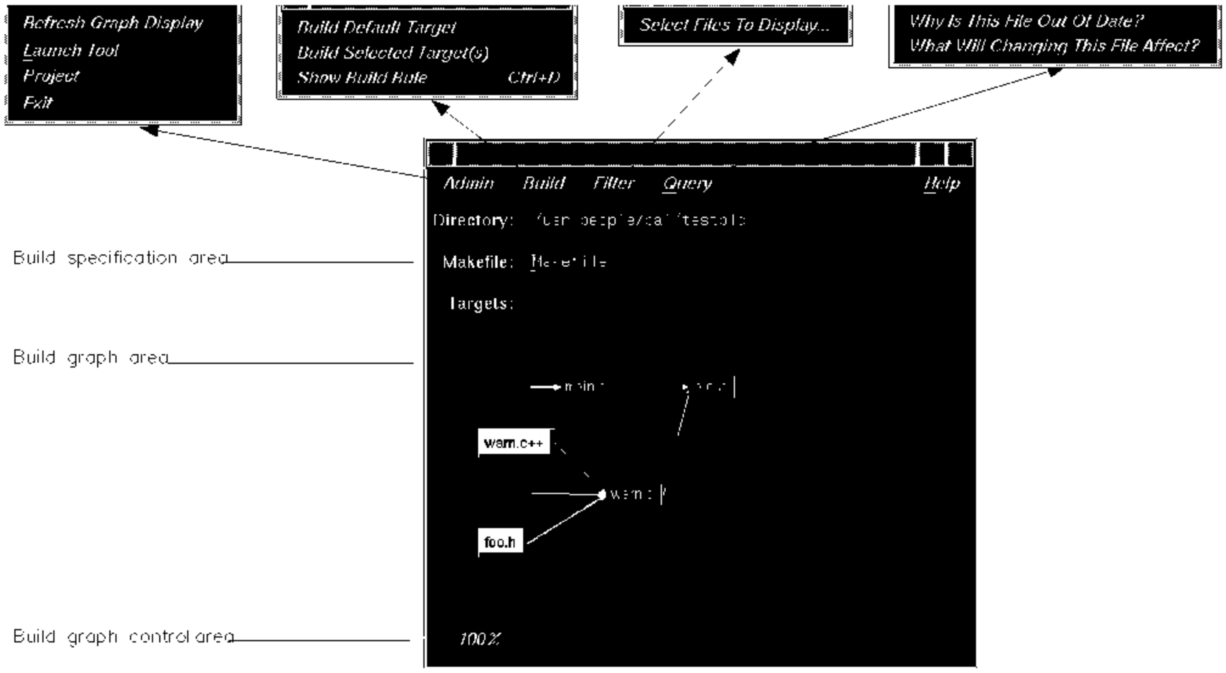
7. Click *Build* to restart the build.

Build Analyzer

Build Analyzer displays a graph indicating the source files and derived files in the build, their dependency relationships and current status. *Source files* refers to input files, such as code modules, documentation, data files, and resources. *Derived files* refers to output files, such as compiled code. You request builds in Build Analyzer by either:

- double–clicking a derived module
- making a selection from the Build menu

You access Build Analyzer from WorkShop by selecting "Launch Tool" from the Admin menu in Main View. Outside of WorkShop, you can access Build Analyzer by typing **`cvbuild`** at the command line. A typical Build Analyzer window appears in **Figure 1–7** with the menus displayed.



Directory: [illegible]

Makefile: Makefile

Targets:

Figure 1–7 Build Analyzer Window

Build Specification Area

The three fields in the build specification area identify the working directory, makefile script, and target file(s) for compilation. You can edit the *Directory*, *Makefile*, and *Targets* field directly. The *Targets* field also lets you specify a search string for locating a file in the build graph.

Build Graph Area

The *build graph area* displays the specified source and derived files and their dependency relationships. Files are depicted as rectangles; dependency relationships are shown as arrows, with the supplying file at the base of the arrow and the dependent file at the head. The colors used to depict the files depends on your color scheme. Build Analyzer differentiates the two types of files by depicting one with light characters on a dark background and the other with dark text on a light background. If you double−click a source file icon, an editor is brought up for that file. Double−clicking a derived file starts a build and displays Build View.

In addition to dependency relationships, Build Analyzer indicates the status of the files and relationships as follows:

- source file availability status: *normal* or *checked out*
    - *Normal* means that the source file is read−only and needs to be made writable to be edited. Normal files appear as light rectangles with black text.
    - *Checked out* means that you have a writable version of this file available and can thus edit it. A checked out file appears in a different color (from normal files) with a shadow.
- derived file compile status: *current* or *obsolete*
    - When applied to a derived file, the term *current* means that none of the files on which the derived file depends have been edited since the derived file was created. Current derived files appear as dark rectangles with white text.
    - *Obsolete* means that one or more of the source files have been modified since the derived file was created. Obsolete files appear in the same color as current derived files but with a colored outline.
- dependency relationship: *current* or *obsolete*
    - *Current* means that the derived file is up to date with the source files. Note that a relationship can be current even if both files are obsolete. This happens when a file on which both files are dependent has been modified. Current arcs are black.
    - *Obsolete* means that the source file has changed and the derived file has not been updated accordingly. Obsolete arcs appear as colored arrows.

Some typical build graph icons are shown in **Figure 1−8** .

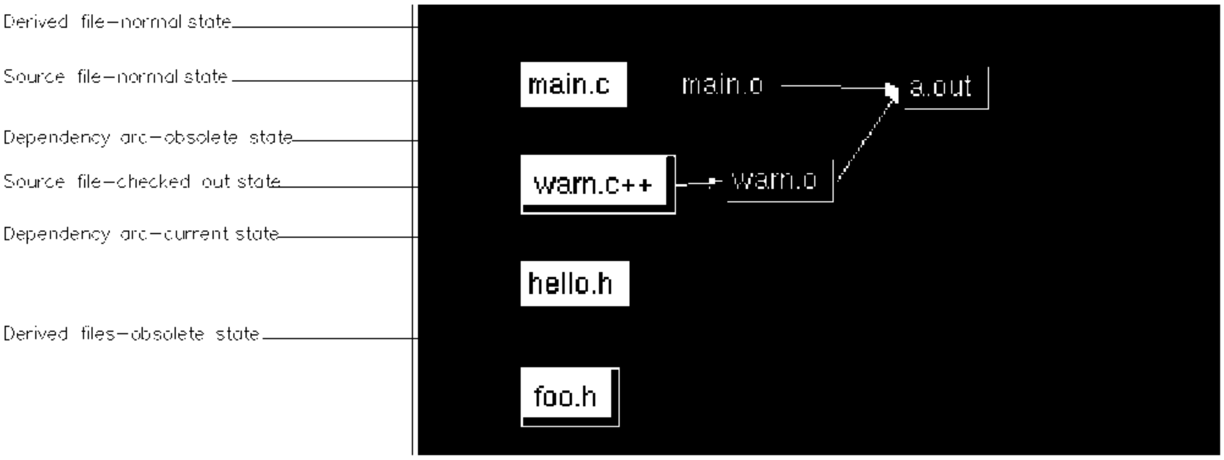


Figure 1–8 Build Graph Icons

The *main.c* and *hello.h* source files are in their normal state. The source files *warn.c++* and *foo.h* are checked out and thus appear highlighted and with dropped shadows. The derived file *main.o* is current, since it has not changed since the last compile. The black dependency arcs indicate that the source and derived files at either end are current with each other. When an arc is highlighted, it indicates that the source has changed since the last compile. The derived files *warn.o* and *a.out* are obsolete because *warn.c++* has changed.

Build Graph Control Area

The *build graph control area* contains a row of graph control buttons similar to the ones in the WorkShop Static Analyzer and the Call Graph View in the Performance Analyzer. The *Overview* button is particularly useful in the Build Analyzer because it helps you quickly find obsolete files where a lot of dependencies are involved.

The build graph control area is shown in **Figure 1–9** . For more information, refer to **Appendix A, "Graphical Views in the CASEVision Environment"** .

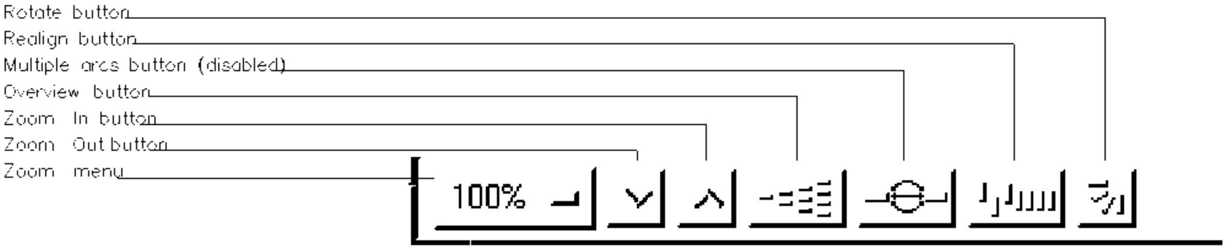


Figure 1–9 Build Graph Control Area

Overview Window

Since build graphs can get quite complicated, an overview mode (similar to those in Static Analyzer and Profiling View) is supplied that lets you view the entire graph at a reduced scale. To display the overview window, you click the *overview* icon (see **Figure 1–9**).

Figure 1–10 shows a typical Overview window with the resulting graph in Build Analyzer. The Overview window has a movable viewport that lets you select the portion of the build graph displayed in Build Analyzer. Source files that have changed and derived files needing recompilation are highlighted for easy detection. In this particular color scheme, the Overview window displays normal source files in turquoise, checked out source files in pink, current derived files in dark blue, and obsolete derived files in yellow. Arcs only appear in black in the Overview window.

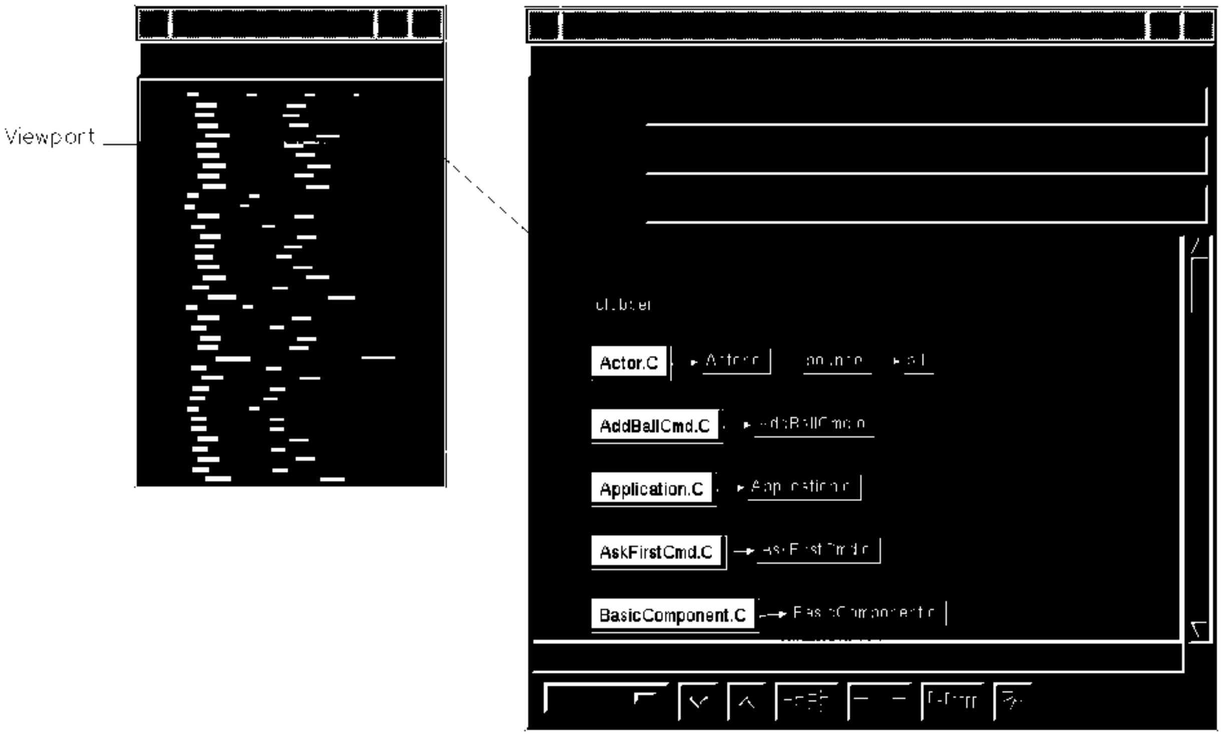


Figure 1–10 Overview Window with Resulting Build Analyzer Graph

Build Analyzer Menus

Build Analyzer contains these menus:

- Admin
- Build
- Filter
- Query
- Operation

Build Analyzer Admin Menu

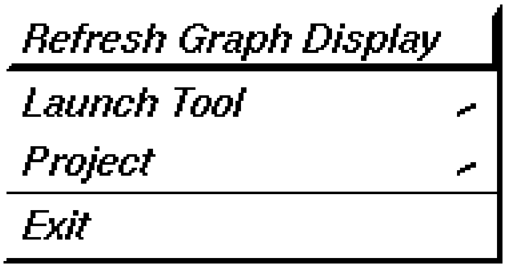


Figure 1–11 Admin Menu in Build Analyzer

The Admin menu (**Figure 1–11**) provides one selection "Refresh Graph Display" in addition to the standard WorkShop selections.
"Refresh Graph Display"

"Launch Tool"

lets you execute the WorkShop tools. For more information, see **"Admin Menu"** .

"Project"

lets you control the WorkShop tools operating on the same executable as a group. For more information, see **"Admin Menu"** .

Build Menu

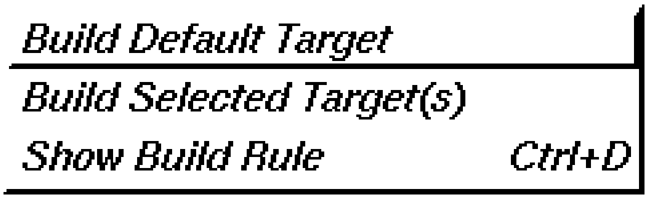


Figure 1–12 Build Menu

The selections in the Build menu (see **Figure 1–12**) let you perform builds as follows:

"Build Default Target"

performs a make with no arguments.

"Build Selected Target(s)"

performs the build(s) as entered in the *Targets* field.

"Show Build Rule"

displays a dialog box showing the makefile line for the selected node.

Filter Menu

The Filter menu has only one selection:

"Select files to show in graph"

lets you enter a regular expression to filter the files displayed in the build graph. The File Filter dialog box appears in **Figure 1–13** .

Figure 1–13 Filter Dialog Box

The upper list area lets you specify files to be excluded from the build graph. The lower list is for specifying files to appear in the graph.

Query Menu

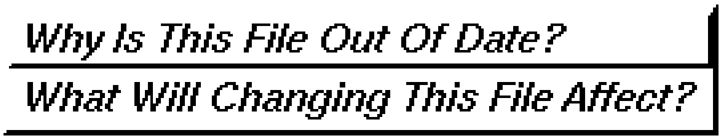


Figure 1–14 Query Menu

The Query menu lets you request information about the build graph (see **Figure 1–14**). The selections are:

"Why Is This File Out Of Date?"
identifies the source files requiring this file to be recompiled. This query only applies to derived files.

"What Will Changing This File Affect?"
shows all derived files dependent on this source file.

Operation Menu

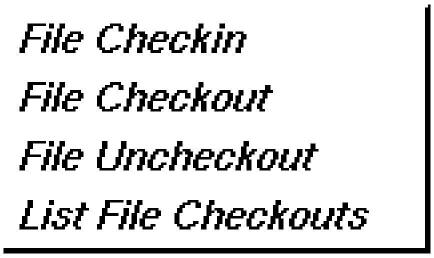


Figure 1–15 Operation Menu

The Operation menu provides access to the configuration management tool, if you have designated one.

Figure 1–15 Operation Menu

The Operation menu provides access to the configuration management tool, if you have designated one. The *cvconfig* script lets you designate CASEVision/ClearCase from SGI, RCS or SCCS. Simply type:

```
cvconfig [clearcase | rcs | sccs]
```

Selecting any of these options displays a shell in which you can access the configuration management tool. The selections in the submenu are:

"File Checkin"	saves the source file and checks it into the database as a new version.
"File Checkout"	recalls the source file from the tool's database if you have the proper authority, locks it, and makes it editable.
"File Uncheckout"	cancels the checkout, with no changes registered.
"List File Checkouts"	provides a list of the files that are currently checked out.

Chapter 2

Introduction to the WorkShop Static Analyzer

The WorkShop Static Analyzer helps you analyze source code written in C, C++, or Fortran by showing you the code's structure: how the functions within programs call each other, where and how variables are defined, how files depend on each other, where you can find macros, and many other structural details you need to help you understand the code. The Static Analyzer lets you analyze source code that won't compile; this may be your first line of attack on a buggy program, helping you to revise its code so that it will compile.

The following is covered in this chapter:

- **"What Does the Static Analyzer Do?"**
- **"Why Use the Static Analyzer?"**
- **"Data Collection Modes"**

If you have purchased WorkShop Professional from Silicon Graphics®, you will have access to the C++ Browser, which lets you gather additional information on the structure, hierarchy, and method interactions of each C++ class in your application or library.

What Does the Static Analyzer Do?

The Static Analyzer is in essence a database program that reads through one or more source code files and creates a database that includes functions, macros, variables, files, and (for C++) classes and methods. The database also includes the interconnections between the elements—which functions call which other functions, which files include which other files, and so on.

The Static Analyzer can perform selective searches (called *queries*) through the database; the Static Analyzer displays the results of the query in the query results area (the interior of its main window). If you've used the UNIX *grep* command, you'll find that the Static Analyzer can perform the same kinds of simple searches through the text of your source code, finding strings of text as well as regular expressions. The Static Analyzer also performs much more sophisticated queries that follow connections between elements of the source code: function calls, file includes, class parenthood, and other similar relationships. Overly general queries often return extensive results that are difficult to comprehend (consider, for example, a query that asks for *all* of the functions defined in millions of lines of source code). The Static Analyzer can restrict the scope of your queries so you can break down large projects into pieces of a manageable size. For example, you can see the connections to and from a single function or take a look at all the classes defined within a single file.

By default, the Static Analyzer displays the results of your query in text form. You can scroll through the results, and you can immediately call up the file that contains any element you see in the results. The file appears in the Source View window, which shows you the exact source code line where that element appears in the Source View window, which shows you the exact source code line where that element occurs. You can also ask the Static Analyzer to display the results of the query in a graphic view that shows not only the elements found but also—using tree form—the relationships between elements. To help you see the structure more clearly, you can set the scale and orientation of the tree, or you can call for a full overview that shows all elements in the structure and helps you scroll to the particular elements you want.

Why Use the Static Analyzer?

Too many programming projects involve working with massive source code files that have inadequate comments and documentation, or they require programmers to work with code they've never seen before. To fully understand the project, it's crucial that you understand the structure of the source code.

The Static Analyzer answers your questions about code structure and shows the answers in easily understood graphic form. Because the Static Analyzer is interactive, you can quickly zero in on the part of the code structure that interests you, or you can step back for an overview. And because the Static Analyzer recognizes the connections between elements of the source code, you can follow those connections, threading your way through calling functions, macro definitions, and other interconnected elements to understand how a proposed change to one element affects related elements.

Data Collection Modes

The Static Analyzer provides two modes for extracting static analysis data from your source files:

- scanner mode—a fast, general-purpose scanner that looks through code with minimal sensitivity to the programming language.
- parser mode—a language-sensitive scanner that can be run at compile time by setting a switch.

The trade-off between the modes is speed versus accuracy. A very effective technique is to perform preliminary analysis in scanner mode, when you need to see the overall structure of a large group of files, and then focus on a smaller subset using parser mode to derive detailed relationship information.

Chapter 3

A Sample Session with the Static Analyzer

This chapter shows how you might use the Static Analyzer in a typical session. It doesn't go into full detail, but it does explain the fundamental concepts you'll need to use the Static Analyzer. It lists related commands and controls after each tutorial so you can experiment on your own. You'll find more specific details about Static Analyzer features in Chapters 4 through 6.

This chapter discusses the following topics:

- **"General Static Analysis Usage"**
- **"Tutorial 1: Applying the Static Analyzer to Scanned Files"**
- **"Tutorial 2: Applying the Static Analyzer to Parsed Files"**
- **"Tutorial 3: Using the Compiler to Create a Static Analysis Database"**
- **"Other Static Analyzer Features"**

General Static Analysis Usage

Typically, in performing static analysis you want to create an overview showing basic relationships and then zero in on the source code requiring further work or analysis. There are five general steps in the static analysis process:

- Decide which files to include in the static analysis fileset.

  If you build a large static analysis database, you will find it not only difficult to navigate to the information you need but time-consuming during the data extraction process. It is good practice to narrow down the set of files to be analyzed as much as possible.
- Choose how the files will be analyzed: parser mode, scanner mode, a combination, or differently in multiple passes.

Scanner mode is good for determining the general structure of a program. It is most appropriate when you are analyzing large filesets, performing preliminary analysis, or working on uncompilable code. Parser mode is better when you need detailed relationship information. You should apply parser mode to smaller filesets, because it takes longer to extract data.

In some situations, it is desirable to use a combination of modes. For example, if you need detail but are having compilation problems, you can apply the scanner to the problem files and the parser to everything else. A different example would be applying the parser to a few files where you need detail and the scanner to the rest of the fileset.

An example of a multiple−pass scenario is to analyze a large fileset in scanner mode, zero in on a subset of the files, and then run that subset through parser mode to get a detailed analysis.

- Build the static analysis database.

  Both scanner mode and parser mode can be invoked within the Static Analyzer. After you have defined your fileset, the database will be built when you make your first query or when you select "Rescan" or "Force Scan."

  Parser mode can also be invoked through the compiler. A particularly convenient method for using the Static Analyzer parser is to modify an existing makefile so that it analyzes the files as part of the build process. This can be done with or without producing object code. You use the flags: `−sa,` *<directory>* and `−nocode`. For more information on this approach, see **"Tutorial 3: Using the Compiler to Create a Static Analysis Database"** .
- Perform static analysis queries and view the results.

  The queries can give you a good idea of the structure and the relationship of components in your program. You can review the results in text form, a list of items and their source or graphically (a tree showing relationships between items).
- Bring up Source View to edit or view a selected component.

  Once you have isolated an area for analysis, you can readily edit the source code from the Static Analyzer. Double−clicking an element brings up source code in Source View.

Tutorial 1: Applying the Static Analyzer to Scanned Files

In this tutorial, you will create a fileset for the demo program *bounce* using scanner mode and perform some basic queries in text mode.

1. Move to the */usr/demos/WorkShop/bounce* directory by entering the following:

   ```
   cd /usr/demos/WorkShop/bounce
   ```

   This directory contains the source code files (C++) for the demo program *bounce*.

2. List the directory's contents to see if the file *cvstatic.fileset* exists (in case someone worked through a tutorial and forgot to remove the file). If it does exist, remove it along with any other files the Static Analyzer may have left:

   ```
   rm cvstatic.* cvdb* vista.taf
   ```

   Whenever you run the Static Analyzer, it checks the directory where you invoked it for *cvstatic.fileset* and uses the file's contents as its fileset. If it doesn't find *cvstatic.fileset*, it creates and saves its own fileset containing the expression **.[cCfF]* so that all possible C files (*.c*), Fortran files(*.f* or *.F*), and C++ files (*.C*) in the current directory are included. When you quit the Static Analyzer, any fileset you or the Static Analyzer created remains in the directory for use in your next Static Analyzer session.

   If you don't care to use the default fileset, you can create your own or modify the default fileset using the "Edit Fileset..." selection in the Admin menu. You can also create your own *cvstatic.fileset* file by hand; you'll find instructions in Chapter 4.

3. Start the Static Analyzer by entering the following:

   ```
   cvstatic &
   ```

   The Static Analyzer window appears (as shown in **Figure 3−1**).

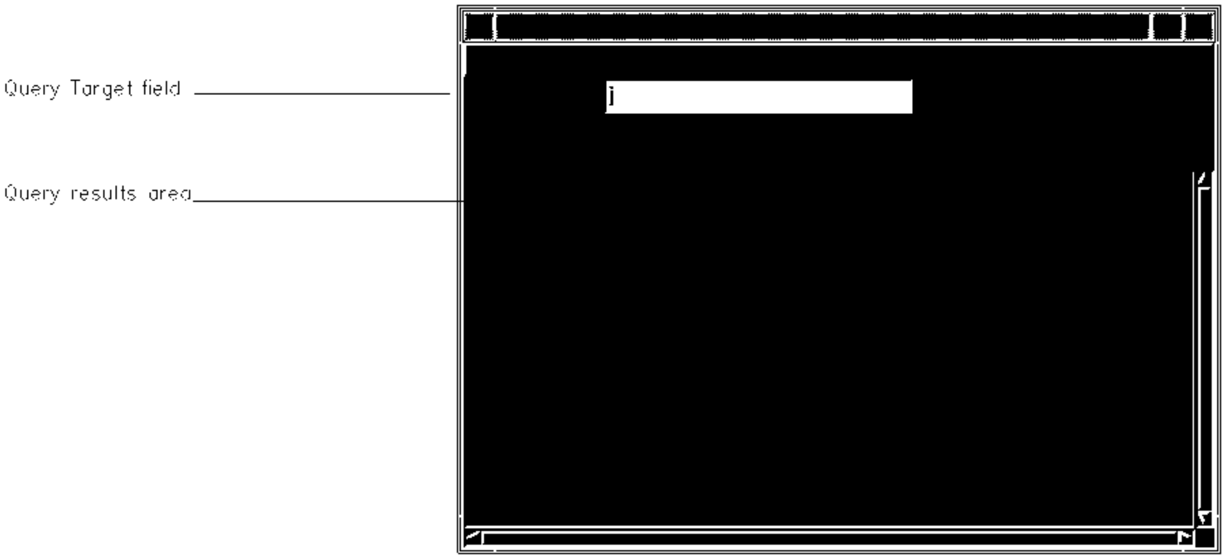


Figure 3–1 The Static Analyzer Window

4. Choose "Edit Fileset..." from the Admin menu to open the Fileset Editor window (shown in **Figure 3–2**).

/usr/demos/WorkShop/bounce/cvstatic.fil

Actor.C
Actor.h
AddBallCmd.C
AddBallCmd.h
Application.C
Application.h
AskFirstCmd.C
AskFirstCmd.h
BasicComponent.C

/usr/demos/WorkShop/bounce/Actor.C
/usr/demos/WorkShop/bounce/AddBallCmd.
/usr/demos/WorkShop/bounce/Application.C
/usr/demos/WorkShop/bounce/AskFirstCmd.
/usr/demos/WorkShop/bounce/BasicCompor

Figure 3–2 The Fileset Editor Window

The current working directory appears in the *Browsing Directory* field at the top left. Subdirectories (if any) appear in the *Directories* list field. The files in the current working directory appear in the *Files* list field. You select the files you wish to include in the fileset from these two lists. For parser mode files, you click the associated *Parser* button; for scanning mode, the *Scanner* button. There are two sets of *Parser* and *Scanner* mode buttons: the upper set moves whole directories and the lower set individual files. The two fileset list fields *Parser Fileset* and *Scanner Fileset* are at the right of the window.

5. Select the expression **.[cCfF]* in both the *Parser Fileset* and *Scanner Fileset* list fields (if it appears), and click the *Remove* button.

   This removes any default expressions from the fileset.

6. Click the *C++* language filter button.

   This filters the *Files* list to include only those files with the .C extension (signifying C++ source files) and selects them all.

7. Now add these source code files to the fileset by clicking the *Move Files:Scanner* button.

   The *Scanner Fileset* list now displays the files selected from the *Files* list. These files will be scanned into the static analysis database when it is created.

8. Click the *OK* button at the bottom of the Fileset Editor window.

   Once you've created the fileset, you can query it for useful information. Your first query prompts the Static Analyzer to extract static analysis data from the files in the fileset and create a cross−reference database. This occurs prior to returning the results of your query. The database includes the relationships between functions, files, classes, and other elements of the code in the fileset, and is saved in a database file along with two accompanying index files. The database file is named cvstatic.xre*f*, the accompanying files are named *cvstatic.index* and *cvstatic.posting.* These files are stored in the same directory as the fileset with which they're associated, and remain there after the Static Analyzer quits.

   Subsequent queries use the same database until you ask the Static Analyzer to rescan the fileset, which creates an updated database. When you quit the Static Analyzer and return to it later, it automatically updates the database, going through any files in the fileset that have changed since the last session. If you use appropriate wild card expressions in the fileset, the fileset will automatically accommodate new files added to specified directories.

9. From a shell, list the contents of the *cvstatic.fileset* file.

   All files and their paths to be included in the fileset should display. If you had selected files for parsing, they would have compiler flags following their paths.

10. Click the Queries menu to open it.

    To query the database, you choose a command from the Queries menu. You'll find the commands grouped in submenus according to types of query (see **Figure 3−3**).

Figure 3–3 Static Analyzer Queries Menu and *Query Target* Field

The query submenus let you perform different searches:

- "General" searches for text strings, regular expressions, and symbols.
- "Macros" searches for locations of macro definitions and places where macros are used.
- "Variables" searches for global and local variables, shows where they're defined, and who references and sets the variables.
- "Functions" searches for functions, shows where they're defined, and shows who calls them and whom they in turn call.
- "Files" searches for files in the fileset (including headers and libraries) and shows which files are included by which other files.
- "Classes" searches C++ files for classes and shows where they're defined. It also shows subclass and superclass relationships, and lists the methods defined within classes.
- "Methods" searches C++ files for methods and shows where they're defined and declared.
- "Common Blocks" searches Fortran files for common blocks.
- "Types" searches C and C++ files for type information.
- "Directories" lets you list directories or the files in a directory.

To start a query, you choose the query you want from the Queries menu. The Static Analyzer then searches through its database or through the original source code to find what you asked for.

If you want to look for a specific function, file, string, or other element, you enter the target text in the *Query Target* field above the query results area (shown in **Figure 3–1**).

Queries that require text in the *Query Target* field (such as "Find String" in the General submenu and "Who Is Called By?" in the Functions submenu) are grayed in the Queries menu if there is no text present. More general queries that require no search text (such as "List All Functions" and "List Global Symbols") are always available.

11. Choose "List All Functions" from the "Functions" submenu of the Queries menu.

The Static Analyzer builds its cross–reference database and notifies you that it's doing so. When it's finished, it displays a list of all functions found in the fileset (as shown in **Figure 3–4**), their file, the line number at which they are first defined or declared, with the actual source line.

Note: During this process, you may get a warning dialog box about multiple function occurrences. This is due to the inaccuracy of scanner mode; it has problems with *#ifdef* statements. You may also get an error message about missing files. This can happen if your include paths are not set correctly. The missing files are not necessary for this tutorial. Usually you can tell by looking at the files if you need to include them in the fileset.

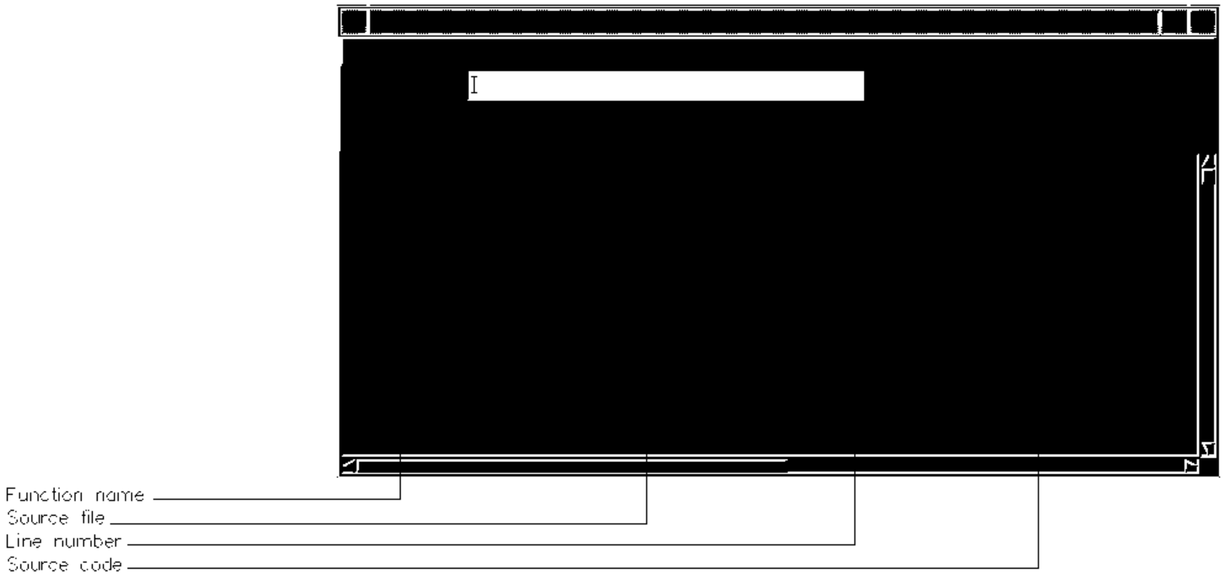


Figure 3–4 The Results of a "List Functions" Query

The Static Analyzer returns the results of all queries in its query results area (below the *Query Target* field). It presents this information in text form (and by the previous type of view if applicable for subsequent queries). You can scroll through a text list to find specific data that interests you. Clicking any part of an element listed (a filename, a function name, a line number, and so on) pastes it into the *Query Target* field so you can use it as the base of your next search. For example, if you'd like to find out what functions a particular function calls, you can left–click its name to put it into the *Query Target* field and then choose "Who Is Called By" from the "Functions" submenu of the Queries menu.

Text View allows you to sort the element lines alphanumerically by any one of the fields in a line. For example, you can sort a list of functions alphabetically by function name or numerically by line number where they occur. To sort, left–click within an element line in the field by which you want to sort, and then choose "Sort" from the Admin menu. The Static Analyzer sorts the results of a query according to the field selected.

12. Left–click the function name Actor in the query results area.

    The Static Analyzer pastes the name into the *Query Target* field.

13. Choose "Who Is Called By" from the "Functions" submenu of the Queries menu.

    The Static Analyzer displays a list of all functions called by *Actor*.

14. Clear the *Query Target* field and then type **`buffer`** in it.

    We are now going to look for any occurrences of the text string "buffer" that might lead us to information in the code concerning z–buffering or data buffering:

15. Choose "Find String" from the "General" submenu of the Queries menu.

    The Static Analyzer returns all the lines of code that contain the text string "buffer," even if it only appears in a comment.

16. Left–click the History menu to open it.

    It displays the queries you've made so far, as shown in **Figure 3–5**.

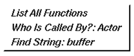


Figure 3–5 Static Analyzer History Menu

17. Choose "List All Functions" from the History menu to see a list of all functions once again.

    This brings back your previous query results.

18. Double–click the *Actor* function.

    Source View now appears, displaying the source code for *Actor*. You can examine it, check it out (if you have a versioning system), or edit it.

19. Select "Close" from the Source View File menu to close it.
20. Select "Exit" from the Static Analyzer Admin menu to end this tutorial.

Tutorial 2: Applying the Static Analyzer to Parsed Files

In this tutorial, you will create a fileset for *bounce* using parser mode and perform some detailed static analysis in both text mode and graphic mode.

1. Move to the */usr/demos/WorkShop/bounce* directory by entering the following:

   ```
   cd /usr/demos/WorkShop/bounce
   ```

2. Type `rm cvstatic.* cvdb* vista.taf`

   We need to remove these files to avoid using the previous fileset.

3. Start the Static Analyzer by entering the following:

   ```
   cvstatic –mode PARSER &
   ```

   The Static Analyzer window appears. The **`–mode PARSER`** option causes the Static Analyzer to use parser files only when queries are performed.

4. Select "Edit Fileset..." from the Admin menu.

   We are going to apply parser mode through the Fileset Editor.

5. Remove the wild card defaults (**.[cCfF]*) if they appear in the fileset lists.
6. Select the file called *BouncingBall.C* in the *File* list at the lower left and click the *Move Files: Parser* button to transfer the file to the *Parser Fileset* list.

   This enters the file *BouncingBall.C* into the fileset and sets it for parsing mode.

7. Click *OK* to save the new fileset and from a shell, type

   ```
   cat cvstatic.fileset
   ```

   The contents of cvstatic.fileset display:

   ```
   /usr/demos/WorkShop/bounce/BouncingBall.C NCC
   ```

   Notice that the *BouncingBall.C* file has a compiler flag indicator as part of the entry. This is how parser mode files are indicated.

8. Select "List All Functions" from the Queries Menu.

   The Static Analyzer builds a new database using parser mode. Since *BouncingBall.C* has a number of include files, this process may take a few minutes. During the process, a small window called Build Shell appears that displays any compiler errors or warnings. At the conclusion of the process, the functions in *BouncingBall.C* and its include files are listed in text form in the query results area.

9. Select "Call Tree View" from the Views menu.

   The query results area now changes to graphical form. The functions are depicted as rectangles. In addition to listing functions, the Static Analyzer now provides us with relationship information, that is, who calls which functions. The function calls are shown as arrows (or arcs) pointing to the functions that were called.

   Besides "Call Tree View", there are two other types of graphical views: "Class Tree View", which displays C++ classes and their hierarchy, and "File Dependency View", which displays the files in the fileset and their dependency on each other.

Whenever you use a tree view, the view interprets the results of your query according to the type of tree displayed. For example, if you perform a "Functions" query while you're in file dependency view, the view changes to show you which files contain the functions returned by the query. Some views don't make sense for displaying the results of a query, in which case the Static Analyzer switches to the view it thinks is most reasonable for the query.

10. Left–click on the *Graph Overview* button (the fourth button from the left at the bottom of the Static Analyzer window).

    This displays the Call Tree Overview window, a feature to help users navigate through a graph. It displays the full call tree in overview, with a small rectangular outline (called the *viewport*) in the upper–left corner. The viewport shows which portion of the tree currently appears in the query results area of the Static Analyzer window and can be dragged by the mouse to expose other portions of the graph. See **Figure 3–6** .

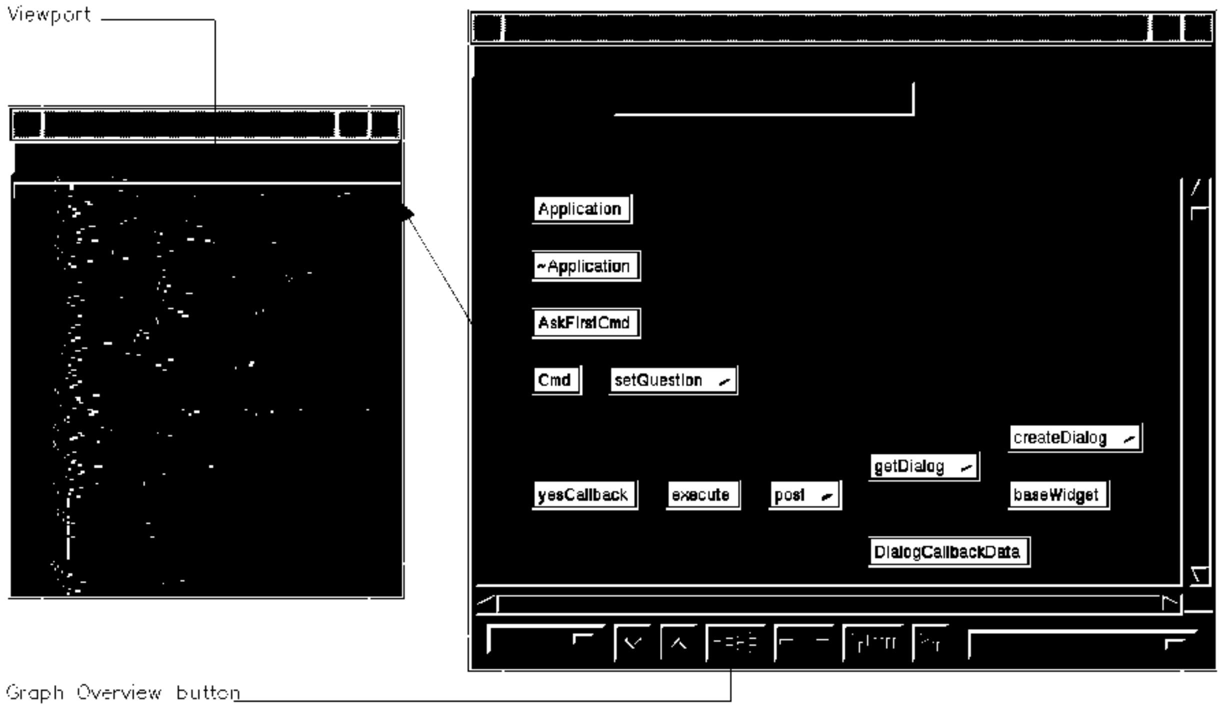


Figure 3–6 Typical Static Analyzer Call Tree

11. Left–click in the center of the Call Tree Overview window.

    The viewport jumps so that its upper–left corner matches the pointer location. The query results area in the Static Analyzer window shifts to display the part of the tree outlined by the viewport in the Call Tree Overview window.

12. Drag the viewport around in the Call Tree Overview window by holding down the left mouse button and moving the mouse. End up by dragging the viewport to the upper–left corner of the call tree.

    As the viewport moves over the call tree overview, the call tree shown in the Static Analyzer query results area scrolls to match.

13. Select "Close" from the Admin menu in the Call Tree Overview window to close it.

14. Enter `colorSelected` in the *Query Target* field and select "Who Calls?" from the "Functions" submenu in the Queries Menu.

    This reduces the graph to three nodes: *colorSelectedCallback* and *cancelCallback*, which call *colorSelected*, and *colorSelected* itself.

15. Hold the right mouse button down over the node labeled *colorSelected* to open its popup menu.

    This displays the individual node menu, which provides the selections: "Hide Node", "Collapse Subgraph", "Show Immediate Children", and "Show Parents". The triangular arrow at the right of the *colorSelected* node indicates that it has undisplayed child nodes: accordingly the "Show Immediate Children" is enabled. Since the parents of *colorSelected* are already displayed, the "Show Parents" selection is disabled.

    If you hold the right mouse button down over a portion of the query results area where there are no nodes, the selected nodes menu displays, which provides the selections: "Hide Selected Nodes", "Collapse Selected Nodes", and "Expand Selected Nodes".

16. Choose "Show Immediate Children" from the popup menu.

    The Static Analyzer displays the functions called by *colorSelected*.

    For more information on the standard graph controls and node manipulation, see **Appendix A, "Graphical Views in the CASEVision Environment"** Note that the view options menu is unique to the Static Analyzer. It offers options that extend the range of the nodes you see in the tree to include nodes not included in the original query.

17. Open the History menu to review the commands you have selected.
18. Choose "Exit" from the Admin menu to exit the Static Analyzer.
19. Remove all the files generated by the Static Analyzer from the directory:

    ```
    rm cvstatic.* cvdb* vista.taf
    ```

Tutorial 3: Using the Compiler to Create a Static Analysis Database

This tutorial takes you through the steps of creating a static analysis database, using parser mode.

1. Type **`cd /usr/demos/WorkShop/bounce`** to go to the *bounce* demonstration directory.

   We will once again analyze the *bounce* demonstration program.

2. Type **`mkdir staticdir`**

   This creates the subdirectory in which we will store the static analysis database. If a directory named *staticdir* already exists, remove it or use a different name.

3. Type **`cd staticdir`** to change directories and then **`initcvdb.sh`**.

   The **`initcvdb.sh`** script creates the initial files *cvdb*.** necessary for producing the database.

4. Change */usr/demos/WorkShop/bounce/Makefile* as follows:

   The most convenient method for applying the Static Analyzer parser to a large group of files is to modify the existing makefile so that it analyzes the files without producing object code.

   Change the following line:

   ```
   CC=CC
   ```

   to

   ```
   CC=NCC -sa,staticdir -nocode
   ```

   Changing to **`NCC`** causes the C++ compiler to be used. The flag **`-sa`** tells the compiler to perform static analysis. Following **`-sa`** with **`,staticdir`** tells the compiler to store the results in the *staticdir* subdirectory; otherwise, the current directory is assumed. The flag **`-nocode`** saves you time by telling the compiler not to create object code.

5. Make sure you are in the */usr/demos/WorkShop/bounce* directory and type **`make -k`**

   This runs the compiler as you've specified in the makefile. This may take a while. Running parser mode performs the major operations of compiling, short of creating the object code.

6. Go to the *staticdir* subdirectory and type

   ```
   cvstatic -mode PARSER -readonly
   ```

   This invokes the Static Analyzer set for parsed files. The other mode options are SCANNER for scanned files and BOTH if you mix scanned and parsed files. The `-readonly` safeguard flag protects against inadvertent changes. You can now perform any valid Static Analyzer operations, as in the previous tutorials.

Other Static Analyzer Features

You'll find more detailed information about querying in Chapter 5. To explore on your own, try these commands in the Admin menu, which also affect queries:

"Rescan"

asks the Static Analyzer to update the cross-reference database by rescanning any source code files in the fileset that have changed since the last database update.

"Force Scan"

asks the Static Analyzer to update the cross-reference database by rescanning all source code files in the fileset, whether they have changed or not.

"General Options..."

offers options that determine how a query treats the text string entered in the *Query Target* field and how filenames are displayed.

"Set Include Path..."

allows you to set a search path of directories where the Static Analyzer looks for include files that are mentioned in the code contained in the fileset.

"Save Query..."

saves the text or graphics results of a query to a file. If the query results are displayed graphically, this command allows you to select a file to save the PostScript representation.

Caution: As you experiment with queries in tree views, you may be tempted to look at a coding project that includes millions of lines of code. If so, be sure to use restricted queries or to use the Results Filter to greatly filter the results of the query. If you use a very comprehensive query such as "List All Functions", the Static Analyzer may be locked into creating a tree view that consists of hundreds of thousands of nodes and even more arcs, something akin to drawing a map of the United States with detail enough to show every driveway of every house in the country! Not only will you have to wait hours for your results, but the results will probably be so complicated that they'll be meaningless to you.

And with that warning, you're left to your own experimentation. For more details on individual Static Analyzer features, see **Chapter 4** through **Chapter 7** .

Chapter 4

Static Analyzer: Creating a Fileset and Generating a Database

Before you can perform any static analysis queries, you need to specify the source code files to be analyzed in a file called a fileset and then generate a database containing the static analysis information. This chapter covers these topics:

- **"Fileset Specifications"**
- **"Using the Fileset Editor"**
- **"Creating a Fileset Manually"**
- **"Using Command–Line Options to Create and Use a Fileset"**
- **"Generating a Static Analyzer Database"**
- **"Rescanning the Fileset"**
- **"Search Path for Included Files"**
- **"Changing to a New Fileset and Working Directory"**

Fileset Specifications

A Static Analyzer *fileset* is a single file used to specify the source code files to be analyzed. There are five methods for creating a fileset:

- using the Fileset Editor
- creating a file manually
- letting *cvstatic* do it automatically at startup by defaulting to those files in the current directory that match the expression *.[cCfF]
- letting *cvstatic* do it automatically at startup by designating an executable
- using the compiler to create a fileset (and database) by adding the **`-sa,`**<*dbdirectory*> option to your makefile

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
- included directories preceded by the designator `-I`

Note: In parser mode only, an entry can be followed by the name of the compile driver, compilation options such as **`-ansi`**, and other user-specified options such as **`-D`** for defining macros (see **"Parser Mode"**).

Using Regular Expressions

Each line in the fileset can use shell expansion characters, a wild card system in standard use for specifying filenames in UNIX shells. If you enter a standard pathname (either absolute or relative), the Static Analyzer reads the line literally and looks for the file. If you use metacharacters such as brackets ([]) and asterisks (*), you can specify a number of files with a single line of text. For example, the default fileset contains the single line:

```
*.[cCfF]
```

The asterisk specifies any number of characters (zero or greater) before a period, and the bracketed set of characters specifies any of four single characters—*c, C, f,* or *F*—after the period. The result is that the line specifies any filenames in the current directory that use a *.c, .C, .f,* or *.F* extension.

Note: Don't confuse the shell expansion characters used here with the regular expressions used in the Fileset Selection Browser; they're completely different systems. If you want full information about shell expansion characters, you'll find them described in the man pages for *csh*.

Specifying Pathnames

The Static Analyzer resolves absolute pathnames in the fileset from the root; it resolves relative pathnames from the directory in which you invoke the Static Analyzer, referred to as the *browsing directory*. Anytime you change to a fileset in another directory, however, the Static Analyzer changes the working directory to match so that any relative filenames in the fileset are resolved from the fileset's own directory.

Specifying Included Files

Besides specifying filenames, the fileset also can also specify directories to search for included files. The default search files are the current directory and */usr/include*. Any additional search paths are specified with the prefix `-I` followed immediately (without space) by the pathname. For example, the pathname

```
-I/usr/include/gl
```

listed in a fileset asks the Static Analyzer to search through */usr/include/gl* for include files.

Filesets created by the Static Analyzer are named *cvstatic.fileset* by default. If you create your own filesets, you can give them any name you wish, but by convention you should use the *.fileset* extension.

Defining Symbols in the Fileset

The Static Analyzer lets you define macros to be included in the database. When you compile with the

`-sa` flag, the fileset is built with one file per line; lines may also contain a **`-I`** flag for including files, **`-D`** for defining macros, or **`-U`** for undefining macros. The Static Analyzer doesn't normally preprocess source code files before creating a cross-reference database. Some source code, however, requires preprocessing to resolve *ifdef* statements before you can successfully analyze the code.

The way to perform preprocessing is to specify these symbol names and values in the file *cvstatic.fileset* and then run *cvstatic* from the command line with the **`-preprocess`** flag. The macros are specified at the end of the fileset by appending a line of the form

```
-D<symbolname>
```

or

```
-D<symbolname>=<value>
```

for each preprocessor symbol you want to define. For example, to set the macros `DEBUG` and `BUFFERSIZE`, you would append two lines like this to the end of the fileset:

```
-DDEBUG
-DBUFFERSIZE=8
```

In like manner **`-U`** undefines macros. These symbol definitions will be used for processing all of the files in the fileset.

Note: Using the **`-preprocess`** option increases the scanning time tremendously (scanner mode only). Use it only when absolutely necessary, and consider analyzing the code as is, including all the *ifdef*ed sections.

Using the Default Fileset

When you start the Static Analyzer in a directory that doesn't contain a file named *cvstatic.fileset*, the Static Analyzer creates a default fileset and saves it as *cvstatic.fileset*. The contents of the fileset are:

```
*.[cCfF]
```

This line specifies any C, C++, or Fortran files in the working directory. Note that the line assumes that C++ files have a *.C* extension, which may not be the case for all C++ files because there isn't yet a pervasive extension standard. If your C++ files use *.c++, .cc,* or other extensions and you want to use the default fileset, you should edit it to include the extensions you want.

Using the Fileset Editor

The Fileset Editor window (see **Figure 4–1**) lets you edit the contents of a fileset. You invoke it by choosing "Edit Fileset..." from the Admin menu. The contents of the current fileset appear in the two file lists on the right side of the window; directories and files that you can add to the fileset appear in the *Directories* and *Files* lists on the left.

The *Current Fileset* field at the top right of the window is a read–only display that shows the full pathname of the current fileset. The directory displayed here is the Static Analyzer's current working directory. You can't change either the fileset or the working directory here; to do so, you use the "Change Fileset..." selection in the Admin menu.

Below the *Current Fileset* field, there are two list areas. A fileset can contain two kinds of files: those that are scanned into and those that are parsed into the database. (For a complete discussion of scanner and parser mode, see **"Generating a Static Analyzer Database"** .) The top list area shows the files in the fileset to be parsed and the lower one the files to be scanned. Both list areas have vertical scroll bars to scroll through long lists and horizontal scroll bars to move left and right through long filenames.

To see an example of the Fileset Editor, refer to **"Tutorial 1: Applying the Static Analyzer to Scanned Files"** .

Figure 4–1 The Fileset Editor Window

Adding Lines to the Fileset Contents List

Both fileset list areas have direct entry fields immediately below them that allow you to enter lines in the fileset. You put the pointer in the line entry field and type. When you press **`<Enter>`**; the Fileset Editor enters your line in the fileset.

The line entry field interprets each typed line as soon as you press **`<Enter>`**. If you enter literal filenames such as *jello.c* or *../bounce/bounce.C*, that filename appears in the fileset list when you press **`<Enter>`**. If you enter a wild card entry such as **.**, the Fileset Editor interprets it, resolving from the working directory, and places those filenames that match (not the wild card entry itself) in the fileset list.

If you want to enter a wild card entry in the fileset without having it immediately interpreted and replaced with actual filenames, turn on the *Literal Input* toggle button just below the line entry area. When this button is on, the Fileset Editor treats any strings you enter literally; it does not interpret them as shell expansion characters, which allows you to place wild card lines directly into the fileset. The Static Analyzer interprets these strings later when you query the fileset.

Removing Lines from the Fileset Lists

To remove a line from a fileset list, left–click to select it and then left–click the *Remove* button below the lists. The Fileset Editor removes the line from the list. To remove more than one line at a time, left–drag the cursor over a range of files or hold down the **`<Control>`** while left–clicking, then left–click the *Remove* button.

Browsing for Fileset Contents

You can use the lists and buttons on the left side of the Fileset Editor window to browse through available directories for files to add to the fileset.

Directories List

The *Directories* list shows the subdirectories available in the current directory; left–double–click a subdirectory to move to that directory and see its subdirectories in the *Directories* list. The ".." entry is the parent directory of the current directory; left–double–click it to move up a directory.

Browsing Directory

The *Browsing Directory* field just above the *Directories* list shows the current directory in which you're browsing. You can use it to type an absolute pathname to a new directory—put the pointer in the area to type. When you press **`<Enter>`**, the contents of the Directories list change to show the subdirectories of the directory you entered.

Language Filters

The *Files* list below the *Directories* list shows the files contained in the current directory. You can filter the contents you see there by turning on any or all of the three filter buttons below the list: the *C* button, the *C++* button, or the *Fortran* button. If none of these buttons is turned on, the *Files* list shows all files in the current directory. Turning on any single button restricts files listed to C, C++, or Fortran files:

- The *C* button restricts files shown to those with .c extensions.
- The *C++* button restricts files shown to those with *.C, .cc,* or *.cxx* extensions.
- The *Fortran* button restricts files shown to those with *.f* or *.F* extensions.

You can set combinations of these buttons to see different source code file types.

Adding Filenames from Lists

If you wish to add one or more filenames from the *Files* list to one of the fileset lists, select the filename and left–click the *Move Files Parser* button or *Scanner* button to the right of the *Files* list depending on how you want information extracted from the file. The Fileset Editor puts the absolute pathname of each file in the fileset list.

To add all the files in a directory to the *Fileset Contents* list, left–click the directory name (or directory names if you want more than one) in the *Directories* list, then left–click either the *Parser* button or *Scanner* button to the right of the *Directories* list. The Fileset Editor (in its default state) adds only the files contained in that directory, and not files contained within any of its subdirectories.

To add files contained within a directory's subdirectories, turn on the *Include Subdirectories* button. When you left–click the *Add Directories* button with this button turned on, the Fileset Editor adds all files in directories, subdirectories, and so on, to the fileset lists.

You can specify the kinds of files the Fileset Editor puts in the *Parser Fileset* and *Scanner Fileset* lists when you left–click the *Add Directories* button. To do so, turn on any of the filter buttons below the *Files* list.

Transferring Files in the Fileset Between Modes

The Fileset Editor lets you change the method of data extraction (parser or scanner) for files in the fileset. You do this by transferring them from one fileset list to the other using the two *Transfer Files* arrows. This is particularly useful when you discover that a file cannot be parsed, as first thought; you then transfer it to the scanner mode which is not sensitive to programming languages.

Leaving the Fileset Editor Window

You can close the Fileset Editor window by left–clicking the *OK* button or the *Cancel* button. Left–click *OK* to put all the fileset changes you made into effect. Left–click *Cancel* to close the window and return the fileset to the state it was in when you first opened the Fileset Editor Your editing changes are ignored.

Creating a Fileset Manually

You can create a fileset by hand if you wish, either by using a text editor that saves text in a text–only format (*vi*, for example), or by using the output of UNIX commands that return filenames. You may find the UNIX *find* command particularly useful for returning all specified filenames within a directory tree. For example, the command

```
find . –name "*.f" –print > cvstatic.fileset
```

creates a fileset of all Fortran files (those with a *.f* extension) found within the current directory and all of its subdirectories. Note that all the pathnames in the fileset are relative, determined from the current directory.

You can pipe the output of the *find* command through filtering commands such as *sed* to further modify the fileset created. For example, the command

```
find . −name "*.c" −print | sed'/\.\.c/d > cvstatic.fileset
```

finds C files within a directory tree and strips out any *.c* files left by the C++ compiler.

Using Command−Line Options to Create and Use a Fileset

The Static Analyzer provides three special options when you invoke *cvstatic* from the command line:

- The **`−executable`** option followed by the filename of an executable file asks the Static Analyzer to create a fileset that contains the absolute pathname of every file used to compile that executable. For example, entering

  ```
  cvstatic −executable jello
  ```

  while in the */usr/demos/CASEVision/jello* directory starts the Static Analyzer and creates a fileset that includes all the files used to compile *jello*.

  Note that the executable must not be stripped; stripped files do not contain the names of their source files. When using the **`−executable`** option, it's a good idea to use the fileset editor to exclude files with "incomplete" names (which can occur with files compiled into *lib* using compilers prior to 4.0.1 or non−supported languages like assembler or Pascal). The **`−executable`** option requires that the executable be built on the same system performing the static analysis.

  Note also that this command−line option works only if you have the C, C++, or Fortran compiler that's shipped with IRIX™ version 4.0.1 or greater.
- The **`−fileset`** option followed by the filename of a fileset asks the Static Analyzer to start using a fileset other than *cvstatic.fileset*.
- The **`−mode`** flag takes the options `SCANNER`, `COMPILER`, and `BOTH` to indicate the type(s) of files in the fileset to be used in queries. For example, if you enter **`cvstatic −mode COMPILER`**, only parser mode files will be used in queries; scanner files will be ignored.

Generating a Static Analyzer Database

The most time−consuming part of the static analysis process is creating the database, which is a collection of symbols and their relationships. There are two methods for extracting static analysis data from a fileset:

- scanner mode, which is fast but not sensitive to the characteristics of specific programming languages.
- parser mode, which is language−specific and thus more thorough.

If you need a mix of accuracy and speed, you can combine the two modes by flagging the files in the fileset according to mode and building the database with the **`−mode BOTH`** flag. You might use this approach if some files cannot be compiled or if scanner mode is misinterpreting necessary symbols.

Scanner Mode

The quickest way to build a database is to use scanner mode. Since scanner mode is not sensitive to the characteristics of specific programming languages, it may miss or incorrectly parse certain symbols (especially in Fortran). If you are analyzing a large quantity of source code, do not care about minor inaccuracies, and don't need the language−specific relationships (such as C types) available in parser mode, then use scanner mode.

Scanner mode is the default method for building a static analysis database. It is run automatically whenever you create a new fileset or perform a rescan, unless you explicitly specify parser mode.

Scanner mode creates files named *cvstatic.fileset*, *cvstatic.index*, *cvstatic.posting*, and *cvstatic.xref* in the directory in which it is started. These files comprise the Static Analyzer database for the program.

If the Static Analyzer finds cross−reference files to accompany a fileset, it determines when they were last updated. It then scans the fileset to see which files which have been modified or added since that date. The Static Analyzer updates the cross−reference files with cross−references found in modified or added files.

Parser Mode

Parser mode is language−specific and slower as a result. Use parser mode when you need to stress accuracy over speed. Parser mode provides relationship data specific to the programming languages C, Fortran, and C++, such as querying on types, directories, and Fortran common blocks. Parser mode uses the compiler to identify entities in the source code, so you must be able to compile a file in order for it to be parsed. If a source file cannot compile, then you need to flag that file for scanning and run it through scanner mode.

Note: The database generated by parser mode can also be used by the C++ Browser (it must be purchased separately).

Preparing the Fileset

File entries for parser mode take the general form:

```
/fullpath/sourcefile drivername   options
```

where:

drivername
refers to the compiler driver and can be "f77" for Fortran, "ncc" for the Edison C compiler, "NCC" for the standard C++ compiler, or "DCC" for the Delta C++ compiler.

options
lets you choose language level (**`−ansi`**, **`−cckr`**, **`−xansi`**, or **`−ansiposix`**) and user−specified options such as **`−I`** for including files, **`−D`** for defining macros, **`−nostd`**, and **`+p`**. See the man page for *cc* for more information.

The Static Analyzer recognizes the type of language by the file extension. ".c" extension is considered to be C. ".C" and ".cxx" are considered to be C++ files. Parser mode assumes that C files are ANSI unless otherwise specified in the makefile.

Before processing the files, the Static Analyzer must know where to look for include files. If you are using parser mode, you will want to set the include paths before the Static Analyzer scans the files; so do this before performing any queries or selecting "Force Scan".

Invoking the Parser

There are three methods for creating a fileset with parser mode files:

- Enter the files in the parser mode fileset list in the Fileset Editor.
- Edit the *cvstatic.fileset* file directly, specifying the compiler and other options after the file entry.
- Use the compiler to generate the fileset by specifying the flag **`−sa[,`***databasedirectory***`]`** and **`−nocode`**. Without arguments, the **`−sa`** flag stores the static analysis database in the current directory. If you enter a comma (,) and a directory, the static analysis database will be stored in the specified directory. If you specify the flag **`−nocode`**, then the database will be built without creating new object files.

While the database is being built, a window appears, displaying any messages from the parsing process. This helps you find problems if there is code that cannot compile.

Parser mode creates a *cvstatic.fileset* file and some new files named *cvdb*.dat, cvdb*.key, vista.taf,* and *cvdb.dbd* in the current directory. In parser mode, "Force Scan" rebuilds the database. "Rescan" looks at the time stamps of files in the database and only rebuilds pieces when they are out of date.

For more information on creating a database in parser mode, see **"Tutorial 3: Using the Compiler to Create a Static Analysis Database"** .

Parser Mode Shortcuts

If you want to use parser mode but wish to avoid waiting for the process to finish, there are two ways to speed things up:

- You can use the compiler with the `–nocode` flag to skip creating object files.
- You can build the Static Analyzer database using the compiler and bring up the graphic user interface later to read this database.

Rescanning the Fileset

After you have generated a database, you can always go back and rescan the fileset. The Admin menu provides two selections for this purpose:

"Rescan"

asks the Static Analyzer to check for new or modified files since the last scan and to store any cross–references found in new and modified files in the database. Use this command anytime you've modified source code files during a Static Analyzer session and you want to ensure that the Static Analyzer reflects those changes in the cross–reference files.

"Force Scan"

asks the Static Analyzer to completely rebuild the cross–reference files, creating a cross–reference database of all files specified in the fileset, whether or not they've been modified since the last scan. "Force Scan" also returns the Static Analyzer to its initial startup state with no query results in the main window and no past queries stored in the History menu. Use this command to restart the Static Analyzer and to verify the integrity of its cross–reference files.

There are also two command–line options involved with rescanning the fileset:

`–batch`

asks the Static Analyzer to perform the equivalent of the "Rescan" selection; it updates the cross–reference files to accommodate new and modified files in the fileset. It doesn't open the Static Analyzer's main window, however, and it quits the Static Analyzer once the scan is finished. You can use the **`–batch`** option to update cross–reference files for a large set of source code files, using the Static Analyzer as a background process. Note that you must have a fileset in the directory where you start the Static Analyzer or that you must specify a fileset when you start the Static Analyzer, or this option won't work.

`–noindex` (**Note:** scanner mode only)

asks the Static Analyzer not to create an inverted index for the cross–reference database, so it doesn't create the *.index* and *.posting* files. This makes creating a cross–reference database faster than it would be without the option, but the lack of an index makes queries to the database much slower. Use this option with caution.

Search Path for Included Files

Whenever the Static Analyzer scans a fileset and finds an included file in source code, it searches by default for the file in the current directory and then in */usr/include*. If it doesn't find the included file in either of these directories, it posts a Not Found dialog box that shows the names of those included files listed but not found in its search path.

To add directories to the search path for included files, choose "Set Include Path" from the Admin menu to open the Scanning Options dialog box shown in **Figure 4–2**.

The *Include Directories* list at the top of the box lists all directories that the Static Analyzer will search in addition to the default search path. To add a directory to the list, move the pointer to the *Directory* field below the list, type in a directory name, then press **`<Enter>`** (or left–click the *Add Directory* button). The path should be relative to the directory in which *cvstatic* is running. To delete a directory, left–click its name in the *Include Directories* list (this puts it in the *Directories* field), then left–click the *Remove Directory* button.

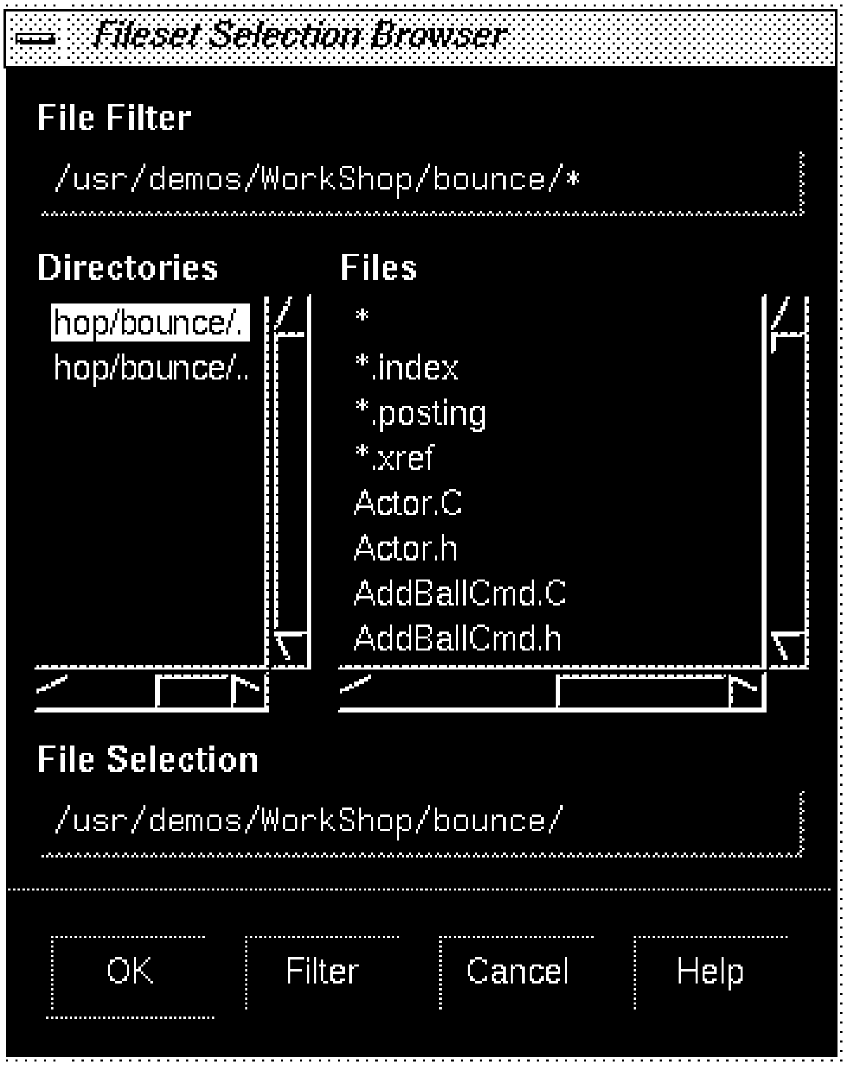
Figure 4–2 The Scanning Options Dialog Box

To exclude */usr/include* from the Static Analyzer's search path, left–click the *No Standard Includes* button to turn on the option. Turn on this option whenever you don't want to scan standard libraries and headers into a *.xref* file. By eliminating these files from a scan, you can greatly reduce the amount of data the Static Analyzer handles, increase its speed, and concentrate query results on your custom code. (Note, however, that you won't be able to find data in the header files normally found in */usr/include*.)

To close the Scanning Options dialog box, left–click the *Close* button. Note that any directories you added to the search path are stored as part of the fileset. You won't see the directories listed if you open the Fileset Editor, but you will see them if you examine the fileset file directly; each added search directory appears in a separate line with a `–I` prefix..

Changing to a New Fileset and Working Directory

The Static Analyzer uses only one fileset at a time, and resolves each relative pathname and general line from its current working directory. To change to a new fileset or a new working directory, use the Fileset Selection Browser window shown in **Figure 4–3** by choosing "Change Fileset..." from the Admin menu.



File Filter: /usr/demos/WorkShop/bounce/*

Directories:
- hop/bounce/.
- hop/bounce/..

Files:
- *
- *.index
- *.posting
- *.xref
- Actor.C
- Actor.h
- AddBallCmd.C
- AddBallCmd.h

File Selection: /usr/demos/WorkShop/bounce/

Figure 4–3 The Fileset Selection Browser Window

To load a new fileset, change to the directory in which it's located using the *Directories* list, then select the fileset in the *Files* list. Once you change to a new fileset, the directory where it's located becomes the new working directory.

You can use the *File Selection* field of the Fileset Selection Browser window to create a new fileset from within the Static Analyzer. If you enter a new filename such as *custom.fileset* in the *File Selection* field (as part of a full pathname) and then left–click *OK* to accept your new fileset, the Static Analyzer creates a file by that name and saves any fileset edits you make to that file.

Chapter 5

Static Analyzer: Queries

This chapter covers queries, which ask the Static Analyzer for specific information about the source code files included in the fileset. You'll find these topics here:

- **"Defining the Scope of a Query"**
- **"Making a Query"**
- **"Viewing Source Code"**
- **"Repeating Recent Queries"**
- **"Saving Query Results"**

For examples of using queries, refer to **"Tutorial 1: Applying the Static Analyzer to Scanned Files"** and **"Tutorial 2: Applying the Static Analyzer to Parsed Files"** .

Defining the Scope of a Query

The Static Analyzer has two types of query: comprehensive queries, such as "List All Functions" and "List Global Symbols", that don't require a query target; and specific queries, such as "Who Is Called By?" and "List Methods In Class", that require a query target. Specific queries are grayed in the Queries menu unless you supply target text in the *Query Target* field.

To enter text in the *Query Target* field, put the pointer in the text area and type. You can also left–click an element in the query results area; the Static Analyzer pastes it into the text area. For example, you can left–click a function name displayed in the query results area to enter the function name in the *Query Target* field.

To make a query based on target text, choose a query from the Queries menu. The Static Analyzer returns all elements that match the parameters of the query and the target text. You can also make a query by pressing **`<Enter>`** while the pointer is in the *Query Target* field; the Static Analyzer repeats the last type of query you made, using the contents of the *Query Target* field as target text. Use this convenient keyboard shortcut to repeat a query with a new target text each time—for example, if you're following a thread of function calls.

Target Text as a Regular Expression

The Static Analyzer reads target text in the *Query Target* field as a regular expression, which is a system of string constructions used by the UNIX *ed* command to construct literal strings or wild card strings. Regular expression syntax is described in the man pages for *ed*; you'll also find the same description on the Static Analyzer's Help screen for regular expressions.

If you enter target text without using any regular expression special characters (which are \, ., *, (,), [, ^, $, and +), the Static Analyzer reads the text as a literal string (without wild cards) and searches only for that text. If you use special characters to create a wild card expression, the Static Analyzer searches for a variety of target text in a single query, a useful tool for expanding the scope of a specific query.

Note: Don't confuse regular expressions with the shell expressions you use to create a fileset. They're completely different systems. For example, the regular expression "q.*" matches strings that begin with "q", while the shell expression "q.*" matches strings that begin with "q.".

Case Sensitivity

The Static Analyzer normally recognizes the difference between upper– and lowercase characters in target text during queries. If you want to ignore case in target text during a query (useful for Fortran code, which is case–insensitive), choose "General Options" from the Admin menu to open the General Options dialog box shown in **Figure 5–1** . Left–click the "Ignore Case In Searches" button to turn it on, then left–click the *Close* button to close the dialog box. You can return to the dialog box to turn off the option when you want to recognize case in queries.

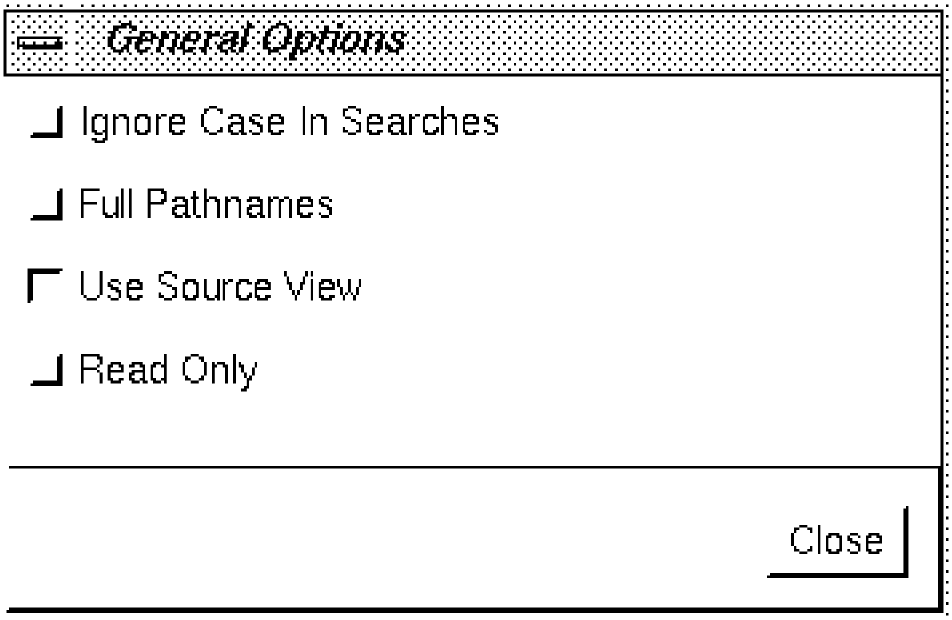


Figure 5–1 The General Options Dialog Box

Making a Query

To make a query, choose a query type from the Queries menu, which organizes queries in submenus of related queries (see **Figure 5–2**). The Static Analyzer shows the results in the query results area of its main window, using the view that is turned on. If the view isn't appropriate for the query (file dependency view for a function query, for example), the Static Analyzer informs you; no elements are listed in the query results area.

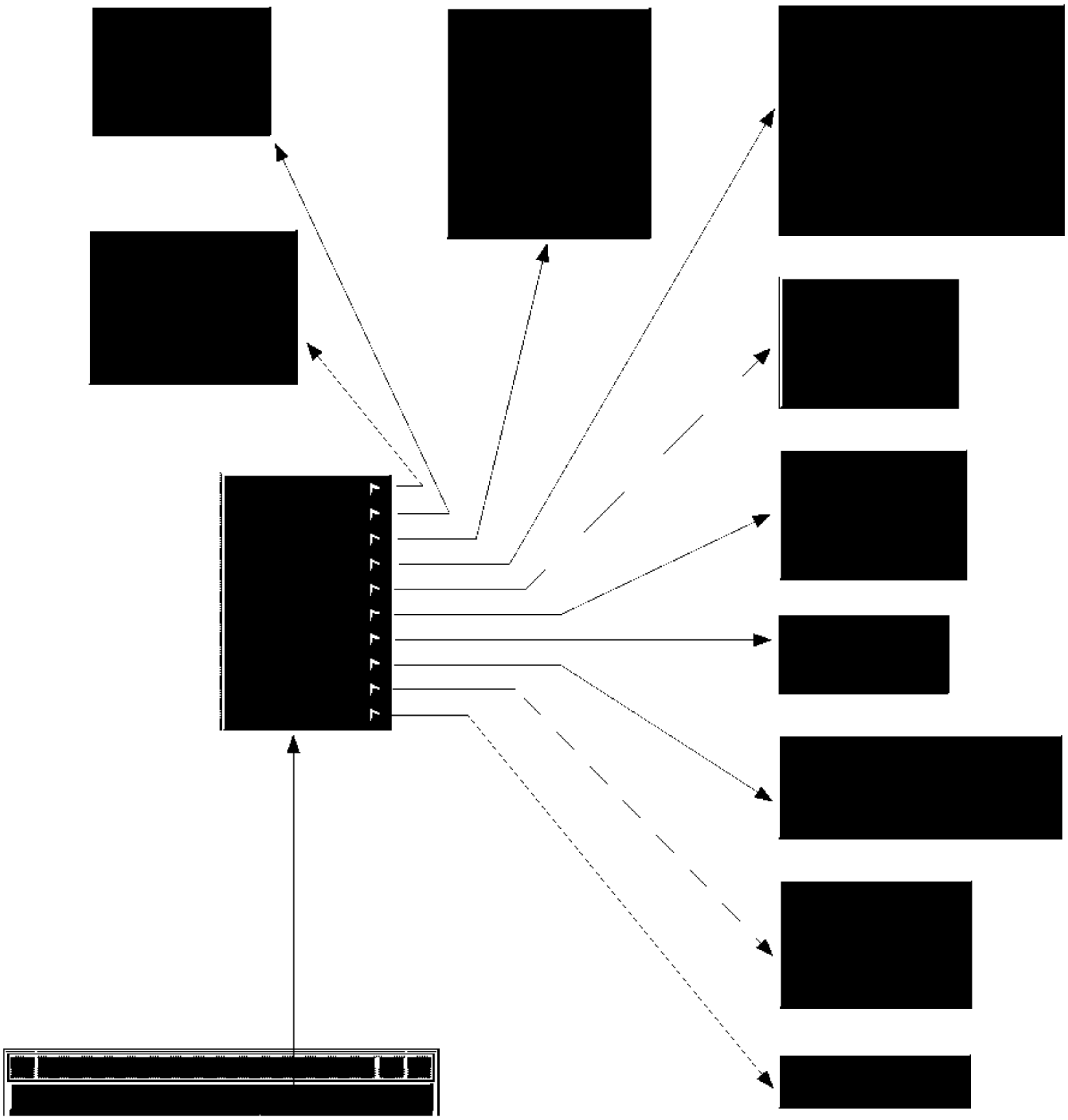


Figure 5–2 Static Analyzer Queries Menu with Submenus

General Queries

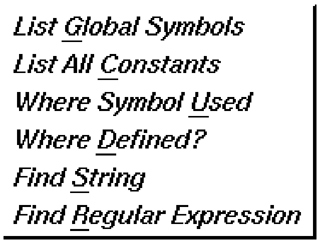


Figure 5–3 Queries Submenu: "General"

The "General" submenu (see **Figure 5–3**) contains a variety of general–purpose queries designed to find strings or nonspecific program elements. Several of these queries find symbols, which are programmatic tokens sent to the compiler—macro names, functions, variables, and other source code elements. The general queries are:

"List Global Symbols"

returns all global symbols it finds in the files defined by the fileset. It ignores any target text. (Global symbols are standard elements of code: functions, macros, variables, classes, and so forth.)

"List All Constants"

returns all constants in the source code (includes *enums*, named constants, and Fortran 77 parameters).

"Where Symbol Used?"

expects a symbol name in the *Query Target* field. It returns the source code locations of all references to the symbol.

"Where Defined?"

expects a symbol name in the *Query Target* field. It finds all symbols that match the target text and returns the source code locations where those symbols are defined.

"Find String"

expects a literal string in the *Query Target* field. It returns the source code locations of all strings that match the target text. When you use this query, you ask the Static Analyzer not to interpret the target text as a regular expression, which allows you to use regular expression special characters as part of a literal text string.

"Find Regular Expression"

expects a general expression in the *Query Target* field. It returns the source code locations of all strings that match the target text.

Macro Queries

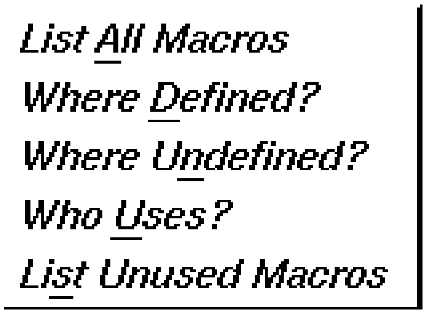


Figure 5–4 Queries Submenu: "Macros"

The "Macros" submenu (see **Figure 5–4**) contains queries that deal with macros:

"List All Macros"

returns all macros it finds in the files defined by the fileset. It ignores any target text.

"Where Defined?"

expects a macro name in the *Query Target* field. It finds all the macros that match the target text and returns the source code locations where those macros are defined.

"Where Undefined?"

expects a macro name in the *Query Target* field. It finds all the macros that match the target text and returns the source code locations where those macros are undefined (using *#undef*).

"Where Used?"

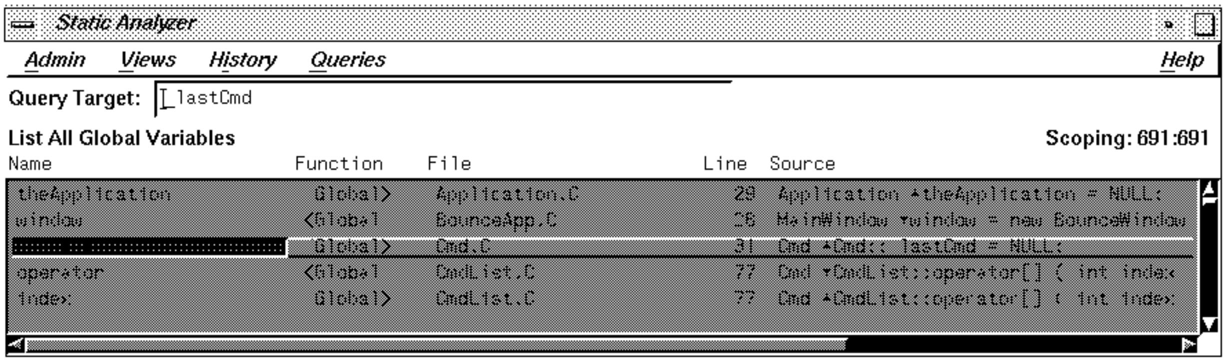
finds all locations where the macro entered in the *Query Target* field is used.

"List Unused Macros"

lists macros defined but never used.

Variable Queries

The "Variables" submenu contains queries that deal with variables. In performing a variable query, you typically list variables first and then select an individual variable for further information. **Figure 5–5** shows the results of the "List All Global Variables" selection with the *_lastCmd* variable selected. Notice that the variable list has five columns: *Name*, *Function*, *File*, *Line*, and *Source*. These identify the variable, its function or *<global>* if no function, the file in which it is defined or declared, the line number at which it is first defined or declared, and the actual source line.



Query Target: _lastCmd

List All Global Variables   Scoping: 691:691

```
Name                 Function    File               Line  Source
theApplication        Global>    Application.C        29  Application *theApplication = NULL;
window               <Global     BounceApp.C          26  MainWindow *window = new BounceWindow
                      Global>    Cmd.C                31  Cmd *Cmd:: lastCmd = NULL;
operator             <Global     CmdList.C            77  Cmd *CmdList::operator[] ( int index
index                 Global>    CmdList.C            77  Cmd *CmdList::operator[] ( int index
```

Figure 5–5 "List All Global Variables" Results with Variable Selected

From a list resulting from "List All Global Variables", you can select individual variables for specific queries. You do this by clicking the variable name. **Figure 5–6** shows the results of a "Who References?" query.

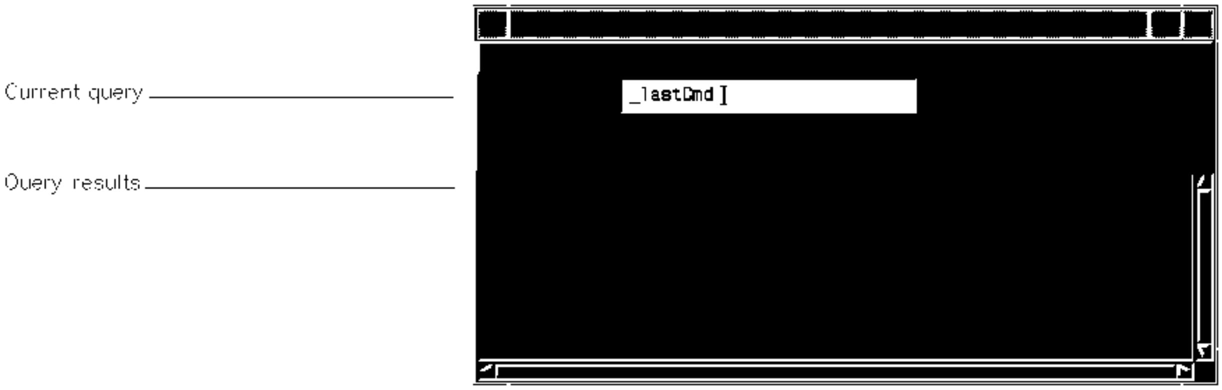


Figure 5–6 "Who References?" Results

The column headings are the same as for the "List All Global Variables" query results. In this case, the *Line* and *Source* fields refer to the line where the reference took place.

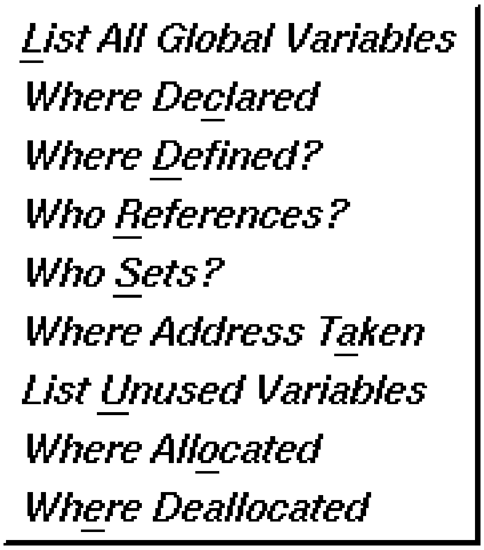


Figure 5–7 Queries Submenu: "Variables"

The "Variables submenu" appears in **Figure 5–7** and offers these selections:

"List All Global Variables"
returns the global variables it finds in files defined by the fileset. It ignores any target text.

"Where Declared?"
finds the location where the selected variable was declared.

"Where Defined?"
finds the location(s) where the variable was defined.

"Who References?"
expects a variable name in the *Query Target* field. It finds all variables that match the target text and returns all references to those variables.

"Who Sets?"
expects a variable name in the *Query Target* field. It finds all variables that match the target text and returns all source code locations where the values of those variables are set.

"Where Address Taken"
finds all locations where the address of the variable is taken.

"List Unused Variables"
lists all variables that have been defined or declared but not otherwise used in the source code.

"Where Allocated"
lists all locations where memory was allocated for the selected variable.

"Where Deallocated"
lists all locations where memory was deallocated for the selected variable.

Function Queries

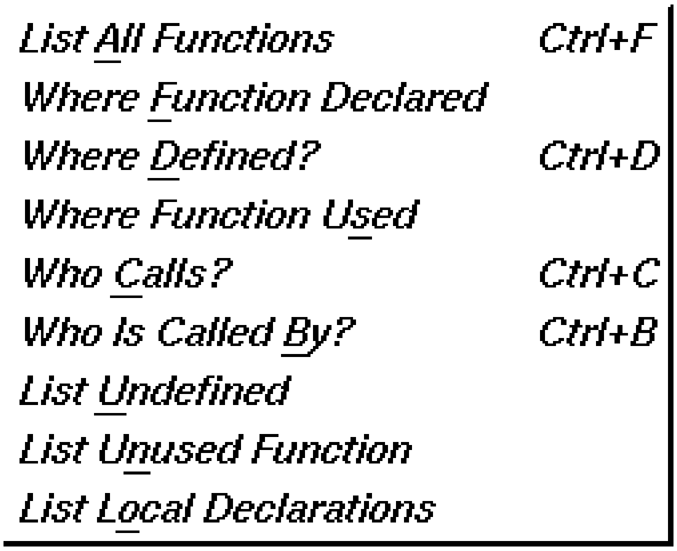


Figure 5–8 Queries Submenu: "Functions"

The "Functions" submenu (see **Figure 5–8**)contains queries that deal with functions. It operates in similar fashion to the variable queries—you create a list of functions and select individual functions for detailed queries. The selections are:

"List All Functions"
returns all functions it finds implemented in the fileset. It ignores any target text.

"Where Function Declared"
returns the location where the function was declared.

"Where Defined?"
returns all source code locations where those functions are defined.

"Where Function Used"
returns all source code locations where the function appears.

"Who Calls?"
returns all source code locations where the function is called.

"Who Is Called By?"
returns the names of all functions called by the selected (or entered) function, including the line number and source code where the call is made.

"List Undefined"
returns all functions called but not implemented in the fileset (usually library functions).

"List Unused Function"
returns functions that were declared or defined but not otherwise used in the source code.

"List Local Declarations"
returns all local variables and arguments in the source code and the line and source code in which the declaration is made.

Files Queries

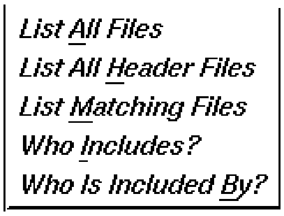


Figure 5–9 Queries Submenu: "Files"

The "Files" submenu (see **Figure 5–9**) contains queries that deal with files:

"List All Files"

returns all files included in the fileset as well as any included files specified by files within the fileset (such as header files). It ignores any target text.

"List All Header Files"

returns all header (*<filename>.h*) files in the fileset.

"List Matching Files"

expects either a filename in the *Query Target* field or no target text at all. If it finds target text, it returns all filenames that match the regular expression. If it finds no target text, it returns the same results as the "List All Files" query.

"Who Includes?"

expects a filename in the *Query Target* field or a selected filename. It returns the names of all files that include the file(s) specified by the target text.

"Who is Included By?"

expects a filename in the *Query Target* field or a selected filename. It returns the names of all files that are included by the specified file(s).

Class Queries

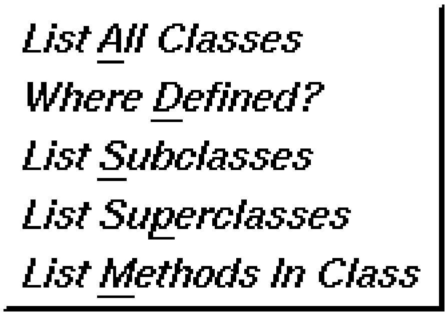


Figure 5–10 Queries Submenu: "Classes"

The "Classes" submenu (see **Figure 5–10**) contains queries that deal with C++ classes:

"List All Classes"

returns all classes it finds in the files defined by the fileset. It ignores any target text.

"Where Defined?"

expects a class name in the *Query Target* field. It finds all classes that match the target text and returns the source code locations where those classes are defined.

"List Subclasses"

expects a class name in the *Query Target* field. It returns the immediate subclasses of the classes matching the target text.

"List Superclasses"

expects a class name in the *Query Target* field. It returns the immediate superclasses of the classes that match the target text.

"List Methods In Class"

expects a class name in the *Query Target* field. It returns those methods defined within the classes that match the target text.

Method Queries

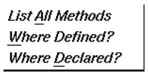


Figure 5–11 Queries Submenu: "Methods"

The "Methods" submenu (see **Figure 5–11**) contains queries that deal with C++ member functions, also called methods:

"List All Methods"
returns all methods in the fileset. It ignores any target text.

"Where Defined?"
expects a method name in the *Query Target* field. It finds all methods that match the target text and returns all source code locations where those methods are defined.

"Where Declared?"
expects a method in the *Query Target* field. It returns source code locations of all class declarations that include methods that match the target text.

Common Blocks Queries

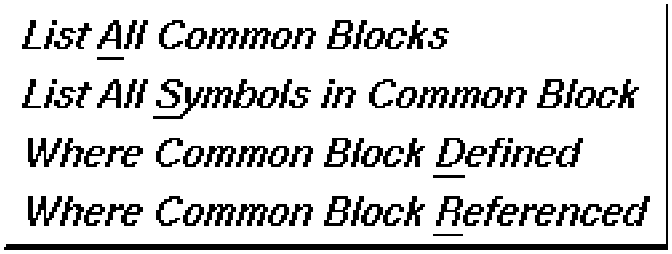


Figure 5–12 Queries Submenu: "Common Blocks"

The "Common Blocks" submenu (see **Figure 5–12**) applies to Fortran source code only. It contains these selections:

"List All Common Blocks"
lists all common blocks in the fileset.

"List All Symbols in Common Block"
lists all symbols used in common blocks in the fileset.

"Where Common Block Defined"
expects a common block in the *Query Target* field. It finds all common blocks that match the target text and returns the source code locations where the common blocks are defined.

"Where Common Block Used"
returns all source code locations where the common block appears.

Types Queries

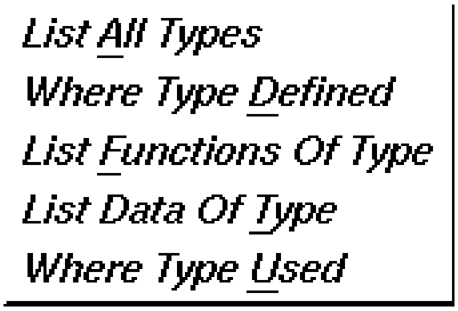


Figure 5–13 Queries Submenu: "Types"

The "Types" submenu (see **Figure 5–13**) helps you get type information. It contains

"List All Types"

"Where Type Defined"

expects a type in the *Query Target* field. It finds all types that match the target text and returns the source code locations where the types are defined.

"List Functions of Type"

returns all functions of the given type and the source code locations where they are declared or defined.

"List Data of Type"

returns all data declarations and definitions using the given type and the source code locations where they are declared or defined.

"Where Type Used"

returns all source code locations where the type and functions and data items using the type appear.

Directories Queries

Figure 5‒14 Queries Submenu: "Directories"

The "Directories" submenu (see **Figure 5‒14**) helps you determine the organization of the current fileset. It contains these selections:

"List Directories"

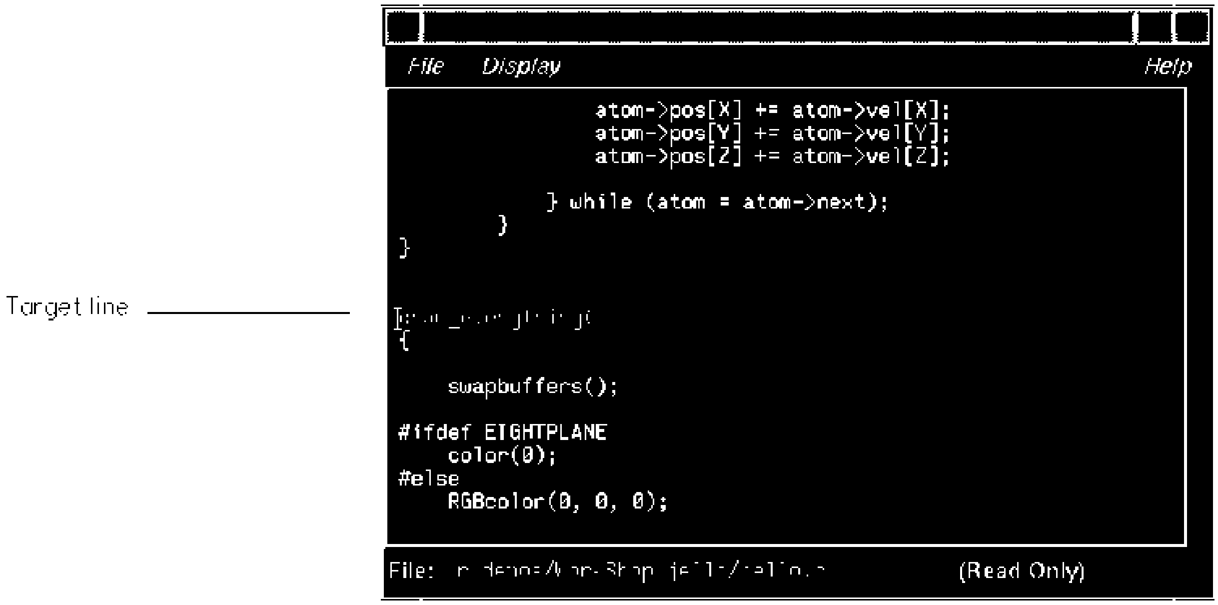
lists all directories in the fileset.

"List Files"

lists all files in the fileset.

Viewing Source Code

When the Static Analyzer returns query results, you can look at each element's source code. To do this, left‒double‒click an element in the query results area, or left‒single‒click an element and then choose "Edit" from the Admin menu. Either of these actions opens up the Source View window shown in **Figure 5‒15** .



```
                atom->pos[X] += atom->vel[X];
                atom->pos[Y] += atom->vel[Y];
                atom->pos[Z] += atom->vel[Z];

            } while (atom = atom->next);
        }
}

[illegible]
{

    swapbuffers();

#ifdef EIGHTPLANE
    color(0);
#else
    RGBcolor(0, 0, 0);
```

File: [illegible] (Read Only)

Figure 5–15 The Source View Window with Highlighted Source Code

The Source View window opens the file containing the element. The line of code that contains the element displays in the center of the window and is highlighted. The line location is also indicated in the vertical scroll bar. If you scroll away from the target line, you can quickly scroll back by dragging the scroll bar handle back to the indicator band.

Although the Source View window is set by default to be read only, you can edit text if you wish. If you have a configuration management tool installed, you can use the "Versioning" selection from the File menu to check out the file for editing. You'll find details about using the Source View window in the *CASEVision Environment Guide*.

Alternate Text Editors

If you prefer to view source code in a text editor other than Source View, you can choose "General Options" from the Admin menu to open the General Options dialog box, which offers the option *Use Source View.* Turn this option off to use the default alternate text editor, *vi,* whenever you left–double–click to see source code. To set a different alternate text editor, add the line

```
*editorCommand: <editor>
```

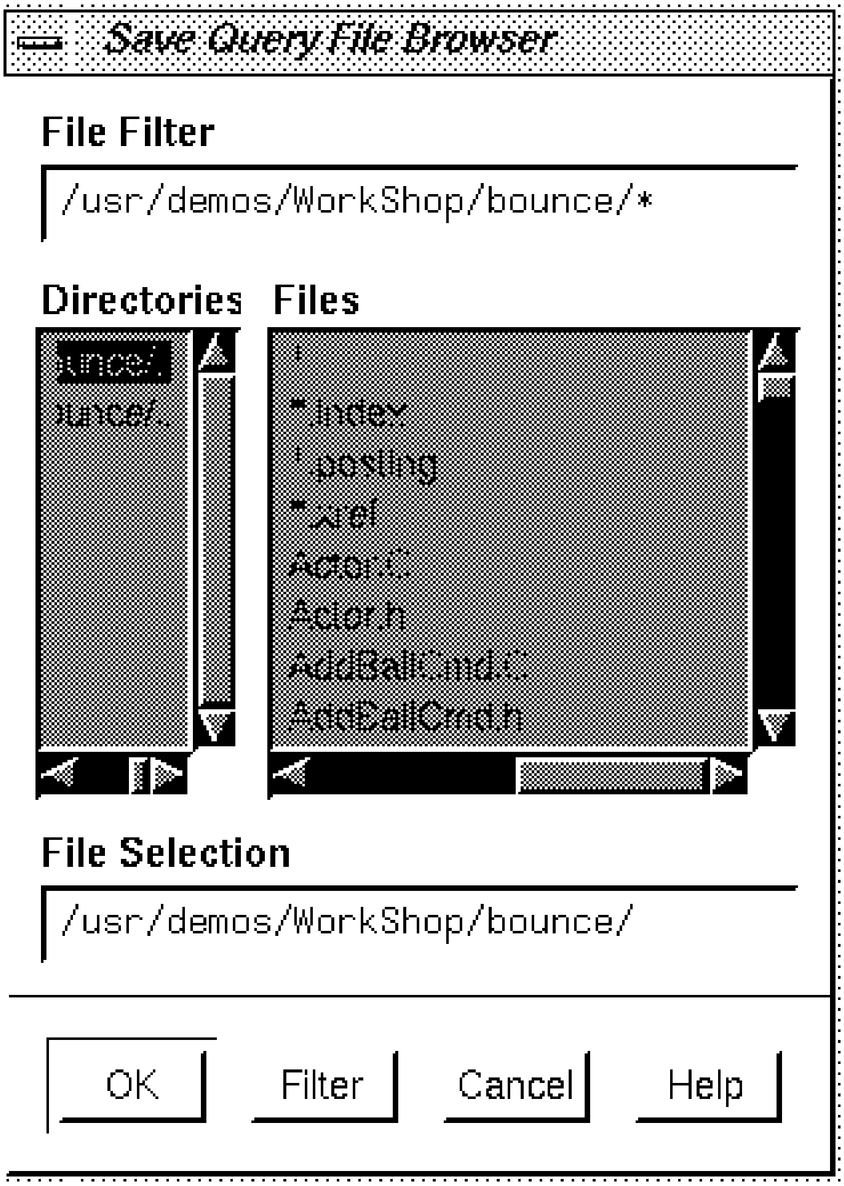
to your *.Xdefaults* file, where <*editor*> is the command for the editor you wish to use. The next time you use the Static Analyzer with the *Source View* option turned off, the editor you specified will appear when you view source code.

Repeating Recent Queries

The Static Analyzer retains a list of your 15 most recent queries and presents them in the History menu. You can choose any of the queries listed in this menu to repeat the query. The Static Analyzer remembers the query type and the target text it used; it doesn't remember any view settings, such as the view type, view options, or Scope Manager settings. If you change view settings and then choose a query from the History menu to repeat the query, the Static Analyzer will return the same query results but will display them differently.

Saving Query Results

You can save query results by choosing "Save Query..." from the Admin menu to open the Save Query File Browser window shown in **Figure 5–16** . This is the standard Save File Browser, described in the *CASEVision Environment Guide*.



File Filter

/usr/demos/WorkShop/bounce/*

File Selection

/usr/demos/WorkShop/bounce/

Figure 5–16 The Save Query File Browser Window

To save query results, use the *Directories* list to move to a directory in which you want to save. Move the cursor into the *File Selection* field, and type the filename under which you want to save (put it at the end of the pathname listed there). Left–click the *OK* button to save the query results and close the Save Query File Browser window.

The Static Analyzer saves the contents of the query results area to the file you named in the Browser. If you're in Text View, the Static Analyzer saves the results in text format; if you're looking at a graphical view, the graph is saved in PostScript format. It adds a heading to the text that lists query type and the target text that specified the query. It also includes field headings (such as Function, File, and so on) to match those at the top of the query results area in the main window.

Chapter 6

Static Analyzer: Views

This chapter discusses the different views available in the Static Analyzer to view your query results. The selections "Text View," "Call Tree View," "Class Tree View," and "File Dependency View" are available in the Static Analyzer Views menu. The "Results Filter..." selection can be accessed from the Static Analyzer Admin menu. You'll find these topics:

- **"Text View"**
- **"Call Tree View"**
- **"Class Tree View"**
- **"File Dependency View"**
- **"The Results Filter"**

Text View

Text View is the Static Analyzer's default view. It displays the results of any query and, because it's limited to text, displays query results faster than any of the tree views.

Text View provides labels at the top of the query results area (as shown in **Figure 6−1**) that identify the query type, show the extent of Results Filter reductions (called *Scoping* field, discussed later in this chapter), and label the columns in the query results area. Below the labels, the Static Analyzer lists the elements returned by a query, one element per line.

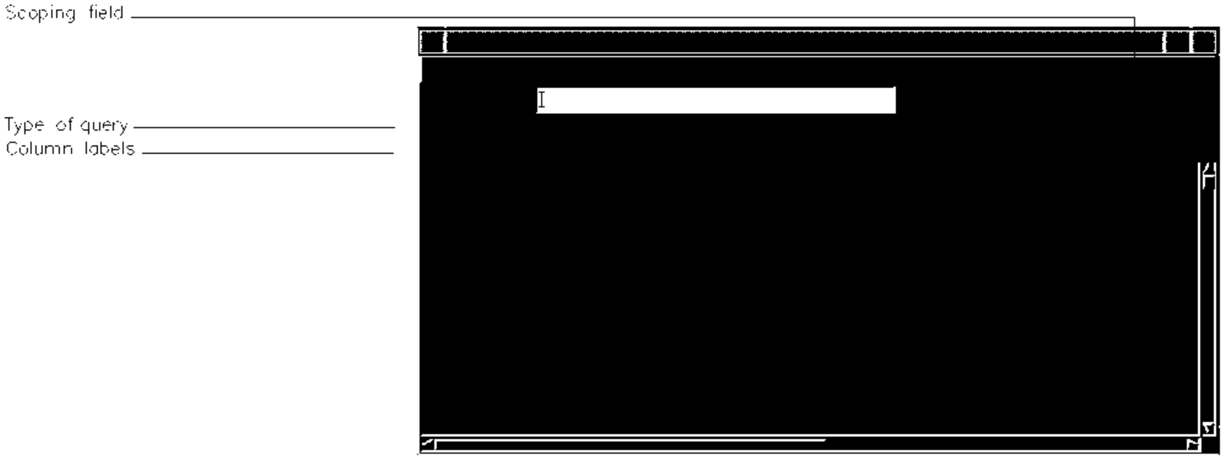


Figure 6−1 Text View Labels

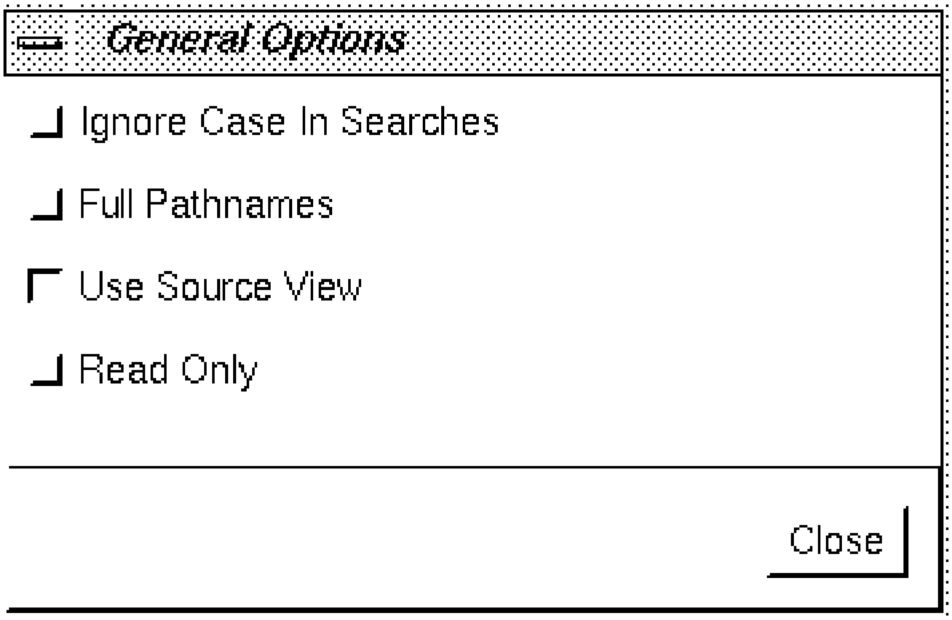
Text View's arrangement of information within each element line depends on the query type. The left field always lists the type of element you searched for; fields to its right show the location of that element and, if applicable, the contents of the source code line where it's located. For example, Text View shows the results of a function query with the function name in the first field, the filename where the function is located in the second field, the line number of the source code line where the function is defined in the next field, and the text of the line in the last field. For class queries, Text View shows any superclasses of returned classes, and for method queries, it shows the class where each method is defined.

Use the horizontal and vertical scroll bars to scroll left and right to see the full contents of long lines or up and down to work through long lists of elements respectively. To see more information at one time, you can enlarge the Static Analyzer window by left−dragging a corner.

To see the source code listing where an element occurs, you can left−double−click any element line to open the Source View window. It displays the selected element in the middle of the window, surrounded by adjacent code.

Viewing Full Pathnames for Files

Text View normally shows filenames in the query results area as short base names. If you want to see the directory as well as the filename (or at least as full a pathname as the Static Analyzer can find), turn on the *Full Pathnames* option: Choose "General Options" from the Admin menu to open the General Options dialog box shown in **Figure 6–2**, then left–click the *Full Pathnames* button to turn on the option. Left–click the *Close* button to close the dialog box.



☐ Ignore Case In Searches

☐ Full Pathnames

☑ Use Source View

☐ Read Only

Figure 6–2 The General Options Dialog Box

To return to base filenames, reopen the General Options dialog box and turn off the *Full Pathnames* option.

Sorting Elements in Text View

The Static Analyzer normally presents elements in the order in which they appear within each file of the fileset. To sort the elements in alphanumerical order by a single field, left–click the field you want within any element line, then choose "Sort" from the Admin menu. The Static Analyzer sorts the elements in ascending order by that field.

Call Tree View

Call Tree View is designed to display functions and the static calls between them in a graphic tree form. Because it's intended for functions, it shows results only for function queries, not for other types of queries such as file and class queries. A line of text above the query results area identifies the last type of query made and shows the extent of Scope Manager reductions.

To use Call Tree View (shown in **Figure 6–3**), choose "Call Tree View" from the Views menu. It presents each function in the query results area as a node, a small movable box labeled with the function name, and each function call as an arc, an arrow drawn from the calling function to the called function. Because the function relationships are presented in a tree structure, higher–level functions normally appear on the left side of the window. They call lower–level functions located farther to the right.

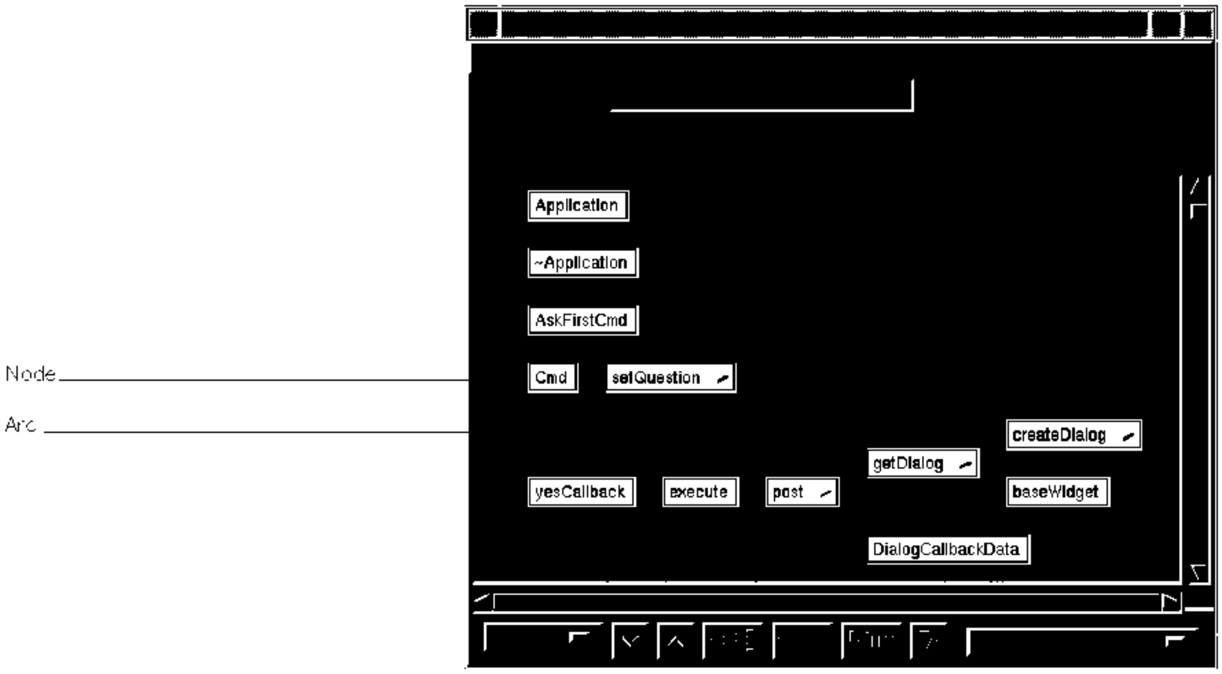


Figure 6–3 Call Tree View Displaying Functions and Function Calls as Nodes and Connecting Arcs

The Static Analyzer Graph Control Panel

The Static Analyzer graph view control panel (shown in **Figure 6–4**) below the query results area offers a set of controls that you can use to change the view. They help you see query results in the form most useful to you.

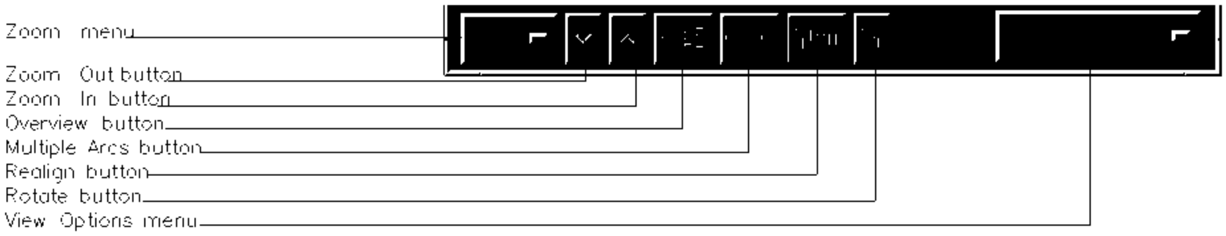


Figure 6–4 The View Control Panel

To change the scale of the call tree in the query results area to see more or less of the tree at one time, use the zoom controls: the *Zoom menu* and the *Zoom In* and *Zoom Out* buttons. If the tree you're viewing doesn't fit entirely within the boundaries of the query results area, you can view other parts of the tree by using the scroll bars or clicking the *Overview* button and navigating in the Overview window. By default, Call Tree View shows only a single arc between two functions, even if the calling function calls more than once. To see multiple calls between functions in the call tree, click the *Multiple Arcs* button. After maneuvering nodes, you can return them to their default positions, by clicking the *Realign* button. The Static Analyzer's default tree orientation is horizontal; the tree grows from left to right. To see vertical tree orientation, that is, top–down, (or toggle back to horizontal), click the *Rotate* button.

Call Tree View allows you to directly manipulate nodes and arcs in the query results area. You can hide, reveal, and rearrange nodes, and you can select a node or an arc to view either a function or a function call in the Source View window.

For more information on the graph controls and node/arc manipulation, see **Appendix A, "Graphical Views in the CASEVision Environment"**

Setting View Options

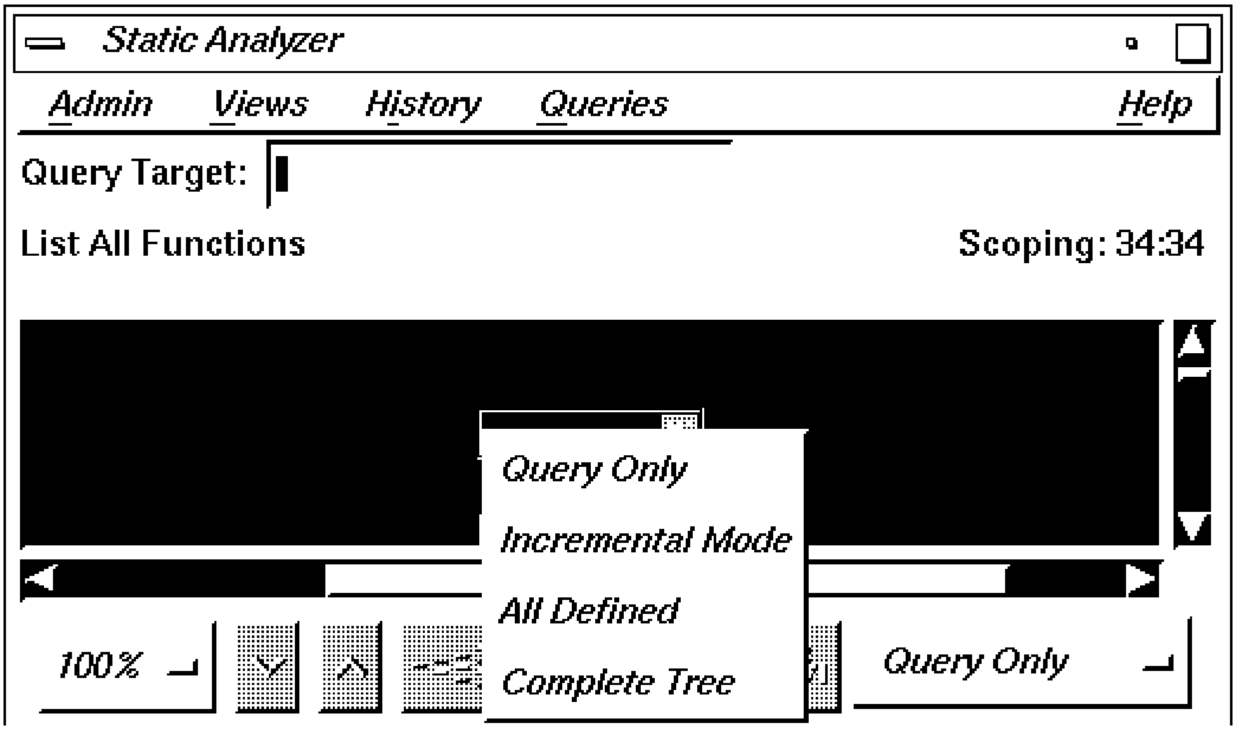


Figure 6–5 View Options Menu

The View Options menu (at the lower right of the Call Tree View window) has four view selections that change the number of nodes you see in the query results area and change the way query results are cleared between queries (see **Figure 6–5**). To open the menu, move the pointer over it and hold the left mouse button down. Drag up or down to the selection you want, then release the button. The selections are:

"Query Only"

is the default selection. Only the target and results of each query appear in the query results area. Each time you make a new query, the results of the old query are cleared before the new results appear.

"Incremental Mode"

is useful for following a chain of function calls. In incremental mode, the Static Analyzer doesn't erase the results of the previous query from the query results area. It adds the results of the latest query to the nodes and arcs already on the screen, so you can incrementally build a tree as you follow function calls. The Static Analyzer shows the target and the results of the latest query in target and result colors. It shows all other nodes in the non–query color so that you can see which nodes were returned by the query and which nodes were there before the query.

"All Defined"

shows at all times a complete tree of all functions defined (that is, implemented) within the fileset. When you make a function query, the Static Analyzer shows the query target and result nodes in target and result colors. All other nodes appear in the non–query color, so that the query results stand out as a subtree within the overall function tree.

"Complete Tree"

shows a complete tree at all times of all the functions known within the fileset, whether they're defined or not. This includes all the defined functions shown in "All Defined" libraries, even a small program can generate a huge complete tree. The "Complete Tree" selection, like the "All Defined" option, shows the results of any queries you make by highlighting in target and result colors, leaving all other nodes in non−query colors.

Caution: The Complete Tree selection can easily create unmanageably large trees for even small programs, so use it with care.

Viewing Function Definitions and Calls in Source View

To view a function definition in Call Tree View, either select the function's node and choose "Edit Selected Item" from the Admin menu, or left−double−click the function's node. The Source View window opens with the beginning of the function definition highlighted amid surrounding code.

Call Tree View offers a Source View function not available in Text View: You can view a function call by left−double−clicking on an arc that connects two functions. The Source View window shows the line of code (listed within the calling function) that calls the called function. You can get the same results by selecting an arc and then choosing "Edit Selected Item" from the Admin menu.

Tutorial: Working in Call Tree View

This tutorial traces function calls in Call Tree View using the "Incremental Mode" and "All Defined" viewing options. It first goes from higher− to lower−level functions using queries, and then returns to higher−level functions by showing parent nodes using the Node menu.

1. Move to the demo directory *jello*:

   ```
   cd /usr/demos/WorkShop/jello
   ```

2. Make sure that no fileset and cross−reference files exist in the directory so that the Static Analyzer will create its own standard default files:

   ```
   rm cvstatic.*
   ```

3. Start the Static Analyzer:

   ```
   cvstatic &
   ```

4. Select "Edit Fileset" from the Admin menu and move the *jello.c* file into the *Scanner Fileset* field using the *Move Files Scanner* button. Click *OK*.

   This creates the fileset for this tutorial.

5. Choose "Call Tree View" from the Views menu to put the Static Analyzer in Call Tree View.
6. Choose "Incremental Mode" from the View Options menu on the bottom right side of the control panel to turn on the "Incremental Mode" view option.
7. Move the pointer into the *Query Target* field and type **`main`**.
8. Choose "Who Is Called By" from the "Functions" submenu of the Query menu to find the functions that *main()* calls.

   The Static Analyzer displays a node named *main* on the left side of the query results area, which displays in the target color for this scheme. It's connected by arcs to a set of lower−order function nodes to the right, all in the result color.

9. Drag the vertical scroll bar of the query results area down until you see the *draw_everything* node, then left−click on it to select it.

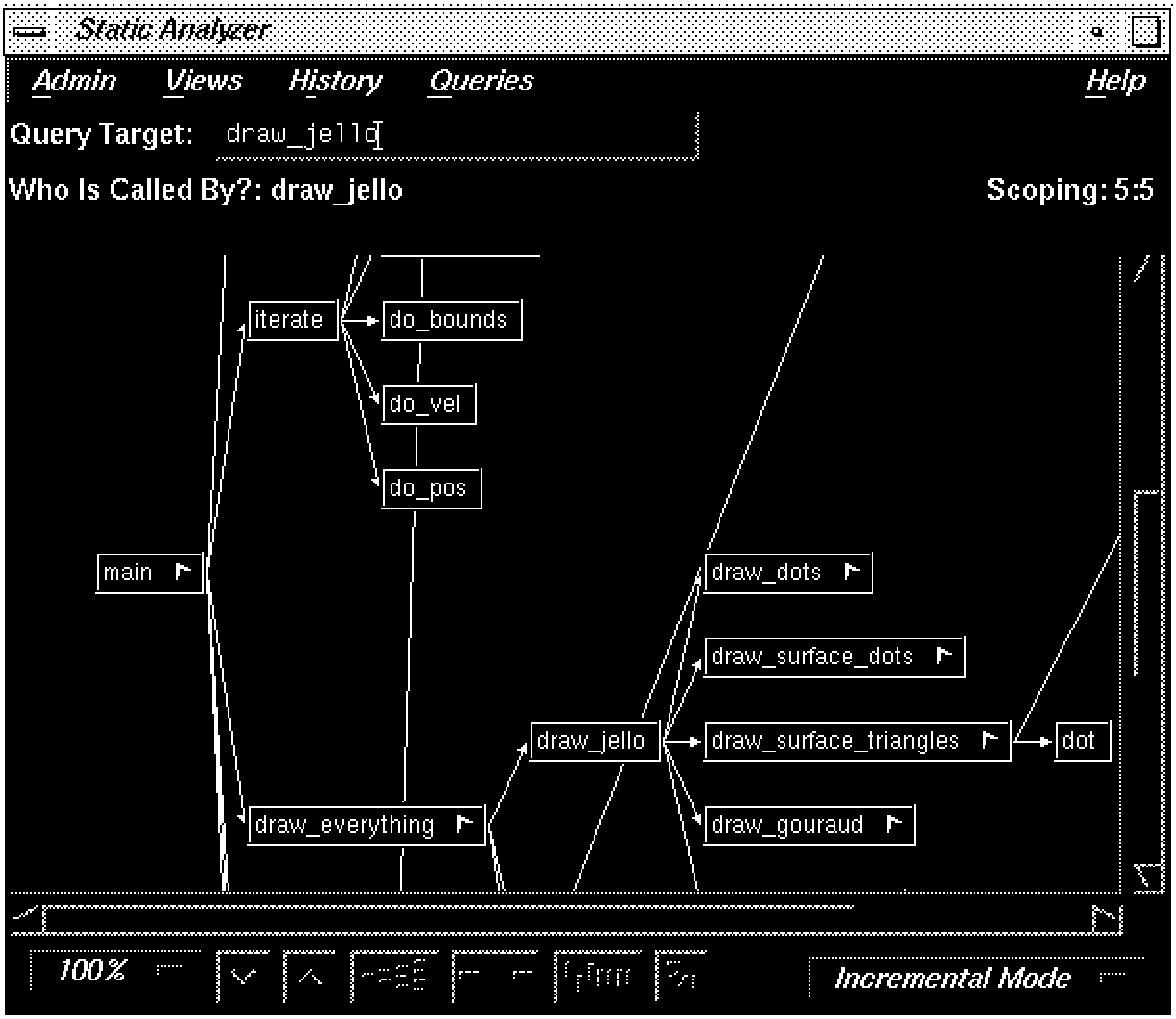
   `draw_everything` appears in the *Query Target* field.

10. Move the pointer into the *Query Target* field, then press **`<Enter>`**.

    The Static Analyzer repeats its last query using the new target and returns *draw_everything* as a target node with several result nodes to its right. The nodes from the previous query—*main* and its other children—still appear in the query results area in non−query color.

11. Select the result node *draw_jello*, move the pointer into the *Query Target* field, and press **`<Enter>`** to search for all functions called by *draw_jello()*.

    The Static Analyzer returns *draw_jello* as a target node with result nodes to its right as shown in **Figure 6–6** . The nodes from the two previous queries are still in the query results area.



Query Target: draw_jello

Who Is Called By?: draw_jello

Scoping: 5:5

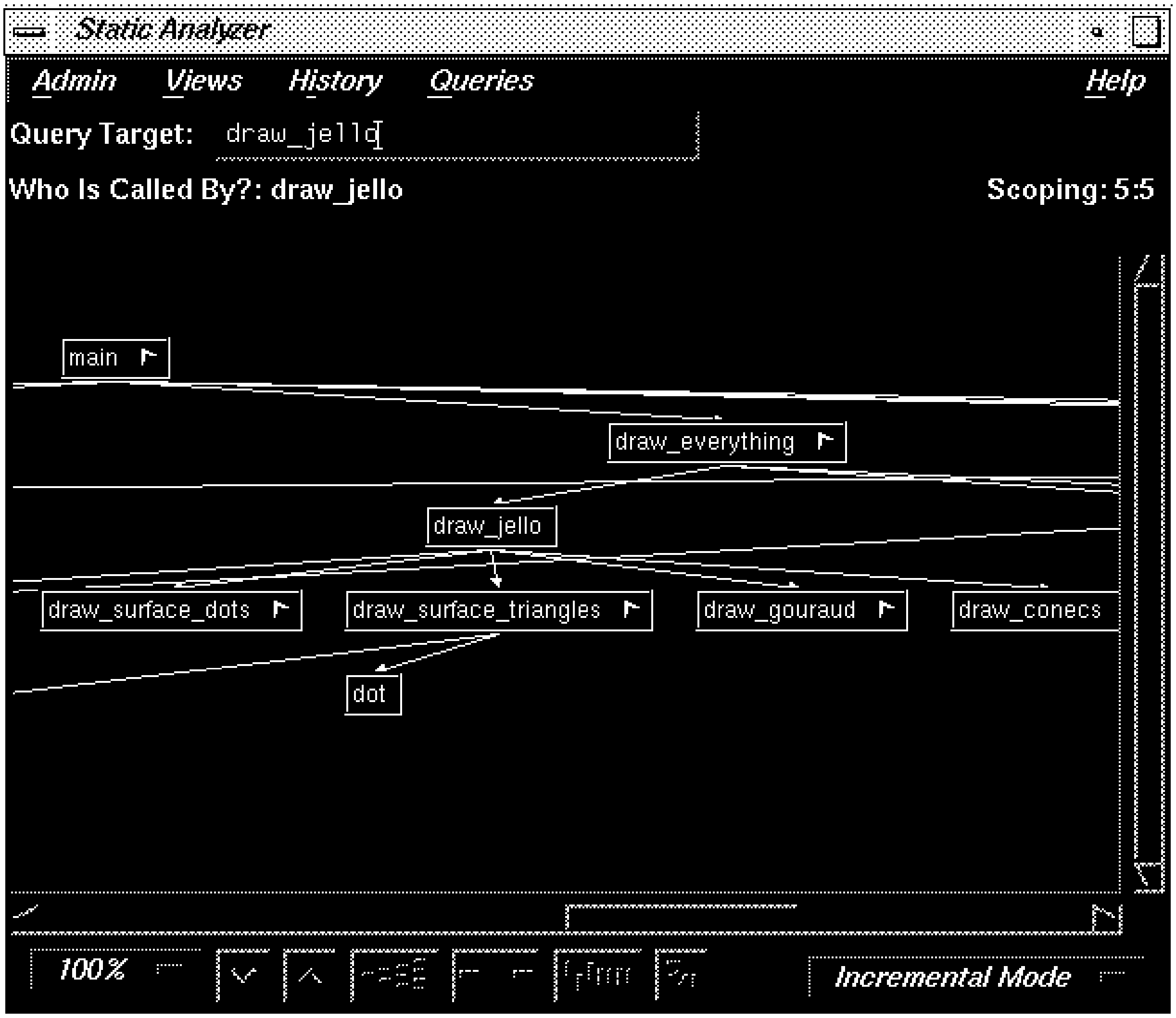
Figure 6–6 Target and Result Colors Marking Nodes

12. Choose "40%" from the Zoom menu to set scaling to 40%.

    The call tree reduces in size so that you can see most of the full call tree.

13. Left–click the *Rotate* button to set the orientation to vertical.

    The call tree turns 90 degrees, growing from the top of the query results area to the bottom. The window width allows the full tree to fit in the query results area as shown in **Figure 6–7** .



Query Target: draw_jello

Who Is Called By?: draw_jello

Scoping: 5:5

Figure 6–7 A Tree in Horizontal Orientation

14. Left–click the *Rotate* button again to return orientation to vertical.
15. Choose "Query Only" from the View Options menu at the bottom right to return to the "Query Only" viewing mode (the Static Analyzer's default view option).

    The non–query nodes disappear from the query results area, leaving only the results of the last query.
16. Choose "100%" from the Zoom menu to return the scale to 100%.
17. Move the pointer over the target node *draw_jello*, hold down the right mouse button to open its node menu, and choose "Show Parents" from the menu.

    The node *draw_everything* appears to the left.
18. Move the pointer over the node *draw_everything,* and choose "Show Parents" from its node menu.

    The node *main* appears to the left.

Class Tree View

Class Tree View, which you set by choosing "Class Tree View" from the Views menu, displays a class inheritance tree containing the classes found in C++ files in the fileset. It's not intended for nonclass elements, and it won't show the results of function, file, and method queries, for example.

Class Tree View looks almost identical to Call Tree View. It includes a line of text above the query results area that lists the last query and the extent of Results Filter reductions; it shows elements in the query results area using nodes and arcs; it offers a control panel to change the view in the query results area. The main difference is that each node in Class Tree View represents a class instead of a function, and each arc shows inheritance instead of a function call. Class trees in horizontal orientation move from superclasses on the left to subclasses on the right.

When you make class queries in Class Tree View, the Static Analyzer uses colors in the same way that it does in Call Tree View: target color to mark target nodes, results color to mark result nodes, and non−query color to mark nodes not returned by the last query. The view controls also work the same way, with one minor variation. The *Multiple Arcs* button has no effect because no multiple inheritances exist in a class tree.

The selections in the Node and the Selected Node menus work the same way they do in Call Tree View, working through parents and children of existing nodes, but they follow class inheritance instead of a chain of function calls. Using the Source View window in Class Tree View has one minor difference: you can left−double−click on a node to view source code for a class, but you can't left−double−click on an arc to see an inheritance.

Note: If you have purchased WorkShop C++ Professional from Silicon Graphics, you will have access to the C++ Browser, which lets you gather additional information on the structure, hierarchy, and method interactions of each C++ class in your application or library.

File Dependency View

File Dependency View, which you set by choosing "File Dependency View" from the Views menu, displays the include relationships between files in the fileset. File Dependency View is similar to Class Tree View and offers the same controls, colors, and menus. The main difference is that each node in this view represents a file in the fileset instead of a function, and each arc shows the inclusion of one file by another. An arc leads from the including file to the included file.

Although File Dependency View only displays files, it can provide useful information when used in conjunction with other types of queries. For example, if File Dependency View is displayed and you select "Where Used" from the "Function" submenu, those files containing the specified function will be highlighted.

An include tree in horizontal orientation places including files on the left and included files on the right. If you use selections from the Node and Selected Node menus to work through parents and children of existing nodes, you follow include relationships. A child of a node is a file included by that node; a parent of the node is a file that includes that node.

The Results Filter

The Results Filter is a view tool that works in all of the Static Analyzer's views; it filters the view to show you a subset of all the results returned by a query. The Results Filter filters only the view of query results, not the results themselves. For example, if a function query returns 18 functions and the Results Filter is set to filter out 5 of them, the query results area shows only 13 functions. The Static Analyzer, however, retains the full 18 functions returned by the query; it simply hides the 5 functions filtered by the Results Filter. If you turn off all filters in the Results Filter, you then see the full 18 functions in the query results area.

When the Results Filter is set to filter, its filters remain turned on to affect the view of any future queries you make. For example, if the Results Filter is set to filter out all elements contained in header files, it does so for all queries that follow. It removes variables found in header files from a "List All Global Variables" query, and it removes header files from a "List All Files" query. You must turn off the filters if you want to see the full results of a query.

The Scoping line, located just above the right corner of the query results area, tells the extent of any filtering performed by the Results Filter. It lists two numbers separated by a colon: the first is the number of elements returned after filtering, the second is the full number of elements returned by the query. For example,

```
Scoping: 78:154
```

tells you that 154 elements were returned by the current query, and after filtering, the Results Filter shows 78 of them in the query results area.

Setting Results Filters

To open the Results Filter window shown in **Figure 6–8**, choose "Results Filter" from the Admin menu.

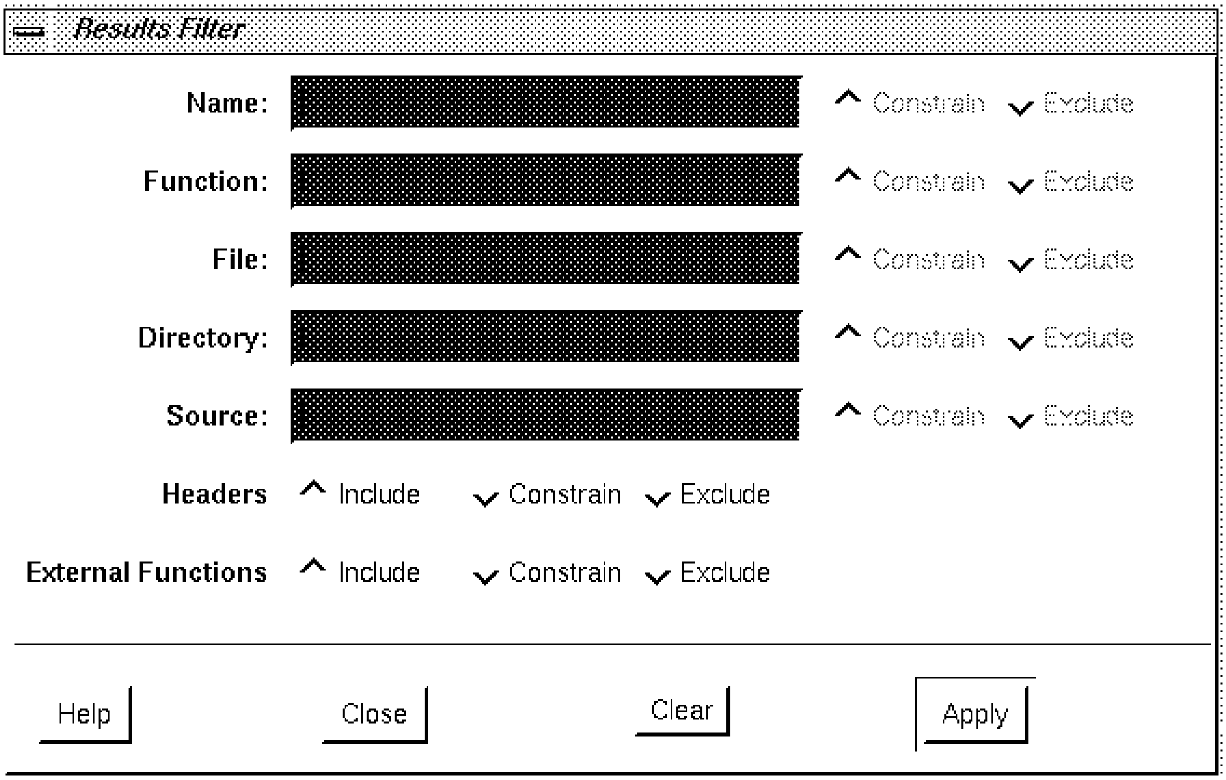


Figure 6–8 The Results Filter Window

The Results Filter has seven different scope filters. The first five filters provide fields in which you can enter regular expressions, which allow you to specify a literal string of characters or a wild card expression that matches a set of strings. The last two filters require specific files and functions. The filters are:

Name

filters by the *Name* field in Text View. The *Name* field can list variables for a variable query, target functions for a function query, or other parts of elements, depending on the query type.

Function

filters by the *Function* field in Text View. This field can list functions called by a target function, functions that define local variables, and other types of functions, depending on the query type.

File

filters by the *File* field in Text View. This field can exclude elements contained in specified files or show only elements contained in specified files.

Source

filters by the *Source* field in Text View. This field can exclude or constrain elements

Directory

filters by the *Directory* field in Text View. This field can exclude elements contained in specified directories or show only elements contained in specified directories.

Headers

filters according to whether elements are contained in a header file or not.

External Functions

filters according to whether elements are contained in externally defined functions or not.

Although the first five scope filters work using fields in Text View, their results are the same in tree views such as Call Tree View. They sort by invisible criteria in these views. For example, you can sort with the *Source* scope filter in Call Tree View, even though Call Tree View doesn't show the *Source* field for each function it displays.

Filtering by Name, Function, File, and Source

To filter using the first five scope filters, enter a regular expression in the appropriate text area, and then left−click on either the *Constrain* or *Exclude* button following the text area. *Constrain* filters elements so that only those that match the regular expression in the appropriate field are displayed in the query results area. *Exclude* filters elements so that elements that match the regular expression in the appropriate field aren't displayed in the query results. For example, if you enter **`jello.c`** in the *File* scope filter and left−click the *Constrain* button, the Static Analyzer displays only elements found in the file *jello.c*.

To turn off filtering by any one of these five filters, delete all text from its text area.

Filtering by Header Files and External Functions

The *Headers* scope filter allows three options:

Include

displays elements found in header files in addition to elements found in other files.

Constrain

displays only elements found in header files.

Exclude

displays only elements not found in header files.

The External Functions scope filter also allows three options:

Include

displays elements found in externally defined functions (functions defined in files outside of the fileset) in addition to elements found in internally defined files.

Constrain

displays only elements found in externally defined functions.

Exclude

displays only elements not found in externally defined functions.

To turn off filtering by either of these two filters, left−click their *Include* button.

Combining Results Filters

You can use results filters singly or in combination to limit the elements you see to a very specific subset of the query results. For example, you can set the File filter to show only elements found in the file *jello.c*. You can then further refine the filtering by setting the Function filter to show only elements found in the function *draw_everything()*. The Static Analyzer combines these two filters to show only elements found in the function *draw_everything()*, which is contained in the file *jello.c*.

Using the Results Filter Buttons

The Results Filter window displays a row of four buttons across the bottom of the window:

Apply

applies current scope settings to the query results area to filter out elements. The Static Analyzer automatically applies scope settings whenever you left−click an *Include, Exclude,* or *Constrain* button, so you don't usually need to left−click on the

Apply button.

Clear	clears text from all text fields and returns the bottom two filters to the *Include* setting. Left−click on *Clear* whenever you want to turn off filtering by the Results Filter.
Close	closes the Results Filter window.
Help	opens the Help window, where you can find information about the Results Filter window.

Tutorial: Using the Results Filter

This tutorial uses the Results Filter to view, in Text View, selected methods in a fileset of C++ files. It first filters the methods by file and then filters them further by a string found within each method's source code line.

1. Move to the demo directory *bounce*:

   ```
   cd /usr/demos/WorkShop/bounce
   ```

2. Make sure that no fileset and cross−reference files exist in the directory so that the Static Analyzer will create its own standard default files:

   ```
   rm cvstatic.*
   ```

3. Start the Static Analyzer:

   ```
   cvstatic &
   ```

4. Use the Fileset Editor to create a fileset for *bounce*. If you need help, refer to **"Tutorial 1: Applying the Static Analyzer to Scanned Files"** .
5. Choose "List All Methods" from the Methods submenu of the Queries menu.

   The Static Analyzer displays all methods found in the fileset. It uses Text View. The *Scoping* field reads `198:198`, which means that all 198 elements returned by the query are displayed in the query results area. Your version of *bounce* may be slightly different.
6. Choose "Results Filter" from the Admin menu to open the Results Filter window. When it appears, drag it from on top of the Static Analyzer window so that you can see the query results area.
7. Move the pointer to the *File* field in the Results Filter window, type **`Application.h`** and click the *Apply* button.

   The Static Analyzer shows only the methods found in the file *Application.h*. The *Scoping* field shows `16:198`, which means that you see only 16 elements of the 198 returned by the current query.
8. Move the pointer to the *Source* field, type **`virtual`**, and click the *Apply* button.

   The Static Analyzer further filters the view as shown in **Figure 6−9** , showing only the methods found in the file *Application.h*, which include the string "virtual" in their source code line. The *Scoping* field shows `5:198`.
9. Left−click the *Clear* button.

   The Static Analyzer clears all text fields and turns off all Results Filter filtering. All elements of the recent query return to the query results area, and the *Scoping* field shows `198:198`.
10. Left−click the *Close* button to close the Results Filter window.

Application.h

virtual

Figure 6–9 The Results Filter Query Results

Chapter 7

Static Analyzer: Working on Large Programming Projects

The Static Analyzer is a flexible tool. It works on uncompilable code, analyzes filesets containing files from completely different programs, and presents query results in a graphic form that's easy to browse through. This same flexibility can bring unproductive results, however, if you use the Static Analyzer carelessly on hundreds of thousands (or millions) of lines of code that are typical of a large programming project. You must narrow your analysis to a meaningful portion of your project, or you may commit the Static Analyzer to spend hours to return results so extensive that they have little meaning.

This chapter recommends techniques to help you get the best results when using the Static Analyzer for large programming projects. It covers these topics:

- **"Creating a Fileset Using a Shell Script"** to find files in separate directories
- **"Customizing the Fileset for Individual Code Modules"**
- **"Using the Results Filter to Focus Queries"**
- **"Applying Group Analysis Techniques"**

Creating a Fileset Using a Shell Script

Creating a fileset for a large programming project can be difficult to do by hand because the source code files may be scattered throughout many different directories. If so, you can use a shell script to create a fileset for you.

A Fileset Shell Script

The shell script in **Example 7–1** is an example that you can modify that searches through a list of directories for filenames with extensions that indicate source code files

Example 7–1 Script for Creating Filesets

```
rm -f cvstatic.fileset
DIRS="/usr/local/src /usr/src "
EXTENSIONS="*.c++ *.c *.f"
for DIR in $DIRS
     for EXT in $EXTENSIONS
           do
           find ${DIRS} -name "$EXT" -print >> cvstatic.fileset
     done
done
```

The first line removes the old fileset. The `DIRS` second line sets the search pattern; it assigns a list of directories you want searched to the variable *DIRS*. Put the pathname of any directory you want searched in between the quotes following *DIRS*, and put a space between pathnames.

The third line creates a list of the file extensions for which you want to search. Use shell metacharacters to create list entries. In this example, the script looks for any filenames that end in *.c++, .c,* or *.f.* To create an extension list that looks for different extensions, use shell metacharacters to spell out the extensions you want, and put the entries between the two quotes following *EXTENSIONS*. Be sure to put a space between each entry.

The six−line double loop at the end of the script looks through each directory in the *DIRS* search path and returns any files that match the list of file extensions in *EXTENSIONS*. It puts the names of all returned files into the file *cvstatic.fileset* in a form that the Static Analyzer reads as a fileset.

Once you create a fileset with a shell script, you should look at the fileset before you make any queries. If you find libraries included in the fileset, you may want to remove them so that you don't have to analyze the internal workings of each library function. You may also want to remove all files that don't apply to your specific area of the project.

Customizing the Fileset for Individual Code Modules

Most programming projects are organized so that the source code is organized in modules, with individual programmers taking responsibility for different sets of modules. The Static Analyzer allows you to analyze each module separately, even if the module won't compile without other parts of the system, so it's wise to bring the Static Analyzer to bear only on the modules you're working on. You can then see your own code in detail and see calls into other modules without having to view the contents of those modules. You also reduce the size of the cross−reference database with which you work, which speeds up the time the Static Analyzer takes to refresh the database and to complete queries of the database.

Using the Results Filter to Focus Queries

Once you create a reduced fileset, you can further improve the efficiency of your analysis by setting the Static Analyzer's Results Filter (shown in **Figure 7−1**).

Two settings are particularly useful for large programming projects: "Headers" and "External Functions". If you set "Headers" to *Exclude*, you prevent the Static Analyzer from taking the time to display query results that come from header files. And if you set "External Functions" to *Exclude*, you ensure that the Static Analyzer doesn't display query results from libraries and other non−fileset files.

For example, consider the function *foo()*, which calls *bar()*, a function in the fileset. It also calls *XtCreateWidget()*, a library function that isn't in the fileset. If you set "External Functions" to *Exclude* and then make the query "Who Is Called By foo?", the Static Analyzer will display only *bar().*

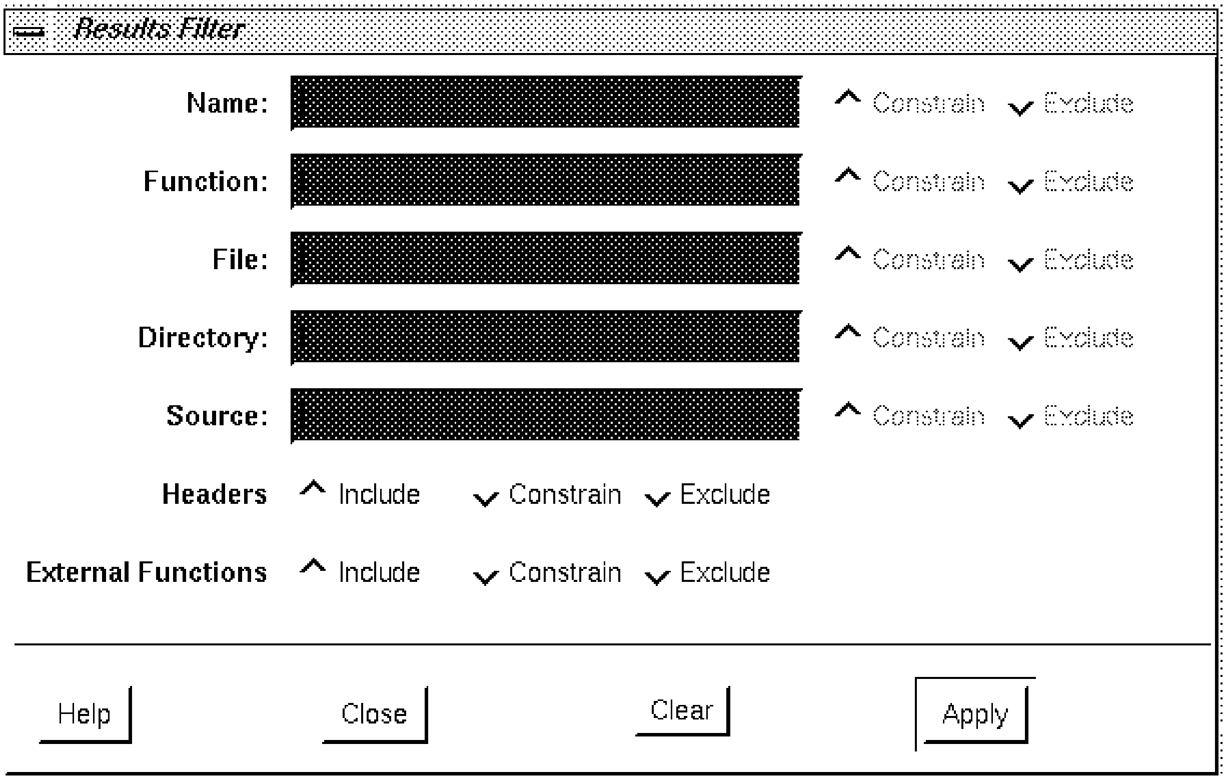


Figure 7–1 Results Filter

Although the Results Filter doesn't reduce the time the Static Analyzer takes to make a query, it does reduce the time it takes to display the results, a substantial gain if you're using a tree view to display the results of comprehensive queries.

Applying Group Analysis Techniques

Although it's good practice for individual programmers to limit the amount of source code they analyze with the Static Analyzer to just the modules for which they're responsible, sometimes it's useful to analyze all the files in the programming project. For example, library programmers may want to know *every* function that calls a specific library function; that way, they know what software is affected by changes they make to the library function.

For this and similar cases, you should create a comprehensive cross–reference database on a project workstation as shown in **Figure 7–2** . This arrangement allows users on personal workstations to query the extensive project database without actually creating the database.

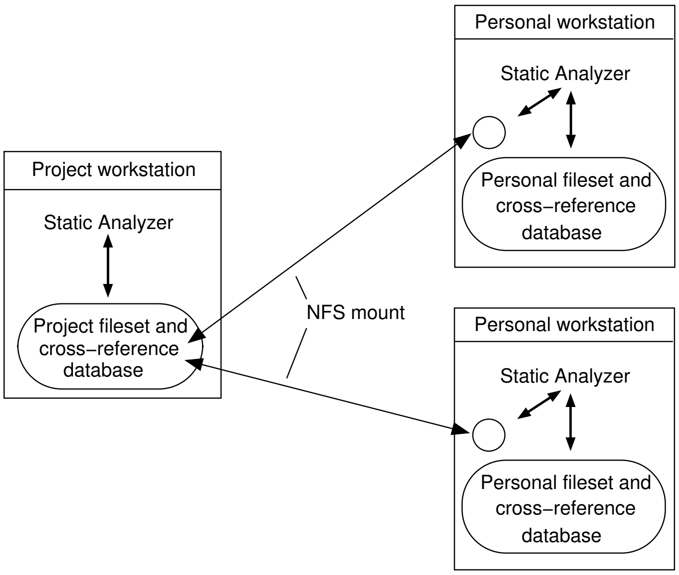


Figure 7−2 A Project Cross−reference Database

Setting Up a Project Database

To create a project cross−reference database, you first need a comprehensive fileset for the programming project. To maintain consistency, the programmer in charge of checking in files for builds should make and maintain the fileset. If the source tree uses a consistent set of directories, the build programmer can use a shell script like the example earlier in this chapter to automatically update the fileset.

Once the fileset is up to date, the build programmer creates a cross−reference database. Because it can take a long time to create a cross−reference database for a large programming project, you can save time by using the
`batch` command−line option when you start the Static Analyzer. It runs the Static Analyzer in the background, keeps the Static Analyzer window from opening, and reduces the time necessary to create a cross−reference database.

It may be useful to run the Static Analyzer in batch mode on the server once a night. This provides a fresh database for programmers who wish to query it from their own workstations. To protect the shared database from automatic modification by outside users, be sure that read and write permissions for all four Static Analyzer files on the server—*cvstatic.fileset, cvstatic.xref, cvstatic.index,* and *cvstatic.posting* —deny write access to outside users.

Querying a Project Database

To query a project database from a personal workstation, you must first mount the project database in a local directory using NFS (the Network File System). You then start the Static Analyzer using command line options to specify the project fileset and to set the Static Analyzer to read only so that it won't try to modify the project database. For example, this command starts the Static Analyzer, sets it to read−only, and directs it to the project fileset, which is NFS−mounted in the directory */project*:

```
cvstatic −readonly −fileset /project/cvstatic.fileset
```

The first command−line option, **`−readonly`**, sets the Static Analyzer so that it won't try to rebuild the project database at any time. The second command−line option, **`−fileset`**, sets the fileset to *cvstatic.fileset*, which is NFS−mounted in the directory */project*.

When you make queries on a large project database, use caution and common sense. Comprehensive queries such as "List All Functions" won't yield useful results—few people find it truly useful to see every function in millions of lines of code displayed at one time. Comprehensive queries like this may also take a good deal of time to complete. You'll find it much more productive to take a task−oriented approach when querying. Ask what you really need to know in the project, then make the most specific query that answers your questions. For example, if you get a bug report on a function, you might use specific queries such as "Where Defined", "Who Calls", or "Who Is Called By" to get the information you need about that function.

Viewing Suggestions

If you do need to make comprehensive queries on a large database, consider using Text View for your queries. Because Text View doesn't require the Static Analyzer to build a tree containing thousands of elements, it's much faster at displaying the results of a comprehensive query than any of the tree views.

Text View doesn't show connections between calling and called functions in the query results area, but you can easily follow a chain of functions. First, left−click a function name you want, then press **`<Alt−B>`** to see which functions it calls, or press **`<Alt−C>`** to see which functions call it.

Tree views show relationships between query elements more clearly than text view, so you may want to use tree views to display the results of some queries. If so, you can reduce the time the Static Analyzer needs to display tree view results by observing a few limitations.

First, don't use either the "All Defined" or the "Complete Tree" view options, which display a huge set of elements in the query results area no matter how limited a query you make. Use the "Query Only" and "Incremental Mode" view options to restrict the number of elements displayed for each query.

In Incremental Mode, you can build a tree from scratch by making very specific queries that identify and then follow only the branch of the tree in which you're interested. For example, you may want to follow a chain of function calls starting with *main()*. If so, start with the query "Who Is Called By main?". Find a function among those called that you want to follow, then query the Static Analyzer for the functions called by that function. As you continue through the call chain, the Static Analyzer displays only the branch of the call tree that applies, not the entire tree.

You should also consider viewing query results in a tree view that offers coarser resolution than you normally use. For example, File Dependency View displays file elements, each of which may contain many functions. This is a much coarser view of the database than that offered by Call Tree View, which displays functions individually in function elements. If you make a query such as "Who Calls", while in File Dependency View, the Static Analyzer shows you each file that contains called functions. You can then open the Source View window for one of those files; it highlights each called function in its display area. The same query in Call Tree View would show you each called function in tree form, but would probably require many more elements to show query results and would take much longer to return results.

Chapter 8

Getting Started with the WorkShop Debugger

The WorkShop Debugger is a UNIX source−level debugging tool that provides special windows (*views*) for displaying program data and execution status. These views update as the program executes. This chapter presents an overview of the WorkShop Debugger and is divided into these sections:

- **"Typical Debugger Usage"**
- **"Main View Menus"**
- **"The Debugger Command Line"**

Typical Debugger Usage

This section provides a general description of debugging software with WorkShop. It covers these topics:

- **"Starting and Exiting the Debugger"**
- **"Using the Debugger from a Remote Host"**
- **"Using Main View"**
- **"Setting Traps"**
- **"Inspecting Debugger Data"**
- **"Changing Source Code"**
- **"Updating and Saving Views"**
- **"Integration with Other WorkShop Tools"**

Starting and Exiting the Debugger

To start the Debugger, you use the following syntax:

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 a loop.

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.

You can also invoke the Debugger and specify a core file (with its executable) to try to determine why a program crashed.

To exit the Debugger, select "Exit" from the Admin menu in the Main View window. You have two other options: you can type **quit** at the Debugger command line or press **<ctrl-c>** where you first entered **cvd**.

Using the Debugger from a Remote Host

The **-host** option lets you designate a host, enabling you to run the Debugger remotely from another machine. This option is useful if:

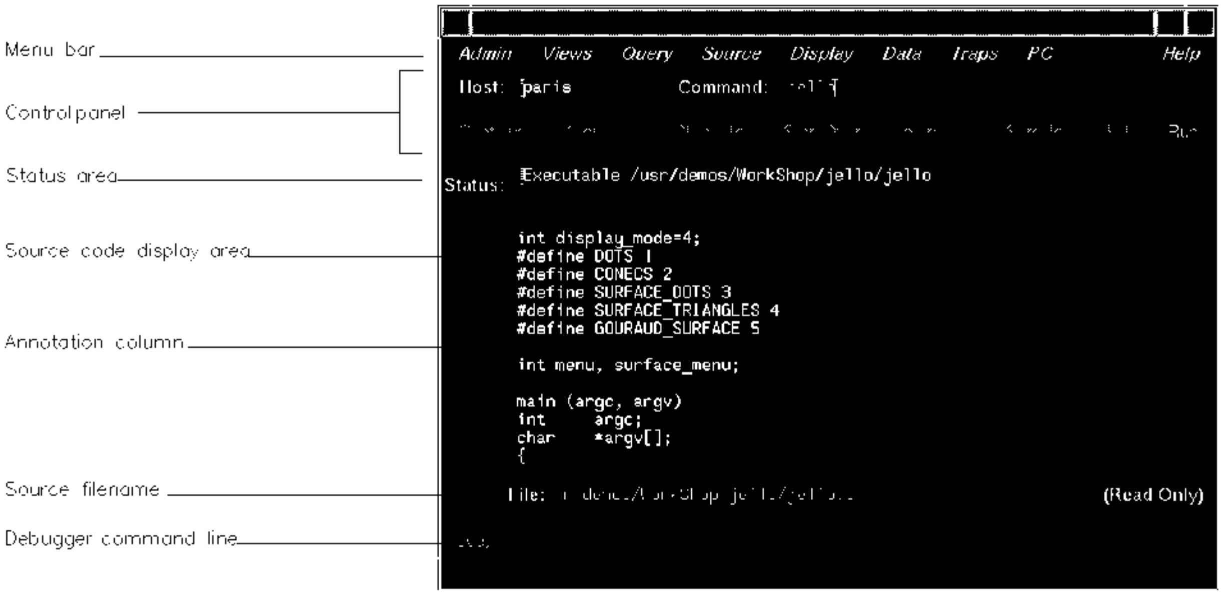
- you don't want the Debugger windows to interfere with the application you are debugging.
- you are supporting an application remotely.
- you don't want to use the Debugger on the target machine for some other reason.

Using Main View

Starting the Debugger with an executable brings up the Main View window, loaded with the source code, and ready to run the process with any specified arguments. You perform most of your work in Main View, which provides:

- a menu bar for performing Main View functions and for accessing other views
- a control panel for specifying and controlling the process to be debugged
- a source code display area for inspecting the source code
- a status line for viewing the state of the program
- the Debugger command line for entering special debugging commands (see **"The Debugger Command Line"** for the syntax)

The major areas of the Main View window are shown in **Figure 8-1** .



Status: Executable /usr/demos/WorkShop/jello/jello

```
int display_mode=4;
#define DOTS 1
#define CONECS 2
#define SURFACE_DOTS 3
#define SURFACE_TRIANGLES 4
#define GOURAUD_SURFACE 5

int menu, surface_menu;

main (argc, argv)
int     argc;
char    *argv[];
{
```

Figure 8–1 Major Areas of the Main View Window

Setting Traps

A major part of the debugging process is inspecting data at points during execution. A trap is a mechanism for gathering this data. A *stop trap* halts the process so that you can manually examine data. A *sample trap* collects specified performance data without stopping.

The Debugger lets you set traps:

- at a line in a file (breakpoint)
- at an instruction address
- on entry to or exit from a function
- when a signal is received
- when a system call is made, at either the entry or exit point
- when a given variable or address is written to, read from, or executed (watchpoint)
- at set time intervals (pollpoint)

For more information on traps, refer to **Chapter 12, "Debugger: Setting Traps in WorkShop"** .

Inspecting Debugger Data

When you stop the process, you then have a number of options for examining the data. You can inspect:

- the call stack at the breakpoint (using Call Stack)
- the value of a selected expression (one–shot evaluation)
- the value of specified expressions (using Expression View)
- the values, types, or addresses of variables (using Variable Browser)
- data structures (using Structure Browser)
- the values of an array variable (using Array Browser)
- values in specified memory locations (using Memory View) or registers (using Register View)
- the disassembled code (using Disassembly View)

For more information on the standard Debugger views, refer to **Chapter 14, "Examining Debugger Data"** . For machine–level Debugger data, see **Chapter 15, "Machine–level Debugging"** .

Changing Source Code

To change your source code and recompile, follow these steps:

1. Switch to a text editor ("Fork Editor" selection in Source menu) or edit in Main View ("Make Editable" in Source menu).

   If you are using a configuration management system, then you can check out the source code, by selecting "Versioning" from the Source menu and accessing the source through the configuration management shell.
2. Make any changes and save them by selecting "Save..." from the Source menu in Main View (or the File menu in Source View).
3. Select "Recompile" from the Source menu in Main View (or File menu in Source View).

   The Build View window displays and lets you start the compile. Any compile errors are listed in the window, and you can access the related source code by clicking the errors. For more information on the Build Manager, refer to **Chapter 1, "Using the Build Manager"**.

   When the code is rebuilt successfully, the new executable reattaches automatically to the Debugger (or Performance Analyzer or Static Analyzer). Previously set traps will be intact unless you have traps triggered at line numbers and have changed the line count.

Updating and Saving Views

Updating and saving view data is done through the Admin menu in the particular view (see **Figure 8–2**).

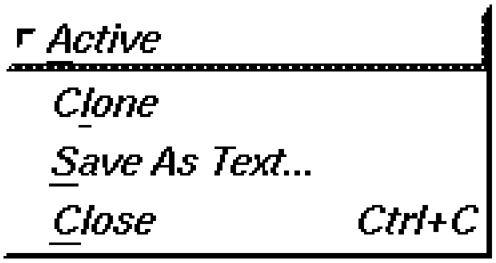


Figure 8–2 Admin Menu in Debugger Views

Typically, you display the WorkShop views of interest when you are stopped at a trap and leave them on the screen as you run the process.

"Active"

updates the view each time the process stops or if the context is changed in the Call Stack View window. The active state is the default.

"Clone"

makes a duplicate, inactive copy of the view, allowing you to save current information for future reference.

"Save As Text..."

lets you save the data currently displayed in a view to a text file. Note that not all views have the "Save as Text..." option.

"Close"

closes the window of the view.

Integration with Other WorkShop Tools

The WorkShop tools are designed so that you can move easily between them in a work session.

Accessing the Performance Analyzer from Main View

You can switch to the Performance Analyzer at any time while debugging. Selecting "Performance Task..." from the Admin menu lets you enable data collection for your experiment. A performance task must be specified before the process is run; this enables the correct data collection. If you are in the middle of a run, you must terminate it, select a task, and restart the target to collect the data. Selecting "Performance Analyzer" from the "Launch Tool" submenu displays the main Performance Analyzer for analyzing the experiment results. See **Chapter 18, "Introduction to the Performance Analyzer"** for more information.

Accessing the Static Analyzer From Main View

You can access the Static Analyzer from the "Launch Tool" submenu, if desired. The Static Analyzer displays source information from Main View, making it easy to set traps in source code located by the Static Analyzer. For more information, see **Chapter 2, "Introduction to the WorkShop Static Analyzer."** Note that the Query menu enables you to conduct some of the same queries as in the Static Analyzer.

Accessing Editors from Main View

After you solve a problem with the WorkShop tools, you may wish to make the change in Source View (or your preferred editor) and then recompile.

Accessing Configuration Management Tools

If you use CASEVision™/ClearCase™ from SGI, RCS, or SCCS for configuration management, you can integrate the tool into the WorkShop environment by typing:

```
cvconfig toolname
```

This enables the "Versioning" in the Source menu, which provides selections for checking files in and out.

Recompiling from Main View or Source View

To access the Build View window (which lets you start the compile), select "Recompile" from the Source menu in Main View or "Recompile" from the File menu in Source View. To examine the build dependencies, select "Build Analyzer" from the "Launch Tool" submenu in the Main View Admin menu. For more information on the Build Manager tools, see **Chapter 1, "Using the Build Manager."**

Main View Menus

This section describes the menus available in Main View (see **Figure 8-3**).

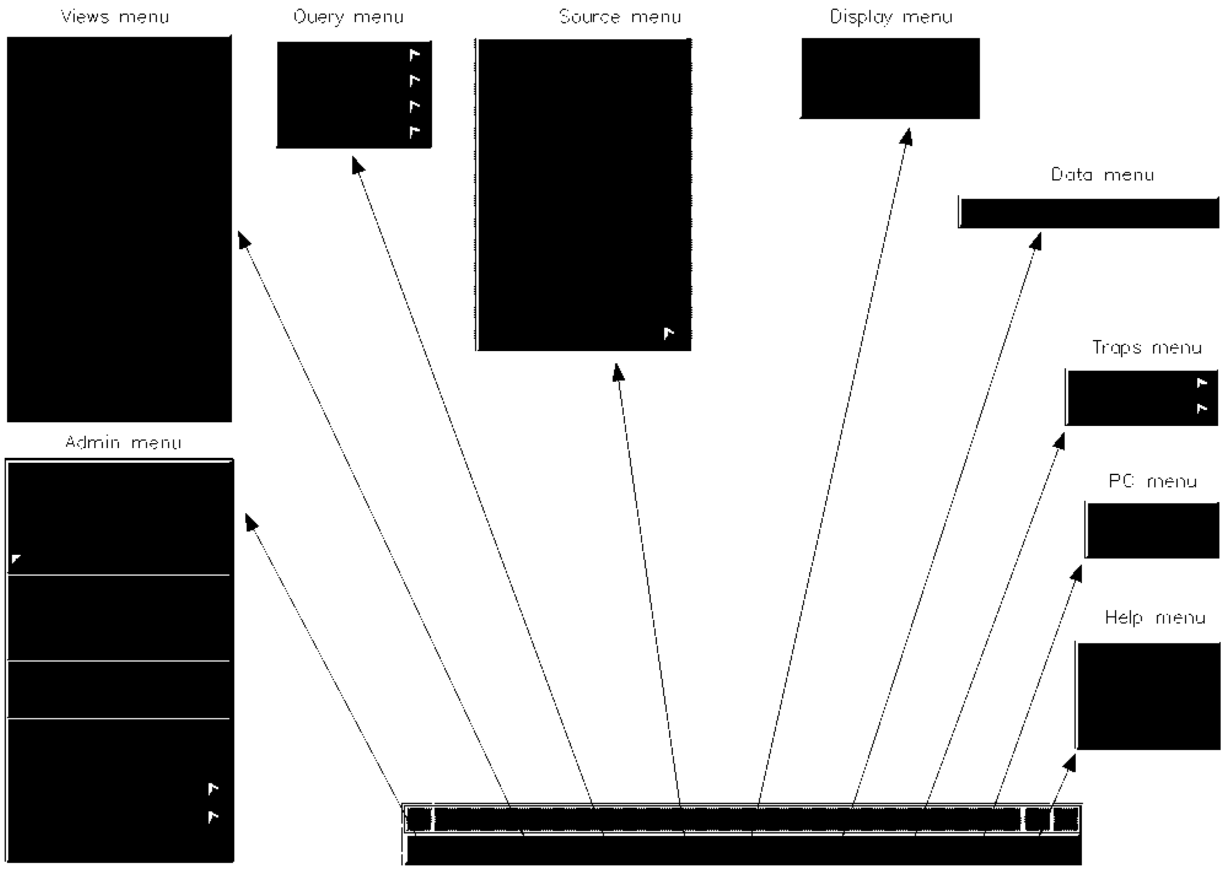


Figure 8–3 Main View Menus

Admin Menu

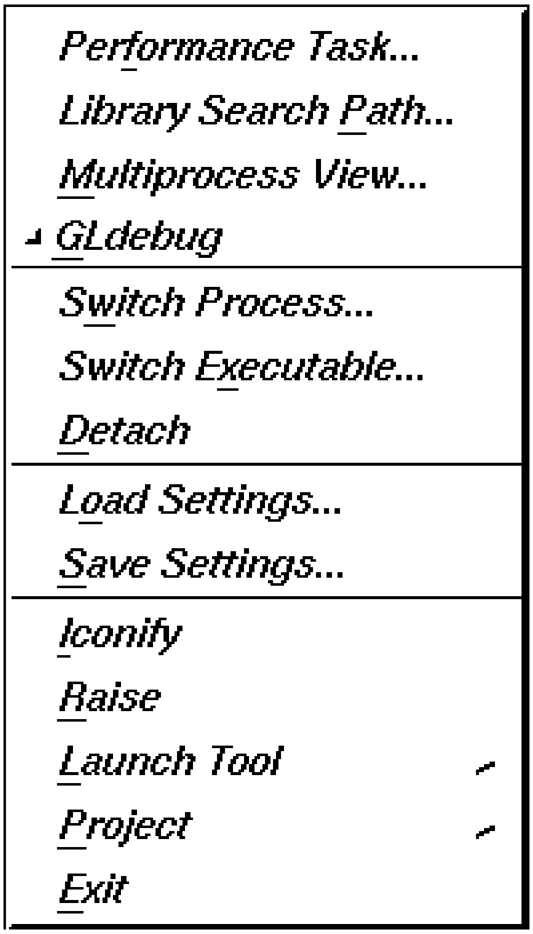


Figure 8–4 Admin Menu in Main View

The Admin menu in Main View performs general management functions dealing with processes, windows, and user preferences (see **Figure 8–4**). The Admin menu provides these selections:

"Performance Task..."

lets you specify the task for a performance analysis experiment. This enables the proper data to be captured. (Used with the Performance Analyzer.)

"Library Search Path..."

controls where the Debugger looks for DSOs when you invoke the Debugger on an executable or core file. The Library Search Path dialog box allows you to reset the environment variables *LD_LIBRARY_PATH* and *_RLD_ROOT*. You can also reset *_RLD_LIST* to control the set of DSOs which will be used by the program. See the man page for *rld* for more information on these variables. Any changes you make to these variables will be propagated into the Execution View shell at the time you run the program.

The Library Search Path dialog is opened automatically when you invoke the Debugger on an executable or core file and the Debugger is unable to find all of the required DSOs. You may also open the Library Search Path dialog box by selecting "Library Search Path..." from the Admin Menu (see **Figure 8–5**). The list of required DSOs displays at the top of the dialog box, annotated by the status of each DSO. The status can be "OK", "Error: Cannot find library", or "Error: Core file and library mismatch". The status "Error: Core file and library mismatch" indicates that the debugger found a DSO that did not match with the core file. There are three fields for the variables below the list area where you can modify their values.

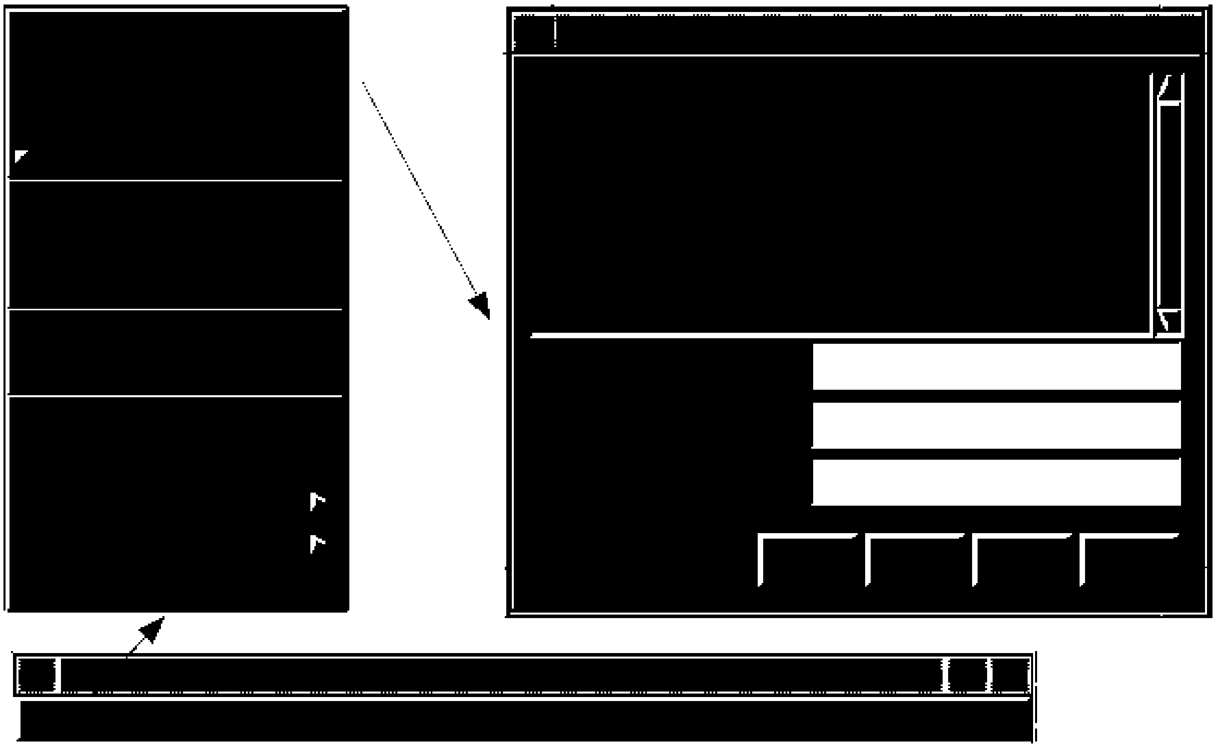


Figure 8–5 The Library Search Path Dialog Box

Insert Before and *Insert After* move the shared object specified in the *Value* field before or after the selected object in the list. *Modify* replaces the selected object in the list with the file entered in the *Value* field. *Remove* deletes the selected shared object from the list.

"Multiprocess View..."

displays the Multiprocess View window, which helps you debug several processes at once.

"GLdebug"

provides a toggle to turn on *GLdebug*. *GLdebug* is 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. For more information, refer to the *GLdebug User's Guide*.

"Switch Process..."

changes the current process. You will be queried for the new process id, as shown in **Figure 8–6**. You can type it in or paste it from another window, if desired. Switching processes changes the session. If you select a process in Multiprocess View, it is used as the default value.

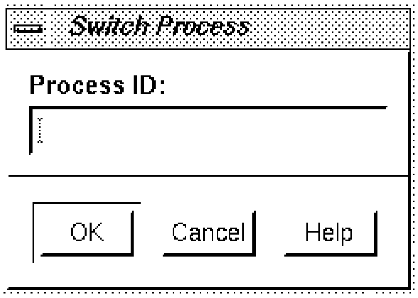


Figure 8–6 The Switch Process Dialog Box

"Switch Executable..."
: changes the current executable. This option also lets you debug a different core file. It brings up the dialog box shown in **Figure 8–7**.

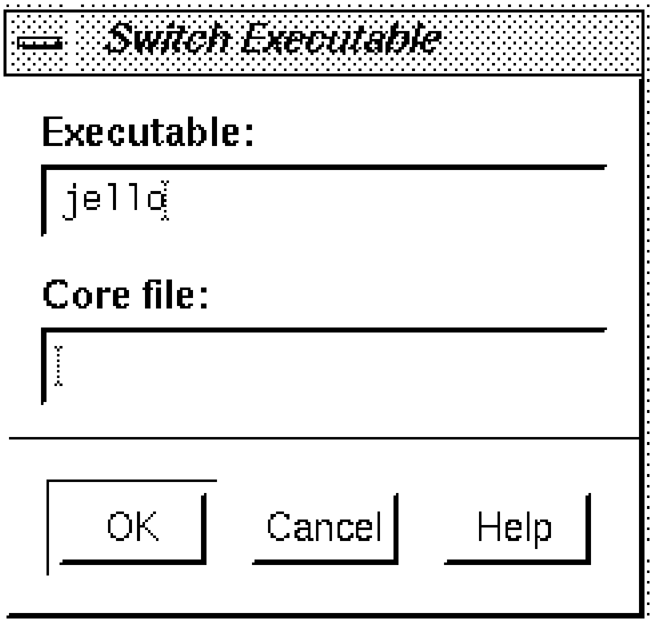


Figure 8–7 The Switch Executable Dialog Box

"Detach"
: releases the process from the Debugger. This allows you to make changes to the source code. You must detach the process before you recompile the program.

"Load Settings..."
: allows you to use the previously saved preference settings to an initialization file used when the Debugger is first started. See the description of "Save Settings...", the following item.

"Save Settings..."
: allows you to save the current preference settings to an initialization file used when the Debugger is first started. These can include such items as window sizes, current views, window configurations, etc.

"Iconify"
: iconifies all of the session's views.

"Raise"
: brings all the session's view windows to the foreground and redisplays any iconified windows.

"Launch Tool"
: lets you run the WorkShop tools. See **Figure 8–8**. You can switch to the other tools by selecting "Build Manager", "Static Analyzer", "Performance Analyzer", or Tester. Selecting "Debugger" lets you start another debugging session. If you buy WorkShop/MP (for multi–process debugging), the "Parallel Analyzer" selection is enabled.

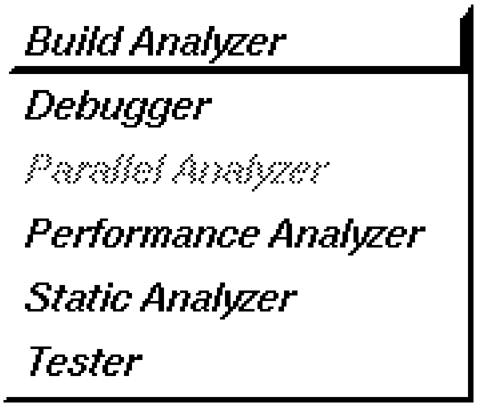


Figure 8–8 "Launch Tool" Submenu

"Project"

lets you control the WorkShop tools operating on the same executable as a group. See **Figure 8–9** . For more information on "Project View", a facility for managing CASEVision tools operating on a common target, see **"Project View"** .

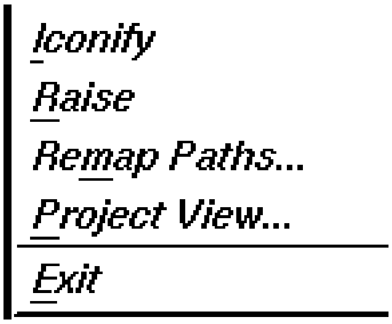


Figure 8–9 "Project" Submenu

"Exit"

exits all views in the session and terminates the session.

Views Menu

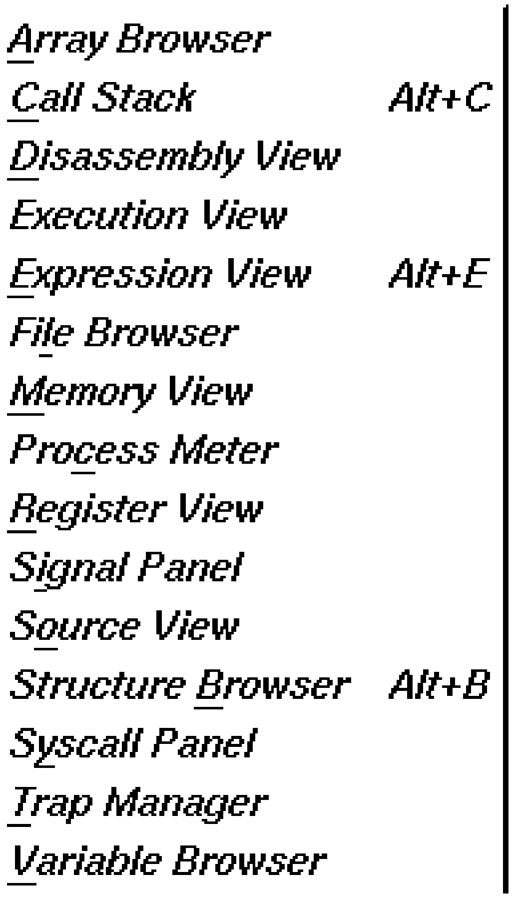


Figure 8−10 Views Menu in Main View

The Views menu in Main View (see **Figure 8−10**) provides these selections for viewing the process(es) and their corresponding data:

"Array Browser"

displays values from an array or array−slice in a two−dimensional spreadsheet and optionally in a three−dimensional representation, that is, a bar graph, surface, multiple lines, or points in space. These help you pick out bad data more readily.

"Call Stack"

displays the call stack along with parameters to the calls. If you double−click a frame in the stack, you can switch the current context to the invocation of that frame and check the state of variables.

"Disassembly View"

displays assembly code corresponding to the source code.

"Execution View"

displays the Execution View window for handling the target process's input and output.

"Expression View"

evaluates expressions in Fortran, C, or C++. To enter an expression, select it in the source code display and paste it into the *Expression View* field, using the middle mouse button.

"File Browser"

displays a scrollable list of source files used by the current executable. Double−click a file in the list to load it directly into the source display area in Main View or Source View. The *Search* field lets you find files in the list quickly.

"Memory View"

displays the value at a given memory address.

"Process Meter"

monitors the resource usage of a running process without saving the data. (Used with

"Register View"

displays the values stored in the hardware registers for the target process.

"Signal Panel"

displays the signals that can occur. You can specify which signals trigger traps and which are to be ignored.

"Source View"

displays source code. Lets you set traps, perform searches, and inspect source code without losing information in Main View.

"Structure Browser"

displays data structures in a graphical format. You can dereference pointers by double–clicking.

"Syscall Panel"

lets you set traps at the entry to or exit from system calls.

"Trap Manager"

allows you to set, edit, and manage traps. (Used in both the Debugger and Performance Analyzer.)

"Variable Browser"

displays the values of local variables and parameters for the current context.

Query Menu

The Query menu (see **Figure 8–11**) lets you perform some of the queries available in the Static Analyzer. If you have previously built a *cvstatic* fileset, this is rather convenient; however, if you need to build the fileset from scratch, the process becomes more involved, as described in **Chapter 4, "Static Analyzer: Creating a Fileset and Generating a Database."**

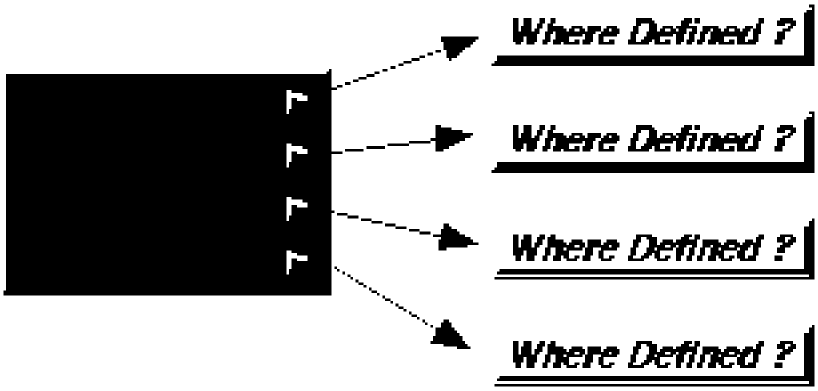


Figure 8–11 Query Menu with Submenus

With a current fileset, you can double–click any defined entity in the source code, select the "Where Defined?" option appropriate to its type, and the source code display area will scroll to the location where the item is defined.

Source Menu

Figure 8–12 Source Menu in Main View

The Source menu in Main View (see **Figure 8–12**) provides these selections to deal with source code files:

"Open..."

loads a source file.

"Save"

records changes made during the debugging session to the source file. You must first select "Make Editable", which appears in the Source menu when the file is read only.

"Save As..."

records changes made during the debugging session to the source file under a different filename.

"Save As Text..."

records the information in the display area as a text file.

"Insert Source..."

inserts the text of a file within your current file.

"Fork Editor"

starts your default editor on the current file. The default editor is determined by the *editorCommand* resource in the *app–defaults* file. The value of this resource defaults to `wsh –c vi +%d,` which means run *vi* in a *wsh* window and scroll to the current line. If the editor lets you specify a starting line, enter **`%d`** in the resource to indicate the new line number.

"Recompile"

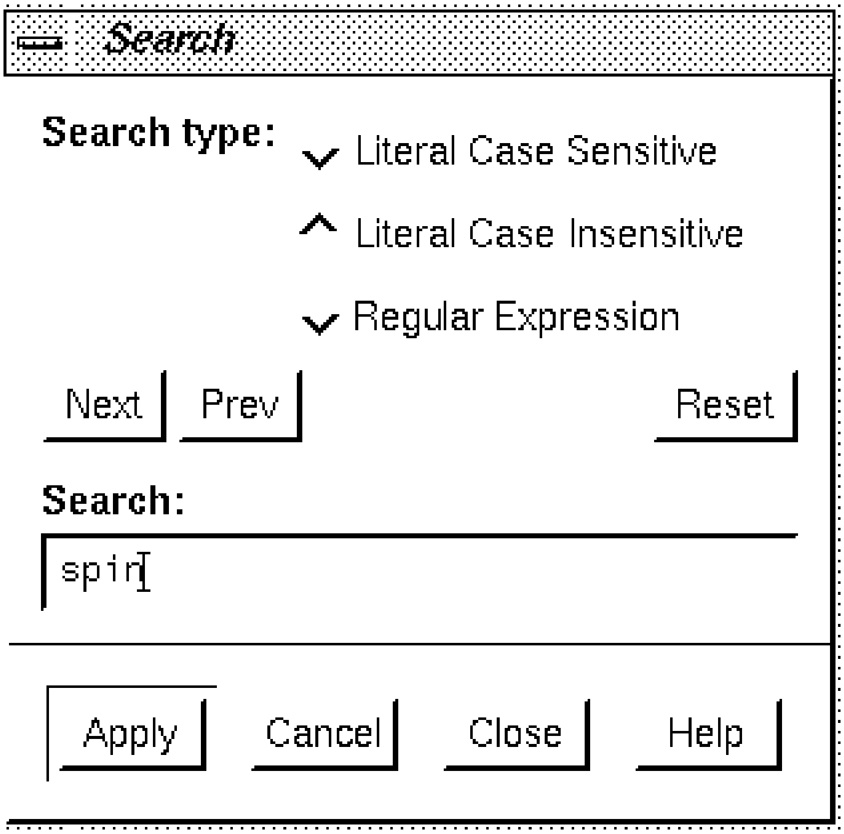
displays the Build View window, which lets you compile the source code associated with the current executable.

"Make Read Only" / "Make Editable"

toggles the source code displayed between read–only and writable states so that you can edit your code.

"Search..."

searches for a literal case–sensitive, literal case–insensitive, or regular expression (see **Figure 8–13**). After you have set your target and clicked *Apply* (or pressed **`<Enter>`**), each instance is marked by a search target indicator in the scroll bar. You can also click an indicator with the middle mouse button to scroll Main View to that point. Clicking *Reset* removes the search target indicators. For an example of "Search", see **"Performing a Search"** .



Search:
spin

Figure 8–13 The Search Dialog Box

"Go to Line..."

lets you scroll to a position in the source code by specifying a line number. "Go to Line..." brings up a dialog box similar to the one shown in **Figure 8–14** .

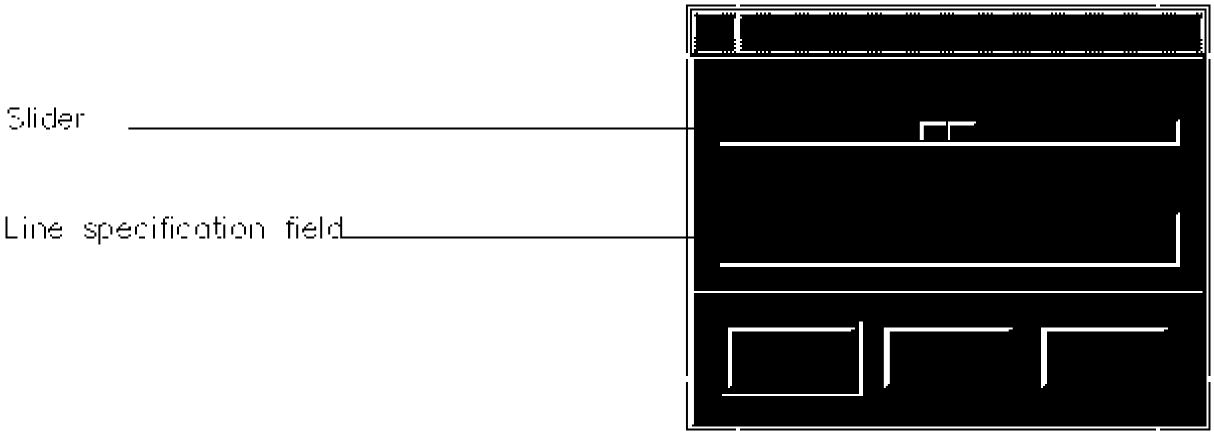


Figure 8–14 Go to Dialog Box

You can enter a line number or use the slider at the top of the box to select a line number. You do not have to display line numbers to use this feature.

Versioning"

provides access to the configuration management tool, if you have designated one. The *cvconfig* script lets you designate CASEVision/ClearCase from SGI, RCS or SCCS. Type:

```
cvconfig [clearcase | rcs | sccs]
```

Figure 8–15 "Versioning" Submenu

The "Versioning" submenu appears in **Figure 8–15** . Selecting any of these options displays a shell in which you can access the configuration management tool. The selections in the submenu are:

"Versioning":"CheckIn"

saves the source file and checks it into the database as a new version.

"Versioning":"CheckOut"

recalls the source file from the tool's database if you have the proper authority, locks it, and makes it editable.

"Versioning":"UncheckOut"

cancels the checkout, with no changes registered.

Display Menu

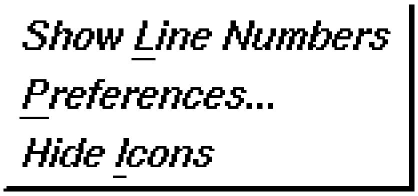


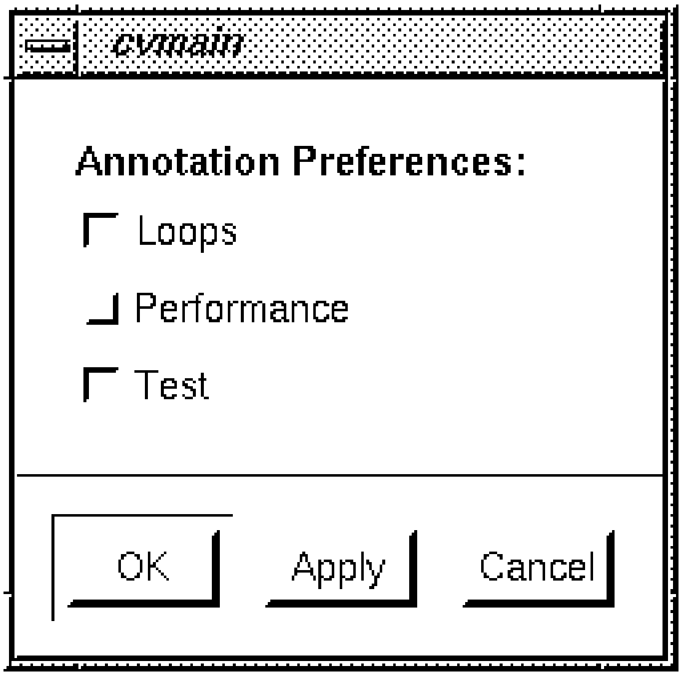
Figure 8–16 Display Menu in Main View

Figure 8–16 Display Menu in Main View

The Display menu in Main View (see **Figure 8–16**) provides these selections to annotate the source code displayed:

"Show Line Numbers"/"Hide Line Numbers"
: displays or hides line numbers in the annotation column corresponding to the source code.

"Preferences..."
: displays the Preferences dialog box (see **Figure 8–17**) which lets you show or hide column annotations and menus specific to the different WorkShop tools. In the Debugger, you can display trap, pc, and context icons. If you have purchased WorkShop/MP, you can display and manipulate loop indicators. The Performance Analyzer displays experiment statistics. The Tester module (if purchased) lets you see coverage statistics.



Annotation Preferences:

☐ Loops

☑ Performance

☐ Test

Figure 8–17 Preferences Dialog Box

"Hide Icons"/"Show Icons"
: removes or displays the annotation column next to the source code display area.

Data Menu

Figure 8–18 Data Menu

The Data menu in Main View (see **Figure 8–18**) provides the selection for one–shot expression evaluation:

"Evaluate Expression..."
: computes the value of the selected expression. Selecting "Evaluate Expression" displays the window shown in **Figure 8–19** . Entering a valid expression in the *Expression* field causes the result to display in the field below it.

Figure 8–19 Evaluate Expression Dialog Box

Traps Menu

Refer to **Chapter 12, "Debugger: Setting Traps in WorkShop"**

PC Menu

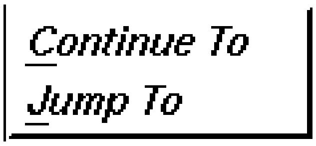


Figure 8–20 PC Menu in Main View

The PC (program counter) menu in Main View (see **Figure 8–20**) provides these selections for controlling the execution of a process:

"Continue To"
continues the process to the selected point in the program unless some other event interrupts. You select a line by placing the cursor in it.

"Jump To"
goes directly to a selected point within the same function, jumping over intervening code. Waits for command to resume execution. You select a line by placing the cursor in it.

Help Menu

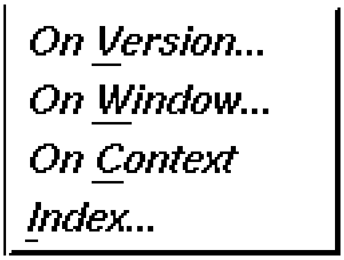


Figure 8–21 Help Menu

The Help menu (see **Figure 8–21**) provides these options:

"On Version..."
provides general copyright and version number information on the current tool.

"On Window..."
provides information on the current window's usage.

"On Context"
provides information on the selected window's feature.

"Index..."
displays the entire list of help topics, alphabetically, hierarchically, or graphically.

The Debugger Command Line

To use the Debugger commands, which are entered at the command line at the bottom of Main View (see **Figure 8–1**), you should be familiar with *dbx* commands. For more information, refer to the dbx

Reference Manual. The syntax for the debugging commands is as follows:

step [_INT_**]**

steps the specified number of source instructions. This command steps into procedures. The default is one instruction.

next [_INT_**]**

steps over the specified number of source instructions. This command does not step into procedures. The default is one instruction.

stepi [_INT_**]**

steps the specified number of machine instructions. This command steps into procedures. The default is one line.

nexti [_INT_**]**

steps over the specified number of machine instructions. This command does not step into procedures. The default is one line.

file [_filename_**]**

displays the name of the current or specified file (_filename_). If a file is specified, it becomes the current file.

stop at [_filename_**:]** _line_number_ **[if** _expression_**]**

traps at the specified line in the specified file. If the **if** option is used, the trap fires only if _expression_ is true.

stop in [_filename_**:]** _function_name_ **[if** _expression_**]**

traps at the entry to the specified function. If the **if** option is used, then the trap only fires if _expression_ is true. If the _filename_ is given, the function is assumed to be in that file's scope.

trace [_variable_**] at [["**_filename_**":] **
[_line_number_ | _function_name_**] **
[if _expression_**]]**

traces the specified variable. You can specify a file and/or test condition. You can also specify a line number or a function where the trace is to take place.

when at [_filename_**:]** _line_number_ **{**_command_ **[;** _command_ **...]}**

stops the process and performs other Debugger commands when the process reaches a specified line number.

when in [_filename_**:]** _function_name_ **{**_command_ **[;** _command_ **...]}**

stops the process and performs other Debugger commands at entry to function. If the _filename_ is given, the function is assumed to be in that file's scope.

list [[_expression_ **[,**_expression_**]] | [**_function_name_**]]**

lists the specified number (_expression_) of lines. The default is 10 lines. You can optionally specify a function where the list is to take place.

status

displays a list of currently set breakpoints and traces.

call function_name [_argument_, **...]**

executes the specified function with any arguments supplied.

clearcalls

cancels pending interactive function calls.

continue

continues executing a program after a breakpoint. Note that you can c and cont as aliases for continue.

continue [_signal_**]**

sends specified signal and continues executing a program after a breakpoint.

return

continues executing the current procedure and returns to the next sequential line in the calling function.

cont to _line_number_

continues execution from the current line until the specified line.

cont in _function_name_

continues execution from the current line to the entry to the specified function.

print _expression_ **[,**_expression_, **...]**

prints the value of the specified expression(s). If the

expression is a character pointer or array, both the string and address print.

printd *expression* [,*expression*, ...]
prints the value of the specified expression(s) in decimal format. You can use pd as an alias.

printo *expression* [,*expression*, ...]
prints the value of the specified expression(s) in octal format. You can use po as an alias.

printx *expression* [,*expression*, ...]
prints the value of the specified expression(s) in hexadecimal format. You can use px as an alias.

display [*expression*, ...]
with *expression*, adds *expression* to the list of expressions displayed whenever the process stops.
with no arguments, lists all expressions on the display list.

undisplay [*displaynumber*, ...]
stops display of expression with specified *displaynumber* when the process stops. Removes the expression from the display list.

clear [**all** | *source_line*]
clears breakpoints. The `all` option clears all breakpoints. The *source_line* option clears the breakpoint at the specified source line.

clearbuffer
clears the currently displayed lines.

delete *displaynumber* [,*displaynumber*, ...]
deletes the specified expression from the display list.

delete all
deletes all traps.

delete *trap_number* [,*trap_number*, ...]
deletes the specified breakpoint from the status list.

enable all
reactivates all inactive traps.

enable *trap_number* [,*trap_number*, ...]
reactivates a disabled stop trap.

disable all
deactivates all inactive traps.

disable *trap_number* [,*trap_number*, ...]
deactivates a trap set by *stop* command.

kill [*pid*]
kills the specified process currently controlled by the Debugger.

which *identifier*
displays the qualification of the specified variable.

whatis *identifier*
displays all the qualifications of the specified variable.

where
performs a stack trace.

func [*func_name*]
moves to the specified frame in the call stack or to the function in the executable if not on the stack.

up [*expression*]
moves up the specified number of frames in the call stack. Up moves in the direction of the caller.

down [*expression*]
moves down the specified number of frames in the call stack. down moves in the direction of the called function.

catch [*signal_name* | **all**]
with no arguments, lists signals to be trapped. If a signal is specified, it's added to the list. If `all` is specified, it traps all signals.

ignore [*signal_name* | all]

with no arguments, lists those signals not to be trapped. If a signal is specified, this command removes it from the list of signals to be trapped. If **all** is specified, ignores all signals.

**syscall catch | ignore [call | return] **
[*sys_call_name* | all]

The **catch** option adds a system call to the list of system calls to be trapped. The **ignore** option removes a system call from the system call trap list. The **call** option specifies the entry to the system call and **return** signifies the return from the call.

run

runs the program.

rerun

runs the program again using the same arguments.

quit

exits the debugging session.

givenfile [*filename*]

with no arguments, displays name of current object file.
with *filename,* specifies object file to be debugged.

corefile [*filename*]

with no arguments, reports whether data referencing commands reference a core file. If so, displays the current core file.
with *filename* provided, specifies core file to be debugged.

pwd

displays the current directory.

use [*path*]

uses the specified path to search for source files.

expression, [*count*] / [*format*] |

displays count number of formatted items. The format options are:

d prints a short word in decimal
D prints a long word in decimal
o prints a short word in octal
O prints a long word in octal
x prints a short word in hexadecimal
X prints a long word in hexadecimal
b prints a byte in octal
c prints a byte as a character
s prints a string of characters that ends in a null byte
f prints a single–precision real number
g prints a double–precision real number

Chapter 9

A Short Debugger Tutorial

This chapter presents a short tutorial for using the Debugger. The tutorial applies the Debugger to a program called *jello*, which provides a walk through some typical debugging situations. The tutorial is divided into four parts:

- **"Starting the Debugger"**
- **"Performing a Search"**
- **"Setting Traps"**
- **"Examining Data"**

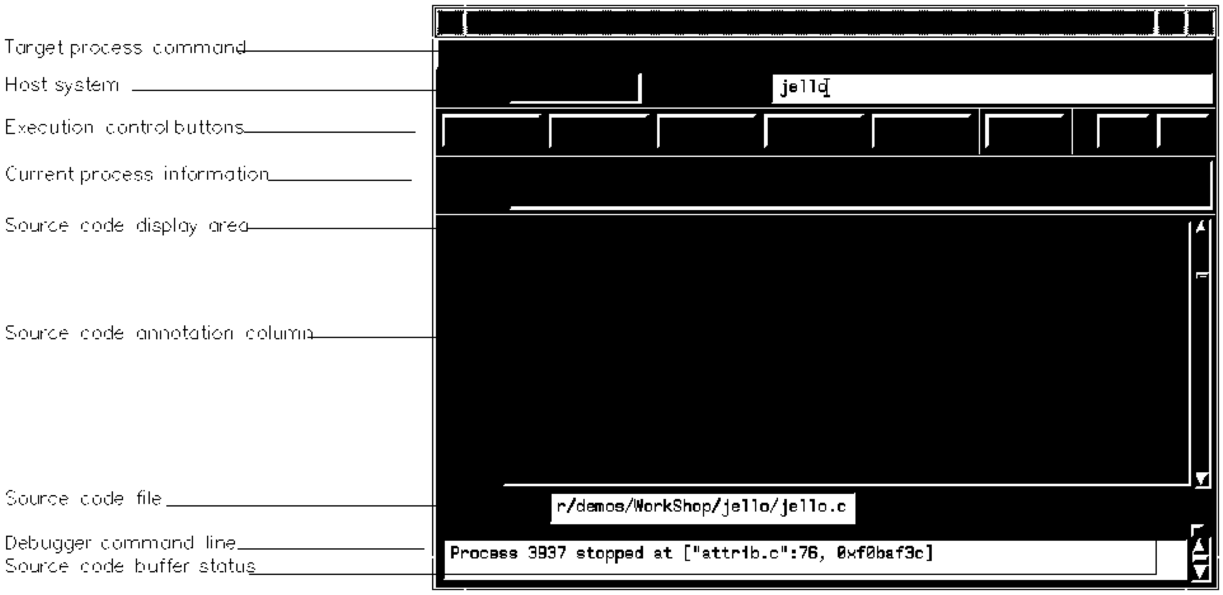
Note: WorkShop identifies files with the pathnames in which they were compiled. The pathnames in the tutorial may not match the ones on your system.

Starting the Debugger

In this part of the tutorial, you invoke the Debugger and start a typical process running. The *jello* program simulates an elastic polyhedron bouncing around inside of a revolving cube. The program's functionality is mainly contained in a single loop that calculates the acceleration, velocity, and position of the polyhedron's vertices.

1. Go to the directory */usr/demos/WorkShop/jello*.
2. If the *jello* executable does not yet exist, type `make jello`
3. To invoke the Debugger, type `cvd jello`

   The Main View window appears as shown in **Figure 9–1**. The display scrolls automatically to the *main* function.



```
jello
r/demos/WorkShop/jello/jello.c
Process 3937 stopped at ["attrib.c":76, 0xf0baf3c]
```

Figure 9–1 The Main View Window with *jello* Source Code

> **Note:** Main View brings up the source file in read–only mode to avoid inadvertent changes during debugging. You can change this mode by selecting "Make Editable" from the Source menu (provided you have the proper file access permissions).

4. Click the *Run* button in the upper–right corner of the Main View to start the *jello* process.

   The *jello* window opens on your display (see **Figure 9–2**). Enlarge this window to watch the program execute. The polyhedron is initially suspended in the center of the cube.
5. Click the left mouse button anywhere inside the *jello* window.

   The polyhedron drops to the floor of the cube.
6. Hold down the right mouse button to display the pop–up menu and select "spin".

   The cube now rotates and the polyhedron bounces. If you select "display" from the menu, you can change the appearance of the polyhedron: points only, lines only, full color, visible points only, or single color.

> **Note:** You may encounter flashing colors inside windows while running *jello*. This is a normal side effect due to GL/X interaction.

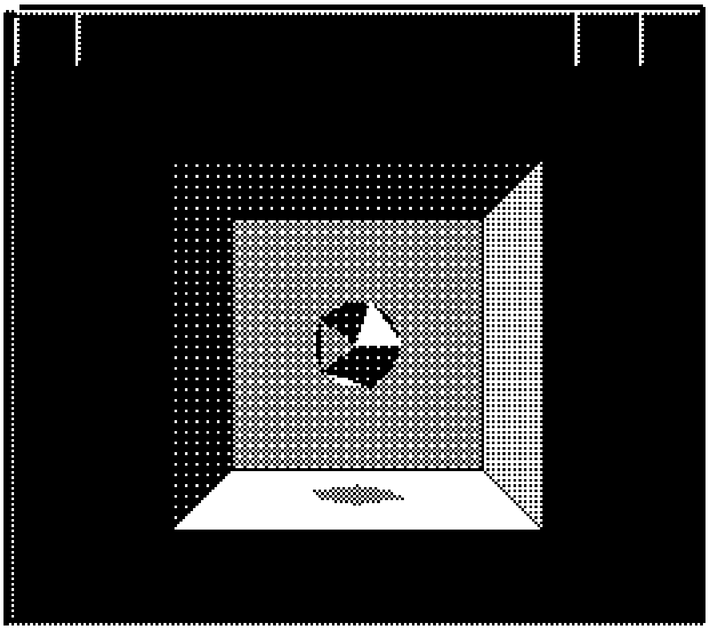


Figure 9–2 The *jello* Window

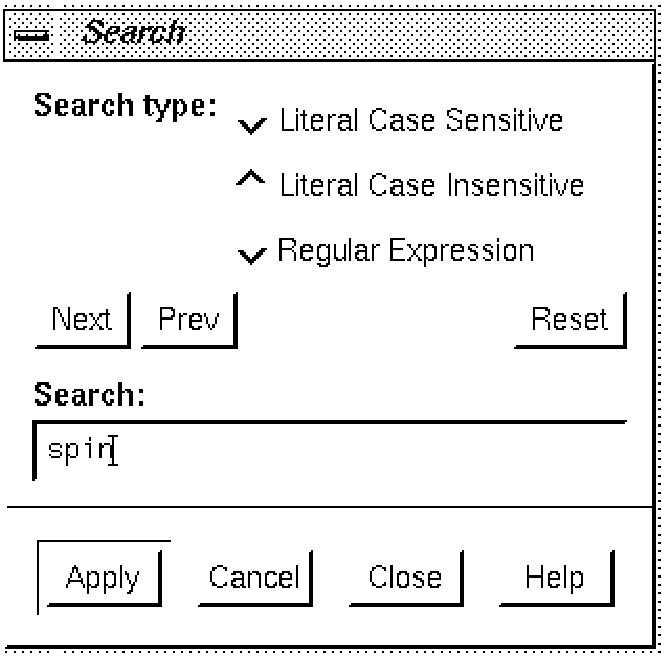
Performing a Search

This part of the tutorial covers the search facility in the Debugger. You will search through the *jello* source file for a function called *spin*. The *spin* function recalculates the position of the cube.

1. Choose "Search" from the Source menu.

   The Search dialog box appears.

2. Type **`spin`** in the entry field in the Search dialog box, as shown in **Figure 9–3**.



Search: spin

Figure 9–3 The Search Dialog Box

3. Click the *Apply* button in the Search dialog box.

   The search takes place. Each instance of "spin", the target string, is highlighted in the source code and flagged in the scroll bar to the right of the display area. **Figure 9–4** shows typical search target indicators. The *Next* and *Prev* buttons let you move from one occurrence to the next in the order indicated. For more information on Search, see **"Source Menu"** .

4. Click the *Close* button in the Search dialog box.

   The dialog box disappears.

5. Click the middle mouse button on the last search target indicator.

   This scrolls the source code down to the last occurrence, which is the location of the *spin* function.

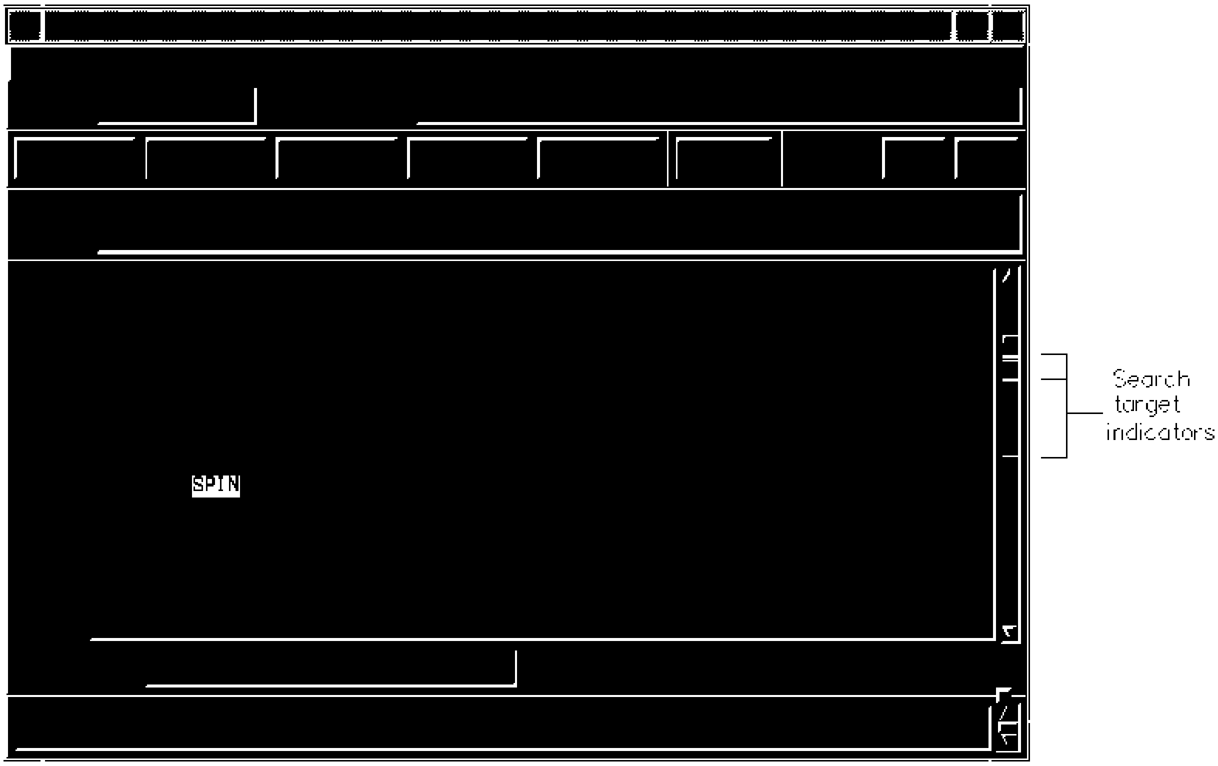


Figure 9–4 Search Target Indicators

Setting Traps

Stop traps (also called breakpoints) stop the program's execution at a specified line in the code, allowing you to track the progress of your program and to check the values of variables at that point. Typically, you set breakpoints in your program prior to running it under the Debugger. For more information on traps, refer to **Chapter 12, "Debugger: Setting Traps in WorkShop"** .

In this part of the tutorial, you set a breakpoint at the *spin* function.

1. Click the left mouse button in the source code annotation column next to the line containing `gl_sincos (a, &sa, &ca)`. This is the first line of code inside the *spin* function.

   A stop trap indicator appears in the annotation column as shown in **Figure 9–5** . This stop trap halts execution of *jello* at the beginning of the next call to the *spin* function. When the process stops, an icon indicating the current PC appears and the line becomes highlighted.

Figure 9‒5 Stop Trap Indicator

2. Click the *Continue* button at the upper‒left corner of the Main View window repeatedly so that *jello* goes through several iterations.

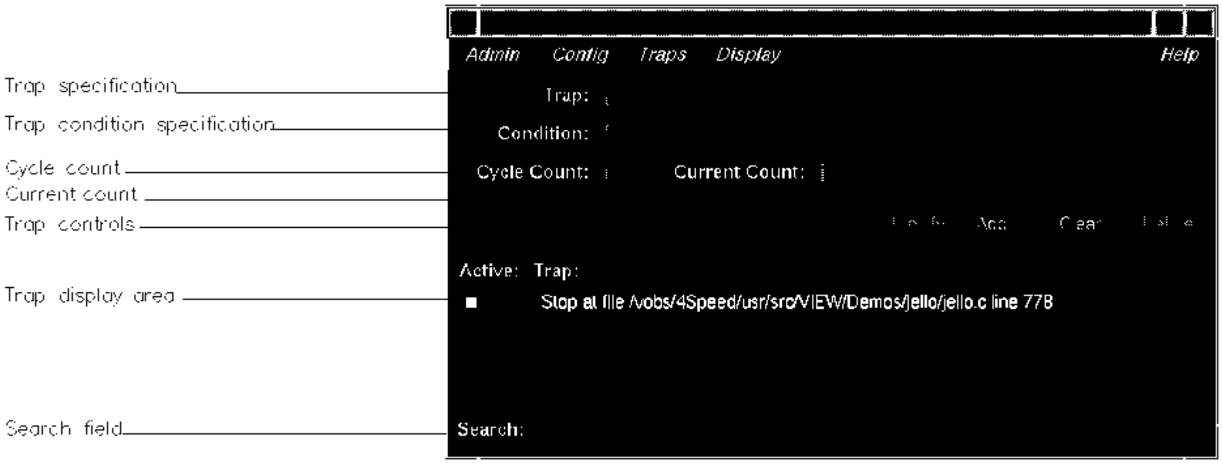
   The *Continue* button resumes execution until the next breakpoint (in this case, *spin*) is encountered. Stopping at the *spin* function allows you to view the *jello* image one frame at a time.

3. Select "Trap Manager" from the Views menu in Main View.

   The Trap Manager window appears as shown in **Figure 9‒6**.

   Trap Manager lets you list, add, edit, disable, or remove traps in a process. You set one breakpoint in the *spin* function by clicking in the source code annotation column. This trap is displayed in the trap display area.

   You can define other traps as well in the Trap Manager. You set conditional traps in the *Condition* field from the top. The count information lets you specify the number of times a trap should be encountered before it fires. The trap controls let you manipulate traps. All traps (active and inactive) are shown in the trap display area.



■ Stop at file /vobs/4Speed/usr/src/VIEW/Demos/jello/jello.c line 778

Figure 9‒6 The Trap Manager Window

4. Click the button to the left of the stop trap in the trap display area.

   The trap is temporarily disabled. Trap Manager lets you turn traps on and off by clicking them.

5. Click the *Clear* button, move the cursor to the *Trap:* field, type

   ```
   watch display_mode
   ```

   and click *Add*.

   This sets a watchpoint for the variable *display_mode*. A *watchpoint* is a trap that fires when a specified variable or address is read, written, or executed.

   After you continue the process, you can fire this watchpoint by holding down the right mouse button in the *jello* window and selecting "display" and a different display option. The variable *display_mode* will be accessed and the watchpoint will fire.

6. Click the *Continue* button to restart the process.

   The process now runs somewhat slower but still at a reasonable speed for debugging.

7. Hold down the right mouse button in the *jello* window to display the pop‒up menu, and select "display" and then the "conecs" option with the right button.

   This triggers the watchpoint and stops the process. If you were tracking the effects of changing display modes, you could bring up other views now.

8. Go to the Trap Manager window and click the button next to the *display_mode* watchpoint to deactivate it. Click the button next to the *spin* stop trap to reactivate it.

   This resets the traps for use in this tutorial.

9. Enter `100` in the *Cycle Count* field, press `<Enter>`, and click the *Continue* button in Main View.

   This takes the process through the stop trap for the specified number of times, provided no other interruptions occur. The *Current Count* field keeps track of the actual number of iterations since the last stop, which is useful if an interrupt occurs. Note that it updates at interrupts only.

10. Select "Close" from the Admin menu in Trap Manager to close it.

Examining Data

This part of the tutorial describes how to examine data after the process stops.

1. Select "Call Stack" from the Views menu in Main View.

   The Call Stack View window appears as in **Figure 9–7**. The Call Stack View window shows each frame in the call stack at the time of the breakpoint with the calling parameters and their values. You can also display the calling parameters' types, locations, and PC (program counter) through the Display menu. For more information, see **"Tracing through Call Stack View"**.

   In this example, the *spin* and *main* stack frames are displayed in Call Stack View, and the *spin* stack frame is highlighted, indicating that it is the current stack frame.

2. Pull down the Admin menu and examine the "Active" selection.

   By default, the "Active" toggle button in the Admin menu is turned on. *Active views* are those that have been specified to change their contents at stops or at call stack context changes. If the toggle is on, the call stack is updated automatically whenever the process stops.

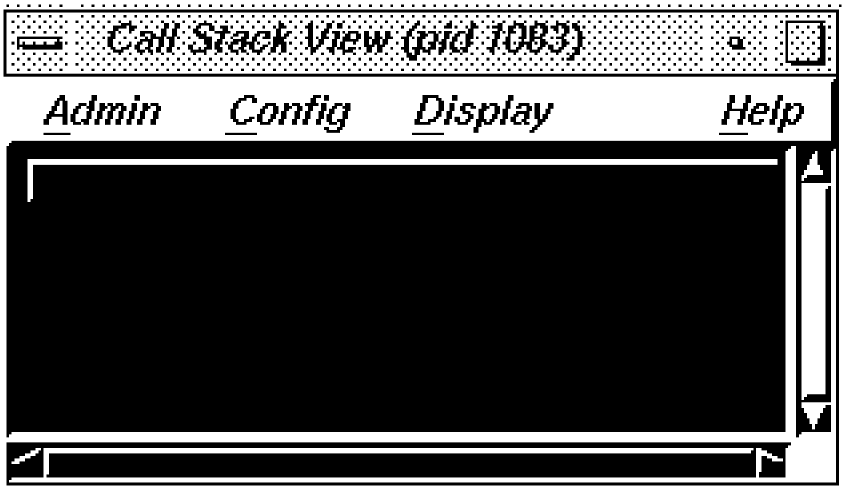


Figure 9–7 Call Stack View at *spin* Stop Trap

3. Double–click the *main* stack frame.

   This shifts the stack frame to the *main* function, scrolls the source code in Main View (or Source View) to the place in *main* where *spin* was called, and highlights the call in the designated context color. Any active views are updated according to the new stack frame.

4. Double–click the *spin* stack frame.

   This returns the stack frame to the *spin* function.

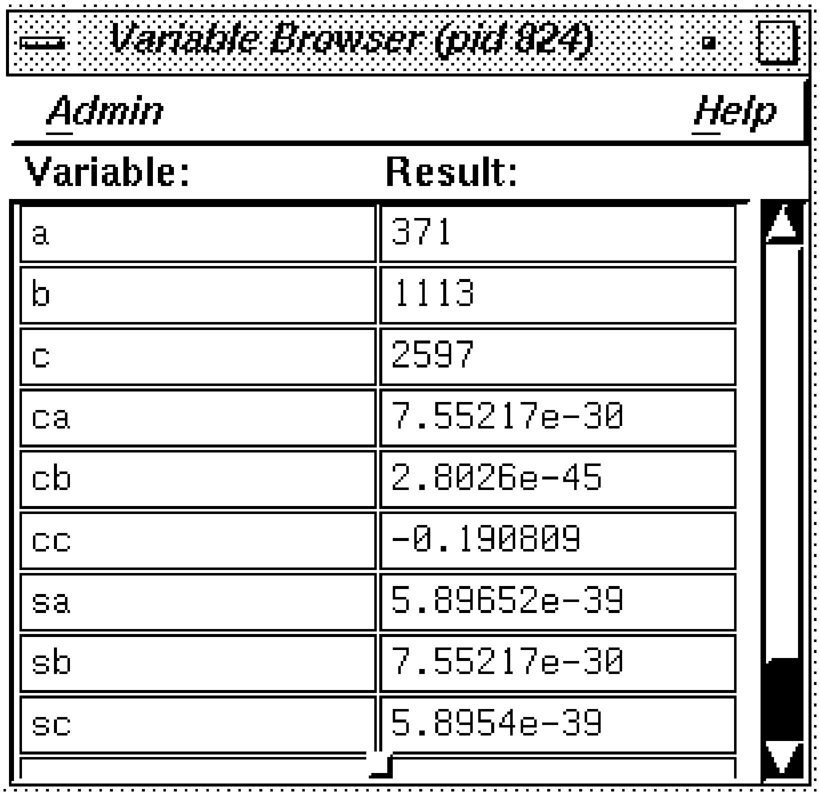
5. Select "Variable Browser" from the Views menu in Main View.

   The Variable Browser window appears. This window shows you the value of local variables at the

The Variable Browser window appears. This window shows you the value of local variables at the breakpoint. The variables appear in the left column (read−only), and the corresponding values appear in the right column (editable). Since the right column is editable, you can change the values of the variables if you want. For more information, see **"Examining Variable Values"** .

Your Variable Browser window should resemble the one in **Figure 9−8** , although you may need to enlarge the window to see all the variables (the values will be different).

The *jello* program uses variables *a, b,* and *c* as angles (in tenths); *ca, cb, cc* as their corresponding cosines; and *sa, sb, sc* as their sines. Whenever you stop at *spin*, these values will change.



Variable:	Result:
a	371
b	1113
c	2597
ca	7.55217e-30
cb	2.8026e-45
cc	-0.190809
sa	5.89652e-39
sb	7.55217e-30
sc	5.8954e-39

Figure 9−8 Variable Browser at *spin*

6. Double−click some different frames in Call Stack View and observe the changes to Variable Browser and Main View.

   These views update appropriately whenever you change frames in Call Stack View. Notice also the change indicators in the upper right corners of the *Result* fields (see **Figure 9−9**). These appear if the value has changed. If you click the "folded" corner, the previous value displays (and the indicator appears "unfolded"). You can then toggle back to the current value.

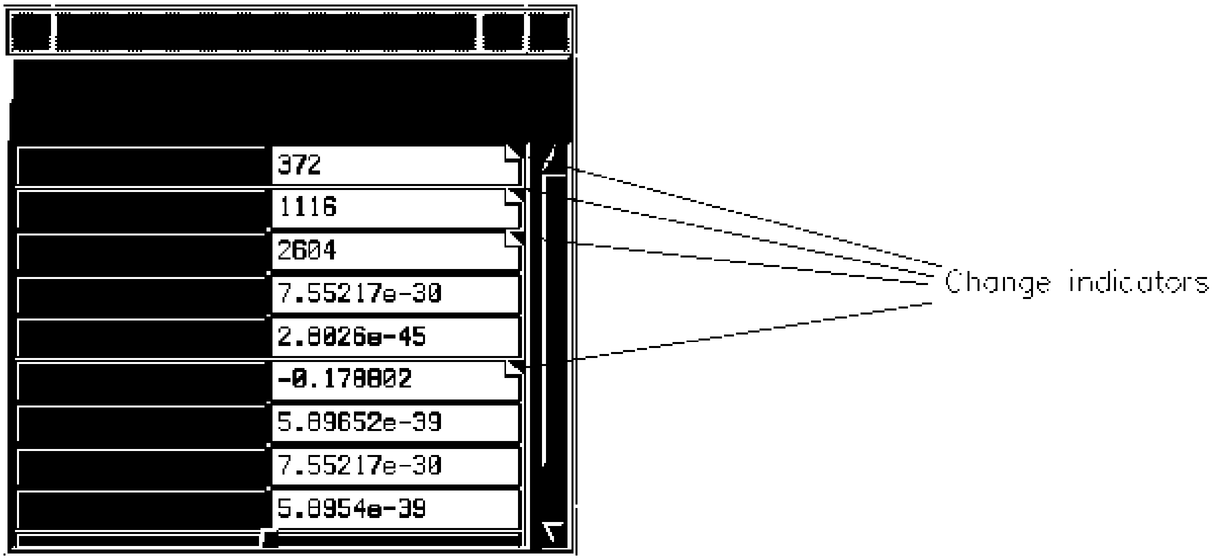


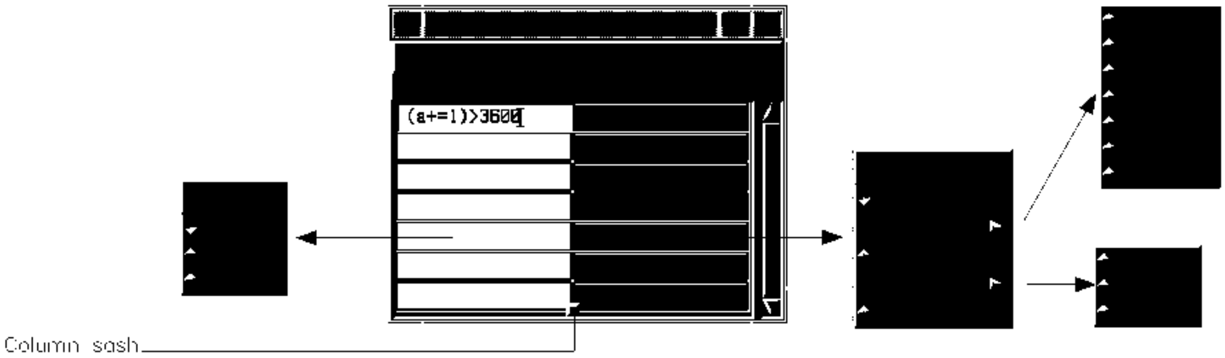
Figure 9–9 Variable Browser after Changes

7. Select "Close" from the Admin menu in Variable Browser and "Close" from the Admin menu in Call Stack View to close them.
8. Select "Expression View" from the Views menu in Main View.

   The Expression View window appears. It lets you evaluate an expression involving data from the process. The expression can be typed in or more conveniently cut and pasted from your source code. You can view the value of variables (or expressions involving variables) any time the process stops. Enter the expression in the left column, and the corresponding value appears in the right column. For more information, see **"Evaluating Expressions"** .
9. Hold down the right mouse button in the *Expression* column to bring up the Language menu. Then hold down the right mouse button in the *Result* column to display the Format menu.

   The Language menu (shown on the left side of **Figure 9–10**) lets you apply the language semantics to the expression.

   The Format menu (shown on the right side of **Figure 9–10**) lets you view the value, type, address, or size of the result. You can further specify the display format for the value and address.



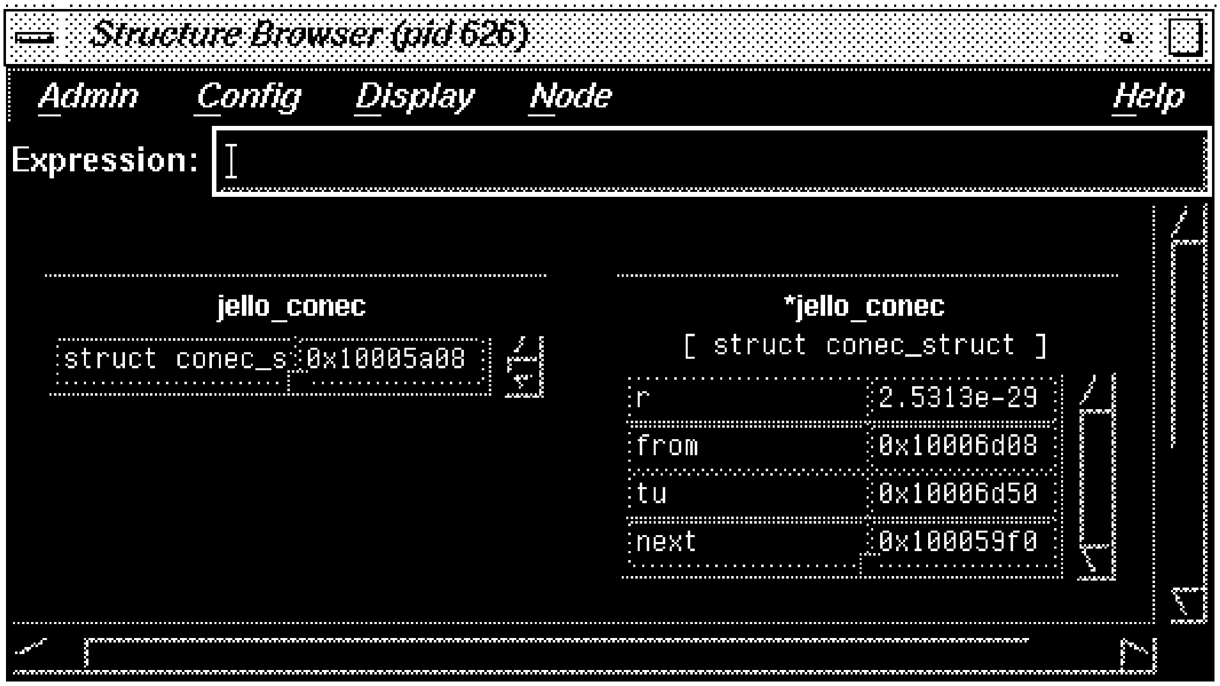
(a+=1))>3600

Figure 9–10 Expression View with Language and Format Menus Displayed

10. Click on the first *Expression* field in the Expression View window. Then enter **`(a+1)>3600`** in the field and press **`<Enter>`**.

    This is a test performed in *jello* to ensure that the value of *a* is less than 3600. This uses the variable *a* that was displayed previously in Variable Browser. After you press **`<Enter>`**, the result is displayed in the right column; 0 signifies false.
11. Select "Close" from the Admin menu in Expression View to close it.
12. Select "Structure Browser" from the Views menu in Main View.
13. Enter **`jello_conec`** in the *Expression* field and press **`<Enter>`**.

    The Structure Browser displays the structure for the given expression; field names are displayed in the left column, and values in the right column. If only pointers are available, Structure Browser will dereference the pointers automatically until actual values are encountered. You can then perform any further dereferencing by double–clicking pointer addresses in the right column of the data structure objects. A window similar to the one shown in **Figure 9–11** now appears. For more information, see **"Looking at Data Structures"** .



jello_conec

struct conec_s	0x10005a08

*jello_conec

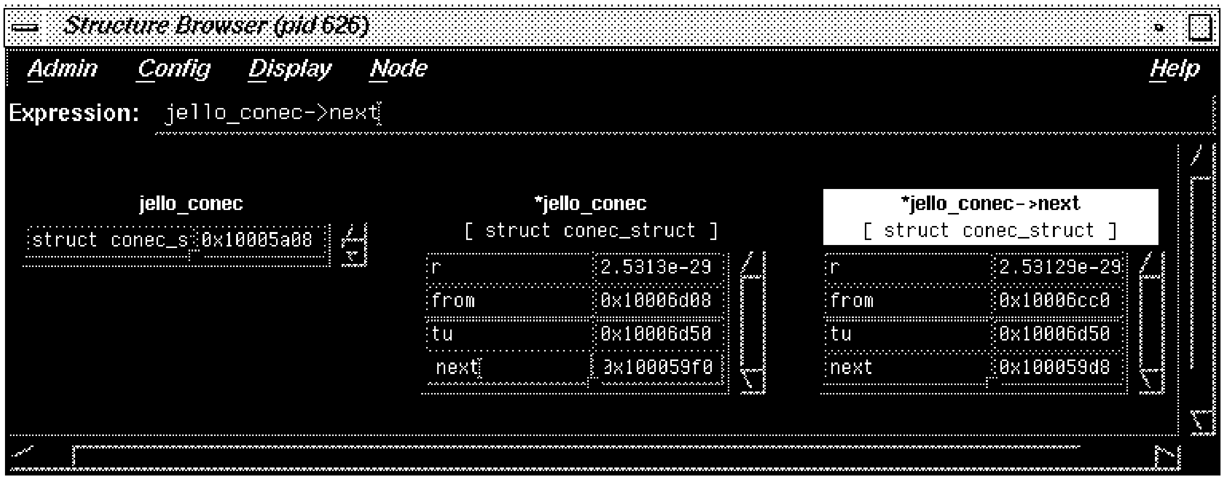
[struct conec_struct]

r	2.5313e-29
from	0x10006d08
tu	0x10006d50
next	0x100059f0

Figure 9–11 Structure Browser Window with *jello_conec* Structure

14. Click once to focus, then double–click the address of the *next* field (in the right column of the *jello_conec* structure).

    Double–clicking the address corresponding to a pointer field dereferences it. Double–clicking the field name displays the complete name of the field in the Expression field at the top of the Structure Browser window. (See **Figure 9–12** .)



Expression: jello_conec->next

jello_conec

struct conec_s	0x10005a08

*jello_conec
[struct conec_struct]

r	2.5313e-29
from	0x10006d08
tu	0x10006d50
next	3x100059f0

*jello_conec->next
[struct conec_struct]

r	2.53129e-29
from	0x10006cc0
tu	0x10006d50
next	0x100059d8

Figure 9–12 Structure Browser Window with `Next` Pointer Dereferenced

15. Select "Close" from the Admin menu in Structure Browser to close it.
16. Select "Array Browser" from the Views menu in Main View.

    The Array Browser lets you see or change values in an array variable. It is particularly valuable for finding bad data in an array or for testing the effects of values you enter. For more information, see **"Looking at Data Structures"** .
17. Type **`shadow`** in the *Array* field and press **`<Enter>`**.

    You can now see the values of the *shadow* matrix, which displays the polyhedron's shadow on the cube. The Array Browser template should resemble **Figure 9–13** but with different data values. If any fields are hidden, you can drag the sash buttons at the right of the window to expose them.

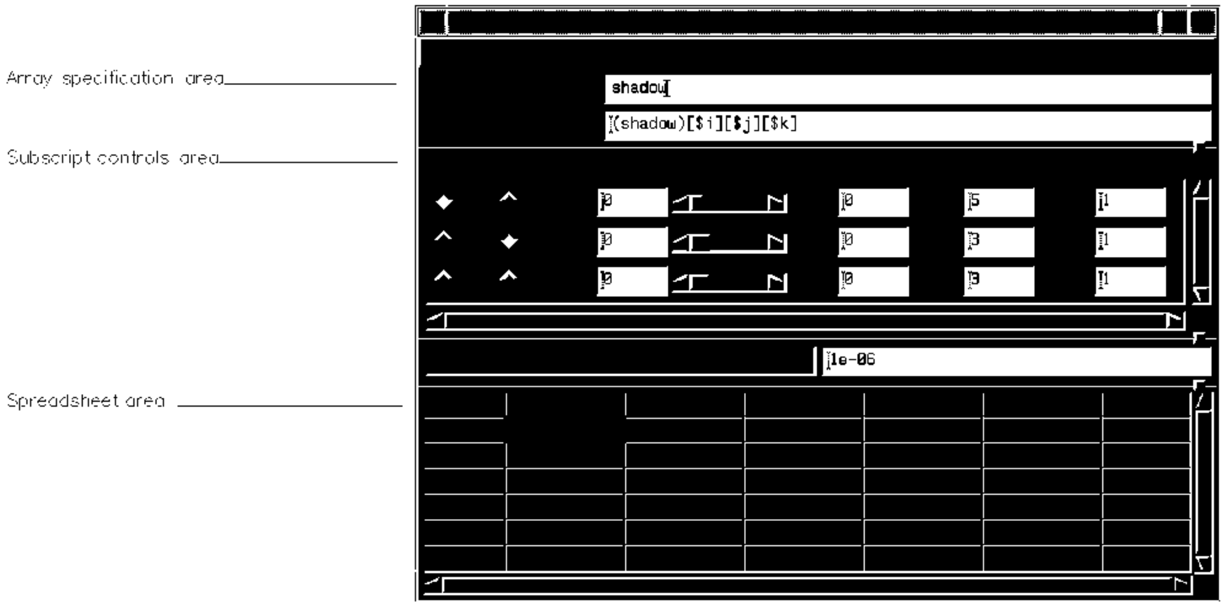


Figure 9–13 Array Browser Window for `shadow` Matrix

18. Select the *Col* button next to the *$k* index.

    The Array Browser can handle matrices containing up to six dimensions but displays only two dimensions at a time. Selecting the *Col* button for *$k* has the effect of switching from a display of *$i* by *$j* to *$i* by *$k*.

    Figure 9–14 shows a close–up view of the subscript control area.

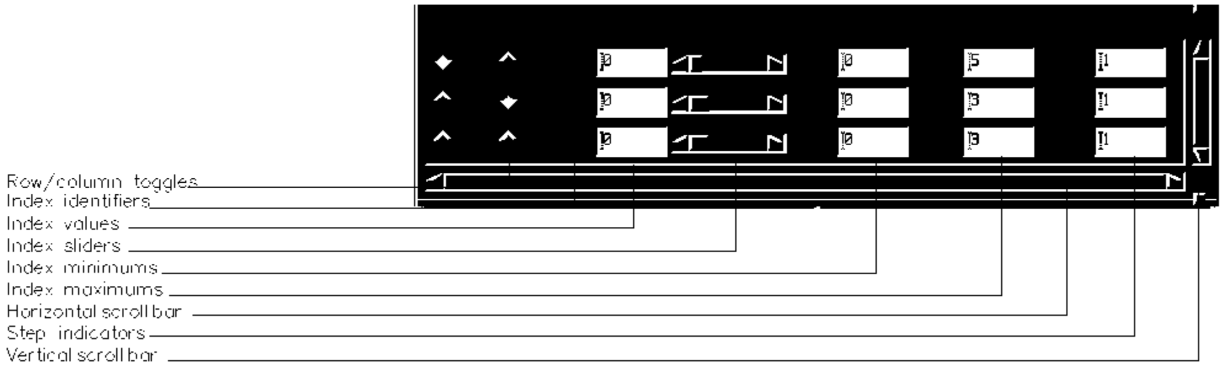


Figure 9–14 Subscript Control Area in Array Browser

The *row/column toggles* indicate whether a vector appears as a row, column, or not at all in the spreadsheet area. Although any number of vectors can reside in an array, you can view only two vectors at a time. The *index values* show the number of elements in a vector and are used to change the dimensions of the matrix. The *index sliders* let you move the focus cell along the particular vector. The *index minimums* and *index maximums* identify the beginning and ending elements, respectively, in the vectors. Use the horizontal and vertical scroll bars to expose hidden portions of the Array Browser window.

19. Select "Surface" from the Render menu.

    The Render menu displays the data from the selected array variable graphically, in this case as a three–dimensional surface. The selected cell is highlighted by a rectangular prism. The selected subscripts correspond to the *x*– and *y*–axes in the rendering; the corresponding value is plotted on the *z*–axis. The data can be rendered as a surface, bar chart, multiple lines, or points. Refer to **"Examining Array Variables"** for more information.

20. Select "Exit" from the Admin menu in Main View to end this tutorial.

Chapter 10

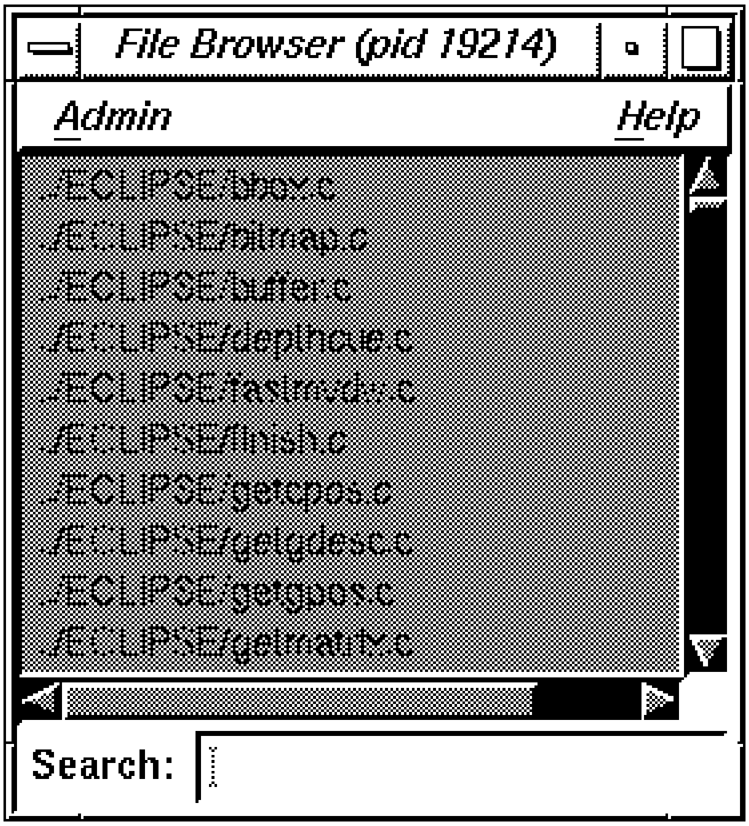
Debugger: Managing Source Files

This chapter looks at the details of working with source files. It covers these topics:

- **"Accessing Files Used by an Executable"**
- **"Opening a New File"**
- **"Path Remapping"**

Accessing Files Used by an Executable

The File Browser, available from the Views menu in the Main View, provides a scrollable list of the source files used by your executable, including files in linked libraries. See **Figure 10–1** .



```
./ECLIPSE/bbox.c
./ECLIPSE/bitmap.c
./ECLIPSE/buffer.c
./ECLIPSE/depthcue.c
./ECLIPSE/fastmode.c
./ECLIPSE/finish.c
./ECLIPSE/getcpos.c
./ECLIPSE/getgdesc.c
./ECLIPSE/getgpos.c
./ECLIPSE/getmatrix.c
```

Figure 10–1 File Browser Window

The File Browser has a field labeled *Search* for quickly locating files in the list. File searching is incremental—as you type the string you are searching for in the *Search* field, the first string that matches the entered string is highlighted.

To load a file directly into Main View from the File Browser window, simply double–click the filename.

Opening a New File

Another way to load a file is to specify it by using the "Open..." selection from the Source menu. The dialog box in **Figure 10–2** displays, listing the directories in the search path and the files in the currently selected directory.

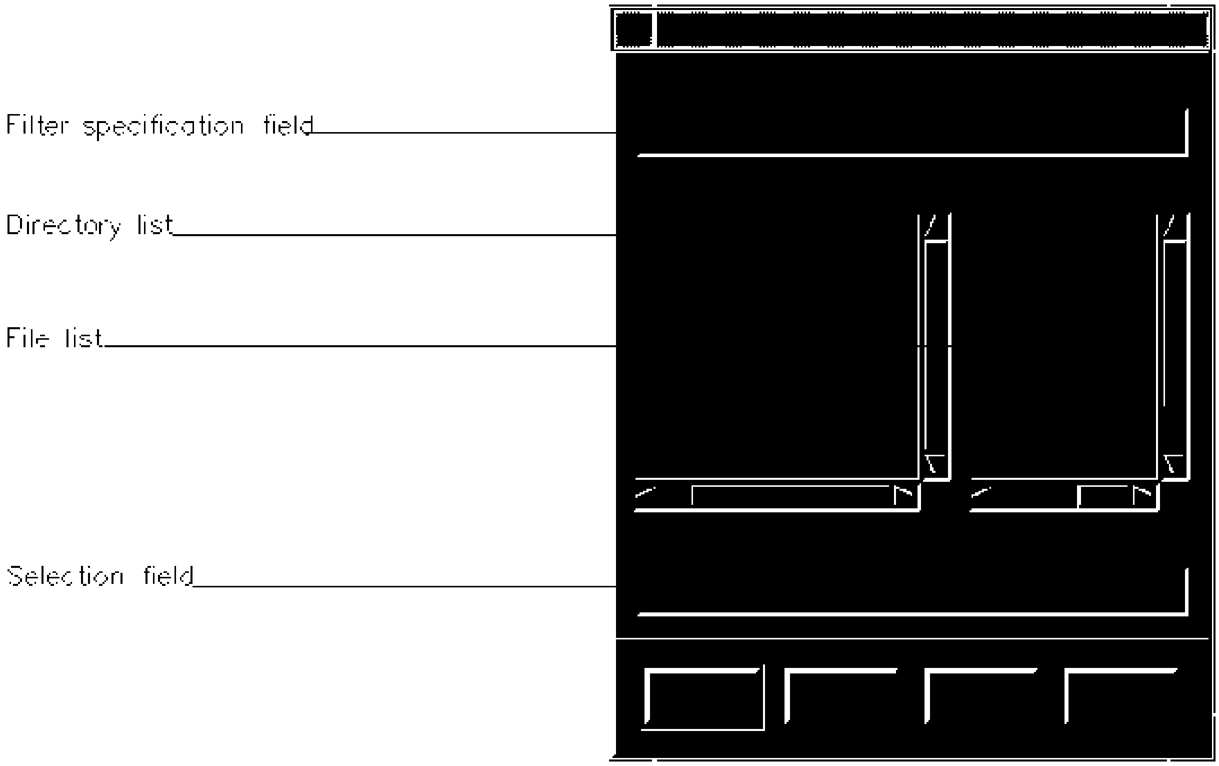


Figure 10–2 Open Source File Dialog Box

To load a file listed in the *Files* list, double–click the file's name. You can also type the full pathname of the file in the *Selection* field and click the *OK* button to load the file.

If the file you want to load is not in the current directory, select the appropriate directory by double–clicking its name in the *Directories* list. The files in the new directory are listed in the *Files* list.

If the desired directory is not in the *Directories* list, type its path in the *Filter* field near the top of the dialog box.

If you specify a file name without a full path, the Debugger will use the current path remapping information to attempt to locate the file.

Another method for opening a file from Main View is to enter its full name in the *File* field below the source code display area and press **`<Enter>`**.

Path Remapping

The path remapping option allows you to modify the set of mappings used to redirect file names located in your executable to their actual locations in your file system. Since WorkShop uses full (absolute) pathnames, path remapping is generally not necessary. However, if you have mounted executable files on a different tree from the one on which they were compiled, you will need to remap the root prefix to get access to the files in that hierarchy.

The most basic remapping is for ".", which allows you to specify the directories to be searched for files. This basic function works just like *dbx* and can be modified using the *use* and *dir* commands in the command line. To open the Path Remapping dialog box, choose "Remap Paths..." from the "Project" submenu in the Main View Admin menu. The Path Remapping dialog box appears (see **Figure 10–3**).

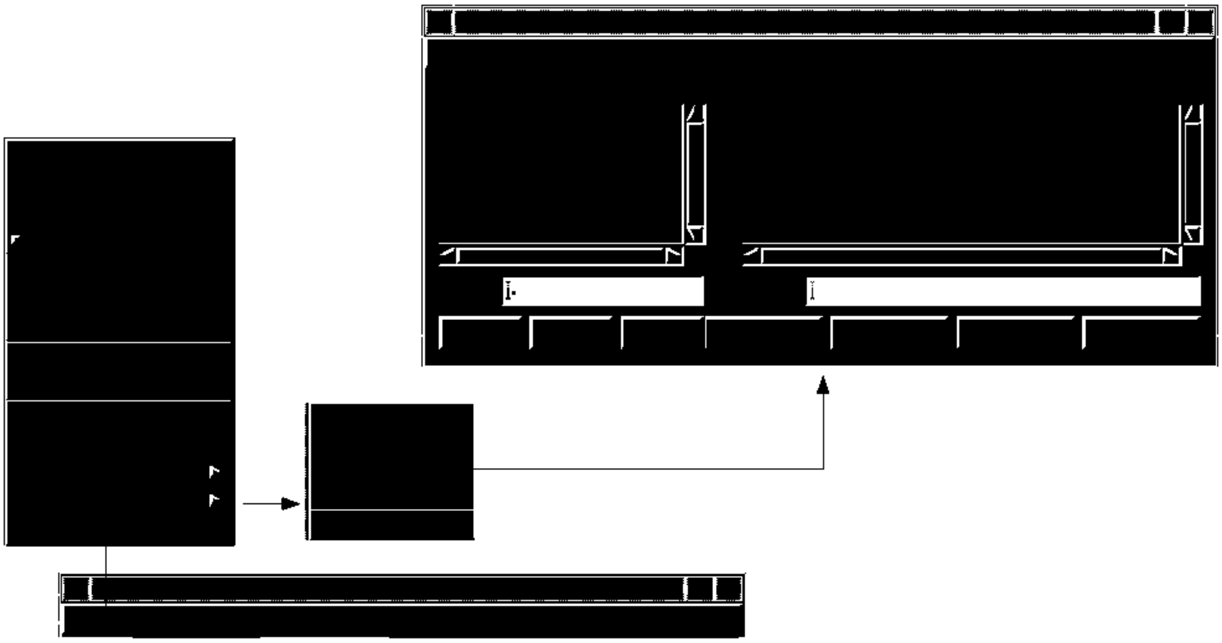


Figure 10–3 Path Remapping Dialog Box

For each prefix listed in the *Prefix* list, there is an ordered set of substitutions that are used to find a real file. By default, the path remapping is initialized so that "." is mapped to the current directory. The *Substitution Set* list shows the substitution list for the currently highlighted item in the *Prefix* list. Here are some operations you can perform:

- To view the substitution set for a different prefix, click that prefix.
- To add a new prefix, enter the new value in the *Value* field below the *Prefix* list and click the *Add* button. A new, empty substitution set is created. Next, type the desired substitution in the *Value* field below the *Substitution Set* list.
- To modify the currently selected prefix, edit the string in the *Value* field and click the *Modify* button.
- To remove the current prefix and its substitution set, select the prefix and click the *Remove* button.

Chapter 11

Debugger: CASEVision Project Session Management

WorkShop provides an integrated programming environment, with ready access between the CASEVision tools. Since users can run multiple tools on different targets simultaneously, WorkShop supplies a number of facilities to manage project sessions.

This chapter covers these topics:

- **"Project Management Concepts"**
- **"Project View"**

Project Management Concepts

A *project* is a single CASEVision work session focused on a common task. Its purpose is to let you perform operations conveniently on selected project components or on the project as a whole. A project includes all CASEVision tools and windows invoked from the command line (unless invoked with the `-privateProject` switch) and from the graphical user interface of one of those tools.

When you are working on a single project, with only a few windows open simultaneously, you can keep track of your activities with relative ease. In situations where you have many windows open from one or more projects, it is easy to get confused. To simplify the use of multiple windows, WorkShop provides facilities for iconifying and raising windows in:

- Main View Admin menu
- the "Project" submenu
- the Project View window

Figure 11–1 shows where these facilities are located.

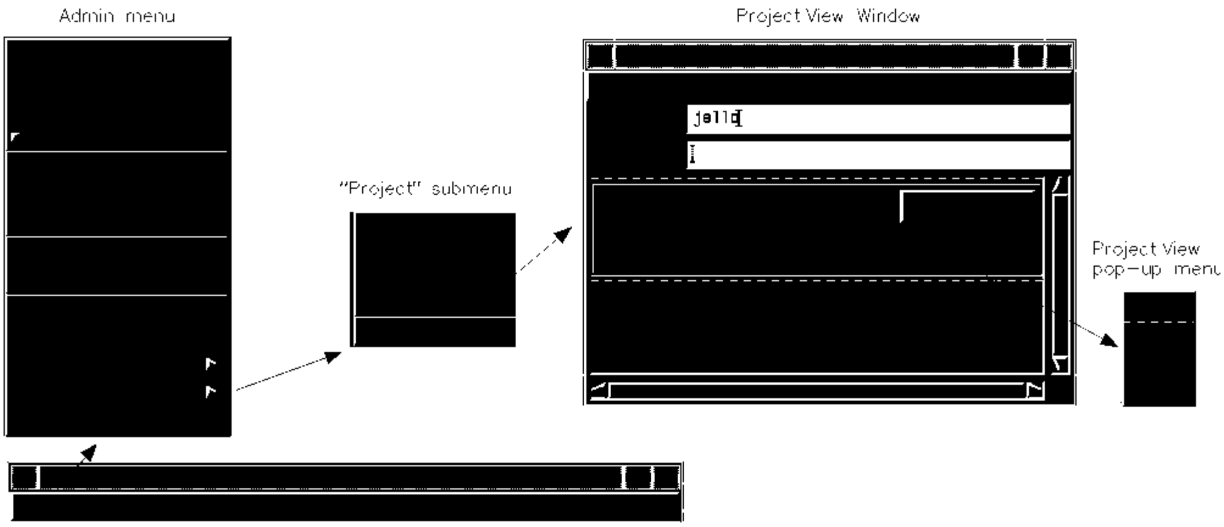


Figure 11–1 Iconify and Raise Facilities

Main View Admin Window Facilities

The window facilities in the Main View Admin menu apply to the current application and its windows only. "Iconify" iconifies all of the application's views. "Raise" brings all the application's view windows to the foreground and redisplays any iconified windows.

"Project" Submenu Window Facilities

The "Project" submenu selections "Iconify", "Raise", "Remap Paths...", and "Exit" operate the same way as their counterparts do in the MainView Admin menu except that they are applied to all tools and windows in the current project.

Project View

The Project View window displays when you select "Project View..." from the "Project" submenu in the Main View Admin menu. The Project View window is shown in **Figure 11–2** with its Admin menu and right–button pop–up menu. Project View represents the components of a project (tools or windows depending on the toggles in the Admin menu) as buttons. Elements from the same project are grouped within a rectangle. A dashed–line rectangle indicates the currently selected project. When a project is selected, you can change its name or change the command.

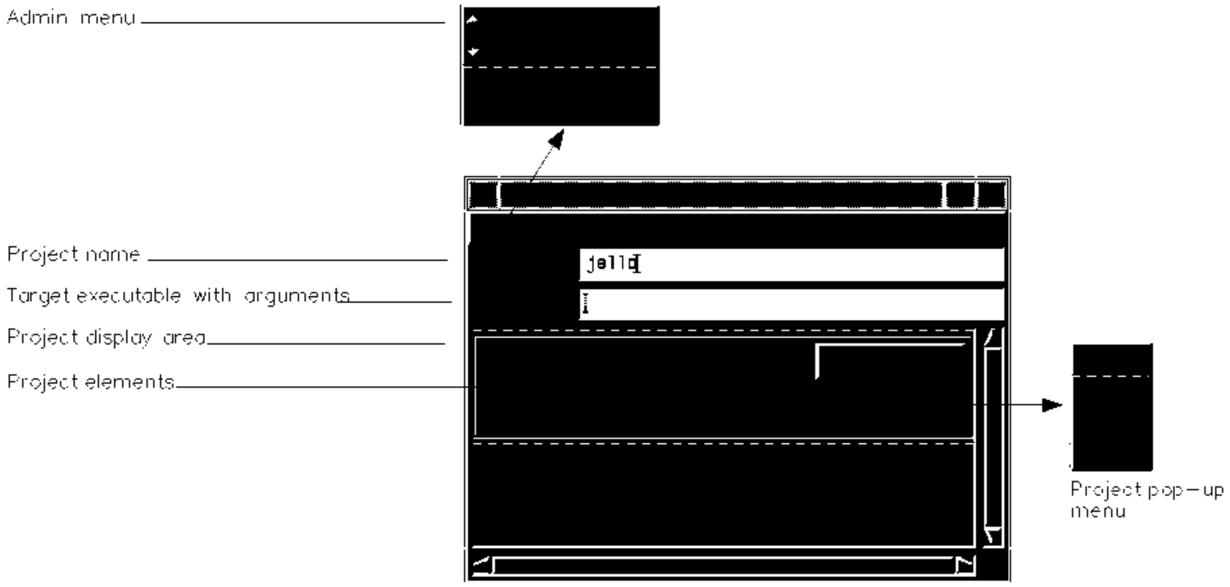


Figure 11–2 Project View Window with Menus

Project View Admin Menu

The "Show Applications" and "Show Windows" toggle selections in the Admin menu determine whether applications or individual windows display as project element buttons. The "Rescan" selection reevaluates the state of your projects and redisplays the current elements. "Exit" closes the Project View window.

Project View Text Fields

The *Project* field lets you enter a name to identify the current project.

The *Command* field lets you invoke other tools to be included in the project. These can be WorkShop tools or integrated tools, such as CASEVision/ClearCase.

Project Display Area

The elements of a project are represented by buttons. When a button protrudes from the screen, the item is currently iconified; when it is recessed, the item is displayed. Clicking the button toggles it between display and iconify modes.

Project Pop–up Menu

When you hold down the right mouse button inside a project rectangle, the Project popup menu displays. It lets you "Iconify", "Raise", or "Quit" the item under the cursor or all items in the project as a whole if the cursor is within the rectangle but not over an item.

Chapter 12

Debugger: Setting Traps in WorkShop

Setting traps is one of the most important functions of a debugger or performance analyzer. A *trap* enables you to select a location or condition within your program at which you can stop the process or collect performance data automatically. In general, you set or clear traps from Main View (Source View) or the Trap Manager. You can also specify traps in the Debugger command line at the bottom of the Main View window. For signal traps, you can also use the Signal Panel window. For system call traps, use the Syscall Panel window.

When you are debugging a program, you typically set a trap in a process to determine if there is a problem at that point. WorkShop lets you inspect the call stack, examine variables, or perform other procedures to get information about the state of the process.

Traps are also useful for analyzing a program's performance. They let you collect data related to resource usage without stopping the process.

This chapter covers the following topics:

- **"Trap Terminology"**
- **"Setting Traps in Main View and Source View"**
- **"Setting Traps in Trap Manager"**
- **"Setting Traps with Signal Panel and Syscall Panel"**
- **"Setting Traps at the Command Line"**

For a tutorial on the use of traps, see **"Setting Traps"** .

Trap Terminology

In WorkShop, the term *trap* refers to any intentional process interruption. A trap has two dimensions: the *trigger,* which specifies when the trap will fire, and the *action* taken when the trap fires, either stopping the process or capturing data.

When used as a verb, the term *trigger* refers to engaging a trap but not necessarily firing it. There are some circumstances where the process may hit a trap but not satisfy all the conditions necessary for firing it.

Trap Triggers

You can set traps at a specified location or when a specified event occurs. The triggers provided in WorkShop are:

- at a given line in a file (traditionally referred to as "breakpoints")
- at a given instruction address
- at the entry or exit for a given function
- after set time intervals (referred to as "pollpoints")
- when a given variable or address is read, written, or executed (referred to as "watchpoints")
- when a given signal is received
- when a given system call is entered or exited

Furthermore, you can specify a condition (as an expression) that must be met before a trap fires. You can also specify the *cycle count*, that is, a specific number of passes through a trap without firing it.

Trap Actions

Two actions can occur when a trap is fired:

- one or all processes can be stopped
- a sample of performance data can be taken

A trap that stops processes is called a *stop trap*, or a breakpoint. A trap that collects performance data is called a *sample trap*.

In single process debugging, a stop trap stops the current process. In multiprocess debugging, you can specify the stop trap to stop all processes or the current process only.

Sample traps are only used in performance analysis, not directly in debugging. They collect data without stopping the process. You can specify sample traps to collect such information as call stack data, function counts, basic block counts, PC profile counts, *mallocs/frees*, system calls, and page faults. Sample traps can use any of the triggers that stop traps use. Sample traps are often set up as pollpoints so that they collect data at set time intervals. For more information on performance analysis, see **Chapter 18, "Introduction to the Performance Analyzer"** .

Setting Traps in Main View and Source View

You can set traps directly in Main View by using the Traps Menu or by clicking the mouse in the source annotation column. You can also specify traps in the Debugger command line, as is discussed in **"Setting Traps at the Command Line"** .

Setting Traps with the Traps Menu in Main View

The Traps Menu in Main View is shown in **Figure 12–1** .

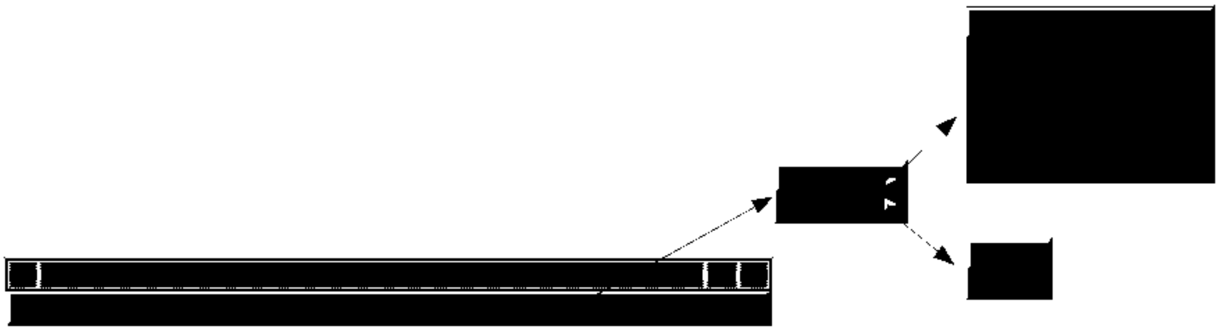


Figure 12–1 Traps Menu in Main View

To set a trap using the Traps menu, you need to identify where the trap is to be located and its type.

To set a stop trap at a line displayed in Main View (or Source View), click the cursor in the source annotation column next to the appropriate line in the source code and select "Set Trap", then "Stop" or "Sample".

For a trap at the beginning or end of a function, highlight the function name in the source code display area and select "Set Trap", then "Stop At Function Entry", "Stop At Function Exit", "Sample At Function Entry", or "Sample At Function Exit", as appropriate.

The traps are indicated by icons in the source annotation column (and also appear in Trap Manager if you have it open). **Figure 12–2** shows some typical trap icons. Sampling is indicated by a dot in the center of the icon. Traps appear in normal color or grayed out, depending on whether they are active or inactive. The active/inactive nature of traps is discussed in **"Enabling and Disabling Traps"** .

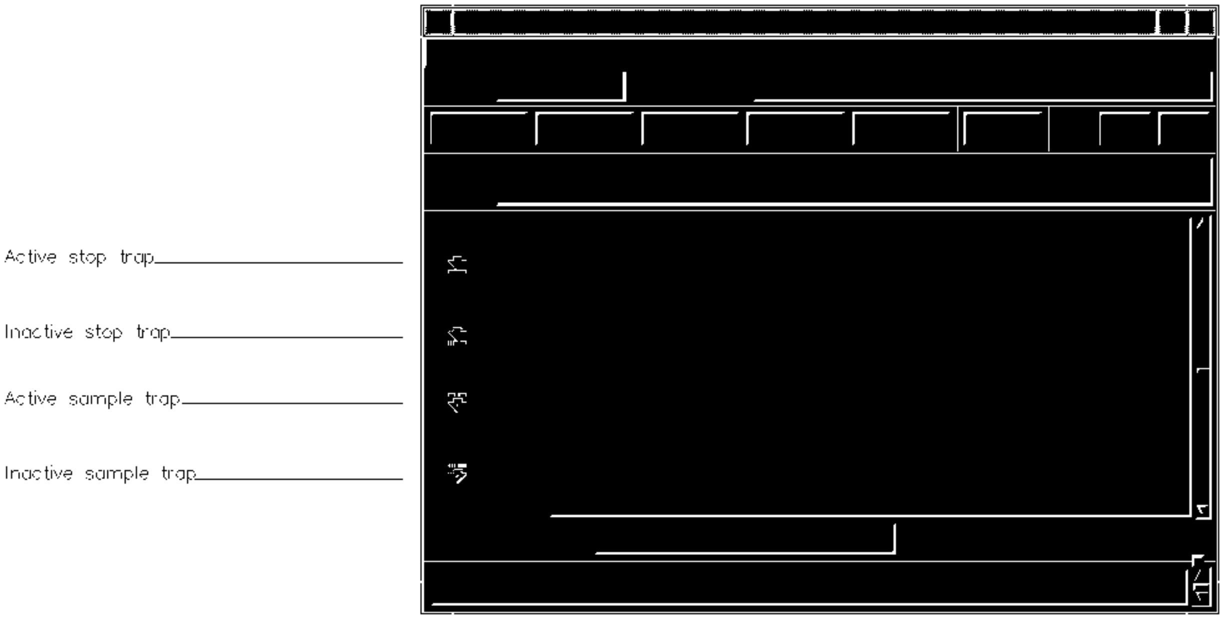


Figure 12–2 Typical Trap Icons

The "Clear Trap" selection in the Traps menu deletes the trap on the line containing the cursor. You must designate "Stop" or "Sample" trap type, since both types can exist at the same location, appearing superimposed on each other.

Setting Traps with the Mouse

The quickest way to set a trap is to click in the source annotation column in Main View and Source View. A subsequent click removes the trap. When the trap is set, an icon appears representing the trap. If data collection mode has been specified in the Performance Data window, clicking produces a sample trap; otherwise, a stop trap is entered. (The way to tell if data collection is on is to check the *Sample* button. It is enabled for data collection and disabled for debugging.)

Setting Traps in Trap Manager

The Trap Manager helps you manage all the traps involved with a process. Its two major functions are to list all traps in the process (except signals) and to let you add, delete, modify, or disable traps. The Trap Manager appears in **Figure 12–3** with the Config, Traps, and Display menus displayed.

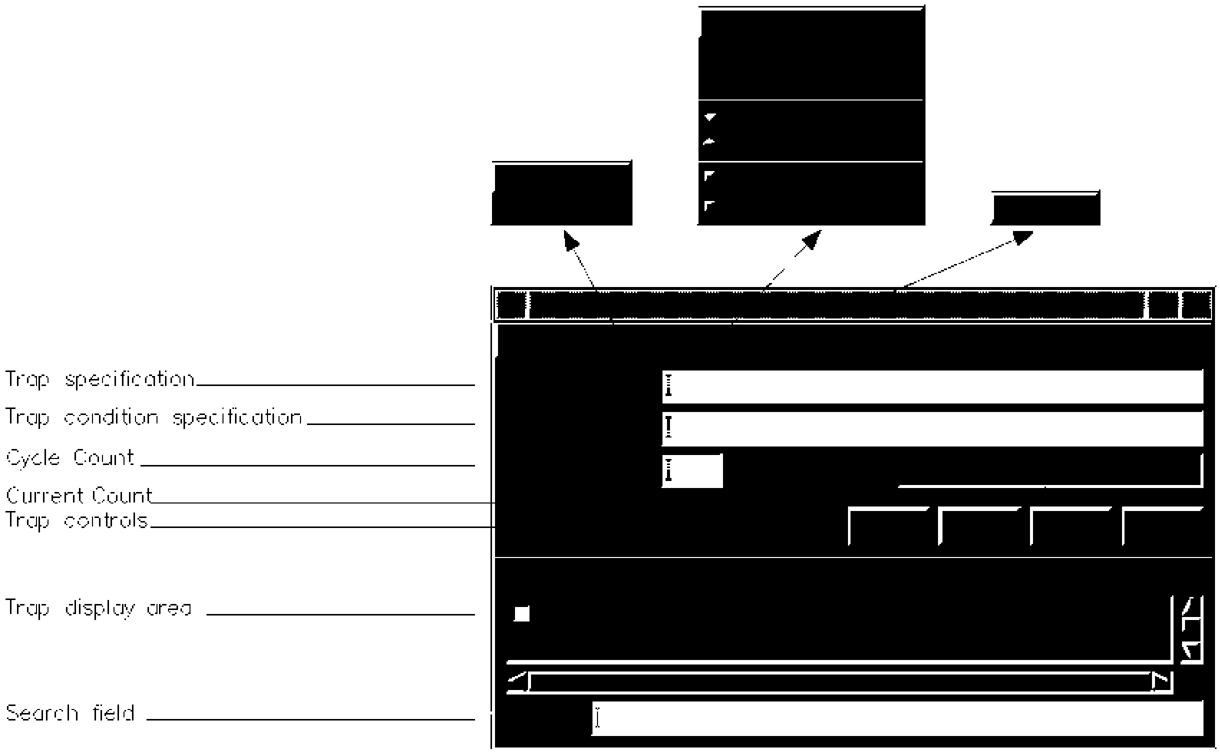


Figure 12–3 Config, Traps, and Display Menus in the Trap Manager

Setting Single– and Multiprocess Traps in the Trap: Field

New or modified traps are entered in the *Trap:* field. They have the general form:

[stop | sample] [all] [pgrp] <*location* **|** *condition***>**

The entry **[stop | sample]** refers to the trap action. You can set a default for it in the Traps menu ("Stop Trap Default" or "Sample Trap Default") and omit entering it at the beginning of the specification.

The entries **[all]** and **[pgrp]** are used in multiprocess analysis. The **[all]** entry causes all processes in the process group to stop or sample when the trap fires. The **[pgrp]** entry sets the trap in all processes within the process group that contain the code where the trap is set. These will be entered by default if the "Stop All Default" and "Group Trap Default" toggles in the Traps menu are on.

The specific syntaxes for the location and condition are shown below. After you enter the trap (by the *Add* or *Modify* button or by **<Enter>**), the full syntax of the specification appears in the field. The *Clear* button clears the *Trap* and *Condition* fields and the cycle fields.

[stop | sample] [all] [pgrp] at [file *filename***] **
[line *line–number***]**
Trap at the specified line in the specified file

[stop | sample] [all] [pgrp] addr *instruction–address*
Trap on the specified instruction address

**[stop | sample] [all] [pgrp] entry function [[file] **
*filename***]**

[stop | sample] [all] [pgrp] in *function* **[[file] **
*filename***]**
Trap on entry to the specified function. If the file name is given, the function is assumed to be in that file's scope.

[stop | sample] [all] [pgrp] exit *function* **[[file] **
*filename***]**
Trap on exit from the specified function. If the file name is given, the function is

[stop | sample] [all] [pgrp] watch *expression* ****
[[for] read | write | execute [access]]
Set a data watchpoint on the specified expression (using the address and size of the expression for the watchpoint span). The watchpoint may be specified to fire on write, read, or execution (or some combination thereof). If not specified, the write condition is assumed.

[stop | sample] [all] [pgrp] watch addr[ess] *address* ****
[[size] *size***] [for] read | write | execute \\ [access]]**
Set a watchpoint for the specified address and size in bytes. The watchpoint may be specified to fire on *write*, *read*, or *execute* (or some combination thereof) of memory in the given span. If not specified, the size defaults to 4 bytes.

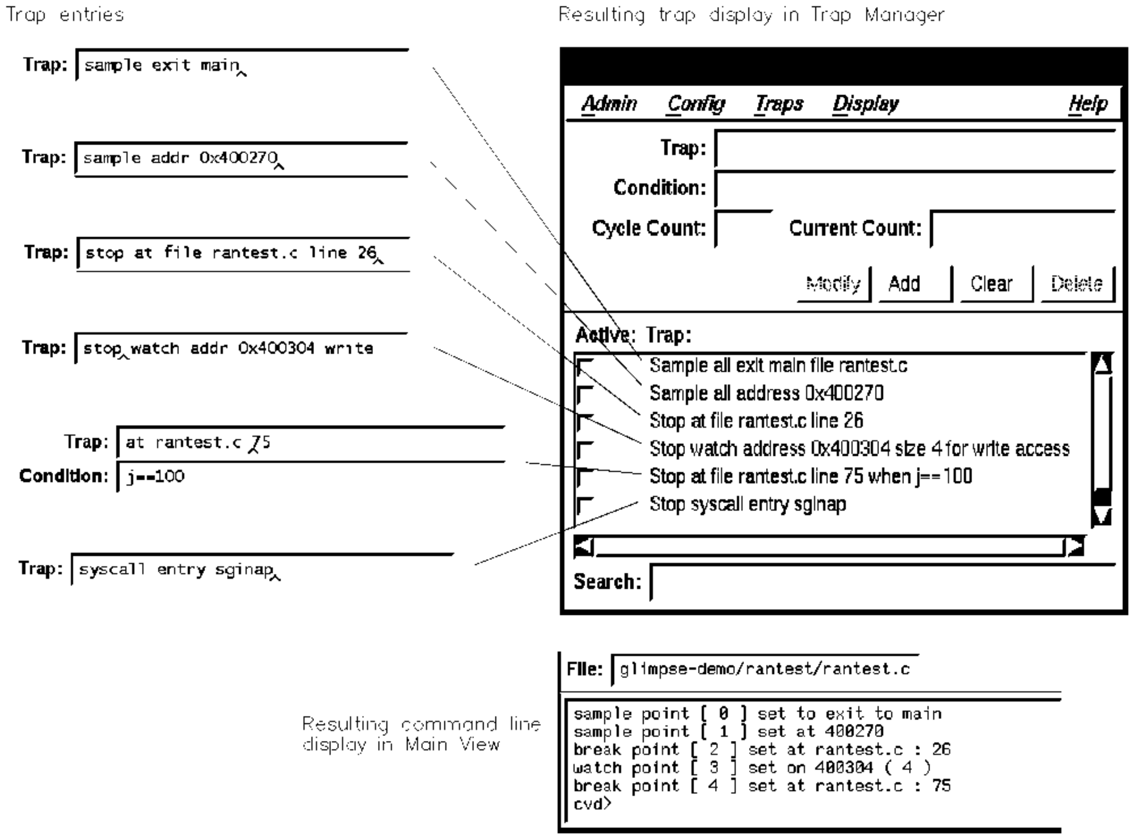
[stop | sample] [all] [pgrp] signal *signal–name*
Trap on receipt of the given signal. Same as "catch" in *dbx*.

[stop | sample] [all] [pgrp] syscall entry *sys–call–name*
Trap on entry to the specified system call. This is slightly different from setting a trap on entry to the function by the same name. A syscall entry trap sets a trap on entry to the actual system call. A function entry trap sets a trap on entry to the stub function that calls the system call.

[stop | sample] [all] [pgrp] syscall exit *sys–call–name*
Trap on exit from the specified system call. This is slightly different from setting a trap on exit from the function by the same name. A syscall exit trap sets a trap on exit from the actual system call. A function exit trap sets a trap on exit from the stub function that calls the system call.

[stop | sample] pollpoint [interval] *time* **[seconds]**
Trap at regular intervals of *time* seconds. This is typically used for sampling only.

Some typical trap examples are provided in **Figure 12–4** . The entries made in the *Trap* field are shown in the left portion of the figure, the trap display in Trap Manager resulting from these entries is shown on the right, and the trap display shown at the command line in Main View is shown at the bottom.



Trap: sample exit main

Trap: sample addr 0x400270

Trap: stop at file rantest.c line 26

Trap: stop watch addr 0x400304 write

Trap: at rantest.c 75
Condition: j==100

Trap: syscall entry sginap

Trap:
Condition:
Cycle Count:
Current Count:

Active: Trap:
Sample all exit main file rantest.c
Sample all address 0x400270
Stop at file rantest.c line 26
Stop watch address 0x400304 size 4 for write access
Stop at file rantest.c line 75 when j==100
Stop syscall entry sginap

File: glimpse-demo/rantest/rantest.c

```
sample point [ 0 ] set to exit to main
sample point [ 1 ] set at 400270
break point [ 2 ] set at rantest.c : 26
watch point [ 3 ] set on 400304 ( 4 )
break point [ 4 ] set at rantest.c : 75
cvd>
```

Figure 12–4 Trap Examples

Setting a Trap Condition

The *Condition:* field lets you specify a condition necessary for the trap to be fired. A condition can be any legal expression and will be true if it returns a nonzero value when the corresponding trap is encountered. The expression must be valid in the context in which it will be evaluated. For example, a Fortran condition like `a .gt. 2` cannot be evaluated if it is tested while the program is stopped in a C function.

There are two possible sequences for entering a trap with a condition:

1. Define the trap.
2. Define the condition.
3. Click *Add*.

and

1. Define the trap.
2. Click *Add*.
3. Define the condition.
4. Click *Modify* or press `<Enter>`.

An example of a trap with a condition is shown in **Figure 12–4**. The expression `j==100` has been entered in the *Condition:* field. (If you were debugging in Fortran, you would use the Fortran syntax, for example, `j .eq. 1`.) After the trap has been entered, the condition appears as part of the trap definition in the display area. During execution, the requirements set by the trigger must be satisfied first for the condition to be tested. A condition is true if the expression (valid in the language of the program you are debugging) evaluates to a nonzero value.

Setting a Trap Cycle Count

The *Cycle Count* field lets you pass through a trap a specific number of times without firing. If you set a cycle count of *n*, the trap will fire at the *n*th time the trap is encountered and every *n* iterations thereafter. The *Current Count* field indicates the number of times the process has passed the trap since either the cycle count was set or the trap last fired. The current count updates only when the process stops.

Setting a Trap with the Traps Menu and Source Display

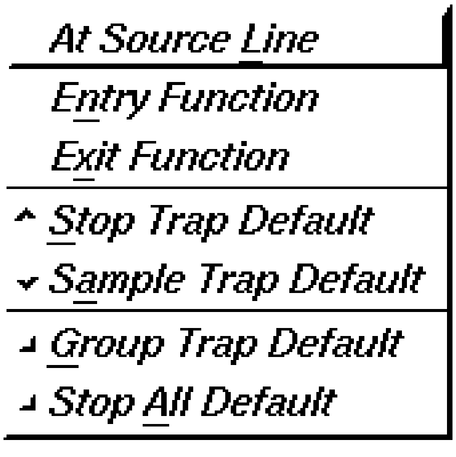


Figure 12–5 Traps Menu in Trap Manager

Figure 12−5 Traps Menu in Trap Manager

The Traps menu in Trap Manager (see **Figure 12−5**) lets you specify traps in conjunction with Main View or Source View. Clicking "At Source Line" sets a trap at the line in the source display area containing the current selection.

To set a trap at the beginning or end of a function, you select the function name in the source display and then click "Entry Function" or "Exit Function". The trap set in all of these cases is governed by the defaults you have set in the menu.

Moving around the Trap Display Area

The trap display area shows all traps set for the current process. You have vertical and horizontal scroll bars for moving around the display area. The *Search* field lets you incrementally search for any string in any trap.

Enabling and Disabling Traps

Each trap has an indicator to its left for toggling back and forth between its active and inactive states. This feature lets you accumulate traps and turn them on only as needed. Thus, when you don’t need the trap, it won’t get in your way. When you do need it, it is readily re−enabled.

Saving and Reusing Trap Sets

The "Load Traps..." selection in the Config menu lets you bring in previously saved trap sets by means of a file browser window. This is useful for reestablishing a set of traps between debugging sessions. "Save Traps..." lets you save the current traps to a file.

Setting Traps with Signal Panel and Syscall Panel

You can trap signals using Signal Panel and system calls using Syscall Panel (see **Figure 12−6**).

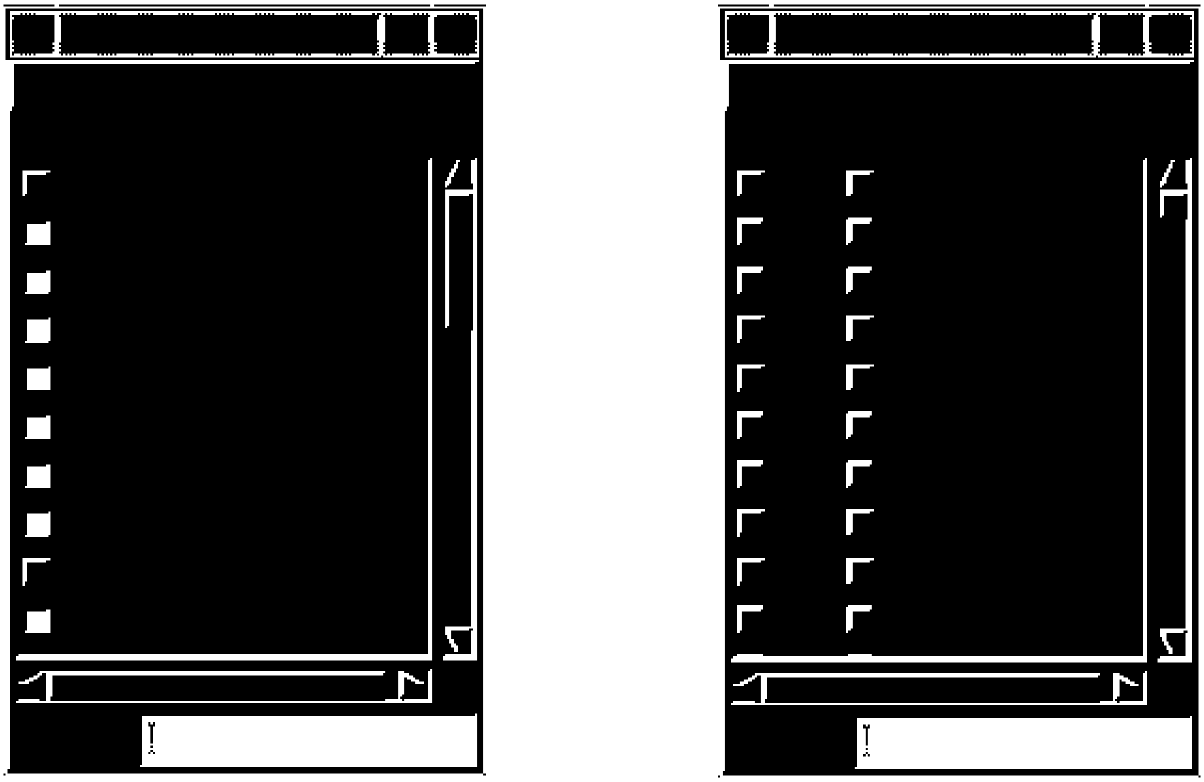


Figure 12–6 Signal Panel and Syscall Panel

Both are selected from the Views menu in Main View. Signal Panel sets a trap on receipt of the signal(s) selected. Syscall Panel sets a trap at the selected entry to or return from the system call.

Setting Traps at the Command Line

You can also enter the trap commands at the Debugger command line at the bottom of Main View. From the Debugger command line, you can enter any of the commands described below as well as those shown in **"Setting Single– and Multiprocess Traps in the Trap: Field"** ; these are convenient for those users familiar with *dbx*.

`catch [signal_name | all]`

With no arguments, lists the signals to be trapped. If a signal is specified, adds it to the list. If **`all`** is specified, traps all signals.

`clear [all |` *source_line*]

Clears breakpoints. The **`all`** option clears all breakpoints; the *source_line* option clears the breakpoint at the specified source line.

`clearcalls`

Cancels pending interactive function calls.

`disable` *trap_number* [, *trap_number* ...]

Disables a trap set by the *stop* command.

`enable` *trap_number* [, *trap_number* ...]

Re–enables a stop trap set that has been disabled.

`ignore` [*signal_name* | **`all`**]

With no arguments, lists those signals not to be trapped. If a signal is specified, removes it from the list of signals to be trapped. If **`all`** is specified, ignores all signals.

`status`

Displays a list of currently set breakpoints and traces.

`stop at` [*filename*:] *line_number* [**`if`** *expression*] only fires if *expression* is true.

`stop in [`*filename***`:]`** *function_name* **`[if`** *expression***`]`**

Traps at the entry to the specified function. If the **`if`** option is used, then the trap only fires if *expression* is true.

`syscall catch | ignore [call | return] \`
`[`*sys_call_name* **`| all]`**

The **`catch`** option adds a system call to the list of system calls to be trapped. The **`ignore`** option removes a system call from the list of system calls to be trapped.The **`call`** option specifies the entry to the system call and **`return`** signifies the return from the call.

`trace [`*variable***`] at [["filename":] \`**
`[`*line_number* **`|`** *function_name***`] \`**
`[if` *expression***`]]`**

Traces the specified variable. A file and/or test condition can be optionally specified. You can also specify a line number or a function where the trace is to take place.

`when at [`*filename***`:]`** *line_number* **`{`***command***`} \`**
`[; {`*command***`} ...]`**

Stops the process and performs other Debugger commands when the process reaches a specified line number.

`when in [`*filename***`:]`** *function_name* **`[{`***command***`}; \`**
`{`*command***`} ...]`**

Stops the process and performs other Debugger commands at entry to a function. If the filename is given, the function is assumed to be in that file's scope.s

---

Chapter 13

Debugger: Controlling Process Execution

This chapter tells you how to control process execution in CASEVision. It includes the following topics:

- **"Main View Control Panel"**
- **"Controlling Process Execution with PC Menu"**
- **"Controlling Execution from the Debugger Command Line"**
- **"Execution View"**

Main View Control Panel

Process execution is controlled using the top portion of the Main View window. See **Figure 13–1** .

Figure 13–1 Main View Control Panel

The Main View window contains a row of execution control buttons that enable you to control program execution. The execution control buttons are located above the source display area. To use any of these commands, click on the appropriate button with the left mouse button. The Main View control panel is described below.

Status and Entry Fields in the Main View Control Panel

The panel contains the following fields:

Host

identifies the host system for the target program. The host system is specified at startup with the **`-host`** argument.

Command

lets you enter the command for running the process with any argument(s).

Status

displays information about the execution status of the program you are debugging. The top line in this box tells you whether the program is running or stopped. The next line lists the current call stack frame, if applicable. (To see all of the stack frames, open the Call Stack View from the Views menu.)

Execution Control Buttons

The execution control buttons enable you to control program execution. The two control buttons for starting and terminating a process are:

Run

creates a new process for the program and starts execution. It is also used to rerun the program.

Kill

kills the active process.

The control buttons used for process interruptions are

Continue

resumes program execution after a halt and continues until a breakpoint or other event stops execution.

Stop

stops execution of the program. When program execution stops, the current source line is highlighted in the Main View and annotated with an arrow indicating the program counter (PC).

Step Into

steps to the next source line and into function calls. To step a specific number of lines, hold down the right mouse button over the *Step Into* button. This displays the pop−up menu shown in **Figure 13−2** . You can select one of the fixed values or enter your own number of steps by selecting "N...". Selecting "N..." displays the dialog box shown at the right of **Figure 13−2** .

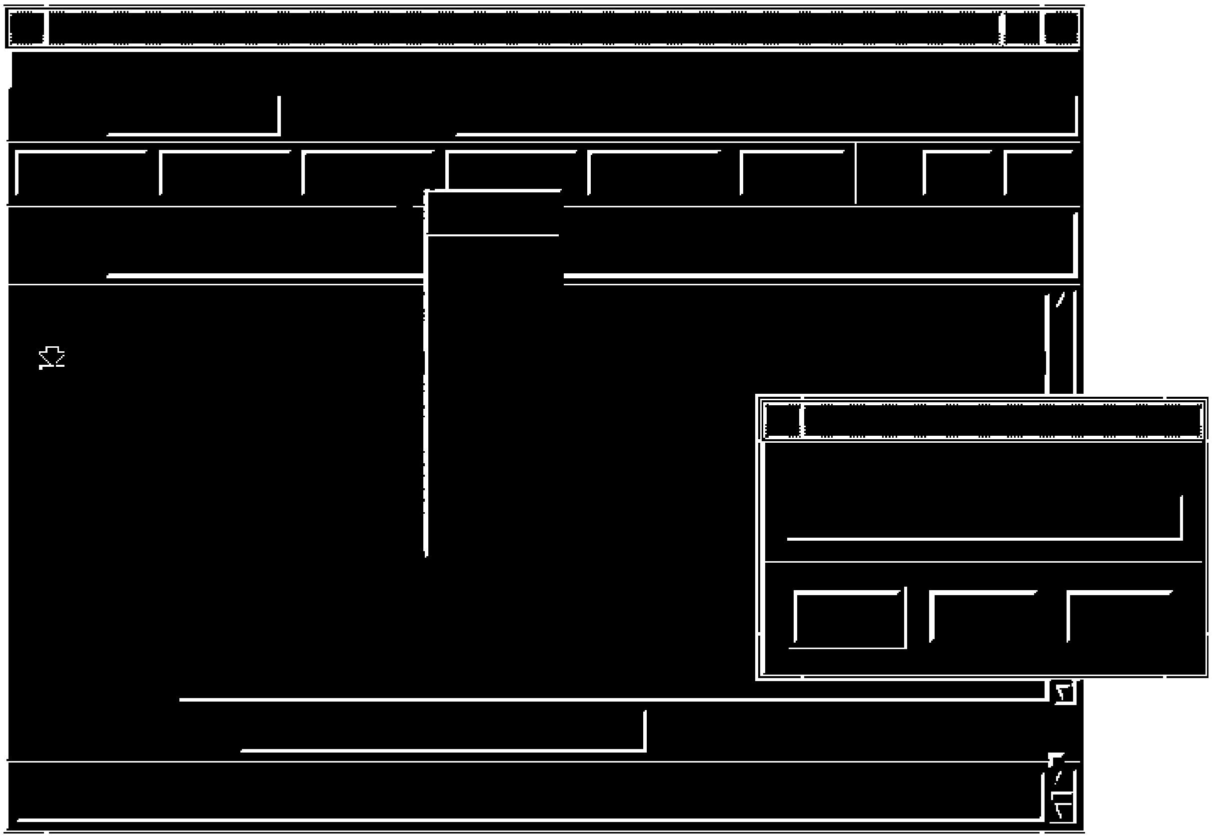


Figure 13–2 Step Into Pop–up Menu and Dialog Box

Step Over

steps to the next source line and over function calls. To step a specific number of lines, hold down the right mouse button over the *Step Over* button. This displays the pop–up menu shown in **Figure 13–3** . You can select one of the fixed values or enter your own number of steps by selecting "N...". Selecting "N..." displays the dialog box shown at the right of **Figure 13–3** .

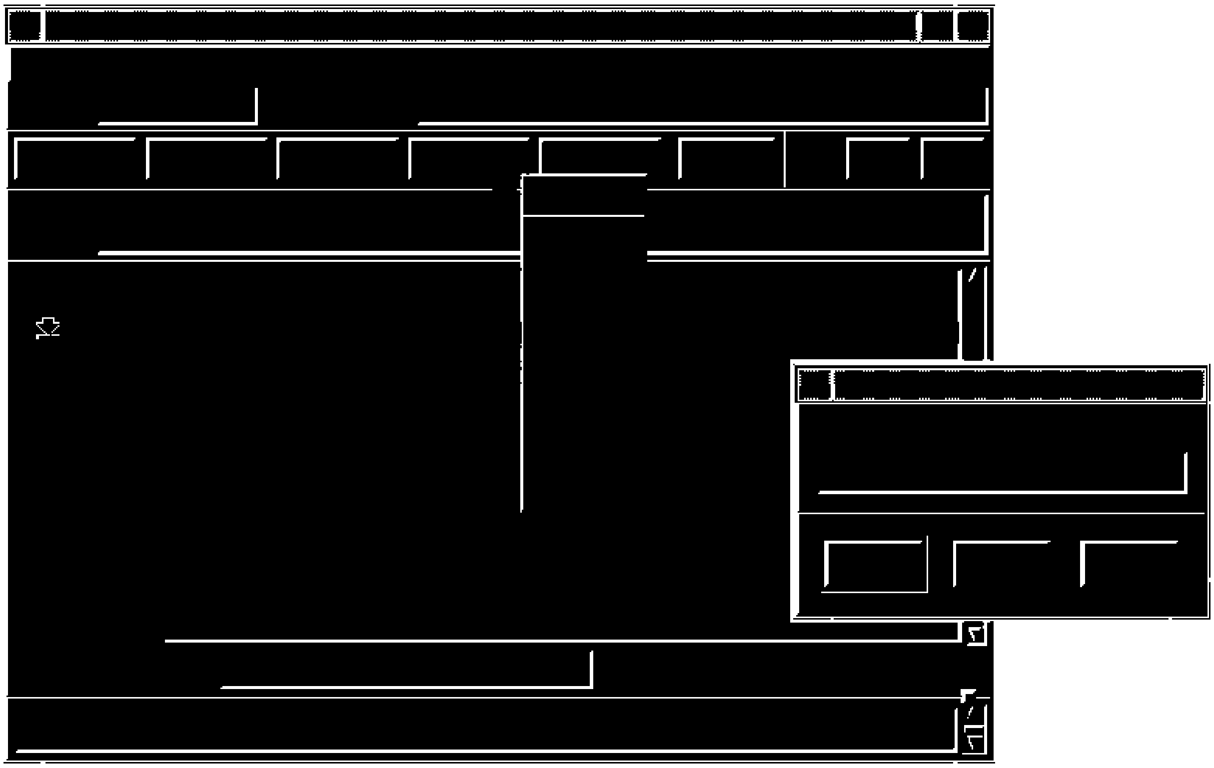


Figure 13–3 Step Over Popup Menu and Dialog Box

Return

executes the remaining instructions in the current function. Program execution stops upon return from that procedure.

There is one button in the control panel for spontaneous sampling:

Sample

collects performance data when clicked. A performance task must have been previously specified in the Performance Task window and data collection must have been enabled.

Controlling Process Execution with PC Menu

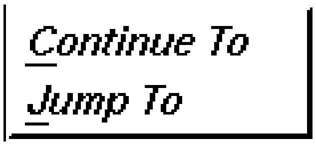


Figure 13–4 PC Menu in Main View

The PC (program counter) menu in Main View provides a quick and informal means of controlling process execution. See **Figure 13–4** .

These options let you manually control process execution without setting traps. The target location is determined by the location of the cursor in the source display area. There are two selections:

"Continue To"

lets you select a target location in the current process (by placing the cursor in the line). The process proceeds from the current PC to that point (provided there are no to setting a one–time trap. If the process is interrupted before reaching the target location, then the command is cancelled.

"Jump To"

lets you select a target location in the current process (by placing the cursor in the line). The location must be in the same function. Instead of starting from the current PC, "Jump To" skips over any intervening code and restarts the process at the target. This is particularly useful if you want to get around bad code or irrelevant portions of the program. It also lets you back up and re–execute a portion of code.

Controlling Execution from the Debugger Command Line

WorkShop also offers a number of commands for controlling execution that can be entered at the Debugger command line:

step [*INT***]**

steps the specified number of source instructions. This command steps into procedures. The default is one instruction.

next [*INT***]**

steps over the specified number of source instructions. This command does not step into procedures. The default is one instruction.

stepi [*INT***]**

steps the specified number of machine instructions. This command steps into procedures. The default is one line.

nexti [*INT***]**

steps over the specified number of machine instructions. This command does not step into procedures. The default is one line.

continue

continues executing a program after a breakpoint.

continue [*signal***]**

sends a specified signal and continues executing a program after a breakpoint.

return

continues executing the current procedure and returns to the next sequential line in the calling function.

cont to *line_number*

continues execution from the current line until the specified line.

cont in *function_name*

continues execution from the current line to the entry to the specified function.

display [*expression***, ...]**

with *expression*, this command adds *expression* to the list of expressions to be displayed whenever the process stops.
with no arguments, lists all expressions on the display list.

undisplay [*displaynumber***, ...]**

stops display of expression with specified *displaynumber* when the process stops. Removes the expression from the display list.

call *function_name* **[***argument***, ...]**

executes the specified function with any arguments supplied.

run

runs the program.

rerun

runs the program again using the same arguments

Execution View

The Execution View window is a simple shell that lets you set environment variables and inspect error messages. Your target program I/O, if any, will be displayed in the Execution View window. If the program is I/O–based, then all interaction takes place in Execution View.

Chapter 14

Examining Debugger Data

After you have learned how to set traps in CASEVision, the next step is to look at the facilities for examining the data. This chapter covers:

- **"Tracing through Call Stack View"**
- **"Evaluating Expressions"**
- **"Examining Variable Values"**
- **"Examining Array Variables"**
- **"Looking at Data Structures"**
- **"Examining Data at the Command Line"**

The Debugger also lets you examine data at the machine level. The tools for viewing disassembled code, machine registers, and data by specific memory location are covered in **Chapter 15, "Machine-level Debugging"** .

Tracing through Call Stack View

The Call Stack View window displays the functions in the call stack (referred to as *frames*) when the process has stopped. The window is shown in **Figure 14-1** with the major menus displayed.

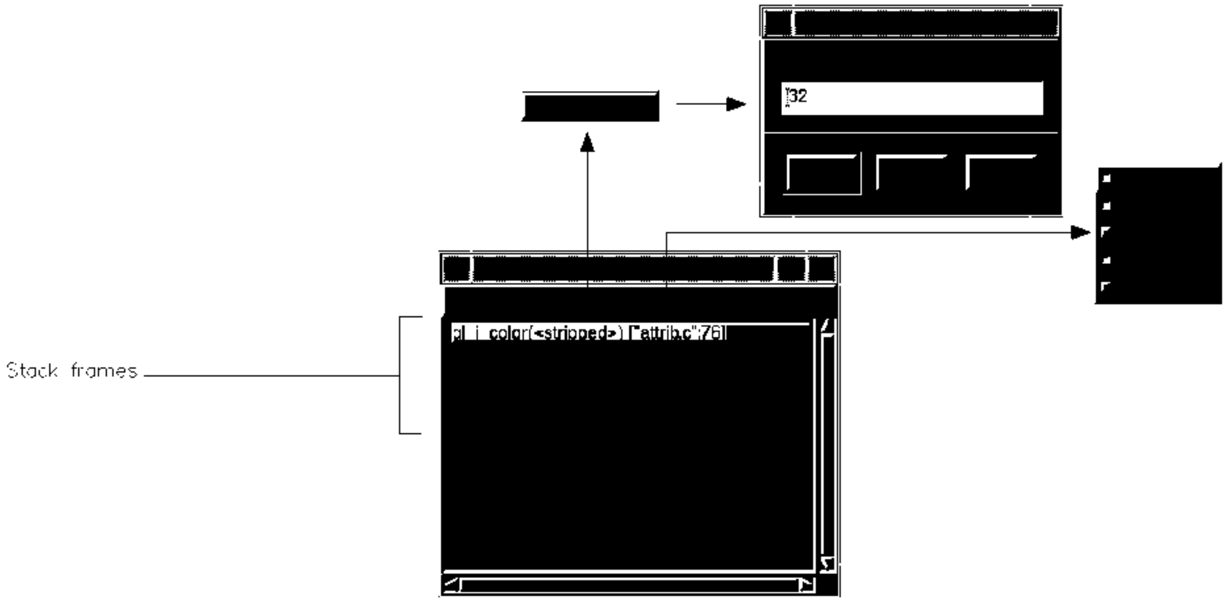


Figure 14-1 Call Stack View Window with Config and Display Menus and Preferences Dialog Box

In addition to the functions, the Call Stack View window lets you see the argument names, values, and types as well as the locations of the functions and the PC. If symbolic information for the arguments has been stripped from the executable, the label `<stripped>` will appear in place of the arguments, as in the function *_start* in the above example. You also have the option of setting the maximum depth of the Call Stack View by selecting "Preferences..." from the Config menu.

To move through the call stack, you simply double-click on a frame in the stack. The frame becomes highlighted to indicate the current context. The source display in Main View (or Source View) scrolls automatically to the location where the function was called and any other active views (such as the Variable Browser or Structure Browser) will update. The source display has two special annotations:

- The location of the current program state is indicated by a large green (depending on color scheme) arrow representing the PC.

- The location of the call to the function selected in the Call Stack View window is indicated by a smaller blue (depending on color scheme) arrow representing the current context, and the source line becomes highlighted.

Figure 14–2 illustrates the correspondence between a frame and the source code when a frame is clicked in the Call Stack View window. In this example, the stack frame *spin* has been selected; Main View scrolls to the place where the trap occurred. If the second stack (*main*) had been selected, then the window would have scrolled to the place where the function *main* calls *spin*.

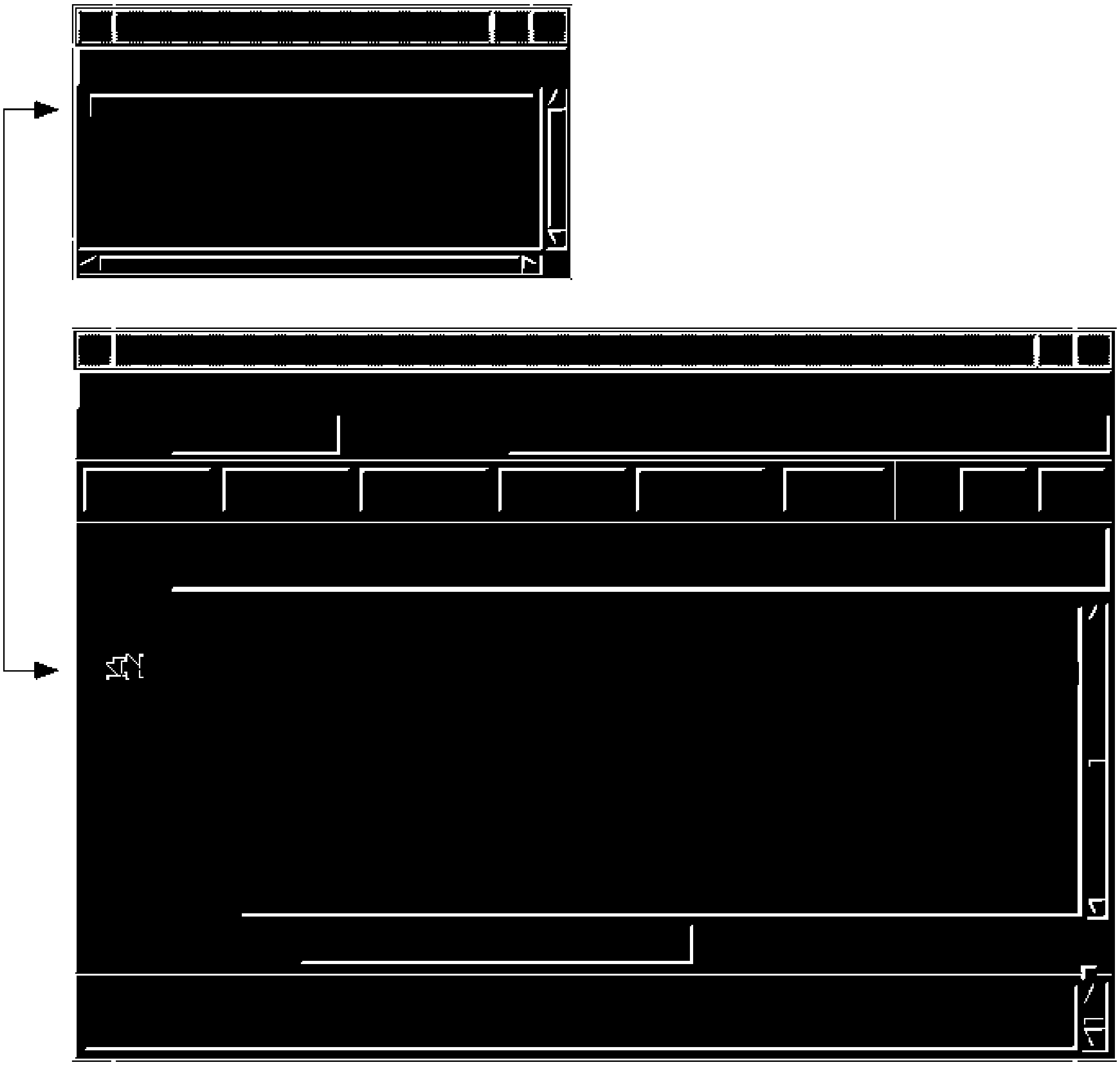


Figure 14–2 Tracing through Call Stack View

Evaluating Expressions

You can evaluate any valid expression (in C, C++, or Fortran) at a stopping point in the process and trace it through the process. Expressions are evaluated by default in the frame and language of the current context. Expressions may contain data names or constants; however, they may not contain names known only to the C pre–processor, as in a *#define* directive or a macro.

To evaluate expressions, you can use one of these methods:

Expression View
: lets you evaluate multiple expressions simultaneously, updating their values each time the process stops.

Evaluate Expression from the Data menu
: lets you quickly evaluate a selected expression.

Expression View

The Expression View window is shown in **Figure 14–3** with its major menus displayed. Note that Expression View has two pop-up menus. The Language menu is invoked by holding down the right mouse button while the cursor is in the Expression column. The Format menu is displayed by holding down the right mouse button in the Result column.

To specify the expression to be evaluated, first click in the Expression column, on the left side of the window, then enter the expression in the selected field. This expression can be typed directly or pasted in from the source code display. It must be a valid expression in the current or selected language: C, C++, or Fortran. To change languages, display the Language menu and make your selection. When you press **`<Enter>`**, the result of the expression is displayed in the right column.

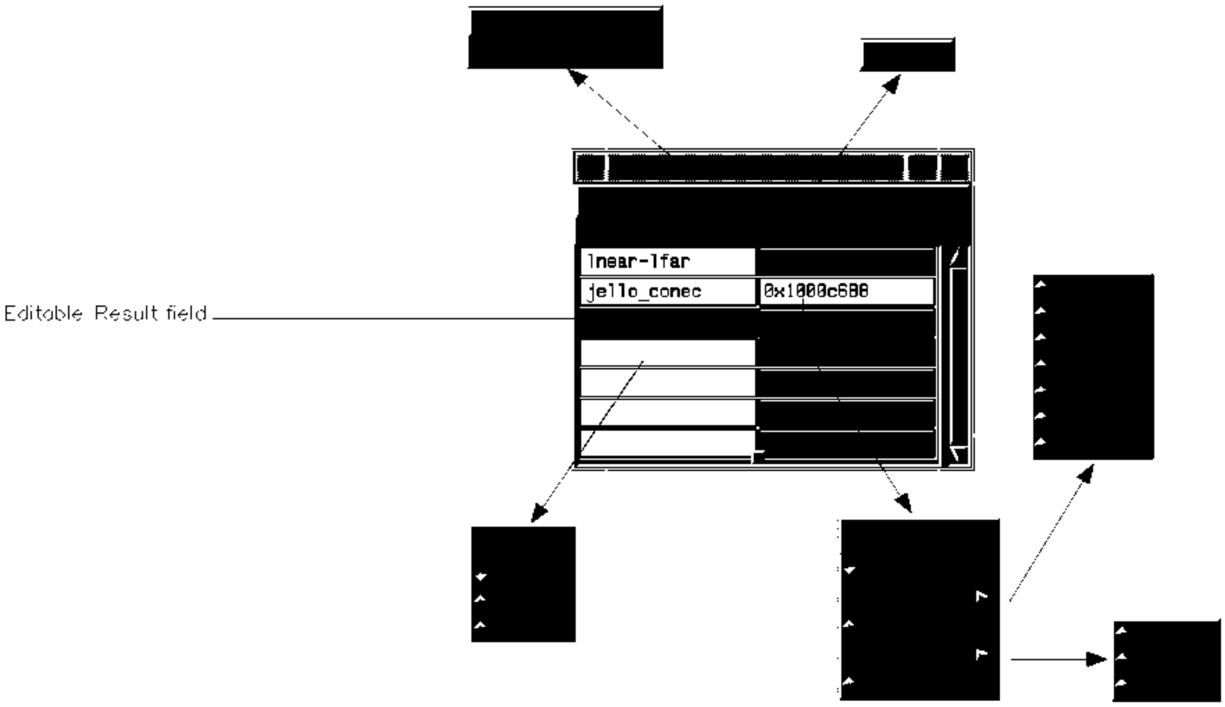


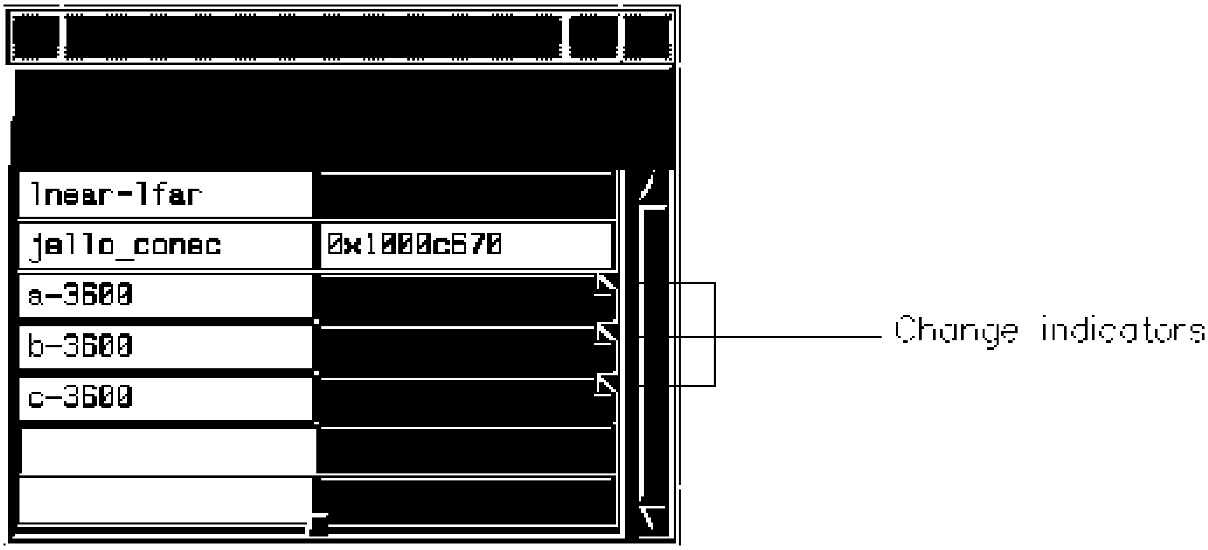
Figure 14–3 Expression View with Major Menus Displayed

If you want to change the type of result information displayed in the right column, hold down the right mouse button over the right column. This displays the Format menu. You can see the value as decimal, unsigned, octal, hexadecimal, float, characters, or as a string. You can also display the type, address (in decimal, octal, or hex), or the size of the result in bits.

Caution: The CASEVision Debugger uses the target program's symbol table to determine the types of variables. Some variables in libraries, such as *errno* and *_environ*, are not fully described in the symbol table. As a result, the Debugger may not know their types. When the Debugger evaluates such a variable, it assumes that the variable is a fullword integer. This gives the correct value for fullword integers or pointers, but the wrong value for non–fullword integers and for floating–point values.

To see the value correctly of a variable of unknown type, you can cast the address of the variable to a pointer to the correct type, using C type–cast syntax. For example, the global variable *_environ* should be of type `char**`. You can see its value by evaluating **(char***)&_environ*.

After you display the current value of the expression, you may find it useful to leave the window open so that you can trace the expression as it changes value from trap to trap. Like the other views involved with variables, Expression View has variable change indicators for value fields that let you see previous values, as shown in **Figure 14–4** (see **"Viewing Variable Changes"**).



1near-1far	
jello_conec	0x1000c670
a-3600	
b-3600	
c-3600	

Figure 14–4 Change Indicators in Expression View

Another useful technique is to save your expressions to a file for later reuse. Expressions are saved by choosing "Save Expressions..." from the Config menu and retrieved by selecting "Load Expressions...".

Assigning Values to Variables

To assign a value to a variable, click the left column and enter the variable name. The current value will appear in the right column. If this *Result* field is editable (highlighted), you can click it and enter a new value or legal expression. Pressing **`<Enter>`** performs the assignment. You can assign to any expression that evaluates to a legal *lvalue* (in C). The C operator = is not valid in Expression View. The valid expression operations are shown in the following paragraphs.

Evaluating Expressions in C

The valid C expressions are shown in **Table 14–1** .

Table 14–1 Valid C Operations

Operation	Symbol
Arithmetic (unary)	+ – ++ –– (increment and decrement do not have side effects)
Arithmetic (binary)	+ – * / %
Logical	&& || !
Relational	< > <= >= == !=
Bit	& | ^ << >> ~
Dereference	*
Address	&
Array indexing	[ ]
Conditional	? :
Member extraction	. –> (these operations are interchangeable)

Sizeof
Type−cast
Function call

C Function Calls

Function calls can be evaluated in expressions, as long as enough actual parameters are supplied. Arguments are passed by value. Following the rules of C, each actual parameter is converted to a value of the same type as the formal parameter, before the call. If the types of the formal parameters are unknown, integral arguments are widened to full words, and floating−point arguments are converted to doubles.

Functions may return pointers, scalar values, unions, or structs. Note that if the function returns a pointer into its stack frame (rarely a good programming practice), the value pointed to will be meaningless, since the temporary stack frame is destroyed immediately after the call is completed.

Function calls may be nested. For example, if the user's program contains a successor function *succ*, the Debugger will evaluate the expression *succ(succ(succ(3)))* to 6.

Evaluating Expressions in C++

C++ expressions may contain any of the C operations. You can use the word *this* to explicitly reference data members of an object in a member function. When stopped in a member function, the scope for *this* is searched automatically for data members. Names may be used in either mangled or demangled form. Names qualified by class name are supported (for example, *Symbol::a*).

If you wish to look at a static member variable for a c++ class, you need not specify the variable with the class qualifier if you are within the context of the class. For example, you would specify **`myclass::myvariable`** for the static variable *myvariable* outside of class *myclass* and **`myvariable`** inside *myclass*.

Limitations

Constructors may be called from Expression View, just like any other member function. To call a constructor, you must pass in a first argument that will point to the object to be created. C++ function calls have the same possibilities of side effects as C functions.

Evaluating Expressions in Fortran

Fortran expressions may contain any of the arithmetic, relational, or logical operators. Relational and logical operator keywords may be spelled in upper case, lower case, or mixed case.

The usual forms of Fortran constants, including complex constants, may be used in expressions. String constants and string operations, however, are not supported. The operators in **Table 14−2** are supported on data of integral, real, and complex types.

Table 14−2 Valid Fortran Operations

Operation	Symbol
Arithmetic (unary)	− +
Arithmetic (binary)	− + * / **
Logical	.NOT. .AND. .OR. .XOR. .EQV .NEQV.
Relational	.GT. .GE. .LT. .LE. .EQ. .NE.
Array indexing	( )
Intrinsic function calls (except string intrinsics)	
Function subroutine calls	

Fortran Variables

Names of Fortran variables, functions, parameters, arrays, pointers, and arguments are all supported in expressions, as are names in common blocks and equivalence statements. Names may be spelled in upper case, lower case, or mixed case.

Fortran Function Calls

The Debugger evaluates function calls the same way that compiled code does. If an argument can be passed by reference, it is; otherwise, a temporary expression is allocated and passed by reference. Following the rules of Fortran, actual arguments are not converted to match the types of formal arguments. Side effects can be caused by Fortran function calls. A useful technique to protect the value of a parameter from being modified by a function subroutine is to pass an expression such as *(parameter + 0)* instead of just the parameter name. This causes a reference to a temporary expression to be passed to the function rather than a reference to the parameter itself; the value is the same.

Evaluate Expression from the Data Menu

Figure 14–5 Data Menu

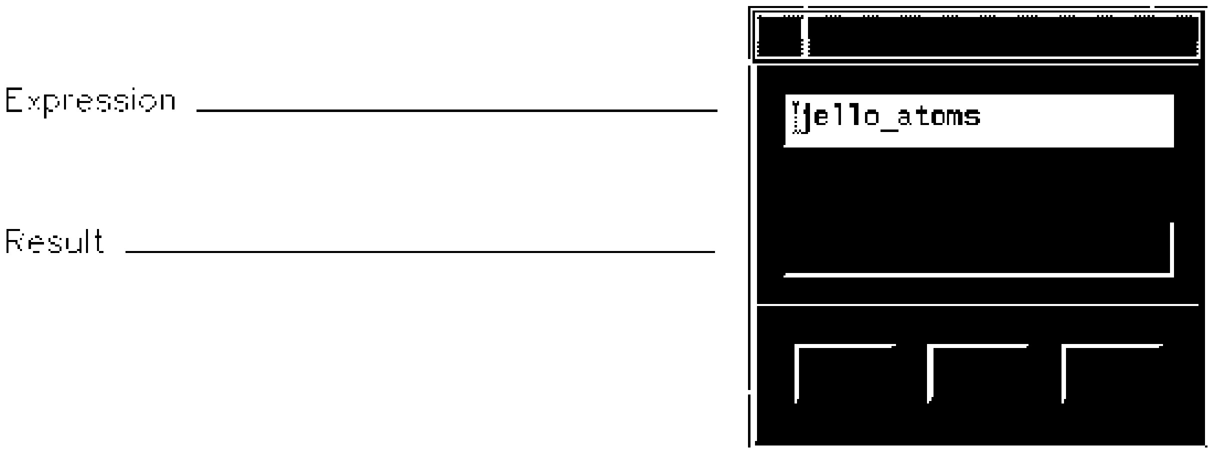
There are two methods for quickly evaluating expressions from the Data menu (see **Figure 14–5**):

- Evaluate Expression dialog box.
- one–shot command line display

The choice is determined by the *useTtyForExpressions* resource setting in your *.Xdefaults* file. Before choosing "Evaluate Expression..." from the Data menu when the process is stopped, you must first select the expression to be evaluated from any open window.

Evaluate Expression Dialog Box

The default response when you click "Evaluate Expression..." is the opening of the Evaluate Expression dialog box (see **Figure 14–6**). This displays the selected expression and its current value. You can enter another expression in the *Expression* field and click *Apply* to display its evaluation results in the *Value* field.



jello_atoms

Figure 14–6 Evaluate Expression Dialog Box

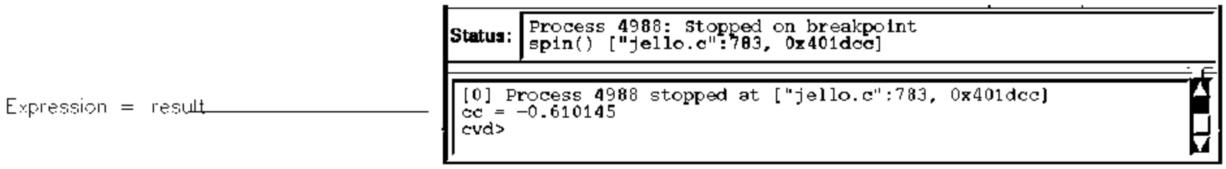
Unlike Expression View, the *Value* field cannot be modified to alter the value of the expression. The *Value* field is not automatically updated when the program is stopped. You must press *Apply* each time to examine the change in the expression value.

One−shot Command Line Display

You can initialize the Debugger for fast one−shot expression evaluation without using the Evaluate Expression dialog box. Set the following resource in your *.Xdefaults* file before initiating the debugging session:

```
*useTtyForExpressions: True
```

Selecting an expression and clicking "Evaluate Expression..." displays the expression *cc* and its result on the command line in Main View as shown in **Figure 14−7** .



```
Process 4988: Stopped on breakpoint
spin() ["jello.c":783, 0x401dcc]
```

```
[0] Process 4988 stopped at ["jello.c":783, 0x401dcc]
cc = −0.610145
cvd>
```

Figure 14−7 Expression Evaluation Displayed on the Command Line

Examining Variable Values

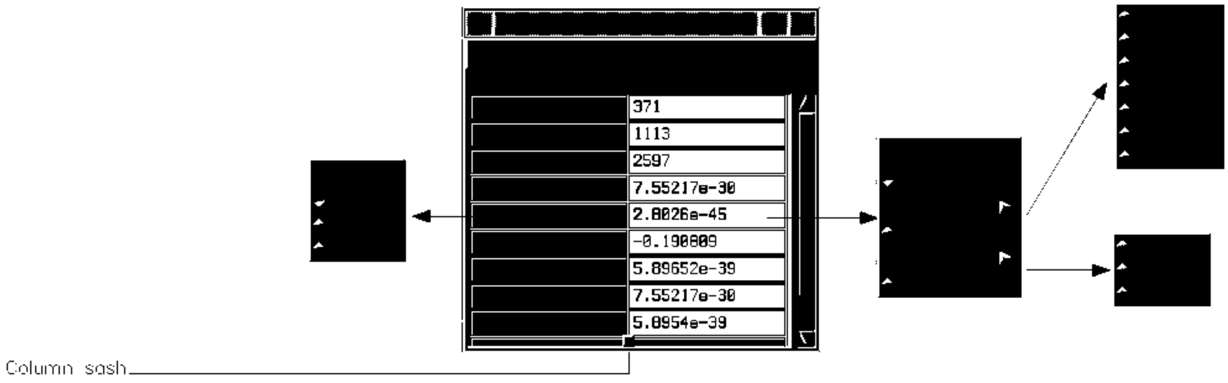
The Variable Browser lets you view and change the values of local variables and arguments at a specific point in a process. In addition to providing the values, Variable Browser is useful for getting a quick list of the local variables in a scope without having to search for their names. A sample Variable Browser window with the Language and Format menus displayed is shown in **Figure 14−8** . (For a tutorial example of the Variable Browser see **"Examining Data"** .)

Typically, you inspect variable values
- at a stop trap
- at a frame in a call stack
- as you step through a process

Note: A useful technique is to set a trap at the entry to a function and inspect the values of the variables there. Some variables may be in an uninitialized state at that point. You can then step through the function and make sure that no uninitialized variables are used inadvertently.

Entering Variable Values

The Variable Browser lets you change the values of variables in the window. You simply enter the new value in the result column and press `<Enter>`. Thus, you can force new values into the process and see their effect.



```
371
1113
2597
7.55217e-30
2.8026e-45
-0.190809
5.89652e-39
7.55217e-30
5.8954e-39
```

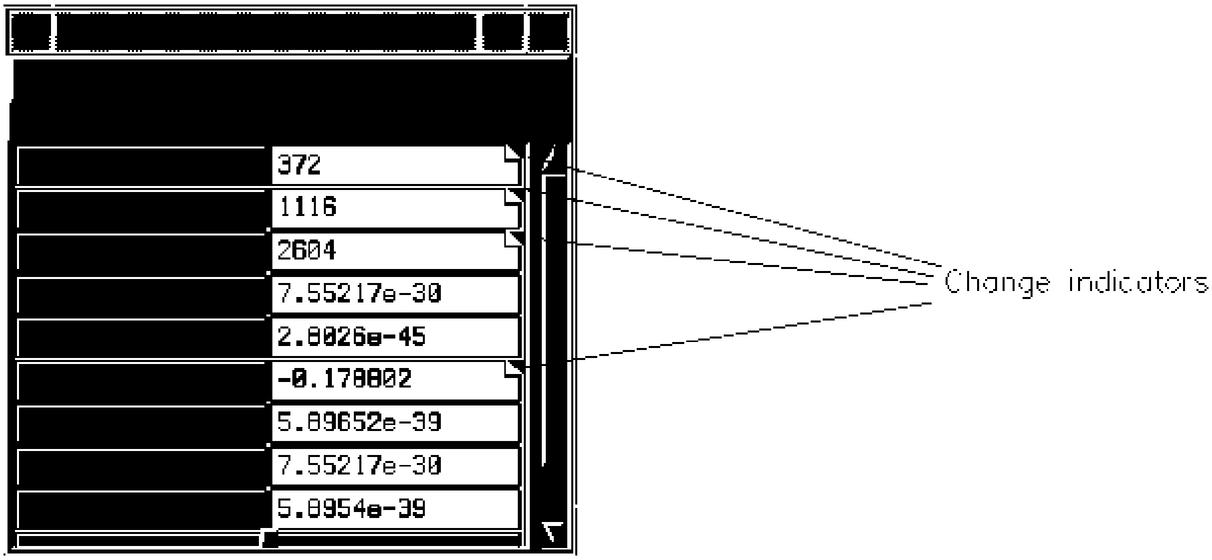
Figure 14–8 Variable Browser with Language and Format Menus

Changing Variable Column Widths

The Variable Browser has a sash between columns that lets you adjust the relative widths of the Variable and Result columns (see **Figure 14–8**). For example, you may wish to adjust for short variable names and long result values.

Viewing Variable Changes

The Debugger views that are involved with variables (that is, the Variable Browser and Expression View) have indicators that show when the variable has changed since the last breakpoint. If you click the indicator, you can view the previous value. The variable change indicators for a Variable Browser window are shown in **Figure 14–9** .



```
372
1116
2604
7.55217e-30
2.8026e-45
-0.178802
5.89652e-39
7.55217e-30
5.8954e-39
```

Figure 14–9 Typical Variable Change Indicators

Examining Array Variables

To examine data in an array variable, select Array Browser from the Views menu at a point in the process where the variable is present. Array Browser lets you view elements in a multi–dimensional array, presented in a spreadsheet and graphically, if desired. (For a tutorial example of the Array Browser, see **"Examining Data"** .)

Array Browser provides a spreadsheet that can display two dimensions of the array data. Since the array may be arbitrarily large, the spreadsheet acts as a window into a rectangular region of the array. It lets you select a cell in the spreadsheet, referred to as the *current element,* and provides the identifier and an editable field for modifying the current element's data. There are scroll bars for moving around large arrays.

Array Browser also lets you display data graphically—as a surface, bar chart, lines, or points. The graphical display is particularly useful for spotting trends in large quantities of data. If you click on a portion of the graph, the corresponding data is displayed in the spreadsheet.

Figure 14–10 shows a typical Array Browser window. The configuration of the window depends on whether Examiner Viewer is selected. With the graphical display enabled, there are four parts to the window:

- array specification area
- subscript control area
- graphical display area
- spreadsheet area

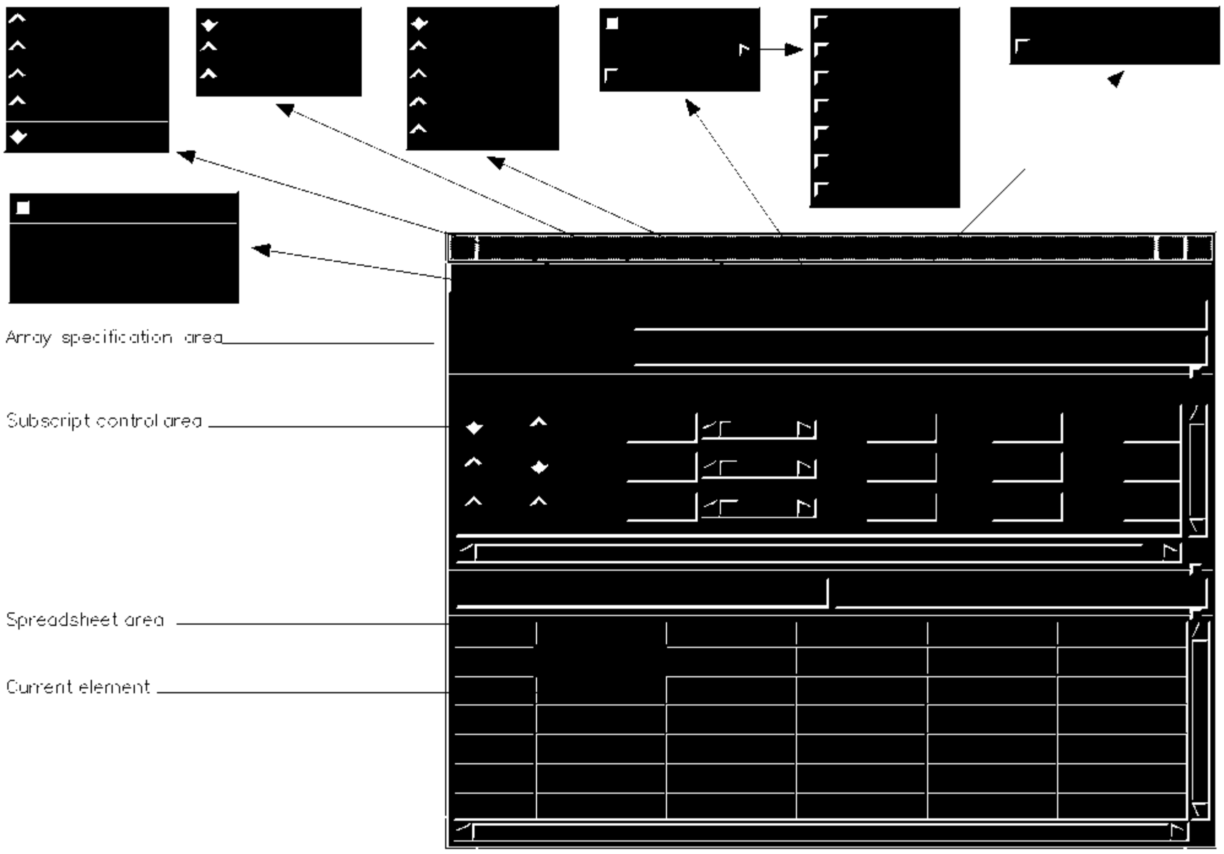


Figure 14–10 Array Browser with Display Menu Options

Array Specification Area

The *array specification area* lets you specify the variable and its dimensions. It consists of these fields:

Array:

lets you enter the name of the array variable. This entry is language–dependent. For Fortran, the expression may be an array or a dummy array variable name. If the last dimension of the array is unspecified (*), a subscript value of 1 is assumed initially.

For C and C++, the entry may be an array, a pointer, or an array pointer. If pointers are used, the expression is treated as though it were a single element, in which case you need to use the subscript controls to see more than the first element.

Indexing Expression:

is the expression used to view an element in the array. It is filled in automatically when you specify an array to view.

The expression supplied is language–specific. It represents the indexing expression used in the language to access a particular element. The subscripts are specified by special indexing variables ($i, $j, $k, and so forth) that can be manipulated in the subscript controls area. For example, the indexing expression for a three–dimensional array in C or C++ appears in the form:

```
array[$i][$j][$k]
```

and in Fortran would be in the format:

```
array($i,$j,$k)
```

Note that the Array Browser cannot determine from a user–supplied indexing expression which portion of the array to display. You must designate which dimensions of the array correspond to the rows and columns in the spreadsheet. When the indexing expression is generated by Array Browser, the correspondence is known.

The indexing expression can, however, be used to access desired data. For instance, to access a member variable for an element in an array of structures, you can modify the indexing expression as follows:

```
array[$i][$j][$k].member
```

Some Fortran codes use one array to index into another array as in the following:

```
data(indexarr($i))
```

For this situation, specify the indexing array *indexarr* as the array in the *Array:* field and then modify the indexing expression with the proper dimensions.

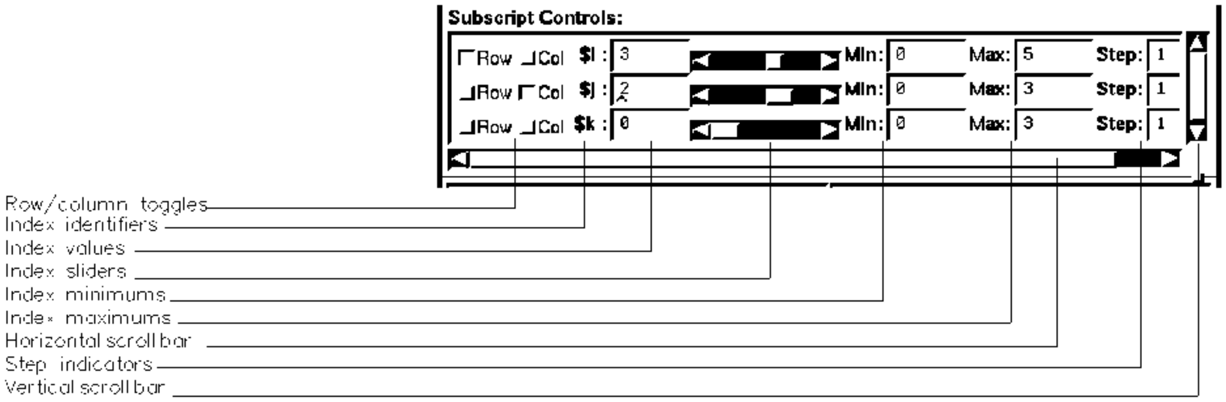
If desired, you can modify the indexing expression dimensions to get a different presentation in the spreadsheet area. For example, you might want to change a one–dimensional array `myArray[x]` into a two–dimensional presentation. To do this, you would change the indexing expression `myArray [$i]` into an expression that incorporates $i and $j like

```
myArray [$i*6 + $j]
```

In this example, we are using $i as the row index and $j as a column index. The above specification treats the array as six columns wide by some arbitrary number of rows. The subscript $j should be set in the subscript controls so it varies from 0 to 5 and $i should be set to vary from 0 to the desired number of rows. If the maximum for $j is greater than 5, there will be overlap in indexing of the elements.

Subscript Control Area

The *subscript control area* serves two functions: (1) it lets you control which elements in the variable are to be displayed and (2) it lets you shift the current element. The number of dimensions in the array governs the number of controls that are displayed. A close–up view of the subscript controls area appears in **Figure 14–11** .



Subscript Controls:

☑ Row ☐ Col $i : 3 Min: 0 Max: 5 Step: 1

☐ Row ☑ Col $j : 2 Min: 0 Max: 3 Step: 1

☐ Row ☐ Col $k : 0 Min: 0 Max: 3 Step: 1

Figure 14–11 Subscript Control Area in Array Browser

The subscript control area provides these features

row/column toggles

control whether an index variable represents rows or columns (or neither) in the spreadsheet area. You are not limited by the number of vectors in an array, but you can only view a two–dimensional orthogonal slice of the array at a time.

index identifiers

indicate to which subscript the controls in the row refer.

index values

show the value of the subscript for the element currently in the focus cell. You can enter a different value if you wish.

index sliders

let you move the focus cell along the particular vector.

index minimums

identify the beginning visible element in a vector.

index maximums

identify the last visible element in a vector. If you have an unspecified array, you can use this field to specify the last element in the vector to be displayed in the spreadsheet.

step indicators

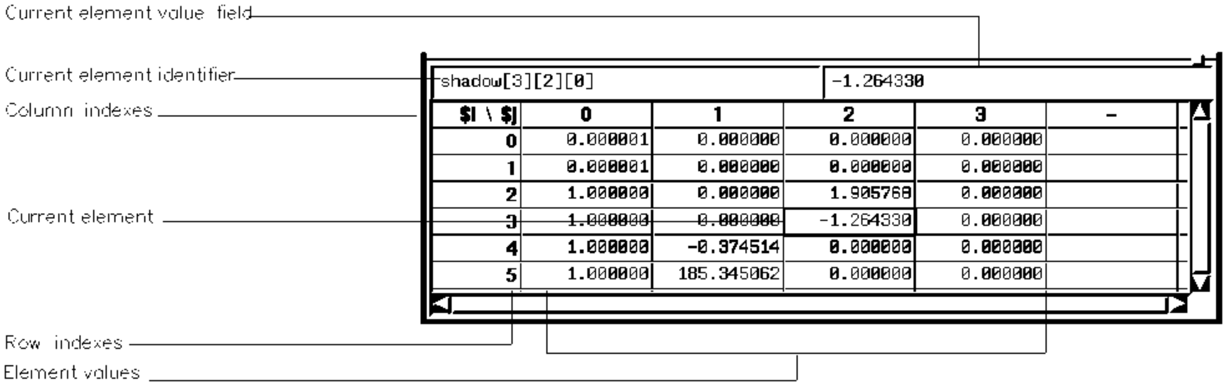
specifies the increment between adjacent elements in a vector to be displayed. A value of 1 displays consecutive data. Specifying some *n* greater than 1 lets you display every *n*th element in a vector.

control area scroll bars

let you expose hidden portions of the subscript control area if your window is not large enough for viewing all of the controls.

Spreadsheet Area

The *spreadsheet area* is where numeric data is displayed. It can show two dimensions at a time (indicated in the upper left corner of the matrix). The column indexes run along the top of the matrix and the row indexes are displayed along the left column. The spreadsheet area has scroll bars for viewing data elements not currently visible in the viewing area. **Figure 14–12** shows a close–up of the spreadsheet area.



shadow[3][2][0]

-1.264330

$I \ $J	0	1	2	3	-
0	0.000001	0.000000	0.000000	0.000000	
1	0.000001	0.000000	0.000000	0.000000	
2	1.000000	0.000000	1.905768	0.000000	
3	1.000000	0.000000	-1.264330	0.000000	
4	1.000000	-0.374514	0.000000	0.000000	
5	1.000000	185.345062	0.000000	0.000000	

Figure 14–12 Array Browser Spreadsheet Area

The current element is highlighted by a colored rectangle in the spreadsheet area. Its corresponding expression is shown in the current element identifier field, and the value is shown in the current element value field.

Spreadsheet Display Menus

Array Browser comes with two menus for controlling the display of data in the spreadsheet area:

- Spreadsheet menu
- Format menu

Spreadsheet Menu

Figure 14–13 Spreadsheet Menu

The Spreadsheet menu (see **Figure 14–13**) lets you change the appearance of data in the spreadsheet area. It provides these selections:

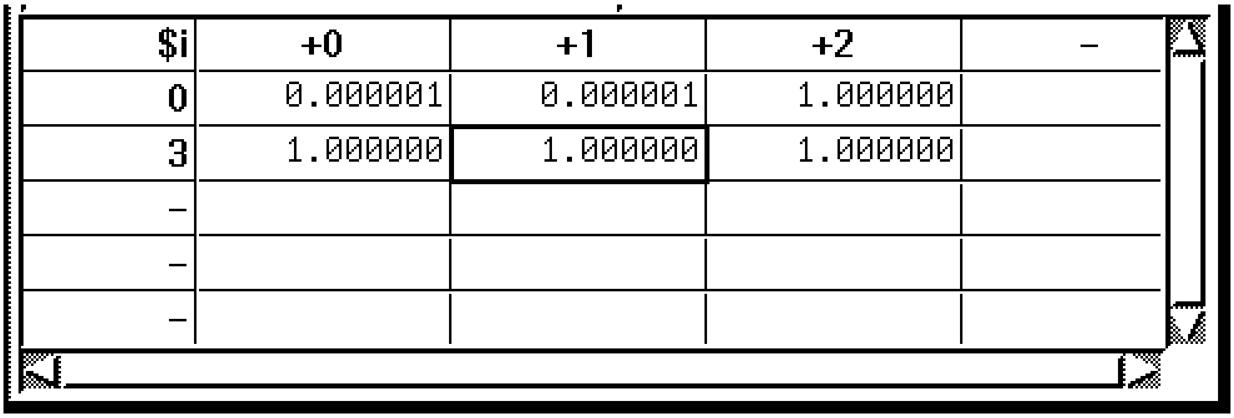
"Column Width..."

lets you specify the width of the spreadsheet cells in terms of characters. For instance, a value of 12 indicates that 12 characters including punctuation and digits will be viewable.

"Wrapped Display"

lets you display a single dimension of an array wrapped around the entire spreadsheet area. The index value for an element is determined by adding the appropriate row index and column index values.

Figure 14–14 shows an example of a wrapped array. There is only one index *$i*. The current cell is element 4 in the array (by adding 3 and +1).



$i	+0	+1	+2	–
0	0.000001	0.000001	1.000000	
3	1.000000	1.000000	1.000000	
–				
–				
–				

Figure 14–14 Example of Wrapped Array

Format Menu

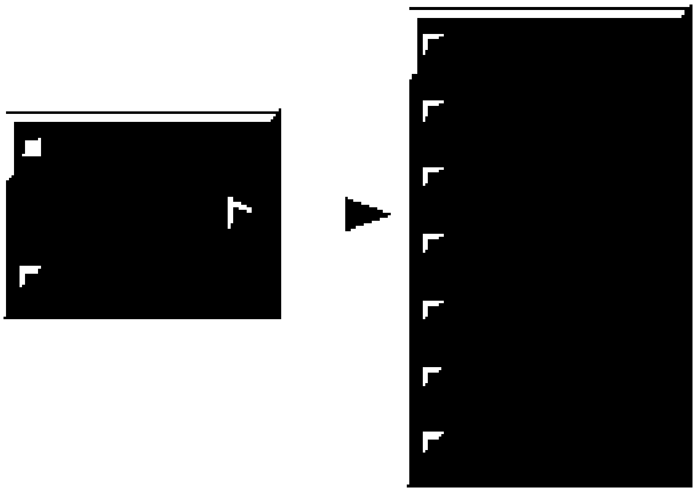


Figure 14–15 Format Menu with Value Submenu

The Format menu (see **Figure 14–5**) displays a separate menu that you lets you display the elements in the default format, as formatted values (decimal, unsigned, octal, and so forth) or as their data types.

Graphical Display Area

The Array Browser graphical display area presents the array data in a 3–D graph in one of the following forms:

- surface (polyhedron)
- bar chart
- points
- multiple lines (array vectors)

The graphical display is useful when you are looking for incorrect values or are trying to spot trends in large quantities of data. Through color and shape, you can make anomalies in your data stand out significantly.

The graphical display is based on Examiner Viewer, a component of IRIS Inventor™. Inventor is a powerful tool for developing 3–D graphical applications. In addition to the basic display formats, Examiner Viewer provides a number of facilities for manipulating the display. Since you only need a subset of these for data inspection, this document only describes the basic functionality of Examiner Viewer. For more information on Examiner Viewer's advanced functions, refer to the special help cards in the Examiner Viewer menu. To learn more about the full range of Inventor capabilities, contact your Silicon Graphics representative.

Array Graphs

The graphical display area plots the array data by scaling the *i* and *j* indices proportionately along the x– and y–axes, respectively. The scaling of *x* and *y* is adjusted by a factor equal to the largest value in the matrix divided by the scale requested in the Scale menu. This produces a graph that fits all values inside the graphical display area (which can be adjusted later as needed). The actual values of the elements in the matrix are plotted in the z–dimension without scaling. **Figure 14–16** is an example of a single, plotted point corresponding to the array element, array (3,1) = 1.25.

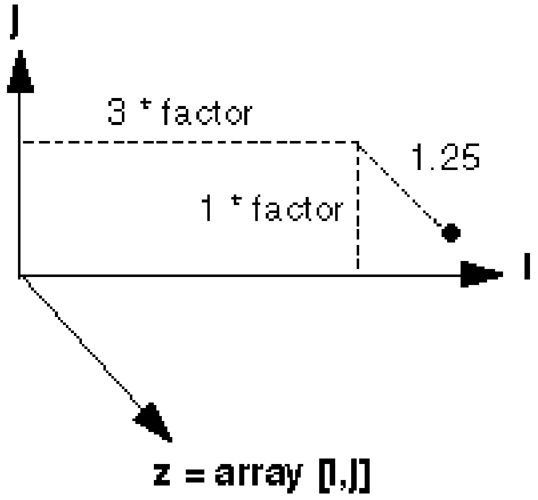


Figure 14–16 Graphical Display Plotting Example

You can manipulate the graphical display through the Render menu, Color menu, and Scale menu accessible from the menu bar at the top of the window; the Examiner Viewer menu, displayed by holding down the right mouse button; and the controls around the display area.

Render Menu

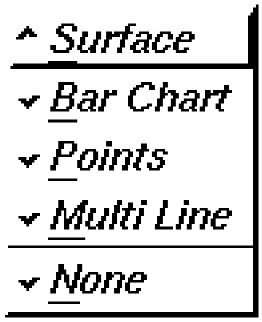


Figure 14–17 Render Menu

You select the graphical display mode through the Render menu. The Render menu (see **Figure 14–17**) has the following items:

"Surface"	exhibits the data as a solid using the data values as vertices in a polyhedron.
"Bar Chart"	presents the data values as 3–D bar charts.
"Points"	simply plots the data values in 3–D space.
"Multi Line"	plots and connects the data values in each row.
"None"	lets you display Array Browser with no graphical display, in effect turning off graphical display mode.

Three of the four graphic modes are illustrated in **Figure 14–20** , applied to the same data, a 30 by 30 array with values = 0 along the diagonals. "Points" mode is not shown because it is hard to see. In general, you'll find only one or two of the modes appropriate for the same data. "Surface" mode is appropriate for really large matrices, particularly where you may be applying isograms to the data. "Bar Chart" mode lends itself to smaller matrices where individual values are largely independent of their neighbors. "Multi Line" mode is useful when values along a row or column are more closely related than the values in the neighboring rows or columns or where you are looking for linear trends. "Points" mode is mainly useful for really small matrices where you can zoom in close.

Color Menu

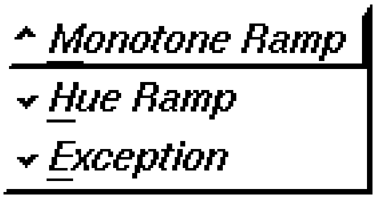
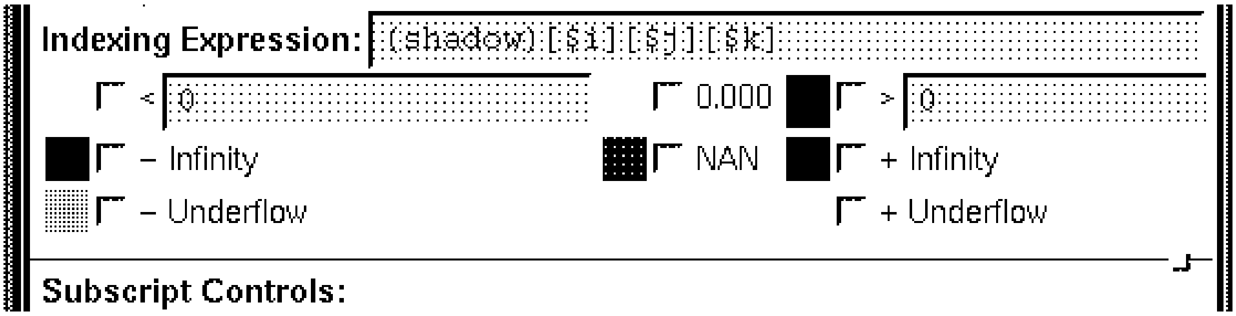


Figure 14–18 Color Menu

The Color menu (see **Figure 14–18**) provides these options:

"Monotone Ramp"	displays the data values in a single tone, with lower numbers being darker and higher values lighter in tone.
"Hue Ramp"	displays the data values in a spectrum of colors ranging from blue (lowest values) through green, yellow, orange, and red (highest values).
"Exception"	lets you flag certain conditions by color, usually for the purpose of spotting bad data. When you select "Exception", the controls shown in **Figure 14–19** appear in the window.



Indexing Expression: (shadow)[$i][$j][$k]

< 0

> 0

Figure 14–19 Color Exception Portion of Array Browser Window

Thus, you can highlight data values less than or greater than specified values, values of plus or minus infinity, values of plus or minus underflow, zero values, and NaN (not a number) values.

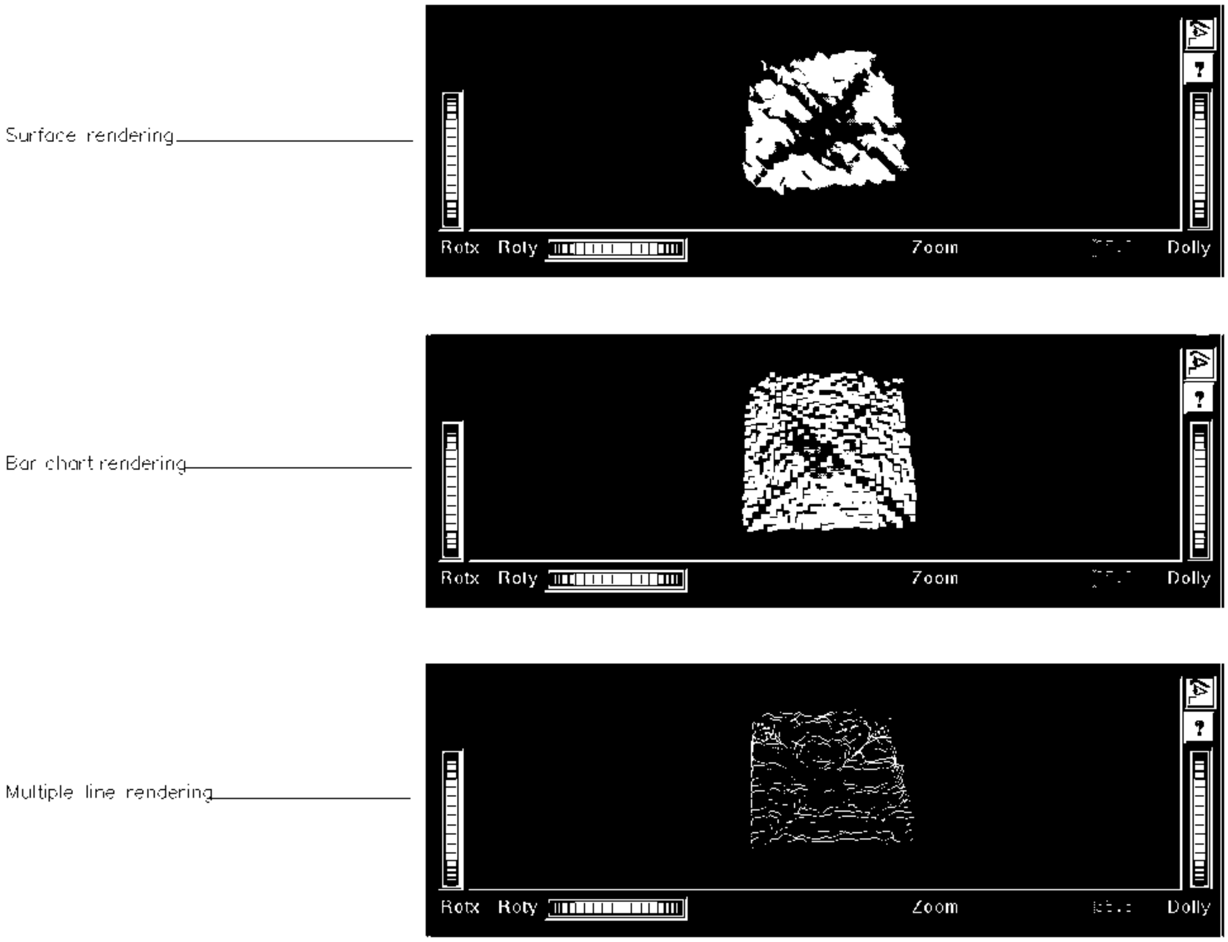


Figure 14–20 Array Browser Graphic Modes

Scale Menu

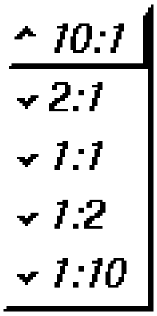


Figure 14–21 Scale Menu

The Scale menu (see **Figure 14–21**) provides options for changing the ratio of the *z*–dimension, which represents the value of the element. The number on the left represents the value of the *x*– and *y* –dimensions (which are always the same as each other); the number on the right is the *z* dimension.

Figure 14−21 Scale Menu

The Scale menu (see **Figure 14−21**) provides options for changing the ratio of the *z*−dimension, which represents the value of the element. The number on the left represents the value of the *x*− and *y* −dimensions (which are always the same as each other); the number on the right is the *z* dimension. Manipulating the z−dimension affects the ease of spotting differences in values. If your data is scattered over a narrow range of values, you may wish to heighten the graph by selecting "10:1" as your scale; this exaggerates the values in the *z*−dimension. If your data is in a wide range, selecting "1:2" or "1:10" as the scale will minimize the differences, flattening the graph.

Examiner Viewer Controls

The graphical display uses controls and menus from Examiner Viewer. Examiner Viewer is based on a camera metaphor and borrows terms from the film industry, such as *zoom* and *dolly*, in naming its controls. The graphical display area of the window is shown in **Figure 14−22** , with its main controls and menus. Note that the buttons on the upper right side of the graphical display area may not be visible if the area is too small; you can expose them by moving either the upper or lower sash to enlarge the display area.

Examiner Viewer provides these controls for viewing the graph. The right side buttons are:

view mode

toggles between a view−only mode (closed eye) and manipulation mode (open eye).
In view−only mode, the cursor appears as an arrow and the graph cannot be moved. Clicking on a portion of the graph selects the corresponding array element in the spreadsheet.
In manipulation mode, the cursor appears as a hand and you can move the graph. Dragging the graph with the left mouse button down, moves the graph in any direction as if it were in a trackball; a quick movement will spin the graph. Dragging the graph with the left mouse button and the ctrl button rolls (rotates) the graph in the plane of the screen. Dragging the graph with the middle mouse button moves it without changing the viewing angle.
If you drag the graph with both the left and middle mouse buttons down, the graph will appear to move into or out of the window (this is the same as the *dolly thumbwheel* which is described in this section).

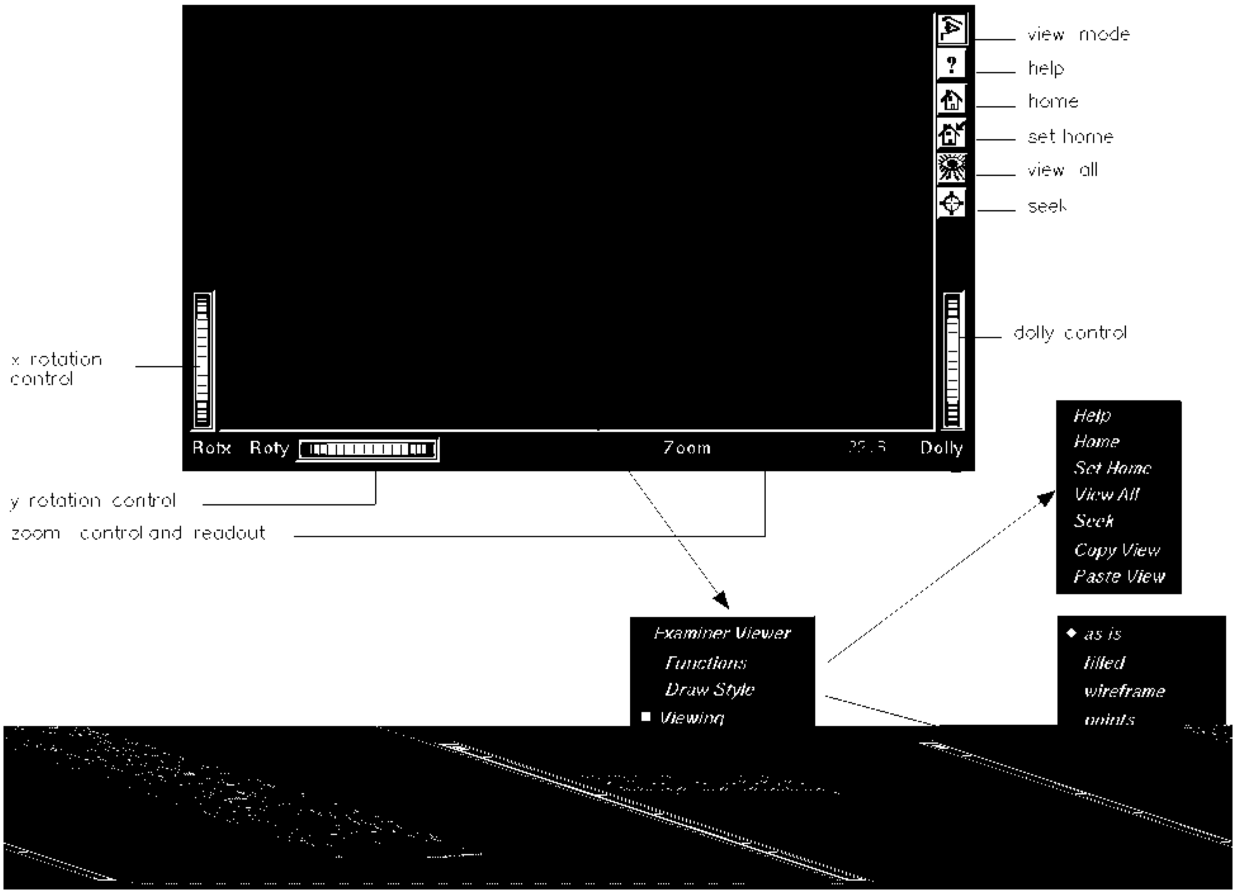


Figure 14–22 Examiner Viewer with Controls and Menus

help	runs a special help system containing Inventor Viewer information.
home	repositions the graph in its original viewing position.
set home	changes the home (original viewing) position for subsequent use of the *home* button.
view all	repositions the display area so that the entire graph is visible.
seek	provides a special cursor that lets you reposition the graph in the center of the display area or lets you center the view on a point you select with the cursor. See "Seek to point <or object>" in Preferences dialog box.

These four controls let you move the graphic display:

x rotation thumbwheel	rotates the graph around its x−axis.
y rotation thumbwheel	rotates the graph around its y−axis.
zoom slider and readout	changes the size of the graph by scaling it.
dolly thumbwheel	changes the size of the graph and adjusts the angles to maintain perspective. The dolly control simulates moving the viewing camera back and forth with respect to the graph.

Examiner Viewer Menu

Figure 14–23 Examiner Viewer Menu and Submenus

You access the Examiner Viewer menu (see **Figure 14–23**) by holding down the right mouse button in the graphical display area. The Examiner Viewer menu provides these selections:

"Functions"

displays a submenu with the selections "Help", "Home", "Set Home", "View All", and "Seek", which are the same as the right side button controls described in the previous section, and the "Copy View" and "Paste View" selections. These operate like standard copy and paste editing commands, enabling you to transfer graphs.

"Draw Style"

displays a submenu that controls how the graph is displayed. The selections "as is", "filled", "wireframe", and "points" control how the graph is displayed when it is static. These override the Render menu selections. The selections "move wireframe" and "move points" control how the graph is displayed while in motion. The selections "single", "double", and "interactive" refer to buffering techniques used in moving the graph. These affect the smoothness of the movement.

"Viewing"

is the same as the *view mode* button described in the previous section. When it is off, you can select points from the graph to display in the spreadsheet but cannot move the graph. When on, it lets you manipulate the graph.

"Decoration"

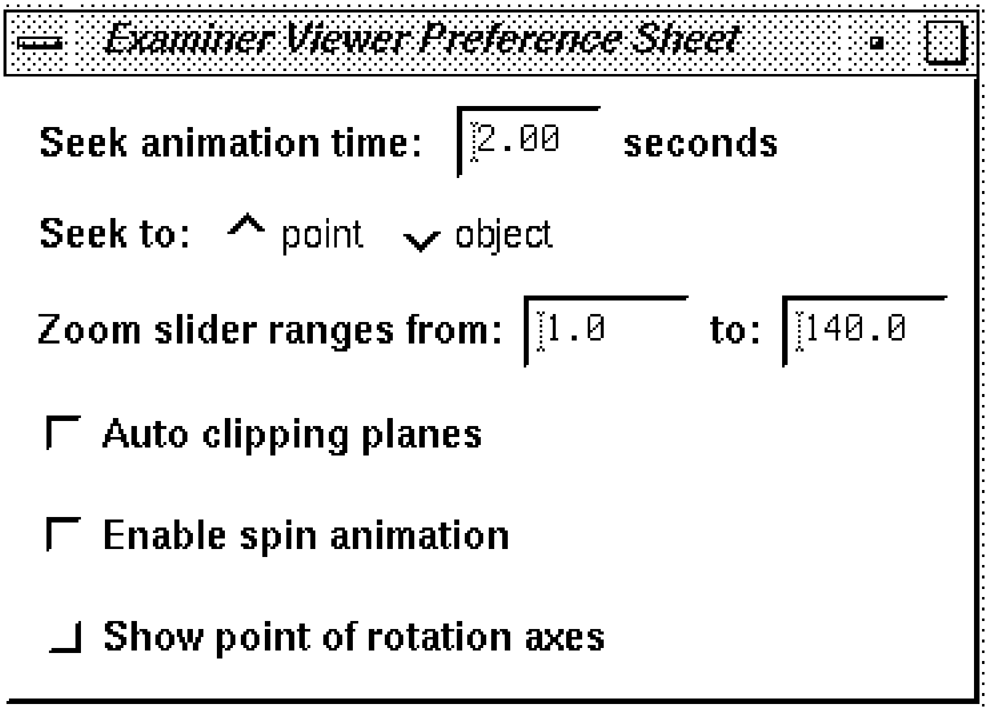
displays the right side buttons when it is on and hides them when it is off.

"Headlight"

controls the shadow effect on the graph. When it is on, the light appears to come from the camera.

"Preferences"

causes the Examiner Viewer Preferences dialog box to display.



Seek animation time: 2.00 seconds

Seek to: point object

Zoom slider ranges from: 1.0 to: 140.0

Figure 14–24 Examiner Viewer Preferences Dialog Box

"Seek animation time"

lets you specify the time it takes for the graph to be repositioned after you change the seek point. See

"Seek to point <or object>"

lets you change the view of the graph to its center (object) or to a point in the graph that you specify with the seek cursor.

"Zoom slider ranges from"

lets you change the Zoom range, that is, the distance that the object appears to be away from the front of the window.

"Auto clipping planes"

centers the graph in your view if enabled. If disabled, it provides controls for removing data from visibility at either end of the z−axis. This is useful if you wish to focus on data above or below a set value.

"Enable spin automation"

lets you spin the graph. You grab the graph with the mouse, move it quickly in the desired direction, and release the mouse button. The graph spins as if in a trackball.

"Show point of rotation axes"

displays a set of three axes. You can move the graph around the x and y axes using the thumbwheel controls described in the previous section. When this option is on, you can set the size of the axes in pixels.

Looking at Data Structures

Structure Browser lets you examine data structures and the relationships of the data within them. It displays complex data structures as separate graphical objects, using arrows to indicate relationships. A typical Structure Browser example is shown in **Figure 14−25** with the Config, Display, Node, and Format menus displayed. (For a tutorial example of the Structure Browser, see **"Examining Data"** .)

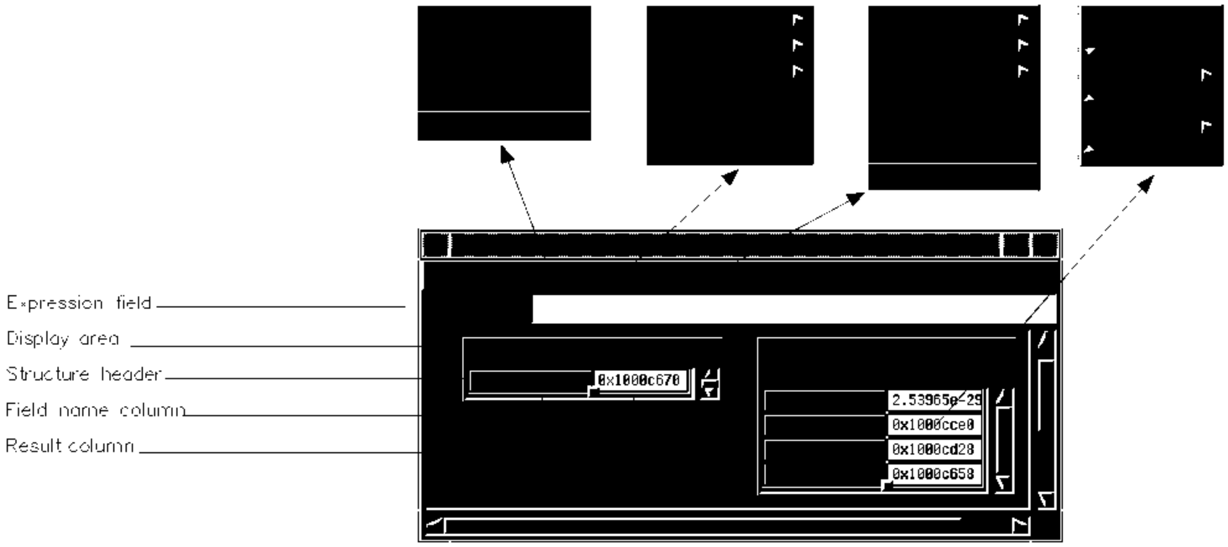


Figure 14−25 Structure Browser with the Config, Display, Node, and Format Menus

The structure name is entered in the *Expression* field. It then appears as an object or set of objects in the display area in the lower portion of the window. Each structure has a header identifying the structure, color coded by data type. Below the header are two columns: the left displays the field name and the right displays the field's value. If the structures to be displayed exceed the size of the Structure Browser window, scroll bars appear.

In addition to the Admin menu, Structure Browser provides four menus that are used to change the way the data displays. The menus are:

Config

for saving and reusing type−specific formats and expressions. You can also set preferences regarding how objects of a given types are to be displayed.

Display	provides operations for all objects in the display area.
Node	provides operations for selected objects in the display area only.
Format	lets you change or reformat a specific value in the result column. It is a pop−up menu that is accessed by holding down the right mouse button while the cursor is over the result column.

Using the Overview Window to Navigate

WorkShop provides the Overview window (from the "Show Overview" selection in the Display menu) as another way to navigate around the display area (see **Figure 14−26**).

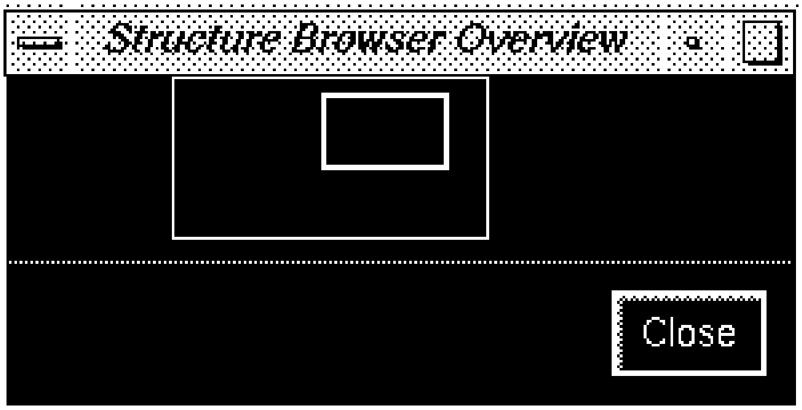


Figure 14−26 Structure Browser Overview Window

The Overview window is a reduced−scale view of the requested structures. The structures are represented by solid rectangles color− coded by data type. The relative position of the currently visible area is represented by a transparent rectangle. This rectangle can be dragged (using the left mouse button) to change the display of the Structure Browser. Clicking the left mouse button in an area of the Overview window repositions the currently visible area.

Entering Expressions

The Structure Browser accepts any valid expression. If the result type is simple, a structure displays showing the type and value. If the result type is a pointer, it is automatically dereferenced until a non−pointer type is reached. If the result type is a structure or union, an object is displayed showing the structures' fields and their values. After the expression is entered, the *Expression* field clears. The Structure Browser can display unrelated structures at the same time—you simply enter new structures using the *Expression* field.

The *Expression* field is also used to enter strings used in searches.

Working in the Structure Browser Display Area

Within the display area, you select objects by clicking in the node headers. Shift−clicks add the selected object to the current selection. You can drag selected objects using the middle mouse button.

Clicking the right button while the cursor is in the right column of an object displays the Format menu, used to change the display of a specific result (see **Figure 14−27**). You can set a default format or request that results be displayed by value, type, address, or size in bits.

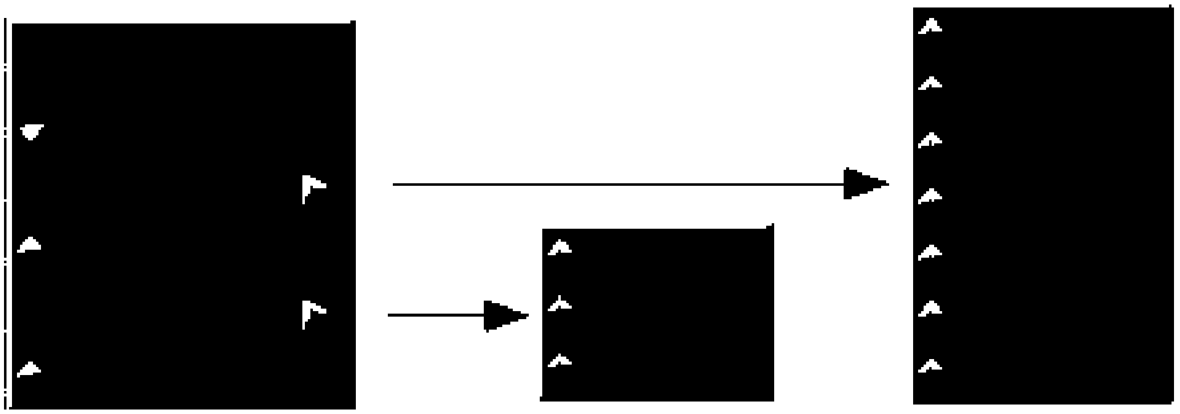


Figure 14–27 Structure Browser Format Menu

Holding down the right button in the header of a Structure Browser object brings up the Node pop–up menu, which is the same as the Node Menu in the menu bar. It is used to change the way selected objects are displayed. When you left–click in the header of an object, it turns on the resizer, which lets you change the size of the object. Left–clicking the handle resizes; middle–clicking moves it.

Graphical arrows show the pointer relationships among the displayed structures. If a pointer field is not visible in a structure, its arrow tail is displayed at the top or bottom of the scrolling area for fields. Otherwise, its tail is adjacent to its field.

Double–clicking a value field (right column) for a pointer dereferences it, so that the data structure it points to is displayed.

Double–clicking a member field (left column) puts the full expression for that member in the *Expression* field.

Structure Browser Display Menu

The Display menu controls the way structures appear in the display area of Structure Browser. The menu is shown in **Figure 14–28** .

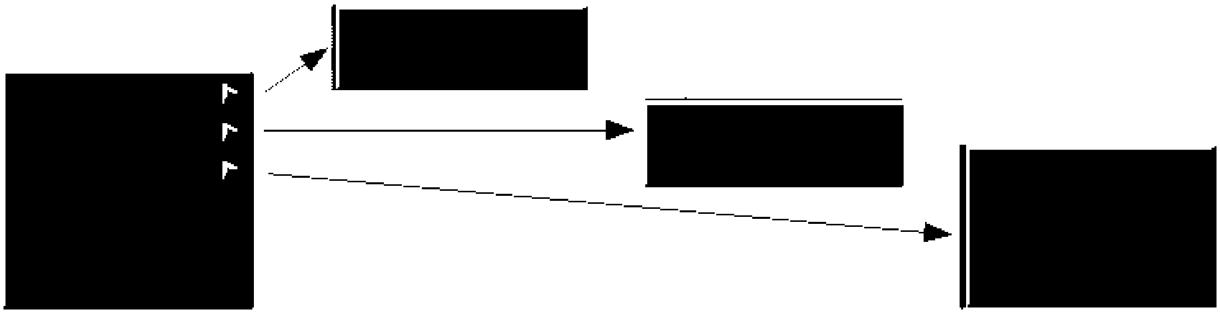


Figure 14–28 Structure Browser Display Menu

The Display menu provides the following selections:

"Display"

has two options:

"Expression" displays the structure of the expression entered in the *Expression* field.

"Selection" displays the structure based on the text from the current selection.

"Arrange"

"Tree" arranges them into a tree−type formation, that is, the hierarchy descends from left to right and child structures are shown as branches to the right of the parent. "Linked List" arranges them into a linked list formation, that is, horizontally.

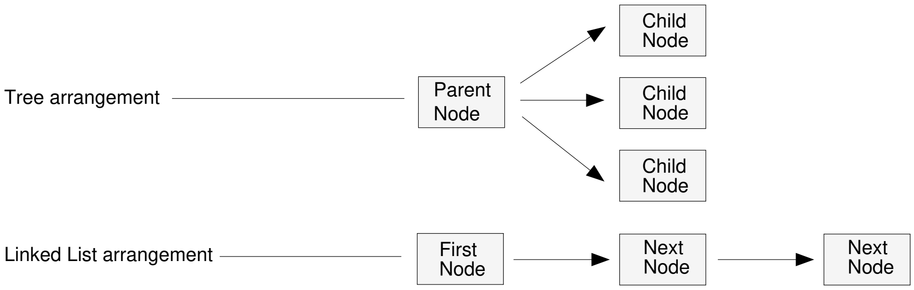


Figure 14−29 Tree and Linked List Arrangements of Structures

"Search"

lets you select structures containing the string specified in the *Expression* field. There are four options:
"Name" selects structures whose names contain the specified string.
"Type" selects structures whose types contain the specified string.
"Field Name" selects structures that have a field whose name contains the specified string.
"Value" selects structures that have a field value containing the specified string.

"Update"

explicitly updates the displayed structures. This happens automatically in the current Structure Browser when the process stops. This can be used in an inactive Structure Browser to update it. It can also be used to update the display after changes have been made in other Debugger views.

"Show Overview"

brings up the Overview window.

"Clear All"

clears all structures from the display area.

Node Menu

The Node menu is shown in **Figure 14−30** .

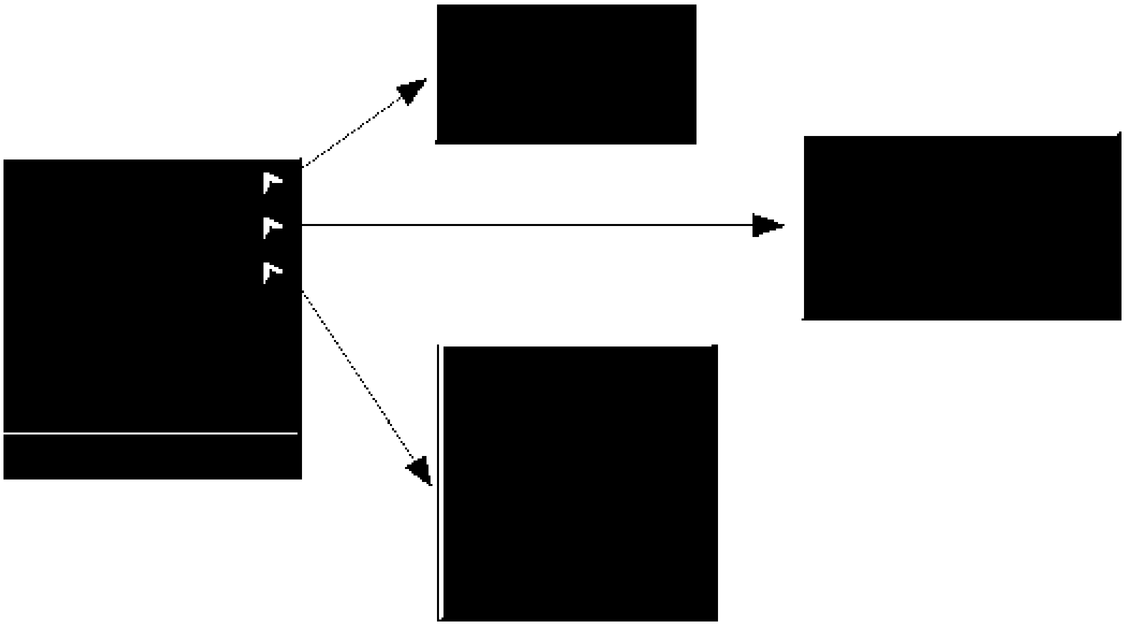


Figure 14−30 Structure Browser Node Menu

The Node menu consists of the following entries and applies to the currently selected objects:

"State"

controls the display of nodes. There are three options:
"Iconic" displays the node header only.
"Normal" uses the default chart display but hides those fields selected to be invisible.
"Detail" uses the default chart display and shows all fields.

"Geometry"

manages graphical objects in the display area. There are four options:
"Minimize" sets the vertical size of an object to the default minimum number of fields. The initial default is four fields but can be changed through either the "Formatting" selection from the Node menu or the "Preferences..." selection from the Config menu.
"Maximize" displays the object as large vertically as necessary to fit all of the fields.
"Raise" raises the selected object(s) to the top of the display.
"Lower" lowers the selected object(s) to the bottom of the display.

"Select"

lets you select objects in various ways. There are six options:
"Parents" selects all objects that have pointers pointing to a selected object.
"Children" selects all objects pointed to by any fields in a selected object.
"Ancestors" selects all objects pointed to a selected object or pointing to an object that has a descendant pointing to a selected object.
"Descendants" selects all objects pointed to by any fields in a selected object or pointed to by any children of a selected object.
"Family" selects all ancestors and descendants of a selected object.
"All" selects all objects.

"Formatting"

brings up the type formatting dialog for this type. See **"Formatting Fields"** .

"Dereference Ptrs"

dereferences any pointers in selected objects.

"Pattern Layout"

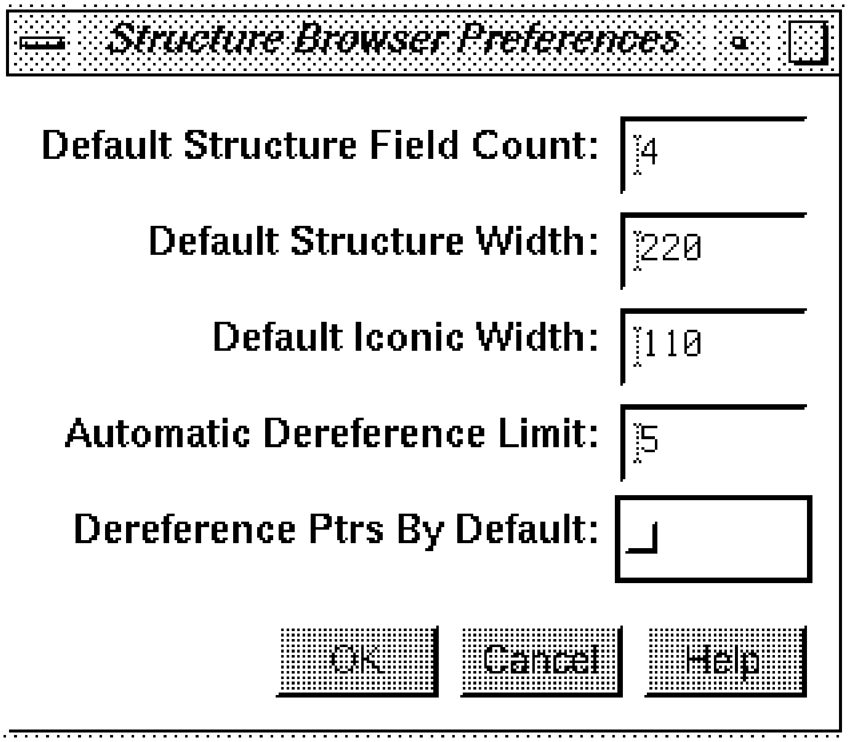
uses the spatial relationship between selected structures connected by pointers to position all similarly related structures in the same way.

"Remove"

removes the selected object from the display.

Formatting Fields

Each field in a data structure has certain display characteristics. These can be specified for all objects in the Structure Browser Preferences dialog box or for type−specific objects only in the Type Formatting dialog box. To display the Structure Browser Preferences dialog box, select "Preferences..." from the Config menu (see **Figure 14−31**).



Default Structure Field Count: 4

Default Structure Width: 220

Default Iconic Width: 110

Automatic Dereference Limit: 5

Dereference Ptrs By Default: ☐

Figure 14–31 Structure Browser Preferences Dialog Box

The Structure Browser Preferences dialog box has the following fields:

Default Structure Field Count
sets the number of fields to be displayed initially.

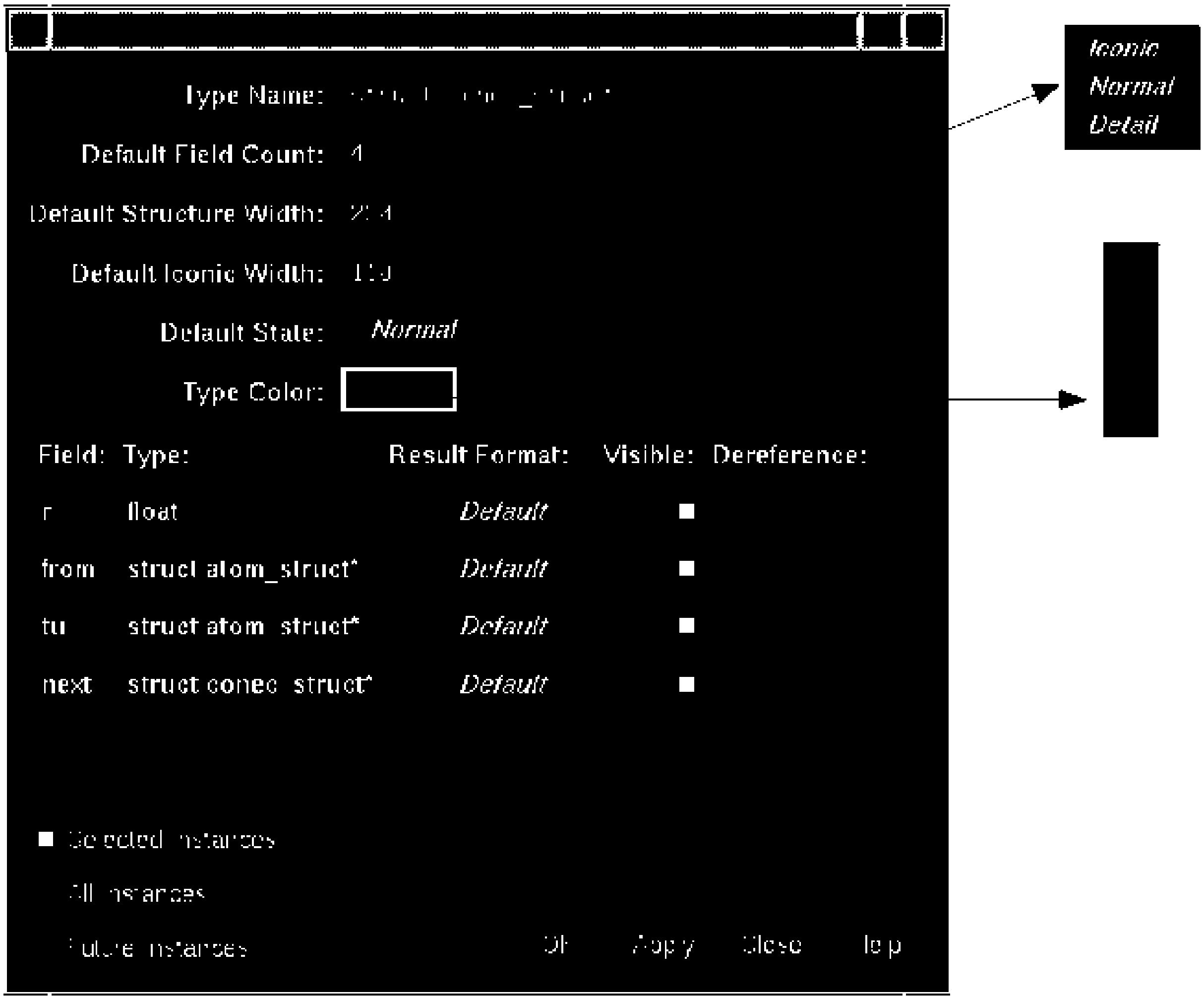
Default Structure Width
is the width in pixels of the object.

Default Iconic Width
is the width in pixels of the object when it is in iconic form.

Automatic Dereference Limit
sets a limit on the number of structures that are automatically dereferenced.

Dereference Ptrs By Default
lets you toggle automatic dereferencing on and off.

To bring up the Type Formatting dialog box, select the set of structures under consideration and select "Node Formatting" from the Node menu (see **Figure 14–32**).



Type Name: [illegible]

Default Field Count: 4

Default Structure Width: [illegible]

Default Iconic Width: [illegible]

Default State: *Normal*

Type Color:

Field:	Type:	Result Format:	Visible:	Dereference:
r	float	*Default*	■	
from	struct atom_struct*	*Default*	■	
to	struct atom_struct*	*Default*	■	
next	struct conec_struct*	*Default*	■	

■ Selected instances

All instances

Future instances

Figure 14–32 Type Formatting Dialog Box

The Type Formatting dialog box has the following fields:

Type Name
displays the current data type.

Default Field Count
is the number of fields to be displayed initially for objects of that type.

Default Structure Width
is the width in pixels of the object.

Default Iconic Width
is the width in pixels of the object when it is in iconic form.

Default State
brings up a pop–up menu that lets you specify whether structures are first displayed as icons ("Iconic"), with the minimum number of fields displayed ("Normal") or with all fields displayed ("Detail").

Type Color
provides a submenu for color coding. It lets you select a color for the header and overview rectangles for objects of a given type.

For structure and union types, the list box shows all the fields with their types. For each field, you can change the result format to one of the following:

- Default
- Decimal
- Unsigned

- Octal
- Hex
- Float
- Char
- String
- Type
- Dec Addr
- Oct Addr
- Hex Addr
- Size in Bits

You can also specify whether a field is visible in normal state, and if it is a pointer field, whether it should be automatically dereferenced.

Once you specify the format for this type, you can apply it to any combination of the following through the toggle buttons in the bottom left portion of the window:

- selected instances
- all existing instances
- any future instances of this type

Examining Data at the Command Line

The Debugger command line lets you examine debugging data through the commands shown below.

file [*filename*]

displays the name of the current or specified file (*filename*). If a file is specified, it becomes the current file.

list [[*expression* [,*expression*]] | [*function_name*]]

lists the specified number (*expression*) of lines. The default is 10 lines. You can optionally specify a function where the list is to start.

print *expression* [,*expression*, ...]

prints the value of the specified expression(s). If the expression is a character pointer or array, both the string and address print.

display [*expression*, ...]

with *expression*, this command adds *expression* to the list of expressions to be displayed whenever the process stops.
with no arguments, lists all expressions on the display list.

undisplay [*displaynumber*, ...]

stops display of expression with specified *displaynumber* when the process stops. Removes the expression from the display list.

which *identifier*

displays the qualification of the specified variable.

whatis *identifier*

displays all the qualifications of the specified variable.

where

performs a stack trace.

func [func_name]

moves to the specified frame in the call stack or to the function in the executable if not in the stack.

up [expression]

moves up the specified number of frames in the call stack. up moves in the direction of the caller.

down [expression]

moves down the specified number of frames in the call stack. down moves in the direction of the called function.

Chapter 15

Machine–level Debugging

The CASEVision Debugger offers three views useful in debugging at the machine level. These views are covered in this chapter:

- **"Disassembly View"**
- **"Register View"**
- **"Memory View"**

Disassembly View

The Disassembly View of the Debugger lets you look at machine–level code rather than source–level code. A typical Disassembly View window appears in **Figure 15–1** , with the Disassemble menu displayed.

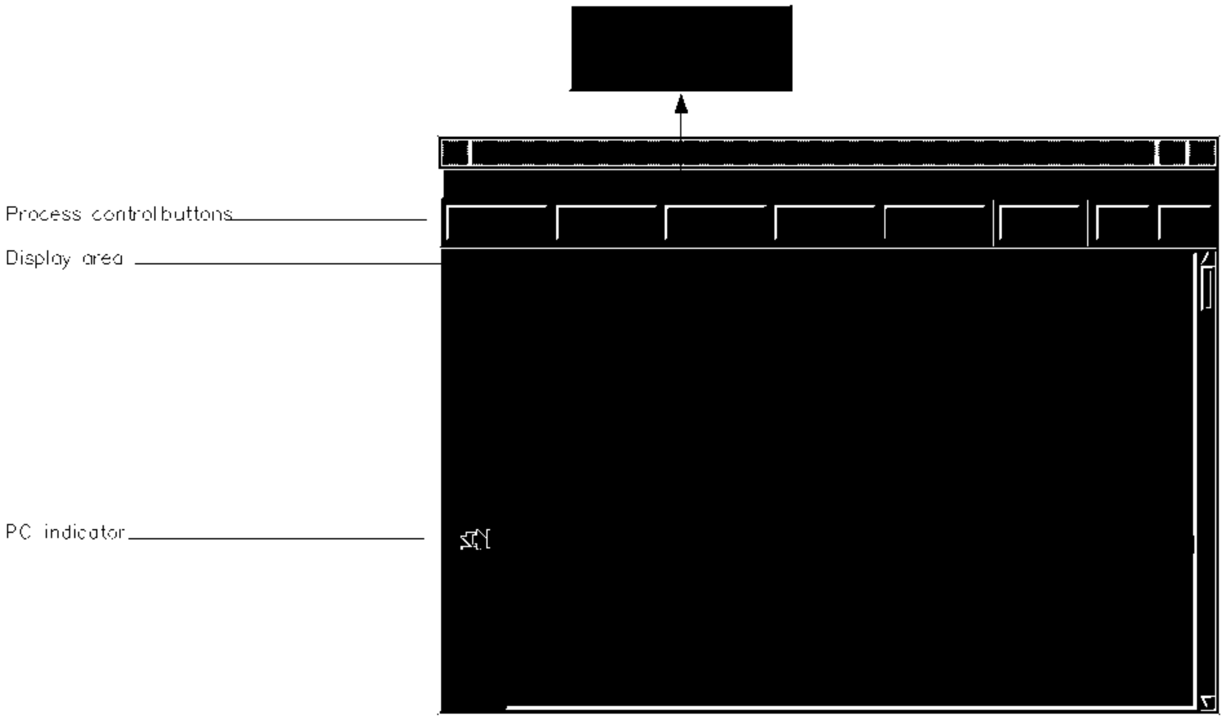


Figure 15–1 Disassembly View with Disassemble Menu Displayed

Similarities with Main View

At the top of the window are the same process control buttons as those in Main View. They behave the same way except for *Step Into* and *Step Over*, which do machine–level instruction stepping instead of source–level. Remember that you select the number of steps by holding down the right mouse button over the *Step Into* and *Step Over* buttons.

The menus are basically the same as in Main View except for the Disassemble menu. The PC menu selections "Continue To" and "Jump To" are based on machine–level instructions rather than source–level steps. The Config menu has a "Preferences..." selection that brings up a dialog box oriented to Disassembly View.

You can set traps either by using the Traps menu or clicking in the annotation column of the source display area that contains the disassembled code.

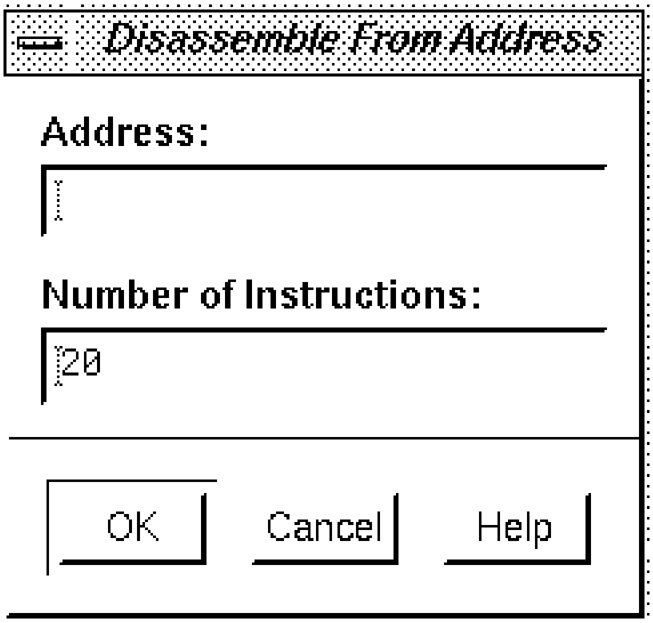
Disassemble Menu

Figure 15–2 Disassemble Menu

The Disassemble menu (see **Figure 15–2**) lets you display disassembled code. It contains the following items:

"Address..."

allows you to disassemble a specified number of lines, starting from a specified source line address (see **Figure 15–3**).



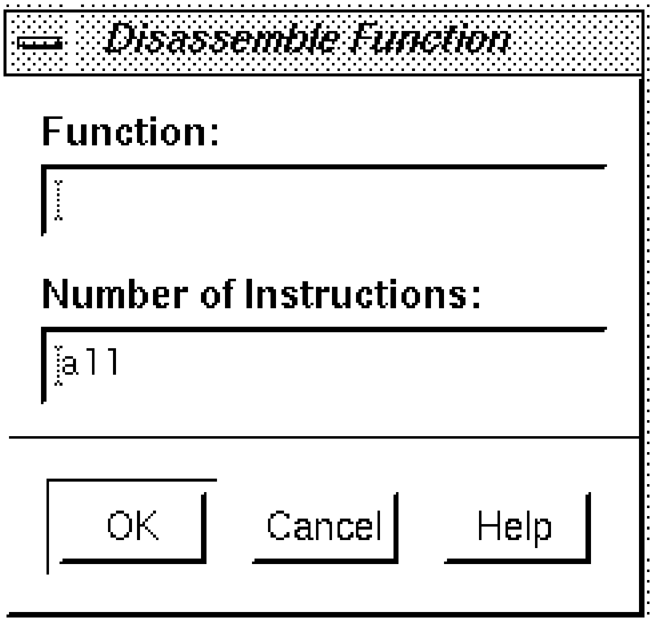
Address:

Number of Instructions: 20

Figure 15–3 Disassemble From Address Dialog Box

"Function..."

allows you to disassemble a specified number of lines, starting from the beginning address of a specified function name (see **Figure 15–4**). By default, the number of lines is all.

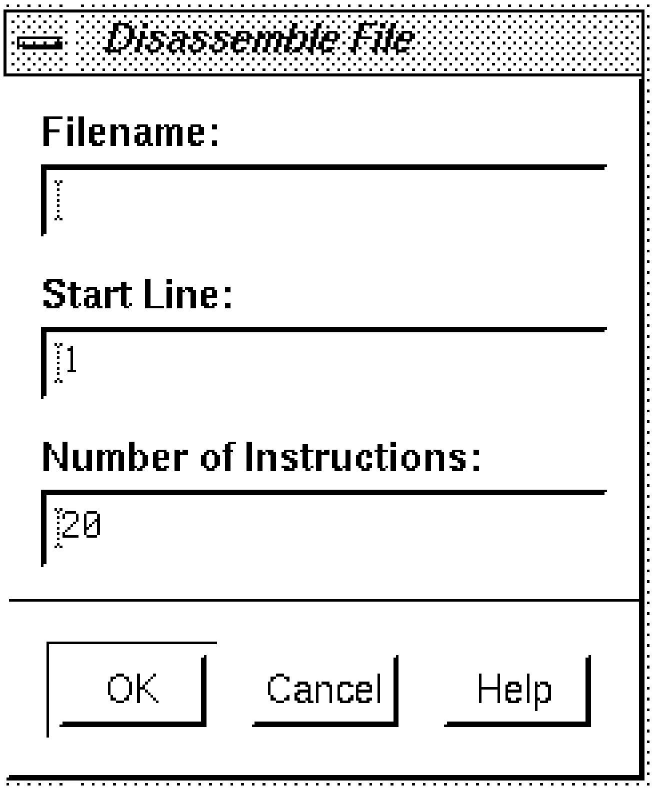


Number of Instructions: all

Figure 15–4 Disassemble Function Dialog Box

"File..."

allows you to disassemble a specified number of lines, starting from the address corresponding to a specified line number in a specified file (refer to **Figure 15–5**). If you have a current selection in Main View or Source View, its file and cursor position are used as the default file name and line number, respectively.



Start Line: 1

Number of Instructions: 20

Figure 15–5 Disassemble File Dialog Box

Disassembly View Preferences

Selecting "Preferences..." from the Config menu brings up the Disassembly View Preferences dialog box (shown in **Figure 15–6**) so that you can change the global preferences.

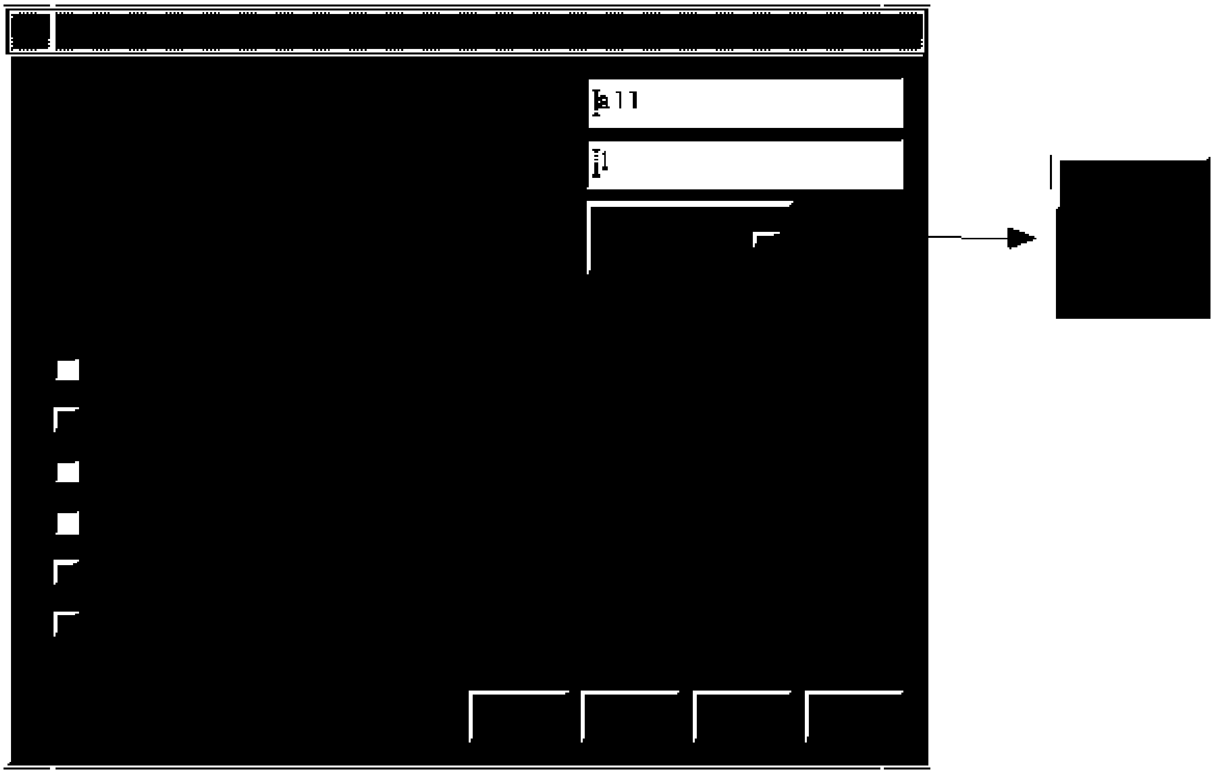


Figure 15–6 Disassembly View Preferences Dialog Box with Format Pop–up Menu

The dialog box provides you with these options:

Number of instructions to disassemble
controls the default number of disassembly lines shown when the process stops. This number appears in the dialog boxes selected from the Disassemble menu (see **Figure 15–3** , **Figure 15–4** , and **Figure 15–5**). The default is all instructions, indicating that the entire function will be disassembled.

Minimum lines around current instruction
controls the display of the disassembled code, enabling you to view at least the specified number of instructions before and after the current instruction.

Register name display format
controls how register names are displayed. The available modes are "Hardware," "Compiler," and "Assembler."
The *Display Options* selections control what information is shown in each disassembled line.

Show embedded source annotation
turns on interleaved source lines in the appropriate positions.

Show source file and line number
displays the file name and file position along with each machine instruction.

Show function name and line number
displays the function name and file position along with each machine instruction.

Show machine address
displays the memory address of each machine instruction.

Show instruction value
displays the instruction word along with each machine instruction.

Show jal targets numerically
controls whether the target address of a jal instruction is displayed as a hex address or symbolic label.

Register View

Register View lets you examine and modify register values. You bring it up by selecting "Register View" from the Views menu in Main View. **Figure 15–7** shows a typical Register View window that has been resized to show all available registers.

Register View displays each register with its current value. A question mark (*?*) displayed immediately before a register value signifies that the value is suspect; it may not be valid for the current frame. This can occur if a register is not saved across a function call. A colored marker indicates that a register value has changed since the last time the process stopped.

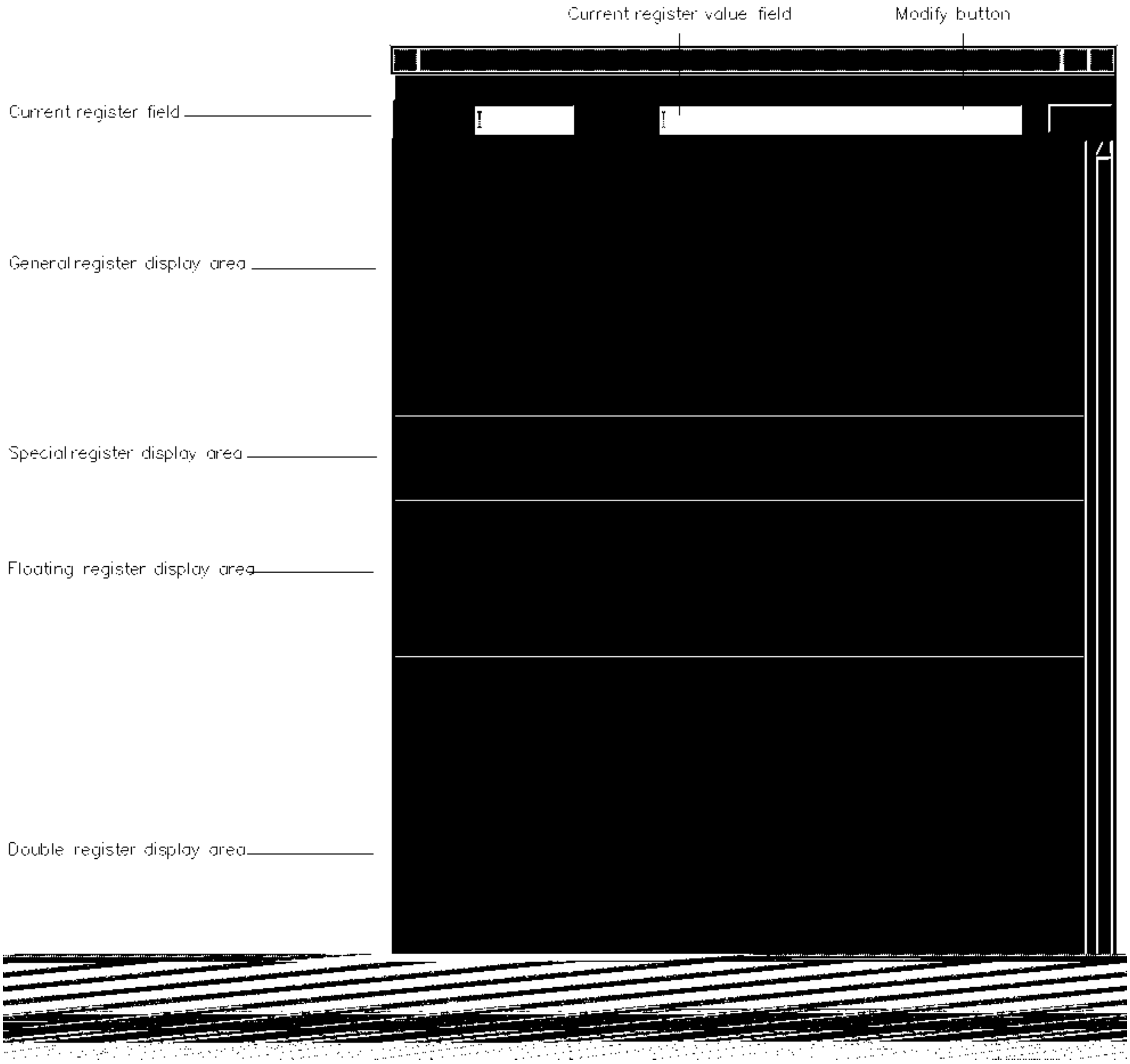


Figure 15–7 Register View

Register View Window

The major features of the Register View window are:

Current register field

identifies the currently selected register. You can switch to a different register by entering its name (either by hardware name or by alias) in this field and pressing **`<Enter>`**. You can also switch registers by clicking on the new register in the display area.

Current register value field

shows the contents of the selected register. You can assign a new value to a register by entering either a literal or an expression into the *Value* field. You then click on the *Modify* button to change the value or press **`<Enter>`**.

Register display area

shows the registers organized into four groups: general, special, floating, and double. Note that the general registers are identified by both their hardware and software names.

Double registers have a one−to−two correspondence with the floating registers.

Note: The special registers *p0, p1,* and *p2* are empty in the figure. These are used for instrumentation and only display values when instrumentation has taken place.

Changing the Register View Display

The "Preferences..." selection in the Config menu lets you change the Register View display. It brings up the Register View Preferences dialog box (see **Figure 15−8**).

The *Register Display* toggle buttons let you specify which types of registers are to be displayed by default.

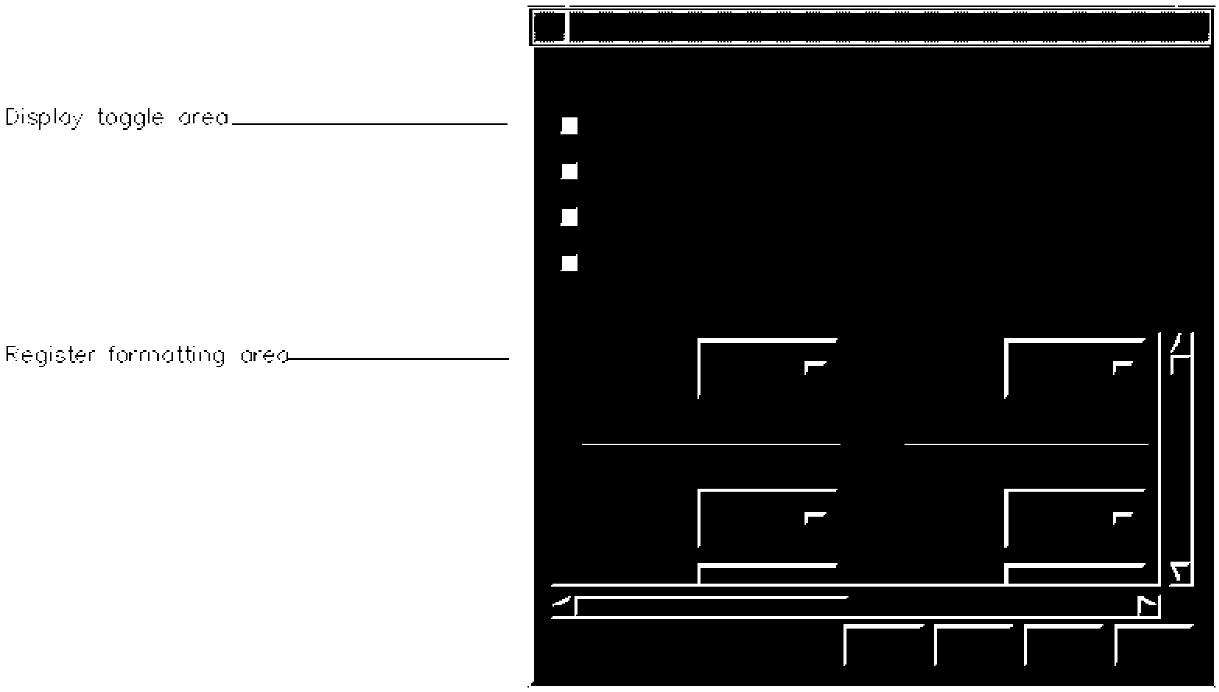


Figure 15–8 Register View Preferences Dialog Box

The *Register Formatting* area lets you select formats for any of the registers. You have a choice of "default", "decimal", "octal", or "hex" format.

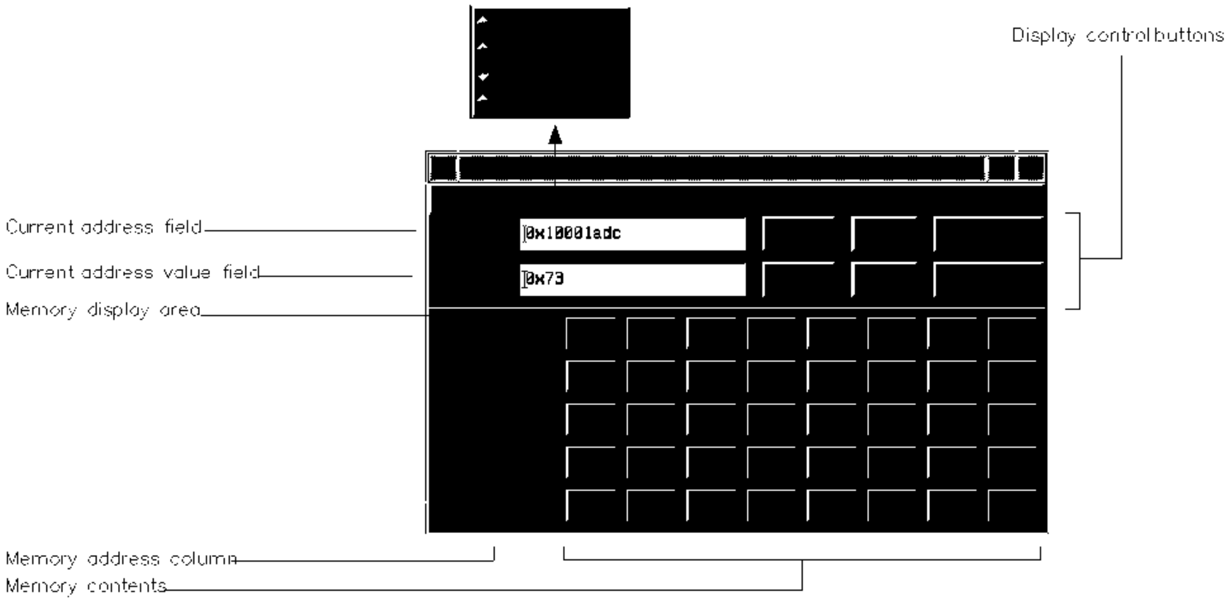
The default fields in the top row let you change the defaults for the four major types, which are set as follows:

- general registers —hexadecimal
- special registers—hexadecimal
- float registers—floating point
- double registers—floating point

The rows in the register formatting area let you change the modes for the individual registers.

Memory View

Memory View lets you examine and modify memory. A typical Memory View window appears in **Figure 15–9** .



0x10001adc

0x73

Figure 15–9 Memory View with Mode Menu Displayed

Viewing a Portion of Memory

To view a portion of memory, enter the beginning memory location in the *Address* field. You can enter the literal value or an expression that evaluates to an integer address. These address specifications must be in the language of the current process as indicated by the call stack frame. For example, you can enter `0x7fff4000+4` as the memory address when stopped in a C function or enter `$7fff4000+4` as the equivalent for a Fortran routine. Press `<Enter>` while the cursor is in the field or click the *View* button to display the contents of that location and the subsequent locations in the display area. This also displays the contents of the first address in the *Value* field where it can be modified.

The *memory display area* shows the contents of individual byte addresses. The column at the left of the display shows the first address in the row. The contents at that address are shown immediately to its right, followed by the contents of the next seven byte locations. If you enlarge the Memory View window, you can see additional rows of memory.

Changing the Contents of a Memory Location

To change the contents of a memory location, you select the address to be changed, either by direct entry or by clicking on the byte value in the display area. You can enter a single value or a sequence of hex byte values separated by spaces (for example, `00 3a 07 b2`) in the *Value* field. You can also enter a quoted string to change a consecutive range of values to the ascii values of that string. Pressing `<Enter>` while the cursor is in the *Value* field or clicking the *Modify* button substitutes the new value(s) starting at the specified location.

Changing the Memory Display Format

The Mode menu lets you change the format of the value field or byte locations to either decimal, octal, hex, or ascii.

Moving around the Memory View Display Area

The four control buttons at the upper right of the window help you move around the display area. These buttons are:

Up

for moving the displayed bytes up a single row.

Down

for moving the displayed bytes down a single row.

Page Up

for moving the displayed bytes upward by as many rows as are currently displayed.

Page Down

for moving the displayed bytes downward by as many rows as are currently displayed.

Chapter 16

Debugger: Detecting Heap Corruption

This chapter describes heap corruption detection and covers the following topics:

- **"Typical Heap Corruption Problems"**
- **"Detecting Heap Corruption Errors"**

Typical Heap Corruption Problems

Due to the dynamic nature of allocating and deallocating memory, the heap is vulnerable to these common corruption problems:

Boundary overrun

occurs when a program writes beyond the *malloc*ed region.

Boundary underrun

occurs when a program writes in front of the *malloc*ed region.

Access to freed memory

occurs when a program attempts to read or write to memory that has been *freed*.

Double frees

occur when a program frees some structure that it had already freed. In such a case, a subsequent reference can pick up a meaningless pointer, causing a segmentation violation.

Erroneous frees

occur when a program calls free() on addresses that were not returned for malloc, that is, a static, global, or automatic variable.

Malloc chain corruption

occurs when a program overwrites the header portion of a memory block, destroying the pointer linkage between blocks.

Unavailable memory

occurs when the virtual memory limit is exhausted.

Detecting Heap Corruption Errors

To detect heap corruption problems, you need to relink your executable with one of the special WorkShop *malloc* libraries (*–lmalloc_cv* or *–lmalloc_cv_d*) instead of the *malloc* library (*–lmalloc*)

The WorkShop Malloc Libraries

Both *–lmalloc_cv* and *–lmalloc_cv_d* support the use of the environment variable, MALLOC_FASTCHK, which provides faster but less accurate malloc checking than the variable MALLOC_DEBUG.

The library *–lmalloc_cv* uses whichever *malloc* you subsequently specify on the link line. It does not permit the use of MALLOC_DEBUG. If MALLOC_DEBUG is set, *–lmalloc_cv* will substitute MALLOC_FASTCHK and display a warning message. The *–lmalloc_cv_d* library permits MALLOC_DEBUG and forces the use of *–lmalloc*.

MALLOC_FASTCHK works by bracketing the *malloc*ed region with "guard" words to detect boundary problems. Any unexpected change to these words causes an error message to be written to *stderr*.

Compiling with the Malloc Libraries

You can compile your executable from scratch as follows:

```
cc -g -o targetprogram   targetprogram.c
 {-lmalloc_cv | -lmalloc_cv_d}
```

or you can relink it by using:

```
ld -o targetprogram targetprogram.o
 {-lmalloc_cv | -lmalloc_cv_d}
```

Setting the Environment Variables

After compiling, you invoke the Debugger with your executable as the target. In Execution View, you set environment variables to enable heap corruption detection from within the *malloc* library, as follows:

MALLOC_FASTCHK	is a fast check but it does miss some errors.
MALLOC_DEBUG	is a slower check, since it uses the entire *malloc* library. It checks chains for consistency. *MALLOC_DEBUG* is only available if you use the *malloc_cv_d* library; if you use *malloc_cv*, then the *MALLOC_FASTCHK* is substituted.

Trapping Heap Errors using MALLOC_FASTCHK

If you are using *MALLOC_FASTCHK*, use the Trap Manager to set a stop trap at the exit from the function *cvmalloc_error* which is called when an error is encountered. Continued execution often results in a core dump.

When you run the program, it will halt at the stop trap if a heap corruption error is detected. The error and the address are displayed in Execution View. You can also examine the Call Stack View at this point to get stack information. To find the next error, click the *Continue* button.

If you need more information to isolate the error, set a watchpoint trap to detect a *write* at the displayed address; then run your program again.

Trapping Heap Errors using MALLOC_DEBUG

If you are using *MALLOC_DEBUG*, then each call to *malloc* and *free* causes the entire *malloc* arena to be scanned and checked for consistency. If corruption is detected in the arena, the checking code calls *abort*(3C) (see man page). This usually results in the calling process exiting and leaving a core file in its current directory.

Chapter 17

Multiple Process Debugging

WorkShop supports performance analysis and debugging of multiprocess applications, including processes spawned either with *fork* or *sproc*. You can perform process control operations on a single process or on all members of a process group. You can attach WorkShop automatically to child processes. You can also specify spawned processes to inherit traps. The Trap Manager provides special trap commands to facilitate debugging multiple processes simultaneously.

This chapter discusses the details of multiprocess debugging in WorkShop and includes the following topics:

- **"Debugging with Multiprocess View"**
- **"Controlling Execution and Setting Traps in a Multiprocess Program"**
- **"Debugging a Multiprocess Fortran Program"**

Debugging with Multiprocess View

Multiprocess View operates on a *process group*. By default, a process group includes the parent process and all descendants spawned by *sproc*. Through a preferences option, processes spawned with *fork* during the session can be added to the process group automatically when they are created. Note that a child that *exec*'s with *setuid* (the user ID) enabled will not become part of the process group. Any process to which you have read/write access can also be added to the process group, if desired. All *sproc*'d processes must be in the same process group, since they share information.

Each process in the session can have a standard Main View session associated with it. All processes in a process group share a single Multiprocess View. Selecting "Multiprocess View..." from the Admin menu in Main View for any process in the group brings up the Multiprocess View window. If the Multiprocess window exists, it will be raised to the front; otherwise, a new window will be created.

Currently, Multiprocess View handles these multiple process situations:

- *True multiprocess program*, which refers to a tightly integrated system of *sproc*'d processes, generated by POWER/Fortran or POWER/C.
- *Auto–fork application*, which is a process that spawns a child process and then runs in the background.
- *Locally distributed application*, which is an application that involves two different executables running in different processes on the same host, coordinated by a rendezvous mechanism. To use the Performance Analyzer, you must have a Main View for each process and enable data collection accordingly.
- *Fork application*, which is a process that spawns child processes and can interact with them. The WorkShop Performance Analyzer supports applications that *fork* but not those that *exec*.

Multiprocess View does not support remotely distributed applications.

Displaying the Multiprocess View

The first step in debugging multiple processes simultaneously is to invoke the Debugger with the parent process. Then select "Multiprocess View" from the Admin menu to bring up Multiprocess View. Main View is attached to the parent process. **Figure 17–1** shows a typical Multiprocess View with Config and Process menus displayed.

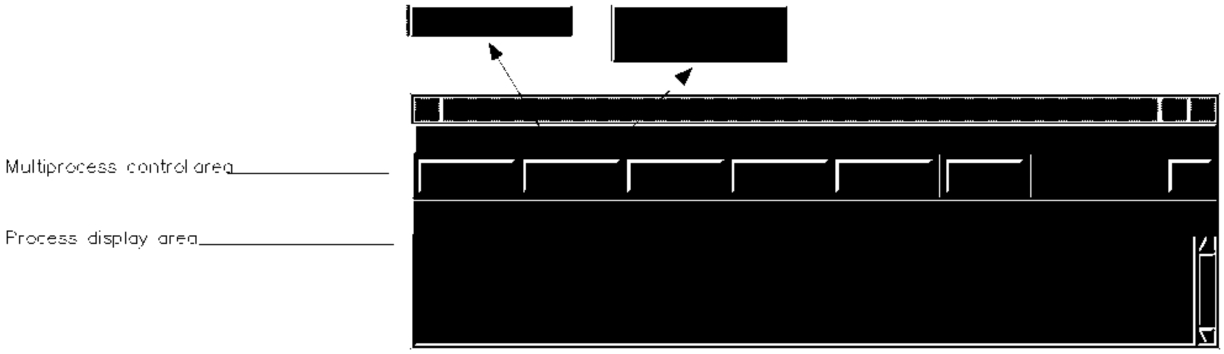


Figure 17–1 Multiprocess View with Config and Process Menus Displayed

To open a Main View (or other debugging views) for another process, double–click the desired process in Multiprocess View. A separate Main View window displays the selected process, and you can select any debugging views desired. If a set of views exists for that process, the views are raised to the foreground. To reuse views already displayed, select "Switch Process..." from the Admin menu in Main View. (If a process is currently highlighted in Multiprocess View, its id is entered automatically in the *Process id:* field in the Switch Process dialog box.)

Viewing Process Status

When Multiprocess View comes up, it lists the status of all processes in the process group. This information includes:

PID:

shows the process identifier (id).

PPID:

lists the parent process ids. Notice in **Figure 17–1** that the first process PID#7748 is the parent process of the second.

State:

represents the state of the process: stopped, running, or created, which appears just prior to running. Terminated processes are not displayed.

Name:

identifies the process by file name.

Function/PC:

indicates the current function and program counter (PC) for any stopped processes.

Multiprocess Control Buttons

Multiprocess View uses the same control buttons as MainView with two exceptions. The buttons are applied to all processes as a group. There is no separate *Run* button. Using a control button in Multiprocess View has the same effect as clicking the button in each process's Main View window. The buttons are:

Continue

resumes program execution after a halt and continues until a stop trap or other event stops execution.

Stop

stops execution of all processes. When program execution stops, the current source line of each process is highlighted in its Main View, if one is active, and annotated with an arrow indicating the PC.

Step Into

steps to the next source line and into function calls. To step a specific number of lines, hold down the right mouse button over the *Step Into* button. A pop–up menu displays that lets you select one of the fixed values or a specified number of steps.

Step Over

steps to the next source line and over function calls. To step a specific number of lines, hold down the right button over the *Step Over* button. A pop–up menu displays that lets you select one of the fixed values or a specified number of steps.

Return

executes the remaining instructions in the current function. Program execution stops upon return from that procedure.

Sample

collects performance data for each process (if performance data collection is enabled).

Kill

terminates all processes in the group.

Multiprocess Traps

As discussed in **Chapter 12, "Debugger: Setting Traps in WorkShop"** , the trap qualifiers *[all]* and *[pgrp]* are used in multiprocess analysis. The *[all]* entry stops or samples all processes when a trap fires. The *[pgrp]* entry sets the trap in all processes within the process group containing the trap location. The qualifiers can be entered by default by the "Group Trap Default" and "Stop All Default" selections in the Traps menu in Trap Manager.

Note that the *Sample* button always samples all processes.

Adding and Removing Processes

Figure 17–2 Process Menu in Multiprocess View

The Process menu lets you manually add or remove a process from the process group (see **Figure 17–2**).

To remove a process, click the process and select "Remove" from the Process menu. Note that a process in a *sproc* share group cannot be removed from the process group.

To add a process, select "Add...". The dialog box shown in **Figure 17–3** displays. Enter the new process id and click *OK*.

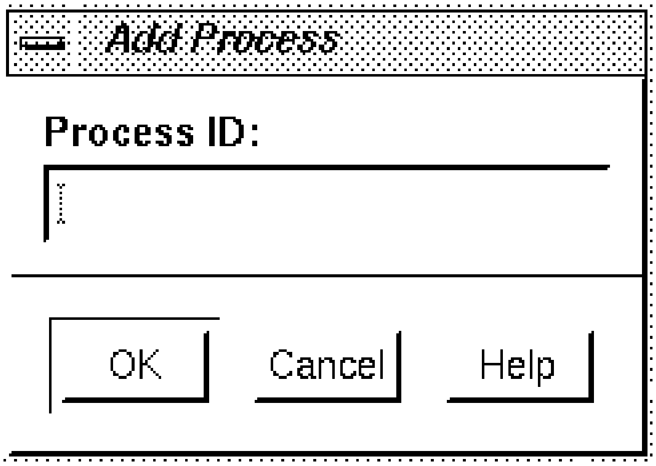
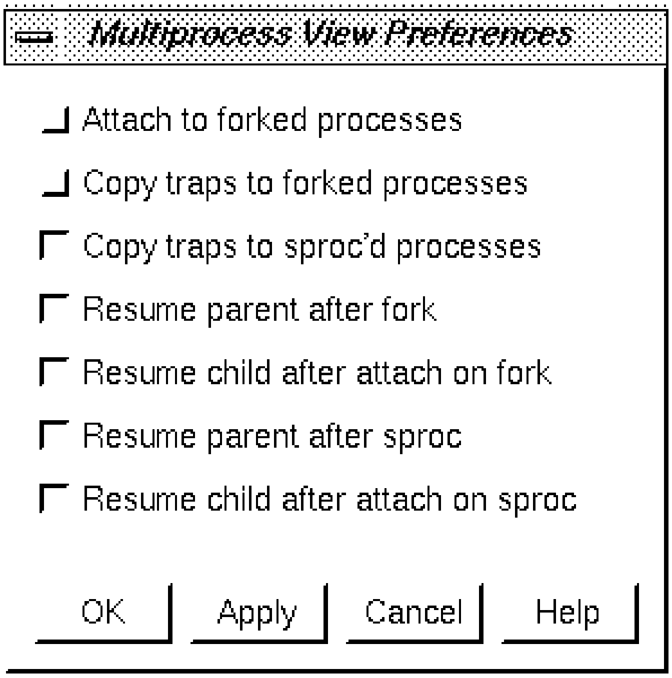


Figure 17–3 Add Process Dialog Box

Multiprocess Preferences

The "Preferences..." option in the Config menu brings up the Preferences dialog box. It lets you control when processes are added to the group, and it specifies their behavior (see **Figure 17–4**).



☑ Attach to forked processes
☑ Copy traps to forked processes
☐ Copy traps to sproc'd processes
☐ Resume parent after fork
☐ Resume child after attach on fork
☐ Resume parent after sproc
☐ Resume child after attach on sproc

Figure 17–4 Multiprocess View Preferences Dialog Box

The Multiprocess View preference options are:

Attach to forked processes

attaches new processes spawned by the *fork* command to the group automatically. (Note that processes spawned by *sproc* are always attached.)

Copy traps to forked processes

copies traps you have set in the parent process to new *fork*ed processes automatically. If you create parent traps with Trap Manager and specify *pgrp*, then the children inherit these traps automatically, regardless of the state of this flag.

Copy traps to sproc'd processes

copies traps you have set in the parent process to new *sproc*'d processes automatically. As in the previous option, if you create parent traps with the Trap Manager and specify *pgrp*, the children inherit these traps automatically, whether this flag is set or not.

Resume parent after fork

restarts the parent process automatically when a child is *fork*ed.

Resume child after attach on fork

restarts the new *fork*ed process automatically when it is attached. If this option is left off, a new process will stop as soon as it is attached.

Resume parent after sproc

restarts the parent process automatically when a child is *sproc*'d.

Resume child after attach on sproc

restarts the new *sproc*'d process automatically when it is attached. If this option is left off, a new process will stop as soon as it is attached.

Controlling Execution and Setting Traps in a Multiprocess Program

This section uses a C program that generates numbers in the *Fibonacci* sequence to demonstrate some of the tasks you'll be performing most often when using *cvd* to debug *mp* code. The tasks demonstrated are:

- stopping a child process on a *sproc*
- using the Multiprocess View buttons to control all processes
- setting traps in the parent process only
- setting group traps

The program *fibo* uses *sproc* to split off a child process, which in turn uses sproc to split off a grandchild process. All three processes churn out Fibonacci numbers until stopped. If you installed the demo programs, you can find the source for *fibo.c* in the directory */usr/demos/WorkShop/mp*.

A listing of *fibo.c* follows:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/prctl.h>

int NumberToCompute = 100;
int fibonacci();
void run(),run1();

int fibonacci(int n)
{
int f, f_minus_1, f_plus_1;
int i;

      f = 1;
      f_minus_1 = 0;
      i = 0;
```

```
      for (; ;) {
          if (i++ == n) return f;
          f_plus_1 = f + f_minus_1;
           f_minus_1 = f;
           f = f_plus_1;
      }
}

void run()
{
int fibon;
      for (; ;) {
          NumberToCompute = (NumberToCompute + 1) % 10;
          fibon = fibonacci(NumberToCompute);
          printf("%d'th fibonacci number is %d\n",
               NumberToCompute, fibon);
      }
}

void run1()
{
int grandChild;

      errno = 0;
      grandChild = sproc(run,PR_SADDR);

      if (grandChild == -1) {
          perror("SPROC GRANDCHILD");
      }
      else
          printf("grandchild is %d\n", grandChild);
      run();
}

void main ()
{
int second;

      second = sproc(run1,PR_SADDR);
      if (second == -1)
          perror("SPROC CHILD");
      else
          printf("child is %d\n", second);

      run();
      exit(0);
}
```

To get started, compile the program and run the Debugger.

1. Compile *fibo.c*.

   ```
   cc -g fibo.c -o fibo
   ```

2. Invoke the Debugger on *fibo*.

   ```
   cvd fibo &
   ```

3. Bring up the multiprocess view by selecting "Multiprocess View..." from the Admin menu.

In the next section, you'll set options to control how the process executes.

Using the Multiprocess View to Control Execution

To examine each process as it appears, you need to stop child processes as they are created with *sproc*. You can control the Debugger's behavior on *sproc* by setting Multiprocess preferences.

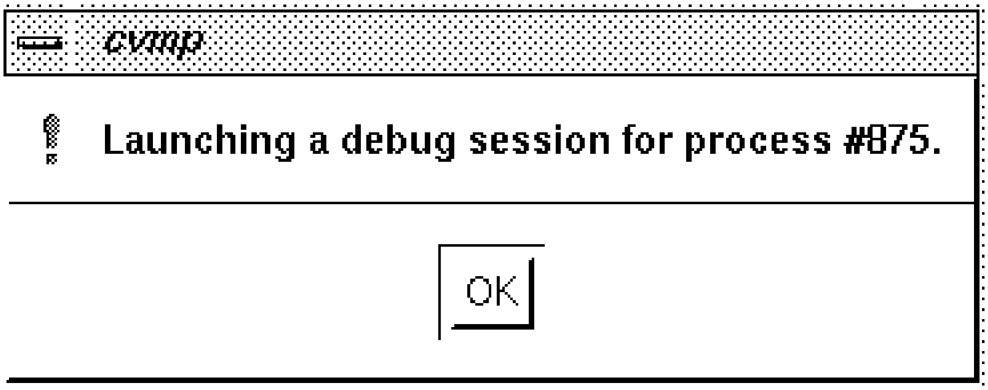
1. Select "Preferences..." from the Config menu in Multiprocess View.
2. Deactivate *Resume child after attach on sproc*.

   At the same time, you can turn off trap inheritance, so you can experiment with trap setting later.
3. Click *OK* to accept the change.

   Now you're ready to run the process.
4. In the Main View, click *Run*.

   If you watch Multiprocess View, you will see the main process appear, and spawn a child process. The child process will stop as soon as it appears, since you turned off the *Resume child after attach on sproc* option. You can now use Multiprocess View to open a new main view for the child process.
5. Double–click the child process in the Multiprocess View window.

   You will see a dialog box like the one in **Figure 17–5** , and the Debugger will create a new window.



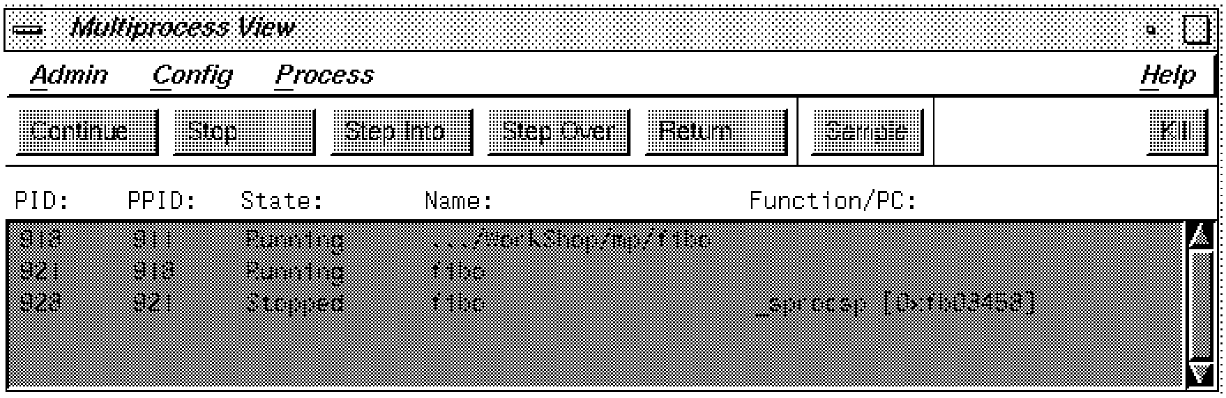
Launching a debug session for process #875.

Figure 17–5 Launching a Debug Session Dialog Box

You can use the buttons in Multiprocess View to control all the processes simultaneously, or use the buttons in each of the Main Views to control each process separately. To send the child process on its way,

6. Click *Continue* in the Multiprocess View window.

   The first child now spawns a grandchild process. The grandchild will stop in sproc, as shown in **Figure 17–6** :



```
PID:    PPID:   State:      Name:                   Function/PC:
918     911     Running     .../WorkShop/mp/fibo
921     918     Running     fibo
923     921     Stopped     fibo                    _sprocsp [0xfb03450]
```

Figure 17–6 Using the Multiprocess View to Examine Process State

Using the Trap Manager to Control Trap Inheritance

This section shows you how to use the Trap Manager to set traps that affect one or all of the *fibo* process group. For complete information on using the Trap Manager, refer to **Chapter 12, "Debugger: Setting Traps in WorkShop"** .

1. In the Main View for the parent process, select "Trap Manager" from the Views menu.

   Right now, traps set using the Traps menu in any of the Main View windows will affect only the process controlled by that Main View. For example, see what happens if you set a stop trap in the first executable line of *run()*, which is line 32:

   ```
   32 NumbertoCompute = (NumbertoCompute + 1) % 10;
   ```

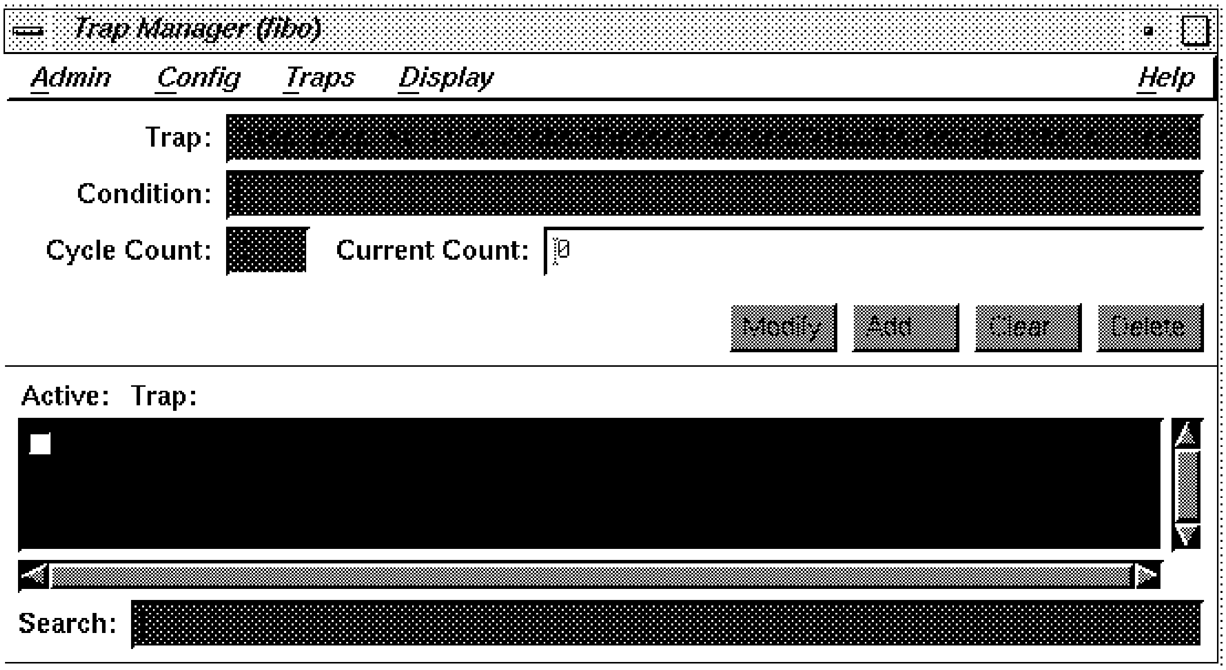
2. Using the Traps menu of the parent process, set a stop trap at line 32 of *fibo.c.*

   Only the parent process halts. The child processes continue running, as a glance at Multiprocess View will confirm.

   You can use the Trap Manager to edit the trap so that it affects the whole process group.

3. Insert the word **`pgrp`** after the word **`Stop`**.

   The trap should read `Stop pgrp at..`, as shown in **Figure 17–7** .



Current Count: 0

Figure 17–7 Modifying a Trap to Affect a Process Group

4. Click *Modify* to accept your change to the trap.

   Now the trap affects the two child processes as well. You can watch the Multiprocess View to see the whole process group stop at the trap on line 32.

   You can set an option to make all traps affect the process group by default for those traps set using the Trap Manager.

5. Select "Group Trap Default" from the Traps menu (see **Figure 17–8**).

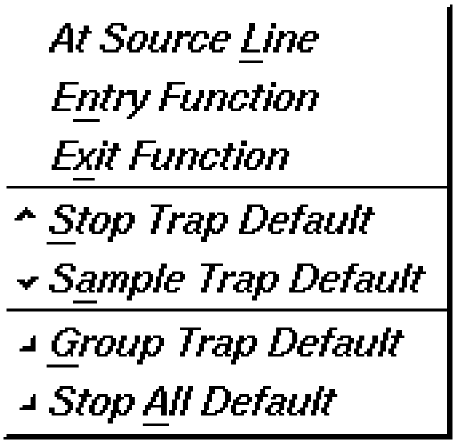


Figure 17–8 Setting the Group Trap Default

6. In the Main View of the parent process, select any executable line in the function *fibonacci*.
7. To set a trap, select "At Source Line" from the Traps menu of the Trap Manager.

   The trap you've just set will include the modifier *pgrp*. It automatically affects both child processes.

   You have now learned the basics of controlling the execution of multiple processes and setting traps.

8. Select "Exit" from the Admin menu in each Main View to end this tutorial.

   Note that the Multiprocess View window must be closed explicitly. It does not close when the Main View windows do.

Debugging a Multiprocess Fortran Program

The first part of this section presents a few standard techniques to assist you in debugging a parallel program. The second part shows you how to use the WorkShop Debugger to debug the sample program from Chapter 6 of the *Fortran 77 Programmer's Guide.*

General Fortran Debugging Hints

Debugging a multiprocessed program is more difficult than debugging a single−processor program; therefore, debug as much as possible on the single−processor version.

Try to isolate the problem as much as possible. If you can, reduce the problem to a single *C$DOACROSS* loop.

Once you've isolated the problem to a specific *DO* loop, try changing the order of its iterations in a single−processor version. If the loop can be multiprocessed, then the iterations can execute in any order and produce the same answer. If the loop cannot be multiprocessed, changing the order frequently causes the single−processor version to fail. If it fails, you can use standard single−process debugging techniques to find the problem.

If this technique fails, you need to debug the multiprocessed version. Compile your code with the flags -g and -mp_keep. The -mp_keep flag saves the file containing the multiprocessed *DO* loop Fortran code. The compiler will save the code in a file named

```
$TMPDIR/P<user_subroutine_name><machine_name><pid>
```

where *user_subroutine_name* is the name of the subroutine containing the DOACROSS, *machine_name* is your machine name, and *pid* is the process id number of the compilation.

If you have not set the environment variable *TMPDIR*, /tmp is used.

Multiprocess Debugging Session

This section walks you through the process of using the Debugger to debug a small segment of incorrectly multiprocessed code. The example used in this section is also treated in Chapter 6 of the Fortran 77 Programmer's Guide with *dbx*. You can use *cvd* to perform the same tasks with less effort.

If you installed the demo programs, you can find the source for the code you will be debugging, *total.f*, in the directory
/usr/demos/WorkShop/mp. A listing follows:

```
program driver
      implicit none
      integer iold(100,10), inew(100,10),i,j
      double precision aggregate(100, 10),result
      common /work/ aggregate
      call total(100, 10, iold, inew)
      do 20 j=1,10
        do 10 i=1,100
          result=result+aggregate(i,j)
10    continue
20  continue
      write(6,*)' result=',result
      stop
      end

      subroutine total(n, m, iold, inew)
```

```
      implicit none
      integer n, m
      integer iold(n,m), inew(n,m)
      double precision aggregate(100, 100)
      common /work/ aggregate
      integer i, j, num, ii, jj
      double precision tmp

      C$DOACROSS LOCAL(i,ii,j,jj,num)
      do j = 2, m-1
        do i = 2, n-1
          num = 1
          if (iold(i,j) .eq. 0) then
            inew(i,j) = 1
          else
          num = iold(i-1,j) +iold(i,j-1) + iold(i-1,j-1) +
&         iold(i+1,j) + iold(i,j+1) + iold(i+1,j+1)
            if (num .ge. 2) then
              inew(i,j) = iold(i,j) + 1
            else
              inew(i,j) = max(iold(i,j)-1, 0)
            end if
          end if
          ii = i/10 + 1
          jj = j/10 + 1
          aggregate(ii,jj) = aggregate(ii,jj) + inew(i,j)
        end do
      end do
      end
```

In the program, the local variables are properly declared. The *inew* always appears with *j* as its second index, so it can be a share variable when multiprocessing the *j* loop. The *iold*, *m*, and *n* are only read (not written), so they are safe. The problem is with *aggregate*. The person analyzing this code reasoned that because *j* is always different in each iteration, *j/10* will also be different. Unfortunately, since *j/10* uses integer division, it often gives the same results for different values of *j.*

While this is a fairly simple error, it is not easy to see. When run on a single processor, the program always gets the right answer. Sometimes it gets the right answer when multiprocessing. The error occurs only when different processes attempt to load from and/or store into the same location in the *aggregate* array at exactly the same time.

Here are the steps in this exercise:

1. First try reversing the order of the iterations. Replace:

   ```
   do j = 2, m-1
   ```

   with

   ```
   do j = m-1, 2, -1
   ```

   This still gives the right answer when running with one process but the wrong answer when running with multiple processes. The local variables look right, there are no equivalence statements, and *inew* uses only simple indexing. The likely item to check is *aggregate*. Your next step is to look at aggregate with the Debugger.

2. Compile the program with the **-g -mp_keep** options:

   ```
   % f77 -g -mp -mp_keep total.f -o total
   ```

   If your debugging session is not running on a multiprocessor machine, you can force the creation of two threads for example purposes by setting an environment variable.

3. If you use the C shell, type:

```
% setenv MP_SET_NUMTHREADS 2
```

4. Start the Debugger:

```
% cvd total&
```

   The Debugger Main View window displays.

5. Choose "Go To Line..." from the Source menu and select line 41.

   This takes you to line 41:

```
aggregate(ii,jj) = aggregate(ii,jj) + inew(i,j)
```

   The subroutine touches *aggregate* in only one place, line 41. You want to set a stop trap at this line, so you can see what each thread is doing with *aggregate*, *ii*, and *jj*. You also want this trap to affect all threads of the process group. One way to do this is to turn on trap inheritance using the Multiprocess View Preferences dialog box. Another way is to use the Trap Manager to specify group traps, as follows.

6. From the Views menu, select Trap Manager.
7. In the Trap Manager window, pull down the Traps menu. Select the "Group Trap Default" option from the menu.

   This sets the group default.

8. Place the cursor in line 41 in the Main View window.

   This selects the line.

9. From the Traps menu in Traps Manager, select "At Source Line."

   This sets the stop trap, which should read something like this trap:

```
Stop pgrp in file /<yourDirectory> line 41
```

10. Bring up the Multiprocess View to keep tabs on the status of the two processes.

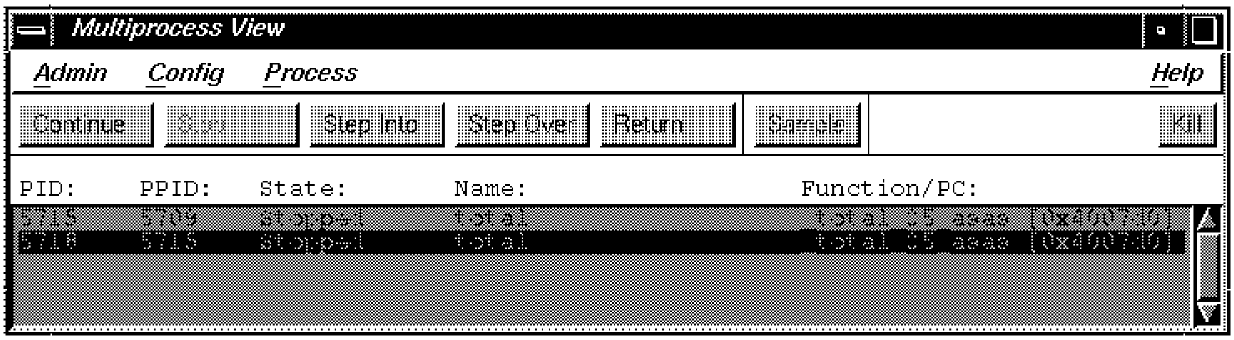
    Now you're ready to run the program.

11. Click *Run* in the Main View window.

    As you watch the Multiprocess View, you'll see the two processes appear, run, and stop in the function _total_25_aaaa. The Main View window is now relative to the master process.

12. Double–click the slave process listed in the Multiprocess View window, as in **Figure 17–9** .

    This invokes a Main View debugging session on the slave process.



```
PID:    PPID:   State:    Name:        Function/PC:
5715    5709    Stopped   total         _total_25_aaaa [illegible]
5718    5715    Stopped   total         _total_25_aaaa [illegible]
```

Figure 17–9 Launching a New Debugging Session from Multiprocess View

Now you can invoke the Variable Browser on each process. Look at *ii* and *jj* in **Figure 17–10** .

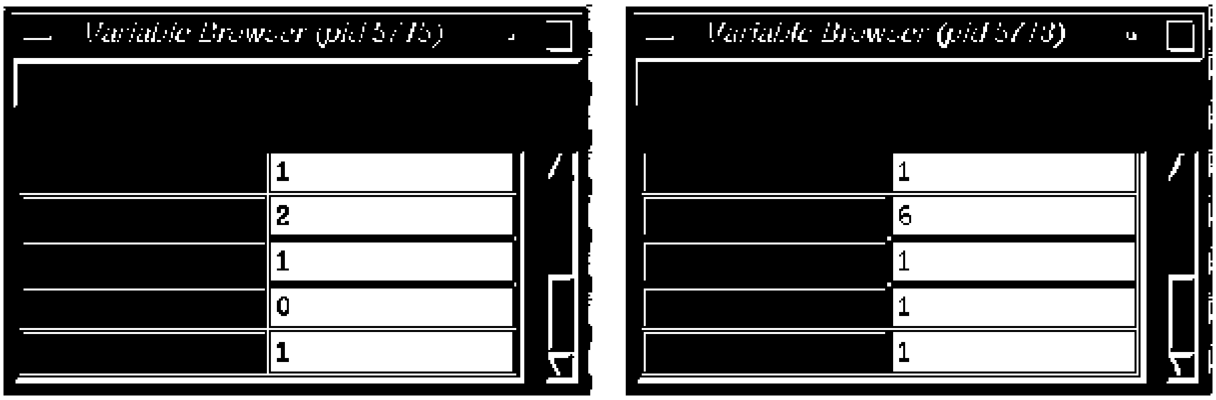


Figure 17–10 Comparing Variable Values from Two Processes

They have the same values in each process; therefore, both processes may attempt to write to the same member of the array *aggregate* at the same time. So *aggregate* should not be declared as a share variable. You've found the bug in your parallel Fortran program.

Chapter 18

Introduction to the Performance Analyzer

The Performance Analyzer helps you understand your program in terms of performance, determine if there are problems, and correct them. This chapter provides a brief introduction to the Performance Analyzer tools and describes how to use them to solve performance problems. It includes the following sections:

- **"Performance Analyzer Overview"**
- **"The Performance Analyzer Tools"**
- **"Sources of Performance Problems"**
- **"Interpreting Performance Analyzer Results"**

Performance Analyzer Overview

To conduct performance analysis, you run a series of experiments to collect performance data. Prior to running an experiment, you specify the objective of your experiment through a task menu. The Performance Analyzer collects the required data and provides charts, tables, and annotated code to help you analyze the results.

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 impact performance, such as *reads* and *writes*, system calls, page faults, floating point exceptions, and *mallocs*, *reallocs*, and *frees*.

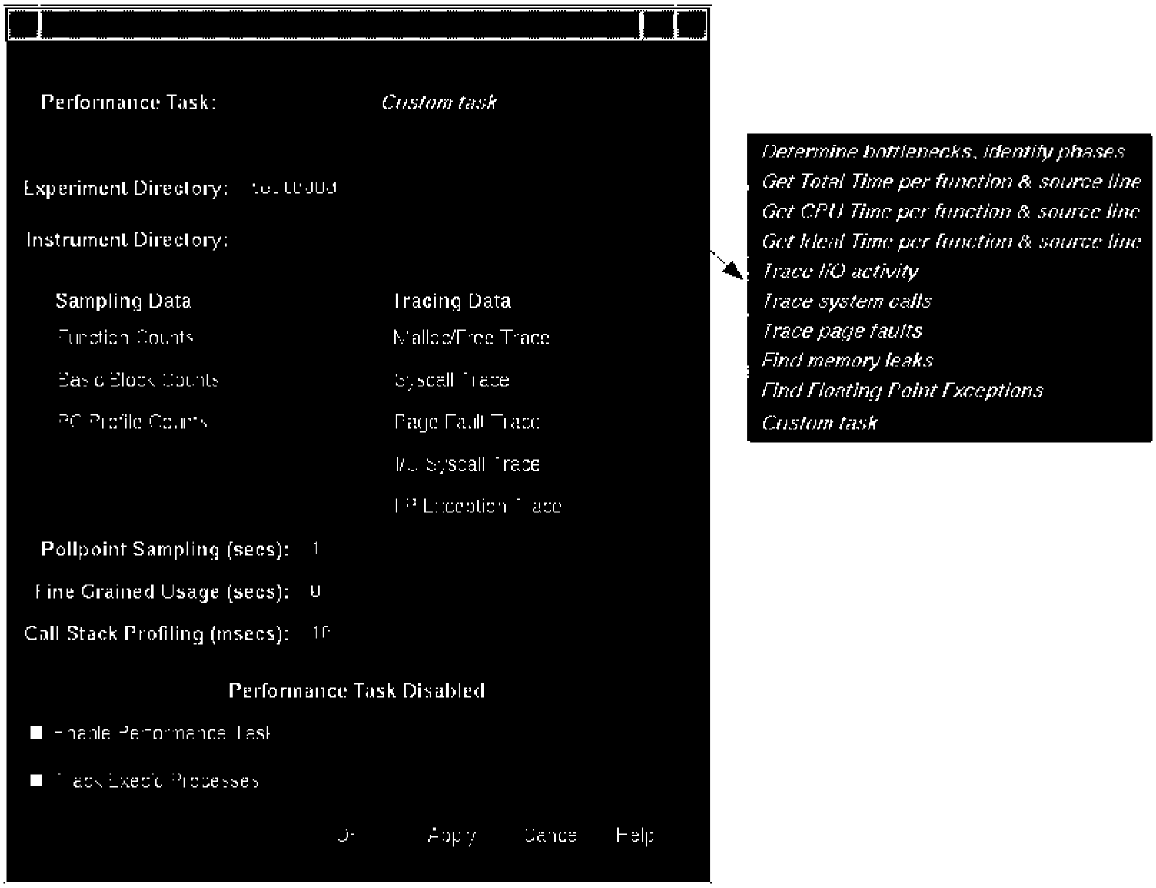
The Performance Analyzer processes the data to provide 28 different types of performance metrics.

The Performance Analyzer Tools

There are twelve major windows in the Performance Analyzer toolset:

- Performance Panel—for specifying data to be collected in the current experiment in terms of predefined tasks (see **Figure 18–1**)

- Performance Analyzer main window—containing the time line which shows when sample events occurred in the experiment and controls the scope of analysis for the Performance Analyzer views; the function list that shows functions with their performance metrics; the system resource usage chart; and the task defining the objective of the experiment (see **Figure 18–2**).
- Usage View (Graphical)—containing charts that indicate different types of resource usage and the occurrence of sample events over time (see **Figure 18–3**)
- Usage View (Textual)—providing the values corresponding to time intervals set by the time line calipers (see **Figure 18–3**)
- Call Graph View—displaying the functions (and their metrics) in a graphical format showing where the calls were made (see **Figure 18–3**)
- Call Stack—displaying the contents of the call stack where specified events occurred (see **Figure 18–3**)
- Heap View—displaying a map of memory indicating where memory may have been allocated or freed incorrectly (see **Figure 18–3**)
- Leak View—for displaying each memory leak that occurs in your executable, its size, the number of times the leak occurred at that location during the experiment, and the corresponding call stack when you select the leak (see **Figure 18–3**).
- Malloc View—for displaying each *malloc* that occurs in your executable, its size, the number of times the *malloc* occurred (a count is kept of *mallocs* with identical call stacks), and the corresponding call stack when you select the *malloc* (see **Figure 18–3**).
- I/O View—displaying a chart devoted to I/O system calls. Identifies up to 10 files involved in I/O (see **Figure 18–3**)
- Source View with performance annotations—a special version of Source View displaying the performance metrics adjacent to the corresponding line of source code (see **Figure 18–9**)
- Disassembly View with performance annotations—a special version of Disassembly View displaying the performance metrics adjacent to the corresponding machine code



Performance Task: *Custom task*

Experiment Directory: [illegible]

Instrument Directory:

Sampling Data

Function Counts

Basic Block Counts

PC Profile Counts

Tracing Data

Malloc/Free Trace

Syscall Trace

Page Fault Trace

I/O Syscall Trace

FP Exception Trace

Pollpoint Sampling (secs): 1

Fine Grained Usage (secs): 0

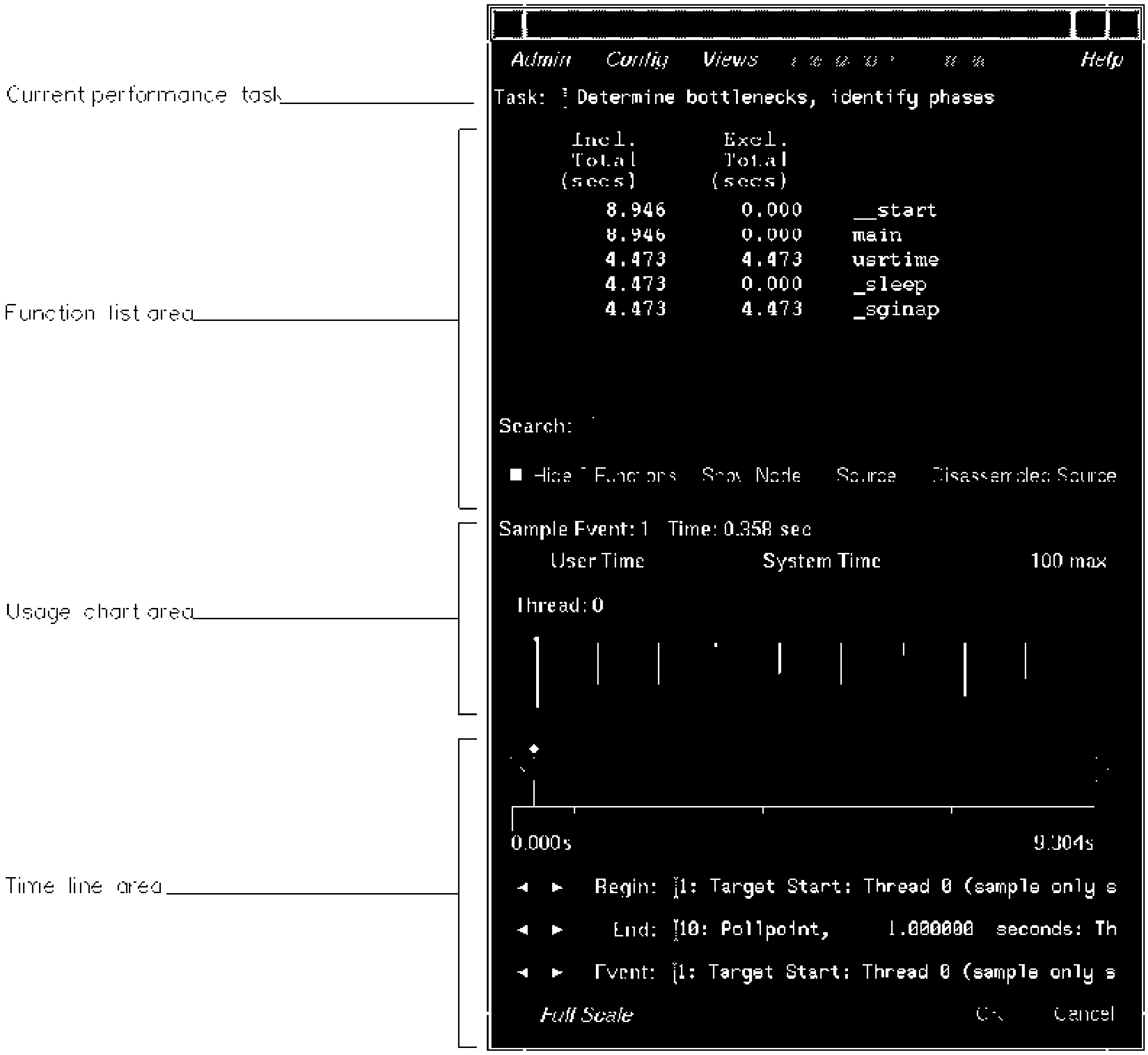
Call Stack Profiling (msecs): 10

Performance Task Disabled

■ Enable Performance Task

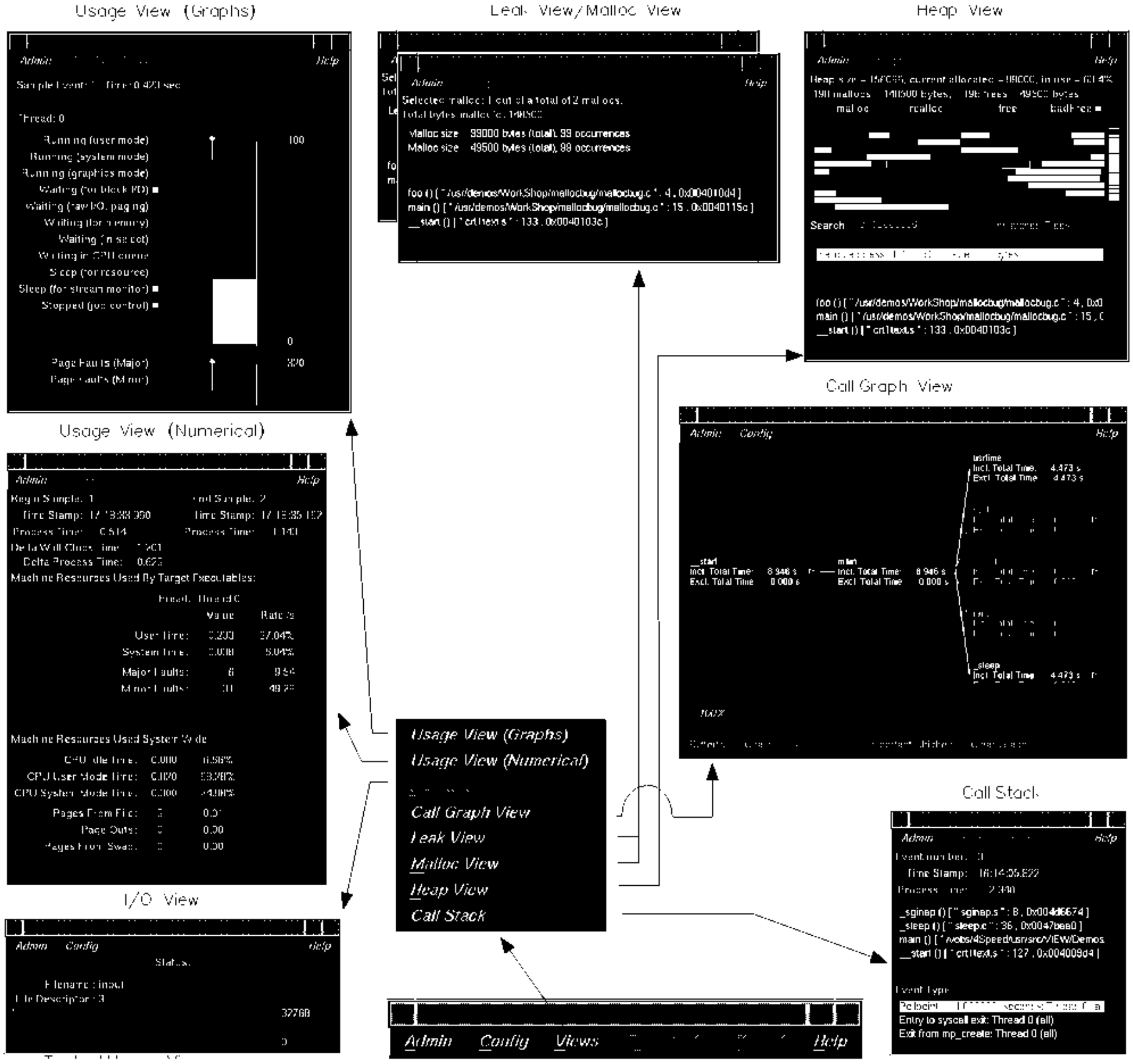
■ Track Exec'd Processes

Figure 18–1 Performance Panel with Task Menu



```
Task: Determine bottlenecks, identify phases

     Incl.      Excl.
     Total      Total
    (secs)     (secs)
      8.946      0.000     __start
      8.946      0.000     main
      4.473      4.473     usrtime
      4.473      0.000     _sleep
      4.473      4.473     _sginap

Search:

Sample Event: 1   Time: 0.358 sec
   User Time          System Time          100 max
 Thread: 0

0.000s                                   9.304s
◄ ►  Begin: 1: Target Start: Thread 0 (sample only s
◄ ►    End: 10: Pollpoint,     1.000000  seconds: Th
◄ ►  Event: 1: Target Start: Thread 0 (sample only s
```

Figure 18–2 Performance Analyzer Main Window



```
Malloc size   99000 bytes (total), 99 occurrences
Malloc size   49500 bytes (total), 99 occurrences

foo () [ "/usr/demos/WorkShop/mallocbug/mallocbug.c" : 4 , 0x004010d4 ]
main () [ "/usr/demos/WorkShop/mallocbug/mallocbug.c" : 15 , 0x0040115c ]
__start () [ "crt1text.s" : 133 , 0x0040103c ]
```

```
_sginap () [ "sginap.s" : 8 , 0x004d6674 ]
_sleep () [ "sleep.c" : 36 , 0x0047bea0 ]
main () [ "/vobs/4Speed/usr/src/VIEW/Demos
__start () [ "crt1text.s" : 127 , 0x00400094 ]

Entry to syscall exit: Thread 0 (all)
Exit from mp_create: Thread 0 (all)
```

Figure 18–3 Performance Analyzer Views Menu with Selections

Sources of Performance Problems

To tune a program's performance, you need to determine its consumption of machine resources. At any point (or phase) in a process, there is one limiting resource controlling the speed of execution. Processes can be slowed down by:

- CPU speed and availability
- I/O processing
- memory size and availability
- bugs
- instruction and data cache size
- any of the above in different phases

CPU–bound Processes

A *CPU–bound* process spends its time in the CPU and is limited by CPU speed and availability. To improve its performance on CPU–bound processes, you may need to streamline your code. This can entail modifying algorithms, reordering code to avoid interlocks, removing nonessential steps, blocking to keep data in cache and registers, or using alternative algorithms.

I/O–bound Processes

An *I/O–bound* process has to wait for I/O to complete, and may be limited by disk access speeds or memory caching. To improve the performance of I/O–bound processes, you can try one of the following techniques:

- improve overlap of I/O with computation
- optimize data usage to minimize disk access
- use data compression

Memory–bound Processes

A program that continuously needs to swap out pages of memory is called *memory–bound*. Page thrashing is often due to accessing virtual memory on a haphazard rather than strategic basis. One telltale indication of a page–thrashing condition is noise due to disk accesses. To fix a memory–bound process, you can try to improve the memory reference patterns or, if possible, decrease the memory used by the program.

Bugs

You may find that a bug is causing the performance problem. For example, you may find that you are reading in the same file twice in different parts of the program, that floating point exceptions are slowing down your program, that old code has not been completely removed, or that you are leaking memory (making *malloc* calls without the corresponding calls to *free*).

Performance Phases in Programs

Since programs exhibit different behavior during different phases of operation, you need to identify the limiting resource during each phase. For example, a program can be I/O–bound initially while it reads in data, CPU–bound while it performs computation, and then I/O–bound again in its final stage while it writes out data. Once you've identified the limiting resource in a phase, you can perform an in–depth analysis to determine which part of the code is causing the problem. And after you have solved that problem, you can check for other problems within the phase—performance analysis is an iterative process.

Interpreting Performance Analyzer Results

Before we discuss the mechanics of using the Performance Analyzer, let's look at these features that help you understand the behavior of your processes:

- **"The Time Line Display"**
- **"Resource Usage Graphs"**
- **"Textual Usage View"**
- **"The Function List Area"**
- **"Call Graph View"**
- **"Source View with Performance Annotations"**
- **"Heap View, Leak View, and Malloc View"**
- **"Call Stack View"**
- **"I/O View"**

The Time Line Display

Have you ever considered timing a program with a stopwatch? The Performance Analyzer time line serves the same function. **Figure 18–4** shows the time line portion of the Performance Analyzer window with typical results.

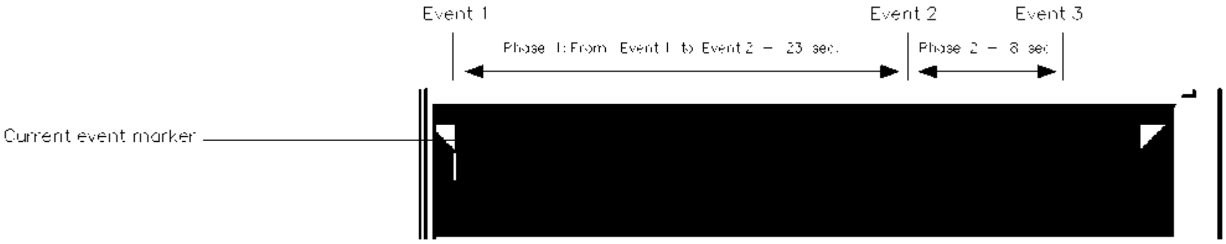


Figure 18–4 Typical Performance Analyzer Time Line

The time line shows where each sample event in the experiment occurred. By setting sample traps at phase boundaries, you can analyze metrics on a phase–by–phase basis. The simplest metric, time, is easily recognized as the space between events. The triangular icons are calipers; they let you set the scope of analysis to the interval between the selected events.

Resource Usage Graphs

The Performance Analyzer lets you look at how different resources are consumed over time. It produces a number of resource usage graphs that are tied to the time line (see **Figure 18–5** , which shows four of the graphs available). These resource usage graphs indicate trends and let you pinpoint problems within phases.

Resource usage data refers to items that consume system resources. They include
- user and system time
- page faults
- context switches
- the size of reads and writes
- read and write counts
- poll and I/O calls
- total system calls
- process signals
- process size

Resource usage data is always recorded (written to file) at each sample point. In addition, setting a time in the *Fine Grained Usage* field enables you to record resource usage data at regular intervals. Fine–grained usage allows you to see fluctuations at a finer gradation than the phases defined by sample points. If you discover inconsistent behavior within a phase, you can set new sample points and break the phase down into smaller phases.

You can analyze resource usage trends in the charts in Graphical Usage View and can view the numerical values in the Textual Usage View.

Fine grained usage has little effect on the execution of the target process during data collection. It is of limited use if the program is divided into phases of uniform behavior by the placement of the sample points.

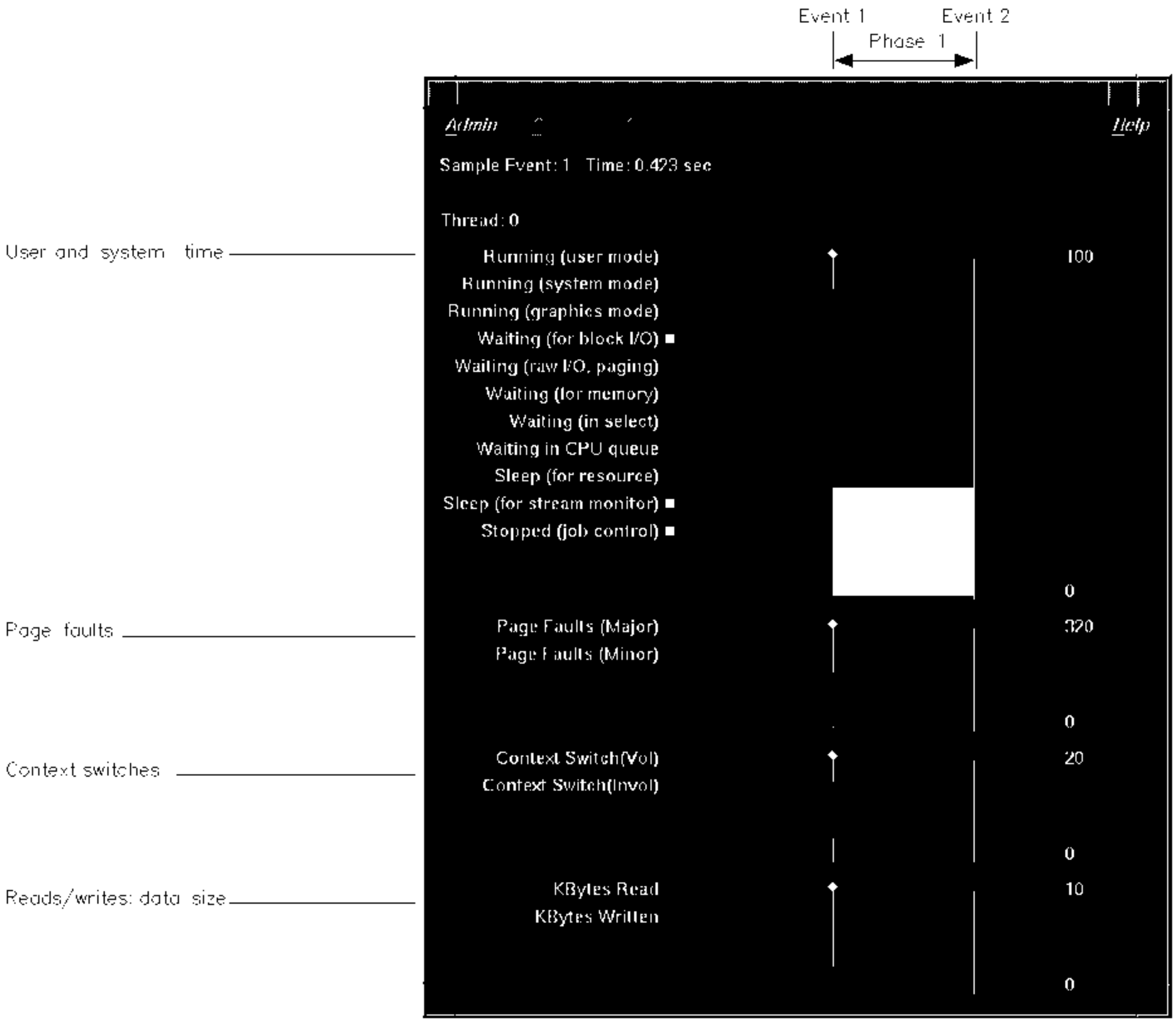
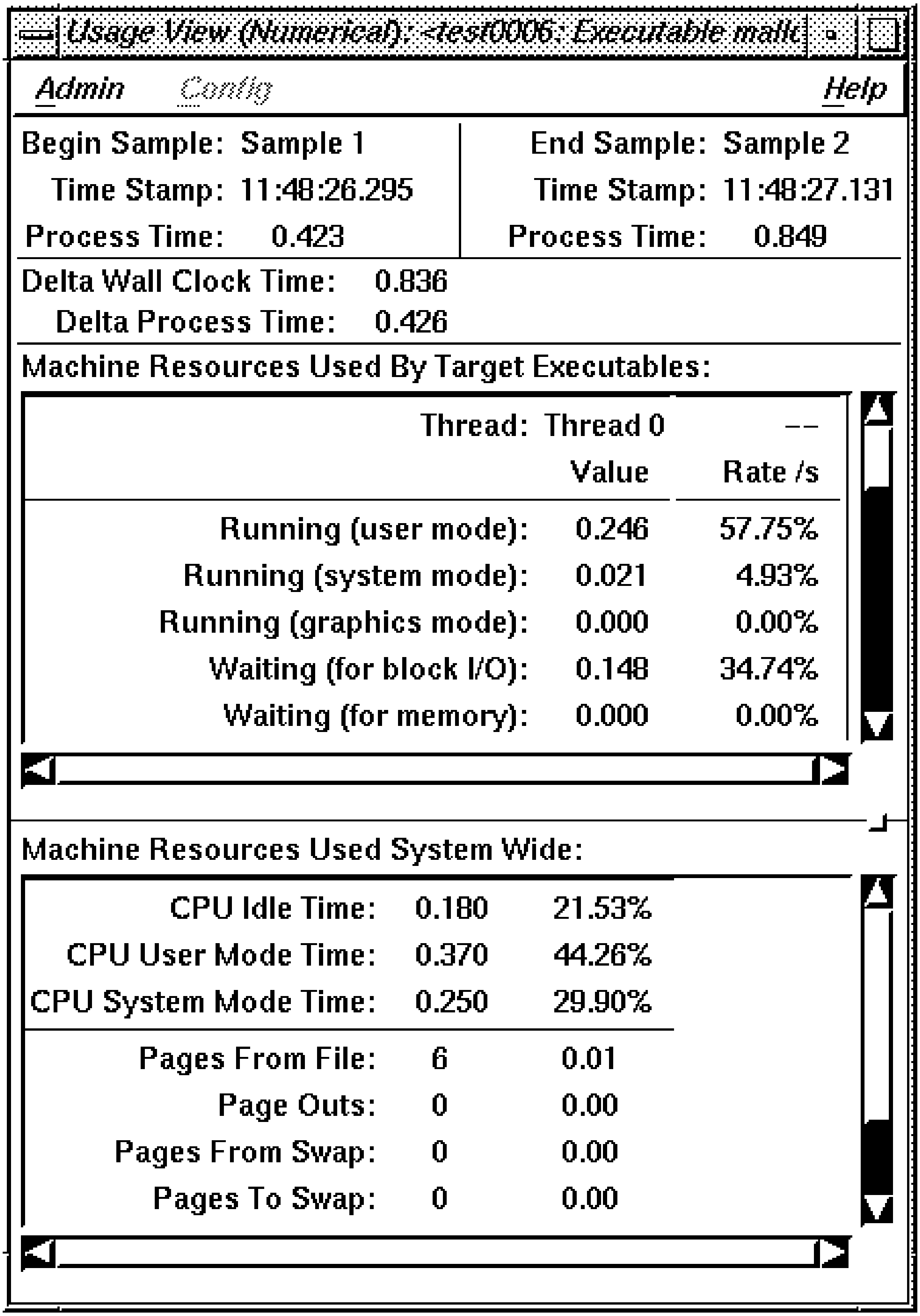


Figure 18–5 Typical Resource Usage Graphs

Textual Usage View

The usage graphs show the patterns; the textual usage views let you view the aggregate values for the interval specified ny the time line calipers. **Figure 18–6** shows a typical Textual Usage View window.



Begin Sample: Sample 1
Time Stamp: 11:48:26.295
Process Time: 0.423

End Sample: Sample 2
Time Stamp: 11:48:27.131
Process Time: 0.849

Delta Wall Clock Time: 0.836
Delta Process Time: 0.426

Machine Resources Used By Target Executables:

Thread:	Thread 0	--
	Value	Rate /s
Running (user mode):	0.246	57.75%
Running (system mode):	0.021	4.93%
Running (graphics mode):	0.000	0.00%
Waiting (for block I/O):	0.148	34.74%
Waiting (for memory):	0.000	0.00%

Machine Resources Used System Wide:

CPU Idle Time:	0.180	21.53%
CPU User Mode Time:	0.370	44.26%
CPU System Mode Time:	0.250	29.90%
Pages From File:	6	0.01
Page Outs:	0	0.00
Pages From Swap:	0	0.00
Pages To Swap:	0	0.00

Figure 18–6 Typical Textual Usage View

The Function List Area

The function list displays all functions in the source code, annotated by performance metrics and ranked by the criterion of your choice, such as counts or one of the time metrics. **Figure 18–7** is an example of the function list, ranked by exclusive CPU time (defined as the time this function spent in the CPU, excluding jumps to other blocks).

Figure 18–7 Typical Performance Analyzer Function List Area

You can configure how functions appear in the function list area by selecting "Preferences..." in the Config menu. It lets you select which performance metrics display, whether they display as percentages or absolute values, and the style of the function name. The "Sort..." selection in the Config menu lets you order the functions in the list by the selected metric. Both selections disable those metric selections that were not collected in the current experiment.

Call Graph View

In contrast to the function list which provides the performance metrics for functions, the call graph puts this information into context by showing you where the calls are made. The call graph displays functions as nodes and calls as arcs. The nodes are annotated with the performance metrics; the arcs come with counts by default and can include other metrics as well.

In **Figure 18–8** , for example, the inclusive time spent by the function *main* is 8.946 seconds. Its exclusive time was 0 seconds, meaning that the time was actually spent in called functions. *main* can potentially call five functions. Call Graph View indicates that in the experiment *main* actually only called two functions: *usrtime* which consumed 4.473 seconds and *sleep* which also consumed 4.473 seconds.

Inclusive time in usrtime

Inclusive time in main

Inclusive time in sleep

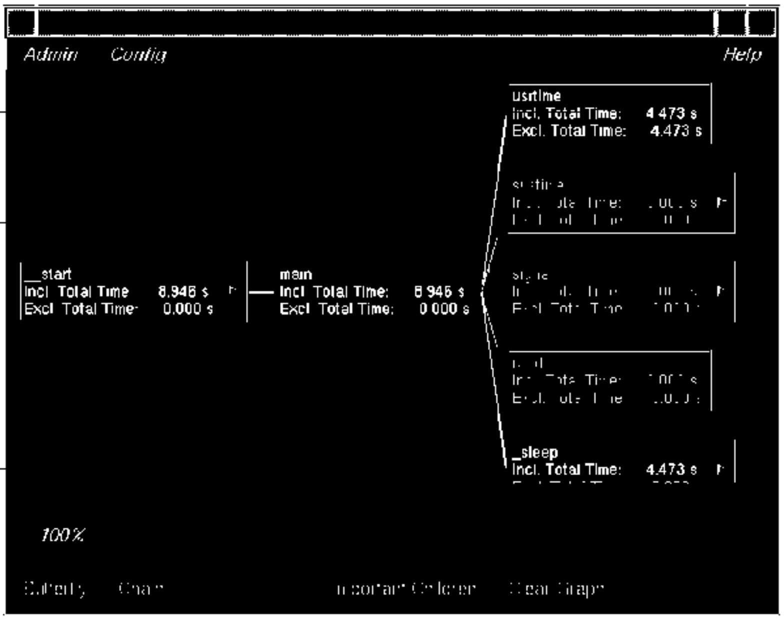


Figure 18–8 Typical Performance Analyzer Call Graph

Source View with Performance Annotations

The Performance Analyzer lets you view performance metrics by source line in Source View (see **Figure 18–9**) or by machine instruction in Disassembly View. Displaying performance metrics is set in the Preferences dialog box, accessed from the Display menu in Source View and Disassembly View. The Performance Analyzer sets thresholds to flag lines that consume more than 90% of a total resource. These indicators appear in the metrics column and on the scroll bar.

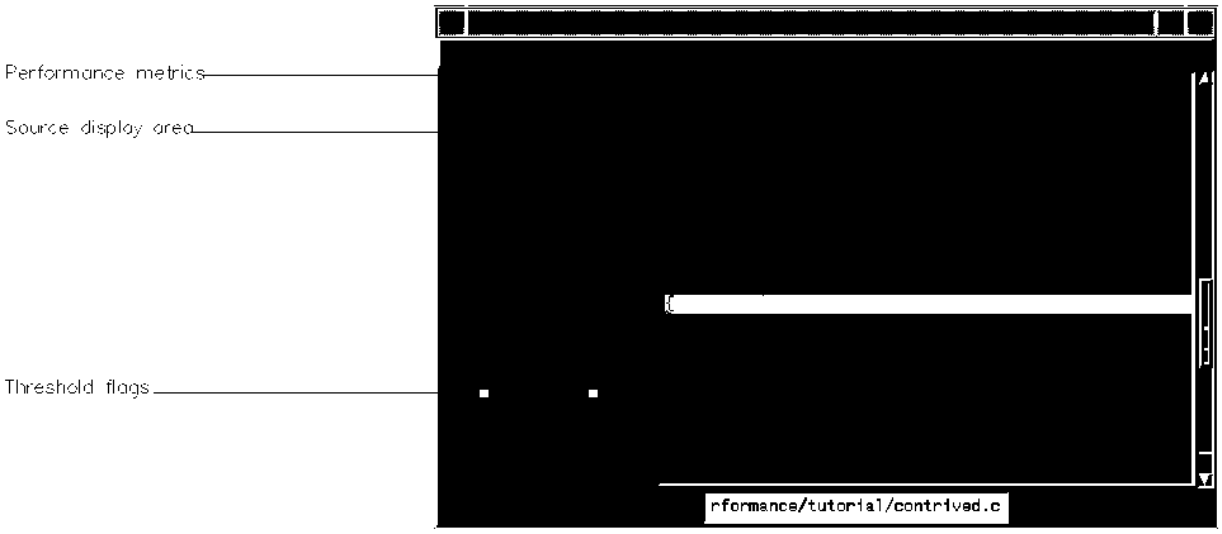


Figure 18–9 Detailed Performance Metrics by Source Line

Heap View, Leak View, and Malloc View

The Performance Analyzer lets you look for memory problems. The Heap View, Leak View, and Malloc View windows address two common types of memory problems that can inhibit performance:

- **"Memory Leakage"**
- **"Bad Frees"**

Memory Leakage

Memory leakage occurs when a program dynamically allocates memory and fails to deallocate that memory when it is through using the space. This causes the program size to increase continuously as the process runs. A simple indicator of this condition is the Process Size stripchart in Process View. The strip chart only indicates the size; it does not show the reasons for an increase.

Heap View displays a map of memory used in a defined interval (set by calipers) and flags all memory that was allocated by *malloc* or *realloc*. Leak View displays each memory leak in the executable, its size, the number of times the leak occurred at that location, and the corresponding call stack (when you select the leak). Malloc View operates in similar fashion to Leak View for *malloc*s occurring in the experiment.

A region allocated but not freed is not necessarily a leak. If the calipers are not set to cover the entire experiment, the allocated region may still be in use later in the experiment. In fact, even when the calipers cover the entire experiment, it is not necessarily wrong if the program does not explicitly free memory before exiting, since all memory is freed anyway on program termination.

The best way to look for leaks is to set sample points to bracket a specific operation that should have no effect on allocated memory. Then any area that is allocated but not freed is a leak.

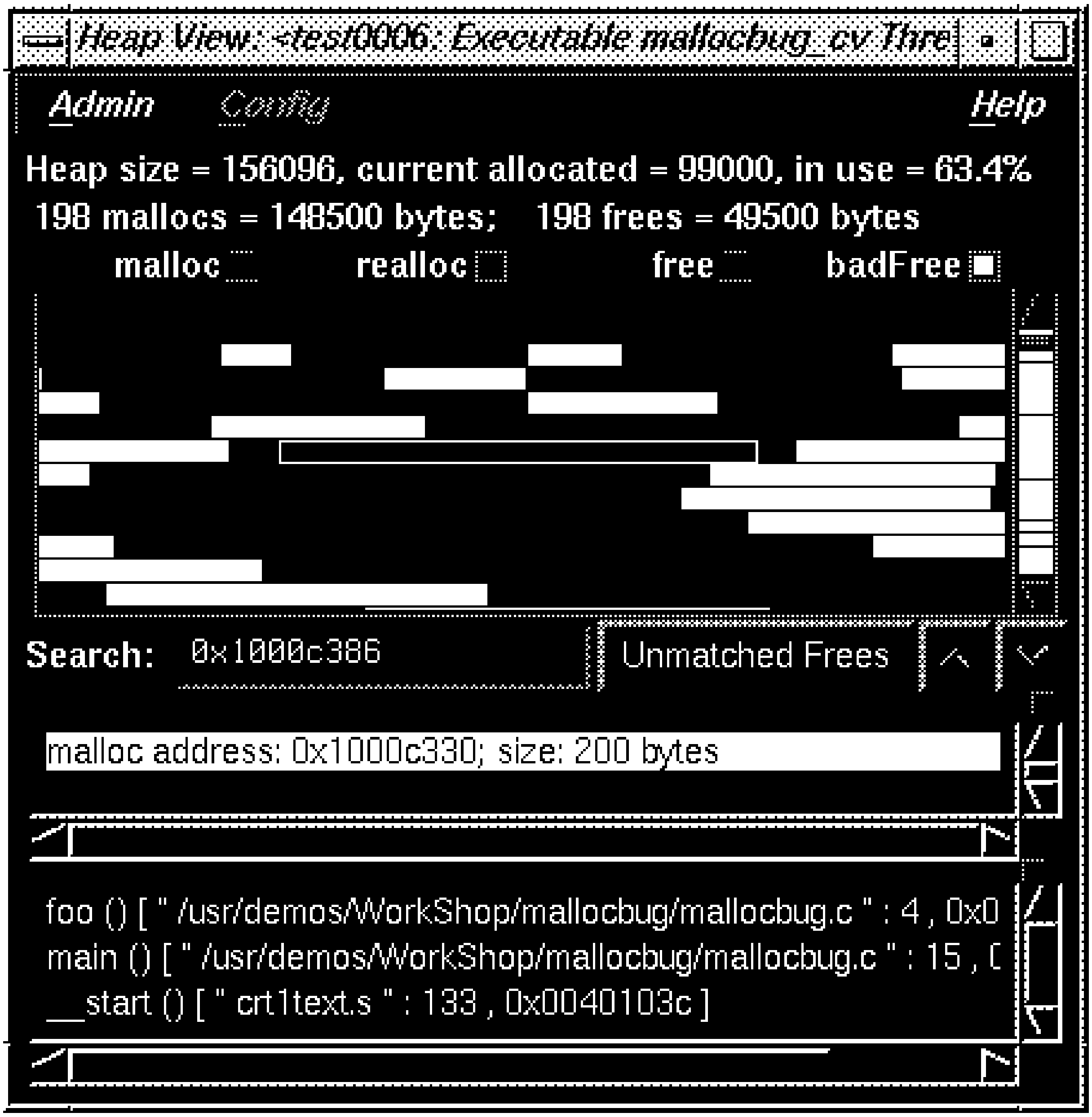
Bad Frees

A bad *free* (also referred to as an anti–leak condition) occurs when a program frees some structure that it had already freed. In many such cases, a subsequent reference picks up a meaningless pointer, causing a segmentation violation. Heap View identifies bad *frees* in its memory map display. It helps you find the address of the *freed* structure, search for the *malloc* event that created it, and the *free* event that released it. Hopefully, you can determine why it was prematurely freed or why a pointer to it was referenced after it had been freed.

Heap View also identifies unmatched *frees* in an information window. An unmatched *free* is a *free* that does not have a corresponding allocation in the same interval. As with leaks, the caliper settings may cause false indications. An unmatched *free* that occurs in any region not starting at the beginning of the experiment may not be an error. The region may have been allocated before the current interval and the unmatched *free* in the current interval may not be a problem after all. A segment identified as a bad *free* is definitely a problem; it has been freed more than once in the same interval.

A search facility is provided in *Heap View* that allows the user to find the allocation and deallocation events for all blocks containing a particular virtual address.

The Heap View window lets you analyze memory allocation and *frees* between selected sample events in your experiment. Heap View displays a memory map that indicates *mallocs*, *reallocs*, bad *frees*, and valid *frees* during the selected period, as shown in **Figure 18–10** . Clicking an area in the memory map displays the address.



```
Heap size = 156096, current allocated = 99000, in use = 63.4%
 198 mallocs = 148500 bytes;   198 frees = 49500 bytes
```

Search: 0x1000c386

```
malloc address: 0x1000c330; size: 200 bytes
```

```
foo () [ " /usr/demos/WorkShop/mallocbug/mallocbug.c " : 4 , 0x0
main () [ " /usr/demos/WorkShop/mallocbug/mallocbug.c " : 15 , 0
__start () [ " crt1text.s " : 133 , 0x0040103c ]
```

Figure 18–10 Typical Heap View Display Area

Call Stack View

The Performance Analyzer enables you to recall call stacks at sample events, which helps you reconstruct the calls leading up to an event so that you can relate the event back to your code. **Figure 18−11** shows a typical call stack. It corresponds to sample event #2 in an experiment.

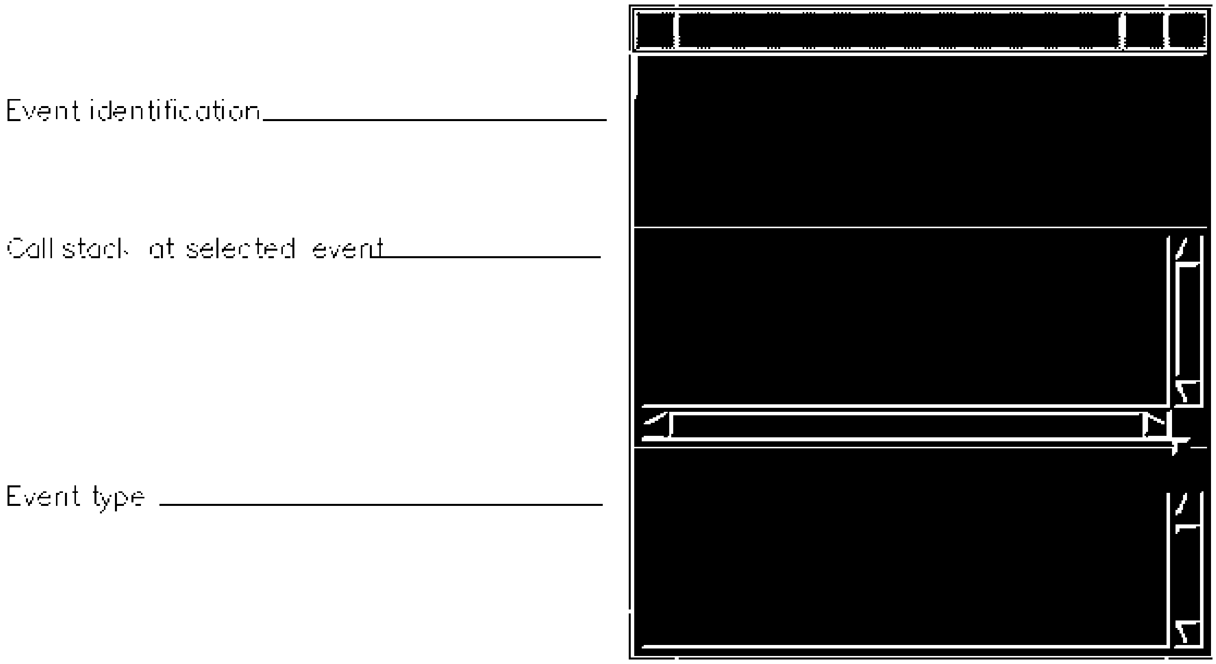
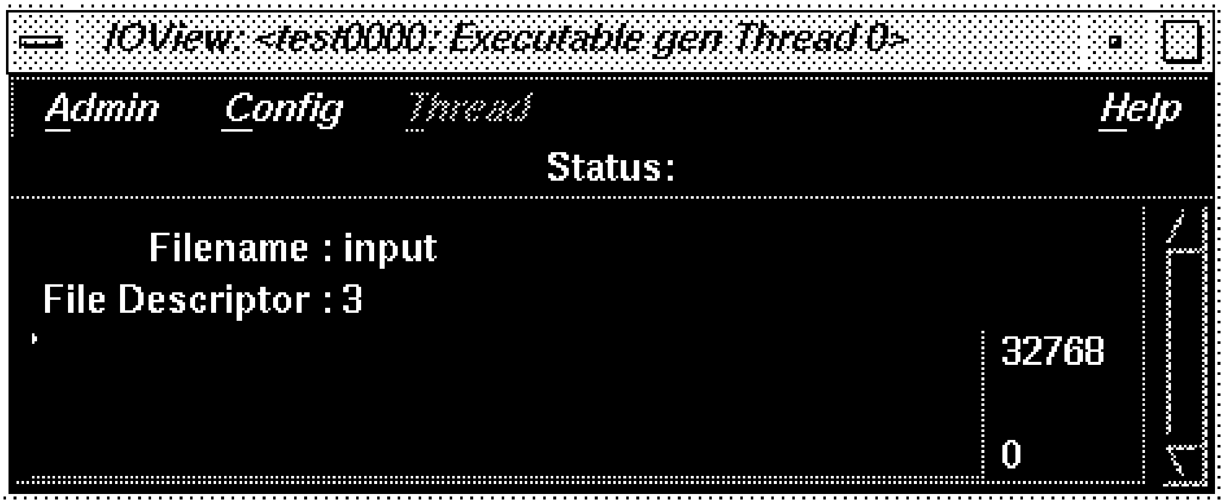


Figure 18−11 Typical Call Stack

I/O View

I/O View helps you determine the problems in an I/O−bound process. It produces a graph of all I/O system calls and identifies up to 10 files involved in I/O. See **Figure 18−12** .



Filename : input
File Descriptor : 3

Figure 18−12 I/O View

Chapter 19

Performance Analyzer Tutorial

This chapter presents a tutorial for using the Performance Analyzer and covers these topics:

- **"Tutorial Overview"**
- **"Tutorial Setup"**
- **"Analyzing the Performance Data"**

Note: Because of inherent differences between systems and also due to concurrent processes that may be running on your system, your experiment will produce different results from the one in this tutorial. However, the basic form of the results should be the same.

Tutorial Overview

This tutorial is based on a sample program called *arraysum.* The *arraysum* program goes through the following steps:

1. defines the size of an array (2,000 by 2,000)
2. creates a 2,000–by–2,000 element array, gets the size of the array, and reads in the elements
3. calculates the array total by adding up elements in each column
4. recalculates the array total differently, by adding up elements in each row

As you probably can already guess, it is more efficient to add the elements in an array row–by–row, as in step 4, than column–by–column, as in step 3. Because the elements in an array are stored sequentially by rows, adding the elements by columns potentially causes context switches, page faults, and cache misses. The tutorial shows you how you can detect symptoms of problems like this and then zero in on the problem. The source code is located in */usr/demos/WorkShop/performance/tutorial* if you wish to examine it.

Tutorial Setup

You need to compile the program first so that you can use it in the tutorial.

1. Change to the */usr/demos/WorkShop/performance/tutorial* directory.

   You can run the experiment in this directory or set up your own directory. You'll need the *arraysum.c* file in either case.

2. Compile the *arraysum.c* file by typing `make arraysum`

   This will provide you with an executable for the experiment.

3. From the command line, type `cvspeed arraysum&`

   Both the Debugger Main View window and the Performance Panel display. The Performance Panel lets you specify the data to be collected. You need the Debugger to run the experiment.

4. Make sure that the performance task "Identify bottleneck resources & phases" in the Performance Panel is selected and click *OK*.

   This is a general–purpose experiment that will help us determine the phases of the program and view basic resource usage. Clicking *OK* enables this task and closes the Performance Panel window.

5. Click *Run* in the Debugger Main View window.

   This starts the experiment. When the status line indicates that the process has terminated, the experiment has completed. The experiment may take one to three minutes, depending on your system.

Analyzing the Performance Data

Performance analysis experiments are set up and run from the Debugger window; the data is analyzed from the Performance Analyzer window.

1. Select "Launch Tool":"Performance Analyzer" from the Admin menu in the Debugger Main View window.

   The main Performance Analyzer window displays the information from the new experiment (see **Figure 19−1**).

2. Look at the Usage Chart in the Performance Analyzer window.

   There are three general phases. The first phase is I/O−intensive, as evidenced by the high system time. The middle phase takes up most of the experiment. We do not have enough information yet, however, to characterize it. The third phase shows high user time and is CPU−intensive.

3. Select "Usage View (Graphs)" from the Views menu.

   The Usage View (Graphs) window displays as in **Figure 19−2** . This indicates that there are significant page faults and context switches in the middle phase. It also shows high read activity and system calls in the first phase, confirming our hypothesis that it is I/O −intensive.

   As a side note, notice that the last chart indicates that the maximum total size of the process is reached at the end of the first phase and does not grow thereafter.

4. Select "Call Stack" from the Views menu.

   The call stack displays for the selected event. An event refers to a sample point on the time line (or any usage chart).

   At this point, no events have been selected so the call stack is empty. To select events, you can click in the time line or usage chart. You can also click the event selector controls to make one event at a time (see **Figure 19−1**).

   The call stack window indicates the state of the call stack when the event occurred. The significance of the call stack is that it lets you map events to the functions in which they occurred.

Task: Determine bottlenecks, identify phases

```
  Incl.      Excl.
  Total      Total
  (secs)     (secs)
  81.419      0.000    __start
  81.419      0.000    main
  62.459     62.459    sum1
  11.153      0.000    getArray
  10.038      0.000    fread
   7.807      7.807    sum2
   4.461      0.000    _findbuf
   4.461      3.346    __filbuf
```

Sample Event: 1   Time: 0.340 sec

Thread: 0

```
Begin: 1: Target Start: Thread 0 (sample only s
  End: 75: Entry to syscall exit: Thread 0 (all
Event: 1: Target Start: Thread 0 (sample only s
```

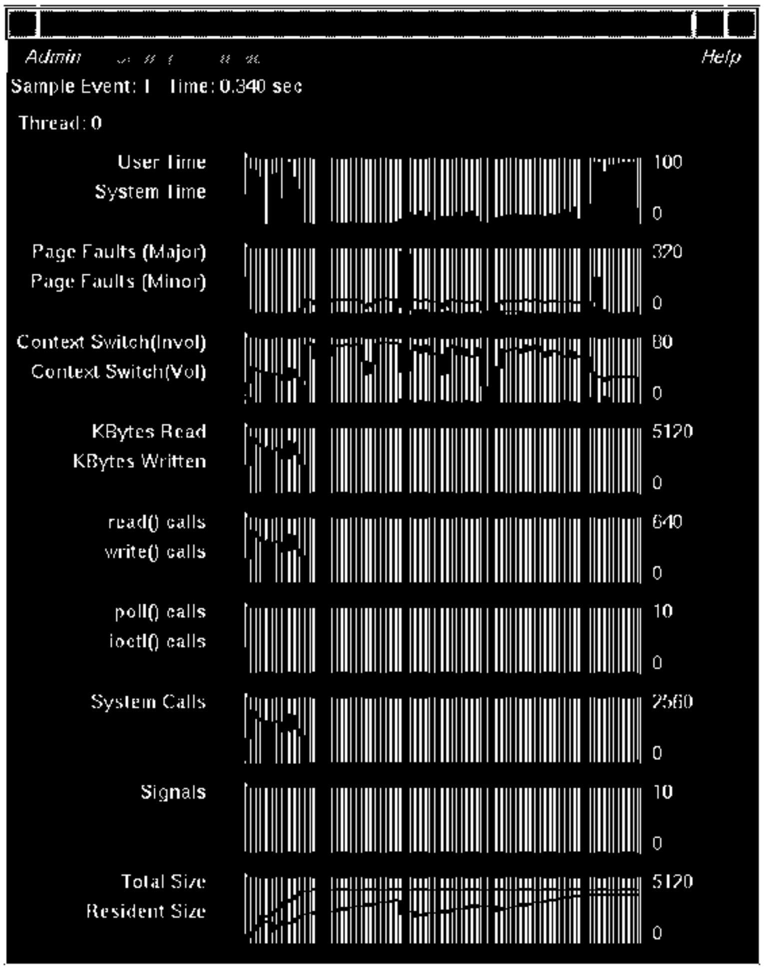
Figure 19–1 Performance Analyzer Main Window—*arraysum* Experiment

5. Select some random events and watch the call stack.

   This exercise helps you see the connection between events and call stacks.

Significant page faults

Significant context switches



Sample Event: 1 Time: 0.340 sec

Thread: 0

Figure 19–2 Usage View (Graphs)—*arraysum* Experiment

The important call stacks are the ones that occur at the beginning and end of phases. The general approach is to click in the vicinity of a usage chart where you think a phase boundary may occur and then check the call stack at that point. In this example, events #11, #12, #67, and #68 are important. The call stacks for these events are shown in **Figure 19–3** , which is drawn out–of–scale to illustrate the relationships. Remember that your results will be different.

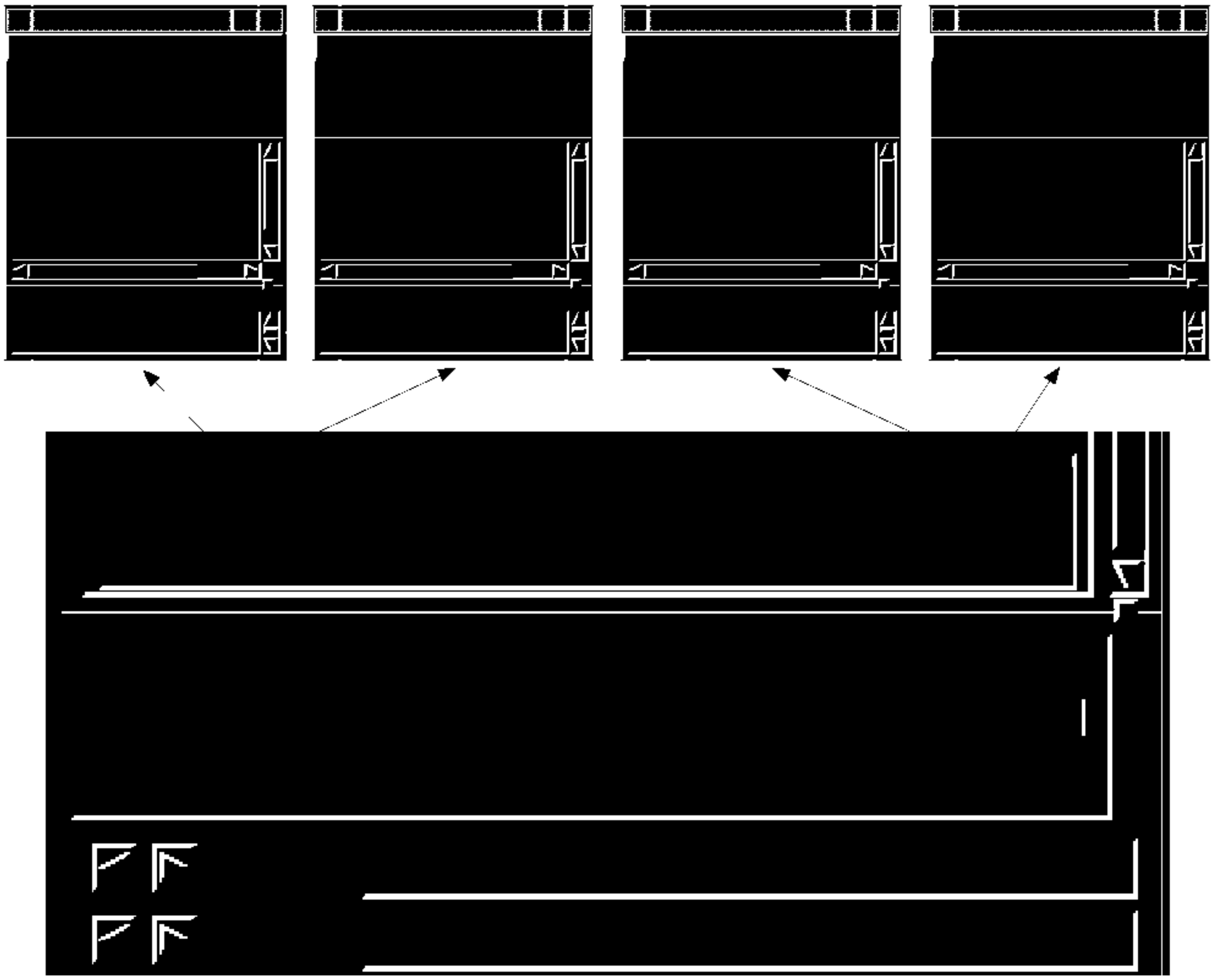
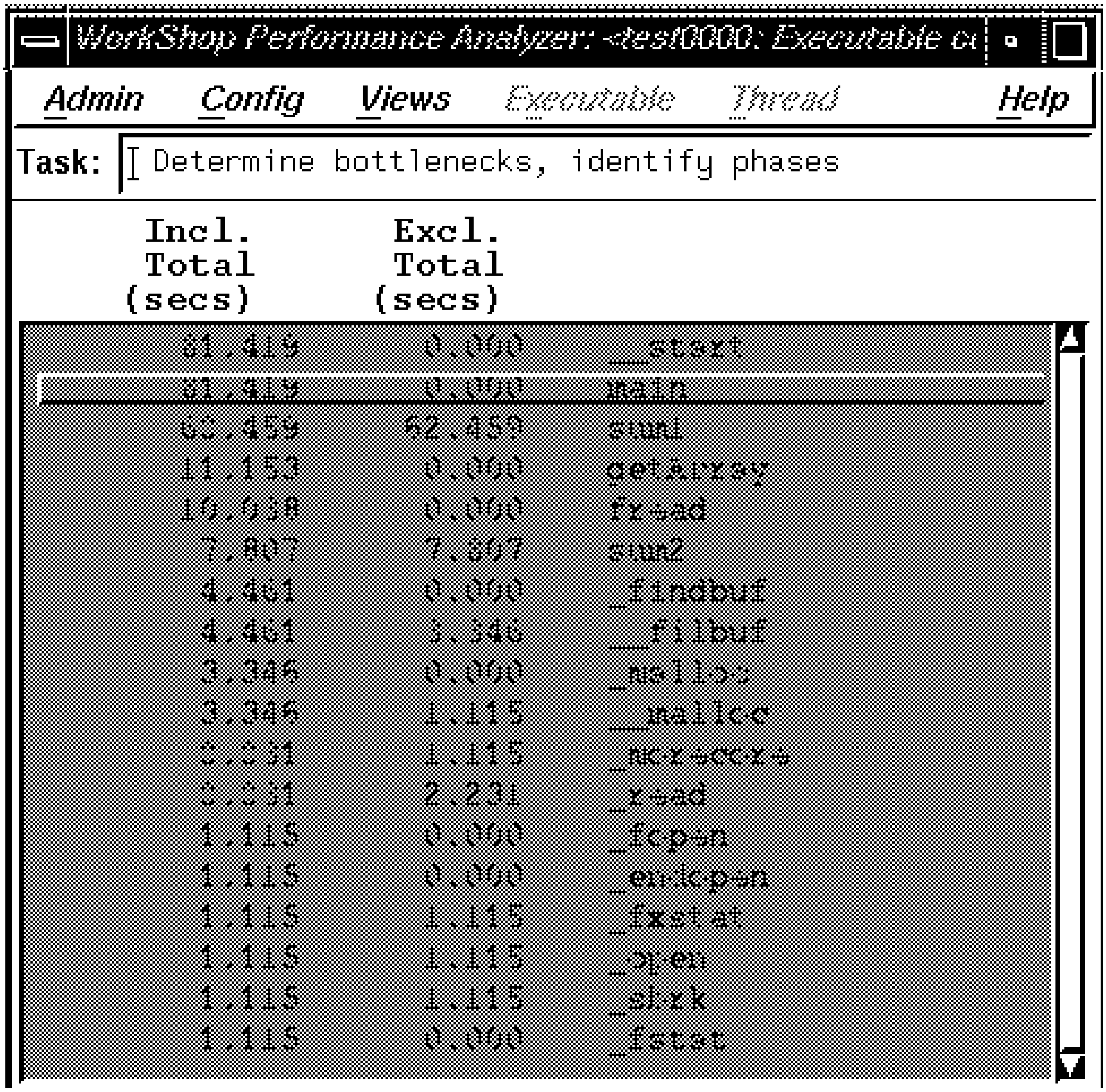


Figure 19–3 Significant Call Stacks in the *arraysum* Experiment

Event #11 is the last event in the first phase. Events #12 and #67 are the first and last events in the *sum1* function. Event #68 shows the switch from *sum1* to *sum2* and represents the beginning of the last phase. The length of time in *sum1* indicates potential problems.

6. Return to the Performance Analyzer window and pull down the sash to expose the complete function list.

   This shows the inclusive time (that is, time spent in the function and its called functions) and exclusive time (time in the function itself only) for each function. As you can see, 62.459 seconds are spent in *sum1* in contrast to 7.807 seconds in *sum2*.



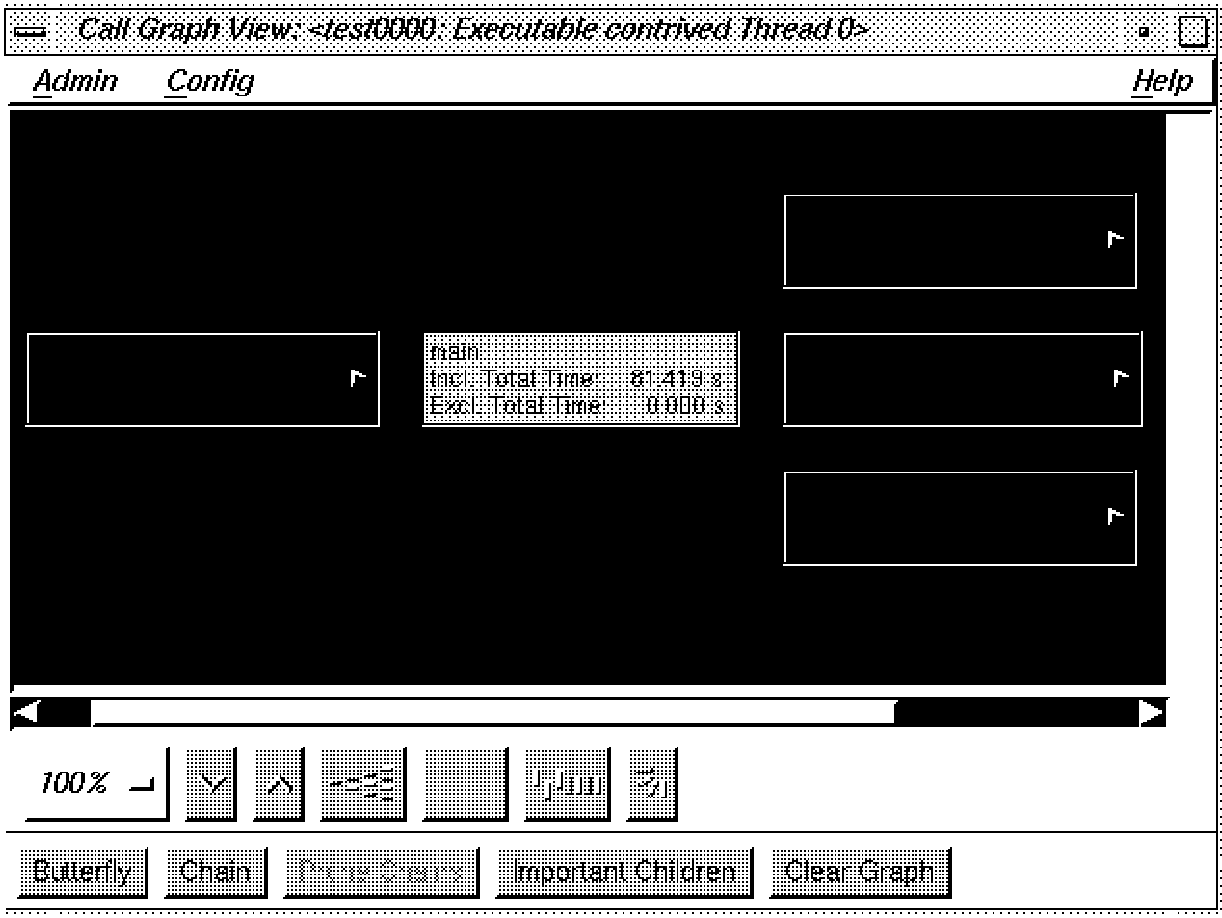
Task: Determine bottlenecks, identify phases

Incl. Total (secs)	Excl. Total (secs)	
81.419	0.000	__start
81.419	0.000	main
63.459	62.459	sum1
11.153	0.000	getarray
10.038	0.000	fread
7.807	7.807	sum2
4.461	0.000	_findbuf
4.461	3.346	__filbuf
3.346	0.000	_malloc
3.346	1.115	__malloc
2.231	1.115	_morecore
2.231	2.231	_read
1.115	0.000	_fopen
1.115	0.000	_endopen
1.115	1.115	_fxstat
1.115	1.115	_open
1.115	1.115	_sbrk
1.115	0.000	_fstat

Figure 19–4 Function List Portion of Performance Analyzer Window

7. Select "Call Graph View" from the Views menu and click the *Butterfly* button.

   The call graph provides an alternate means of viewing function performance data. It also shows the relationships, that is, which functions call which functions. After the *Butterfly* button is clicked, Call Graph View displays as in **Figure 19–5** . The *Butterfly* button takes the selected function (or most active function if none is selected) and displays it with the functions that call it and those that it calls. The Call Graph View provides still further proof that sum1 is consuming an inordinate amount of time.

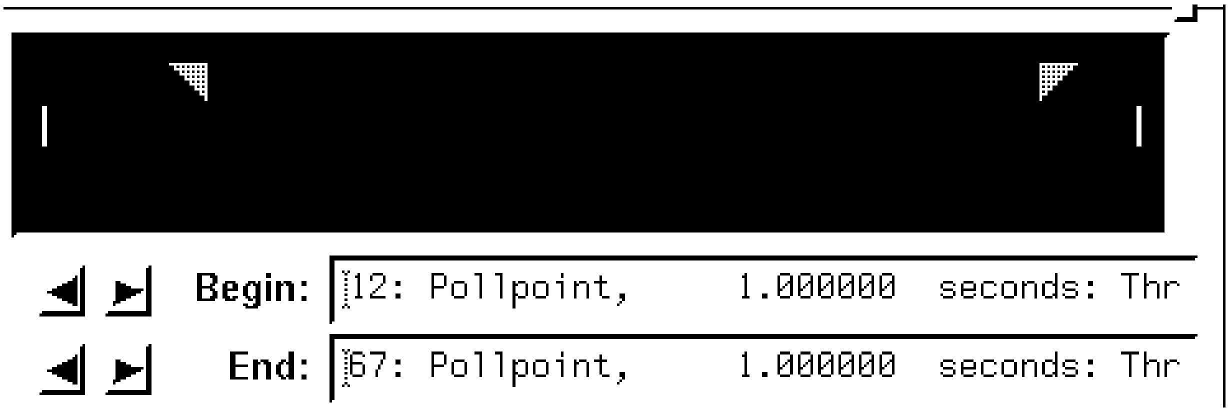


main
Incl. Total Time: 81.419 s
Excl. Total Time: 0.000 s

Figure 19–5 Call Graph View—*arraysum* Experiment

8. Select "Close" from the Admin menu in the Call Graph View to close it. Return to the main Performance Analyzer window and move the left caliper (*Begin*) to event #12 and the right caliper (*End*) to event #67.

   This is shown in **Figure 19–6** . Moving the calipers like this lets us focus on the data between event #12 and event #67.



Begin: 12: Pollpoint, 1.000000 seconds: Thr
End: 67: Pollpoint, 1.000000 seconds: Thr

Figure 19–6 Defining a Phase with Calipers—*arraysum* Experiment

9. Select "Usage View (Numerical)" from the Views menu.

The Usage View (Numerical) window displays as shown in **Figure 19–7** .

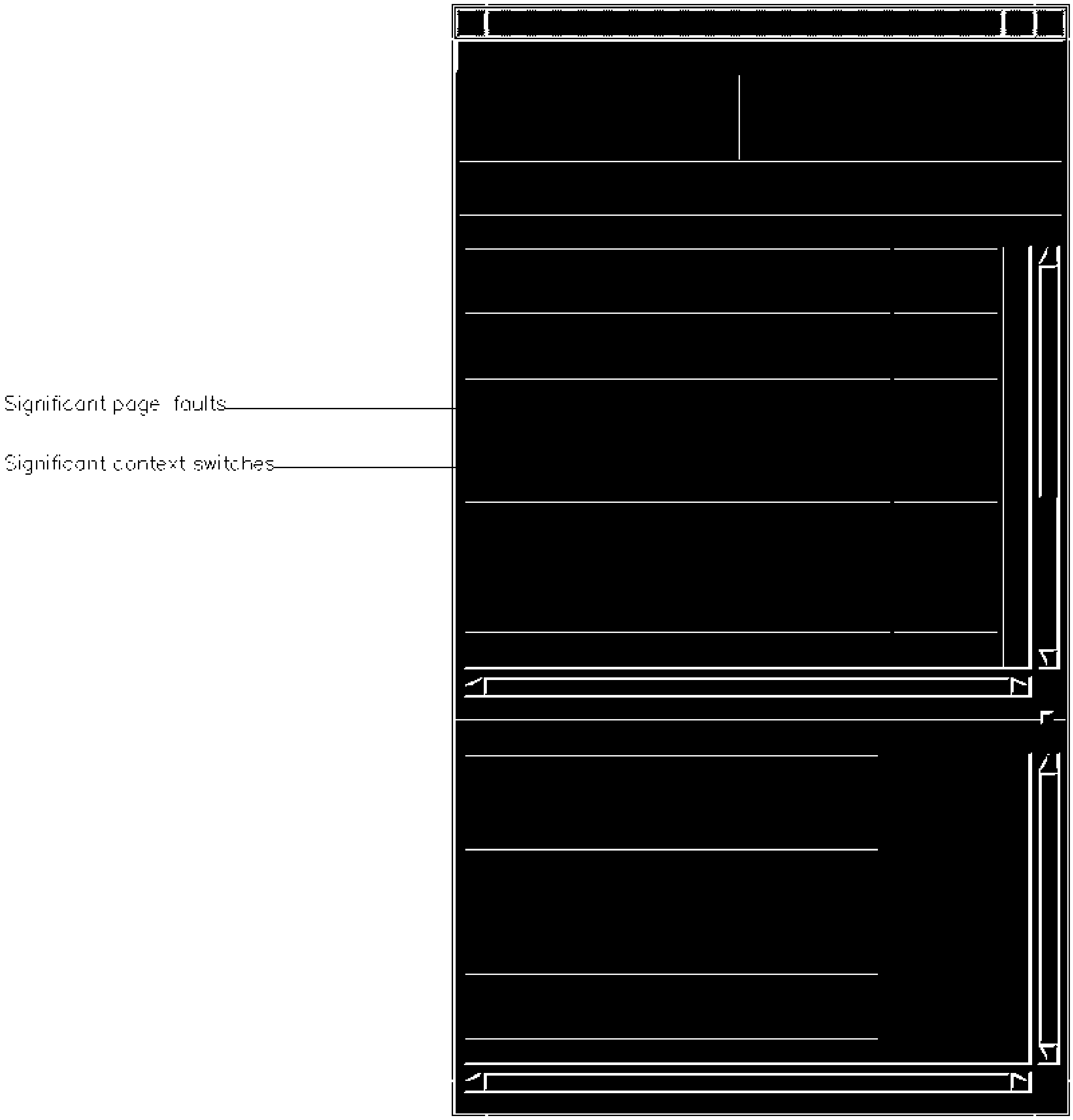


Figure 19–7 Viewing a Phase in the Usage View (Numerical)

This view provides the performance metrics for the interval defined by the calipers, in this case the *sum1* phase. Note the heavy page fault and context switch activity in this phase.

10. Return to the main Performance Analyzer window, select *sum1* from the function list, and click *Source*.

    The Source View window displays as in **Figure 19–8** , scrolled to *sum1*, the selected function. The annotation column to the left of the display area shows the performance metrics by line. Lines consuming more than 90% of a particular resource appear with highlighted annotations.

    Notice that the line where the total is computed in *sum1* is seen to be the culprit, consuming 61,727 milliseconds. As in the other WorkShop tools, you can make corrections in Source View, recompile and try out your changes.

Figure 19–8 Source View with Performance Metrics—*arraysum* Experiment

Note: At this point, we have uncovered the Performance problem, that the *sum1* algorithm is inefficient. As a side exercise, you may wish to take a look at the performance metrics at the assembly level. To do this, return to the main Performance Analyzer window, select *sum1* from the function list, and click *Disassembled Source*. Disassembly View displays, with the performance metrics in the annotation column.

11. Close any windows that are still open.

This concludes the tutorial.

Chapter 20

Setting Up Performance Analysis Experiments

In performance analysis, you set up the experiment, run the executable, and analyze the results. To make setup easier, the Performance Analyzer provides predefined tasks that help you establish an objective and ensure that the appropriate performance data will be collected. This chapter tells you how to conduct performance tasks and what to look for.

It covers these topics:

- **"Experiment Setup Overview"**
- **"Using the Performance Panel"**
- **"Setting Sample Traps"**
- **"Understanding Predefined Tasks"**

Experiment Setup Overview

Performance tuning typically consists of examining machine resource usage, breaking down the process into phases, identifying the resource bottleneck within each phase, and correcting the cause. Generally, you run the first experiment to break your program down into phases and run subsequent experiments to examine each phase individually. After you have solved a problem in a phase, you should then reexamine machine resource usage to see if there is further opportunity for performance improvement.

Each experiment has four steps:

1. Specify the performance task.

   The Performance Analyzer provides *predefined tasks* for conducting experiments. When you select a task, the Performance Analyzer automatically enables the appropriate performance data items for collection.

   You should have an objective in mind when you start an experiment. The predefined tasks ensure that only the appropriate data collection is enabled. Selecting too much data can bog down the experiment and skew the data for collection. If you need a mix of performance data not available in the predefined tasks, you can select "Custom Task" from the task menus, which lets you enable any combination of the data collection options.

2. Specify where data is to be recorded.

   If you have selected the "Identify bottleneck resources & phases" task, which automatically polls for performance data, this step is not needed. If you want data at specific points in the process, you need to set sample traps. See **"Setting Sample Traps"** for a brief description of traps or **Chapter 12, "Debugger: Setting Traps in WorkShop"** , for an in–depth discussion.

   Performance Analyzer sets sample traps at the beginning and end of the process automatically. If you want to analyze data within phases, then you should set sample traps at the beginning of each phase and at intermediate points, if desired.

3. Run the program to collect the data.

You run the experiment from the Debugger Main View window. If you are running a small experiment to capture resource usage, you may be able to watch the experiment in real time in Process Meter. Performance Analyzer stores the results in the designated experiment subdirectory.

4. Analyze the results.

   After the experiment completes, you can look at the results in the Performance Analyzer window and its associated views. Use the calipers to get information for phases separately.

Using the Performance Panel

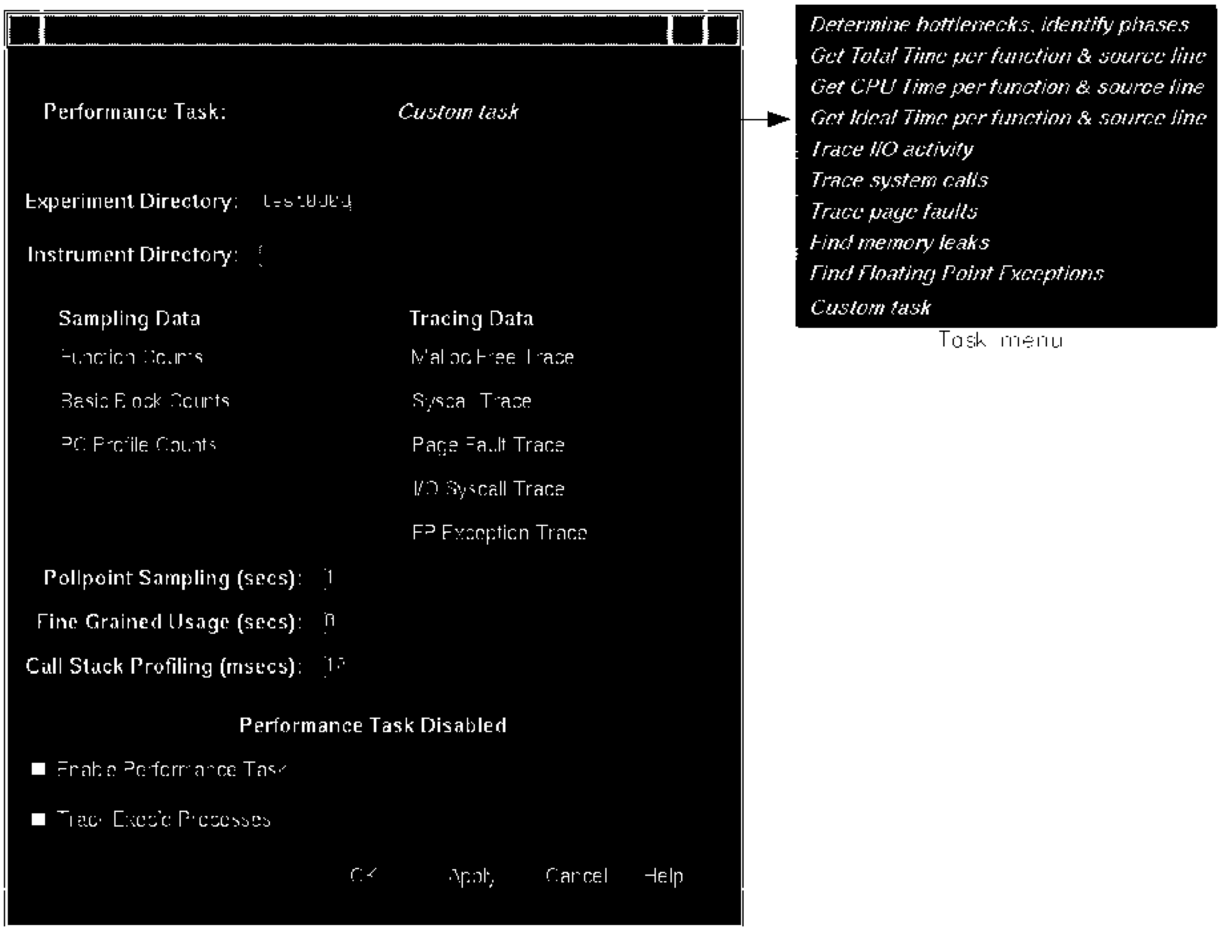
To set up a Performance Analyzer experiment, you need the Performance Panel window. You can access the Performance Panel by selecting "Performance Task..." from the Admin menu in Main View or from a command line, by typing:

```
cvspeed [executable]
```

If you invoke *cvspeed* from the command line, the Debugger Main View window will display as well. A typical Performance Panel window is shown in **Figure 20–1** . The Performance Panel lets you designate:

- the performance analysis task—a feature that selects the appropriate data to be collected automatically
- the experiment directory—for holding the performance data for a specific experiment
- the instrument directory—for reusing a previously instrumented executable
- sampling and tracing data—the data to be recorded (not necessary if you use the task defaults)
- recording intervals—the frequency of data recording (not necessary if you use the task defaults)
- *exec*'d process inclusion—for including processes *exec*'d from the target process for performance analysis

You use the Performance Panel to enable data collection. When you have made your selections, you click *OK* to enable them and close the window or *Apply* to enable the selections and leave the Performance Panel window open.



Performance Task: Custom task

Experiment Directory: test0004

Instrument Directory:

Pollpoint Sampling (secs): 1

Fine Grained Usage (secs): 0

Call Stack Profiling (msecs): 10

Performance Task Disabled

■ Enable Performance Task

■ Trace Exec'd Processes

Figure 20–1 Performance Panel Window with Task Menu

Setting Sample Traps

For a thorough discussion of setting traps, refer to **Chapter 12, "Debugger: Setting Traps in WorkShop"** . Sample traps enable you to record data when a specified condition occurs. You set them from the Debugger Main View, Trap Manager, or Source View. You can define sample traps:

- at function entry or exit points
- at source lines
- for events
- conditionally
- manually during an experiment

Sample traps at function entry and exit points are preferable to source line traps, because they are more likely to be preserved as your program evolves. This better enables you to save a set of traps in the Trap Manager in a file for subsequent reuse.

Manual sample traps are triggered when you click the *Sample* button in the Debugger Main View. They are particularly useful for applications with graphical user interfaces. If you have a suspect operation in an experiment, a good technique is to take a manual sample before and after you perform the operation. You can then examine the data for that operation.

Understanding Predefined Tasks

If you are unfamiliar with performance analysis, it is very easy to request more data collection than you actually need—this can degrade performance of the Performance Analyzer and skew results. To help you record data appropriate to your current objective, the Performance Panel provides predefined combinations of options, which are available in the *Performance Task* menu. When you select a task, the required data collection is automatically enabled.

"Determine bottlenecks, identify phases"

"Determine bottlenecks, identify phases" measures machine resource usage and takes pollpoint samples at 1−second intervals.

Call stack data is captured at each pollpoint sample to compute the total time for each function and source line. In call stack profiling, the time spent at a PC (program counter) is determined by multiplying the number of times the PC appears in any call stack by the average time interval between call stacks. Call stacks are gathered whether the program was running or blocked; hence, the time computed represents the total time, both within and outside of the CPU. If the target process was blocked for a long time as a result of an instruction, that instruction will show up as having a high time.

Gathering machine resource usage data lets you observe resource consumption over time. With it, you can break your program down into phases with similar resource consumption. You can analyze individual phases in detail in subsequent experiments. You can view resource usage in Usage View (Graphical), Usage View (Numerical), and in the Usage Chart in the Performance Analyzer main window.

Figure 20−2 shows a typical example of this task. A resource usage graph has been created, showing user vs. system time. The time line is below the resource graph; it has a time scale so that you can correlate the resource usage with experiment time.

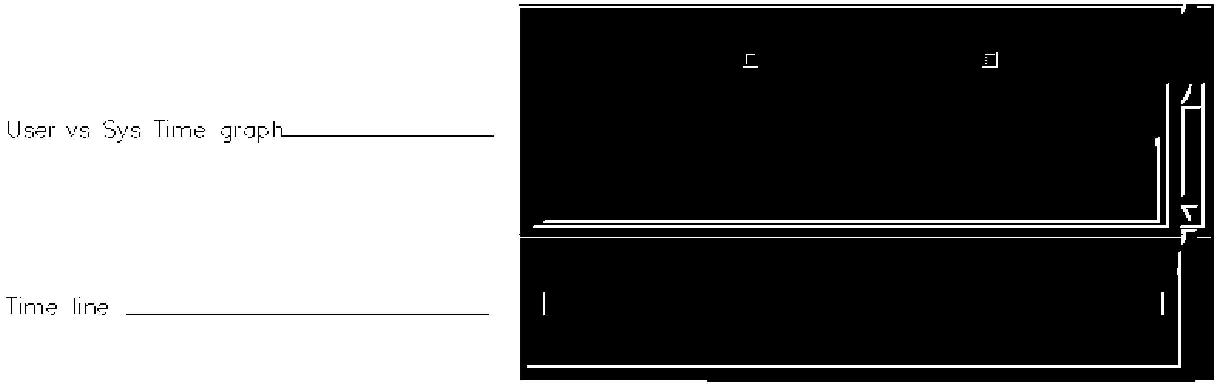


Figure 20−2 Machine Resource Usage in Performance Analyzer Window

"Get Total Time per function & source line"

Use "Get Total Time per function & source line" to tune a phase that has been determined not to be CPU−bound. This task records:

- call stacks every 10 ms, whether the target program is running or blocked
- machine resource usage data at 1−second pollpoints and at sample points

The Total Time values for the PCs are summed up and displayed:

- by function in the function list
- by source line in Source View
- by instruction in Disassembly View

Note that the call stack profiling in "Get Total Time per function & source line" is taken every 10 ms in contrast to one−second intervals for "Determine bottlenecks, identify phases". The 10−ms call stack profiling degrades performance in a way that makes it inappropriate for investigating CPU−bound situations. Gathering call stacks requires stopping the target process temporarily to recover and write the call stack. These context switches are highly intrusive, slowing the execution of the program and causing a different set of machine resources to be consumed. Note also that although the data collection process attempts to collect call stacks every 10 ms, in reality this occurs less often due to the overhead involved.

"Get CPU Time per function & source line"

Use "Get CPU Time per function & source line" to tune a CPU–bound phase. It enables you to display the time spent in the CPU by function, source line, and instruction. This task records:

- PC every 10 ms
- function counts
- machine resource usage data at 1–second intervals and at sample points

The CPU time is calculated by multiplying the number of times a PC appears in the profile by 10 ms. PCs are profiled only when the program is running in the CPU; hence, the time computed is the time spent within the CPU, or the CPU time.

If the target process was blocked for a long time as a result of an instruction, that instruction will show up as having a low or zero CPU time. On the other hand, CPU–intensive instructions will show up as having a high CPU time.

The CPU time values for the PCs are summed up and displayed:

- by function in the function list
- by source line in Source View
- by instruction in Disassembly View

Function count data is computed by inserting machine code that increments a counter at the start of the function (this is called *instrumentation*). This data is used in the function list to show how many times the function was called and also in the call graph to show how many times one function called another function, that is, arc counts.

PC profiling is done by the kernel and is only minimally intrusive. Gathering function counts intrudes substantially due to:

- instrumentation code consuming CPU cycles
- instrumentation code increasing the target executable size
- PC profiling of the instrumented code itself, which distorts the metrics

However, function counts are useful when combined with PC profiling, because they help in the computation of Inclusive CPU times. (Inclusive CPU time is the total time spent in a function and all the functions it calls; exclusive CPU time is the time spent in the function only.) The arc counts indicate what percentage of a function's CPU time can be attributed to each of its callers.

If you only need Exclusive CPU times and are willing to forgo Inclusive CPU times, arc counts, and function count information, you should select "Custom task" and enable *PC Profile Counts* and set *Fine–Grained Usage* to 1 second.

Also look at the task "Get Ideal Time per function & source line".

"Get Ideal Time per function & source line"

Use "Get Ideal Time per function & source line" to tune a CPU–bound phase. This task provides exact counts with theoretical times. It is very useful when used in conjunction with the "Get CPU Time per function & source line" task. This approach lets you examine actual versus ideal time. The difference is the time spent as a result of:

- *load* operations, which take a minimum of two cycles if the data is available in the cache and a lot longer if the data has to be accessed from the swap area or second–level cache
- *store* operations, which cause the CPU to stall if the write buffer in the CPU gets filled
- floating point operations, which consume more than one cycle
- time spent with the CPU stalled as a result of data dependencies

This task records:

- basic block counts
- machine resource usage data at 1–second intervals and at sample points

The following results are shown in the function list, Source View, and Disassembly View:

- execution counts
- resulting machine instructions

- a count of resulting loads, stores, and floating point instructions
- an approximation of the time spent with the CPU stalling (caused by data interlocks)
- the ideal time, that is, the product of the number of the machine instructions executed and the cycle time of the machine (The assumption made in the computation of ideal time is that each instruction takes exactly one cycle to execute.)

This task requires instrumentation of the target executable. This involves dividing the code into basic blocks, which are a set of instructions with a single entry point, a single exit point, and no branches within. Counter code is inserted at the beginning of each basic block.

After the instrumented executable runs, the Performance Analyzer multiplies the number of times a basic block was executed by the number of instructions in it. This yields the total number of instructions executed as a result of that basic block (and similarly for specific kinds of instructions like loads or stores).

Note that the execution of the instrumentation code will skew the behavior of the target executable making it almost entirely CPU−bound; so pay no attention to the User vs Sys Time stripcharts.

"Trace I/O activity"

Use "Trace I/O activity" when your program is being slowed down by I/O calls and you want to find the responsible code. This task records call stacks at every *read* and *write* system call, along with file descriptor information, and the number of bytes read or written.

The number of bytes read and written is presented:
- by function in the function list
- by source line in Source View
- by instruction in Disassembly View

The I/O View window displays a graph of the number of bytes read and written for each file descriptor over time, and displays the files involved in the I/O. You can also see the *read* and *write* system calls.

"Trace system calls"

Use "Trace system calls" when you suspect that system calls are slowing down performance and you wish to determine the responsible code.

The number of system calls made is presented:
- by function in the function list
- by source line in Source View
- by instruction in Disassembly View
- To observe the pattern of system calls over time, look in the syscall event chart of the Usage View (Graphical).

"Trace page faults"

The "Trace page faults" task indicates areas of high page faulting activity and identifies the code responsible. The task records call stacks at every page fault.

The number of page faults is presented:
- by function in the function list
- by source line in Source View
- by instruction in Disassembly View

To observe the pattern of page faulting over time, look in the page fault event chart of the Usage View (Graphical).

"Find memory leaks"

Use "Find memory leaks" to determine where memory leaks and bad *frees* may occur in a process. The task records the call stacks, address, and number of bytes at every *malloc*, *realloc*, and *free*. The currently *malloc*ed bytes (that might represent leaks), and the list of double *frees* are presented in Heap View. The number of bytes *malloc*ed is presented:
- by function in the function list

- by source line in Source View
- by instruction in Disassembly View

"Find Floating Point Exceptions"

Use "Find Floating Point Exceptions" when you suspect that large, unaccountable periods of time are being spent in floating point exception handlers. The task records the call stack at each floating point exception. The number of floating point exceptions is presented:

- by function in the function list
- by source line in Source View
- by instruction in Disassembly View

To observe the pattern of floating point exceptions over time, look in the floating point exceptions event chart in the Usage View (Graphical).

"Custom task"

Use the "Custom task" selection when you need a combination of performance data collected that is not available through the predefined tasks. Remember the basic warnings in this chapter about collecting data:

- Too much data can bog down the experiment.
- Combining PC profiling and basic block counting will cause the instrumented code to be profiled, including the count code.
- Call stack profiling is not compatible with count operations or PC profiling.
- If you combine count operations with PC profiling, the results will be skewed due to the amount of instrumented code that will be profiled.

Chapter 21

Performance Analyzer Reference

This chapter provides detailed descriptions of the Performance Analyzer toolset, including:

- **"The Performance Panel"**
- **"The Performance Analyzer Main Window"**
- **"Usage View (Graphs)"**
- **"Process Meter"**
- **"Usage View (Numerical)"**
- **"I/O View"**
- **"Call Graph View"**
- **"Finding Memory Problems with Leak View, Malloc View, and Heap View"**
- **"Call Stack"**

The Performance Panel

The Performance Panel is used to set up performance analysis experiments. This section covers:

- **"Accessing the Performance Panel"**
- **"The Performance Panel Window"**
- **"Specifying Performance Tasks"**
- **"Specifying the Experiment Directory"**
- **"Specifying the Instrument Directory"**
- **"Specifying Sampling Data"**
- **"Specifying Tracing Data"**
- **"Specifying Polling Data"**
- **"Enabling Performance Data Collection"**

Accessing the Performance Panel

The specifications for a performance analysis experiment are made in the Performance Panel window.

You can access the Performance Panel by selecting "Performance Task..." from the Admin menu in Main View or from a command line, by typing:

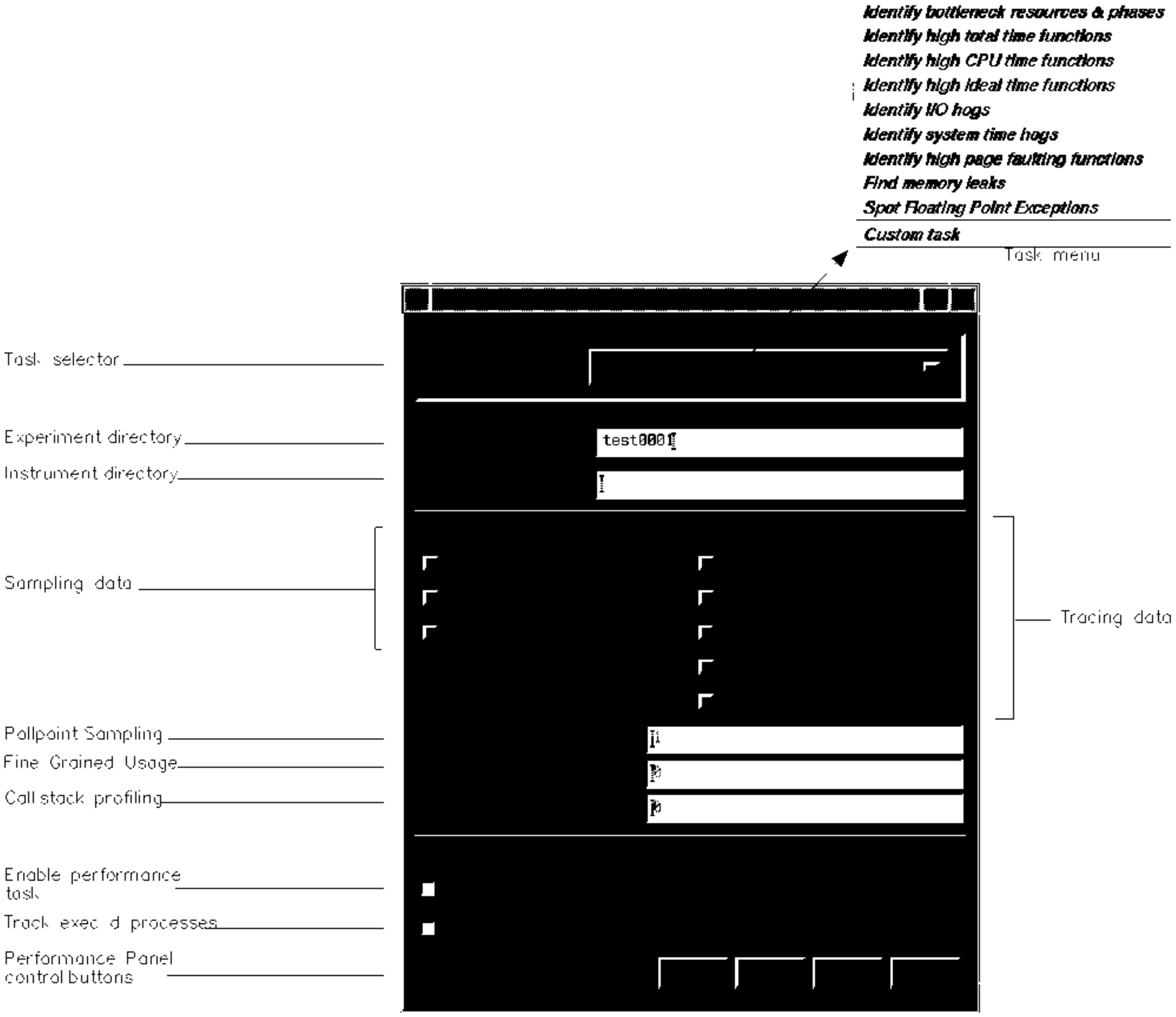
```
cvspeed [executable]
```

Invoking *cvspeed* from the command line also displays the Debugger Main View, which you need to run the experiment.

The Performance Panel Window

A typical Performance Panel appears in **Figure 21–1** . The Performance Panel lets you designate:

- the performance analysis task—a feature that selects the appropriate data to be collected automatically
- the experiment directory—for holding the performance data for a specific experiment



test0001

Figure 21–1 Performance Panel Window with Task Menu

- the instrument directory—for reusing a previously instrumented executable
- sampling and tracing data—the data to be recorded (not necessary if you use the task defaults)
- recording intervals—the frequency of data recording for pollpoint samples, fine–grained usage, and call stack profiling (not necessary if you use the task defaults)
- *exec*'d process inclusion—for including processes *exec*'d from the target process for performance analysis, this activates the Executable menu in the Performance Analyzer main window, causing it to display the resulting executables

In addition, you use the Performance Panel to enable data collection through the *Enable Performance Task* button. When you have made your selections, you click *OK* or *Apply*.

Specifying Performance Tasks

You should have an objective in mind before you start an experiment. The predefined tasks ensure that only the appropriate data collection is enabled. Selecting too much data can bog down the experiment and skew the data for collection.

The predefined tasks are summarized in **Table 21–1** . The Task column identifies the task as it appears in the Performance Task menu in the Performance Panel window. The Clues column provides an indication of symptoms and situations appropriate for the task. The Data Collected column indicates performance data set by the task. Note that call stacks are collected automatically at sample points, pollpoints, and process events. The Description column describes the technique used.

Table 21–1 Summary of Performance Analyzer Tasks

Task	Clues	Data Collected	Description
Determine bottlenecks, identify phases	Slow program, nothing else known	Pollpoint Sampling (1 sec.) Call Stack Profiling (10 msec.) call stacks at sample points	Captures resource usage at the pollpoint sample and displays it in resource usage graphs. Minimal intrusion. Tracks the total time spent by function,

			source code line, and instruction.
Get Total Time per function & source line	Not CPU−bound	Fine−Grained Usage (1 sec.) Call Stack Profiling (10 msec.) call stacks at sample points	Tracks the total time spent by function, source code line, and instruction. Useful for non−CPU−bound conditions. Total time metrics are displayed.
Get CPU Time per function & source line	CPU−bound	Function Counts PC Profile Counts Fine−Grained Usage (1 sec.) call stacks at sample points	Tracks CPU time spent in functions,

			source code lines, and instructions. Useful for CPU-bound conditions. CPU time metrics help you separate CPU-bound from non-CPU-bound instructions.
Get Ideal Time per function & source line	CPU-bound	Basic Block Counts Fine-Grained Usage (1 sec.) call stacks at sample points	Calculates the ideal time, that is, the time spent in each basi

c block with the assumption of one instruction per machine cycle. Useful for CPU–bound conditions. Ideal time metrics also give counts, total machine instructions, and loads/stores/floating point instructions. It is useful to com

			pare ideal time with the CPU time in an "Identify high CPU time functions" experiment.
Trace I/O activity	Process blocking due to I/O.	I/O System call Trace Fine−Grained Usage (1 sec.) call stacks at sample points	Captures call stacks at every *read* and *write*. The file description and number of bytes are available in I/O View.
Trace system faults	Resource usage chart shows high system calls.	System call Trace FP Exception Trace Fine−Grained Usage (1 sec.) call stacks at sample points	Records all syst

em calls and corresponding call stacks. Gives system call counts for the functions, source code lines, and instructions making the system call. Also provides a strip chart showing the chronological sequence of system calls.

Trace page faults	"Noisy disk" due to accesses	Page Fault Trace Fine−Grained Usage (1 sec.) call stacks at sample points	Captures all page faults and corresponding call stacks. Produces event chart showing the page fault pattern. Lists page faults caused by function, source code line, and instruction.
Find memory leaks	Swelling in process size	Malloc/Free Trace Fine−Grained Usage (1 sec.) call stacks at sample points	Determines memory leaks by capt

			uring the call stack, address, and size at all *mallocs*, *reallocs*, and *frees* and displays them in a memory map. Also indicates double *frees*.
Find Floating Point Exceptions	High sys time in usage charts; presence of floating point operations; NaNs	FPE Exception Trace Fine−Grained Usage (1 sec.) call stacks at sample points	Useful when you suspect that time is being wasted in floating point exception

		handlers. Captures the call stack at each floating point exception. Lists floating point exceptions by function, source code line, and instruction.
Custom task	call stacks at sample points user's choice	Enables you to create your own combination of data to be collected. Re

member that too much data can skew results.

Specifying the Experiment Directory

The *Experiment Directory* field lets you specify the directory where the data captured during the next experiment is stored. The Performance Analyzer provides a default directory named *test0000*. If you use the default or any other name that ends in four digits, the four digits are used as a counter and will be incremented automatically for each subsequent experiment. Note that the Performance Analyzer does not remove (or overwrite) experiment directories. You need to remove directories yourself.

Specifying the Instrument Directory

The *Instrument Directory* lets re−use a previously instrumented executable. This technique avoids the processing necessary for a new instrumentation. Often in a series of experiments, you collect the same type of data while stressing the target executable in different ways. Reusing the instrumented executable lets you do this conveniently.

To reuse an executable from a previous experiment, simply enter the old experiment directory.

Specifying Sampling Data

Sampling data is collected and recorded at every sample point. The collection of sampling data requires instrumentation, that is, adding special code to the target executable. You can request four kinds of sampling data:

- call stack
- function counts
- basic block counts
- PC profile counts

Call Stack Profiling

The Performance Analyzer performs call stack data collection automatically, capturing data at every sample point, pollpoint, and process event. There is no instrumentation involved.

Function Count Collection

Function count collection provides this information:

- execution count of each function
- execution count of each call site

This data is a subset of the information provided by basic block counts. However, gathering function count data does not slow down the instrumented executable as much as gathering basic block data.

Note: It is not possible to collect function count data simultaneously with call stack profiling data.

Basic Block Count Sampling

In addition to the data provided by function counts, basic block counting provides you with the execution count of each line of machine code.

Basic block counts are translated to ideal CPU time displayed at the function, source line and machine line levels. The assumption made in calculating ideal CPU time is that each instruction takes exactly one cycle, and ignores potential floating point interlocks and memory latency time (cache misses and memory bus contention). Each system call is also assumed to take one cycle. The end result might be better described as *ideal user CPU time*.

The data is gathered by first instrumenting the target executable. This involves dividing the executable into basic blocks consisting of sets of machine instructions that do not contain branches into or out of them. A few lines of code are inserted for every basic block to increment a counter every time that basic block is executed. The basic block data is actually generated, and when the instrumented target executable is run, the data is written out to disk whenever a sample trap fires. Instrumenting an executable increases its size by a factor of three, and greatly modifies its behavior.

Caution: Running the instrumented executable causes it to run slower. By instrumenting, you might be changing the crucial resources; during analysis, the instrumented executable might appear to be CPU−bound, whereas the original executable was I/O−bound.

Note: It is not possible to collect basic block count data simultaneously with call stack profiling data.

PC Profile Counts

Enabling PC profile counts causes the Program Counter (PC) of the target executable to be sampled every 10 ms when it is in the CPU. PC profiling is a lightweight, high−speed operation done with kernel support. Every 10 ms, the kernel stops the process if it is in the CPU, increments a counter for the current value of the PC, and resumes the process.

PC Profile Counts is translated to the Actual CPU Time displayed at the function, source line and machine line levels. The actual CPU time is calculated by multiplying the PC hit count by 10 ms.

A major discrepancy between actual CPU time and ideal CPU Time indicates:
- cache misses and floating point interlocks in a single process application
- secondary cache invalidations in a multiprocess application run on a multiprocessor

Note: This comparison is inaccurate over a single run if you collect both basic block and PC profile counts simultaneously. In this situation, the Ideal CPU Time will factor out the interference caused by instrumenting; the Actual CPU Time will not. A rough approximation is to divide the Actual CPU Time by three.

A comparison between basic block counts and PC profile counts is shown in **Table 21−2** .

Table 21−2 Basic Block Counts and PC Profile Counts Compared

Basic Block Counts	PC Profile Counts
Used to compute ideal CPU time	Used to estimate actual CPU time
Data collection by instrumenting	Data collection done with the kernel
Slows program down by factor of three	Has minimal impact on program speed
Generates an exact count	*Approximates counts*

Specifying Tracing Data

Tracing data records the time at which an event of the selected type occurred. There are five types of tracing data:
- **"Malloc/Free Tracing"**
- **"System Call Tracing"**
- **"Page Fault Tracing"**
- **"I/O Syscall Tracing"**
- **"Floating Point Exception Tracing"**

Note: These features should be used with care; enabling tracing data adds substantial overhead to the target execution and consumes a great deal of disk space.

Malloc/Free Tracing

Malloc/free tracing enables you to study your program's use of dynamic storage and to quickly detect memory leaks (*mallocs* without corresponding *frees*) and bad *frees* (freeing a previously freed pointer). For this kind of tracing, you must create the target executable by linking with `-lmalloc_cv` instead of the usual `-lmalloc`. This data can be analyzed in Leak View (see **"Detecting Memory Leak Problems with Leak View"**), Malloc View (see **"Analyzing Memory Allocation with Malloc View"**), and Heap View (see **"Analyzing the Memory Map with Heap View"**).

Note that linking with `-lmalloc_cv` is not compatible with MP analysis so that using `-lmpc -lmalloc_cv` will not work.

System Call Tracing

Enabling system call tracing causes the call stack to be recorded whenever your program makes a system call. This data can be viewed in the system call event chart in Usage View (Graphs) which indicates where the system calls took place and in the Call Stack window which displays the call stack for a selected system call.

Page Fault Tracing

Enabling page fault tracing causes the call stack and the faulting address to be recorded every time your program makes a memory reference that causes a page fault.

The Page Fault event chart displays where the page faults took place in Process View. The Call Stack Information window displays the call stack for a selected page fault event.

I/O Syscall Tracing

I/O syscall tracing records every I/O−related system call that is made during the experiment. It traces *read* and *write* system calls with the call stack at the time, along with the number of bytes read or written. This is useful for I/O−bound processes.

Floating Point Exception Tracing

Floating point exception tracing records every instance of a floating point exception. This includes problems like underflow and NaN (not a number) values. If your program has a substantial number of floating point exceptions, you may be able to speed it up by correcting the algorithms.

Note: To use the floating point exception feature, you have to link your program with the library *libfpe.a* .

The floating point exceptions are:
- overflow
- underflow
- divide−by−zero
- inexact result
- invalid operand, e.g., infinity

Specifying Polling Data

There are three categories of polling data:
- **"Pollpoint Sampling"**
- **"Fine Grained Usage"**
- **"Call Stack Profiling"**

Entering a positive nonzero value in their fields turns them on and sets the time interval at which they will record.

Pollpoint Sampling

Setting pollpoint sampling enables you to specify a regular time interval for capturing performance data, including resource usage and any enabled sampling or tracing functions. Since pollpoint sampling occurs frequently, it is best used with call stack data only rather than other profiling data. Its primary utility is to enable you to identify boundary points for phases. In subsequent runs, you can set sample points to collect the profiling data at the phase boundaries.

Fine Grained Usage

Resource usage data is always collected at each sample point. Setting a time in the *Fine Grained Usage* field records resource usage data more frequently, at the specified time intervals. Fine grained usage helps you see fluctuations in usage between sample points.

You can analyze resource usage trends in the charts in Usage View (Graphs) and can view the numerical values in the Usage View (Numerical).

Fine grained usage has little effect on the execution of the target process during data collection. It is of limited use if the program is divided into phases of uniform behavior by the placement of the sample points.

Call Stack Profiling

Enabling call stack profiling causes the call stack of the target executable to be sampled at the specified time interval (minimum of 10 ms) and saved. The call stack continues to be sampled when the program is not running, while it is internally or externally blocked. Call stack profiling is used in the "Identify high total time functions" task to calculate total times.

Call stack profiling is accomplished by the Performance Analyzer views and not by the kernel. As a result, it is less accurate than PC profiling. Collecting call stack profiling data is far more intrusive than collecting PC profile data.

Caution: Collecting basic block data causes the text of the executable to be modified. Therefore, if call stack profiling data is collected along with basic block counts, the cumulative total time displayed in Usage View (Graphs) is potentially erroneous.

A comparison between call stack profiling and PC profiling is shown in **Table 21−3** .

Table 21−3 Call Stack Profiling and PC Profiling Compared

PC Profiling	Call Stack Profiling
Done by kernel	Done by Performance Analyzer process
Accurate, non−intrusive	Less accurate, more intrusive
Used to compute CPU time	Used to compute total time

Tracking Exec'd Processes

This feature allows you to specify whether or not you want the Performance Analyzer to gather performance data for any programs that are launched by an *exec* in any of the target processes. If this feature is enabled and there are *exec*s in the course of the experiment, then you can view the performance data for any of these other executables by using the Executable menu in the Performance Analyzer main window.

Enabling Performance Data Collection

To enable data collection, you use the following control buttons:

Enable Performance Task
turns all data collection on and off. The selected task will be activated.

OK
signifies that the specification of data collection types is complete and closes the Performance Panel window.

Apply
signifies that the specification of data collection types is complete but leaves the

Cancel

cancels any changes made to selections made in the Performance Panel, resets to the default values, and closes the window.

The Performance Analyzer Main Window

The Performance Analyzer main window is used for analysis after the performance data has been captured (see **Figure 21–2**). It contains a time line area indicating when events took place over the span of the experiment, a list of functions with their performance data, and a resource usage chart. This section covers these topics:

- **"Task Field"**
- **"Function List Display and Controls"**
- **"Usage Chart Area"**
- **"Time Line Area and Controls"**
- **"Admin Menu"**
- **"Config Menu"**
- **"Views Menu"**
- **"Executable Menu"**
- **"Thread Menu"**

The Performance Analyzer main window can be invoked from the "Launch Tool" submenu in the Debugger Admin menu or from the command line, by typing:

```
cvperf -exp experimentdirectory
```

where *experimentdirectory* is the directory containing the performance data from the experiment.

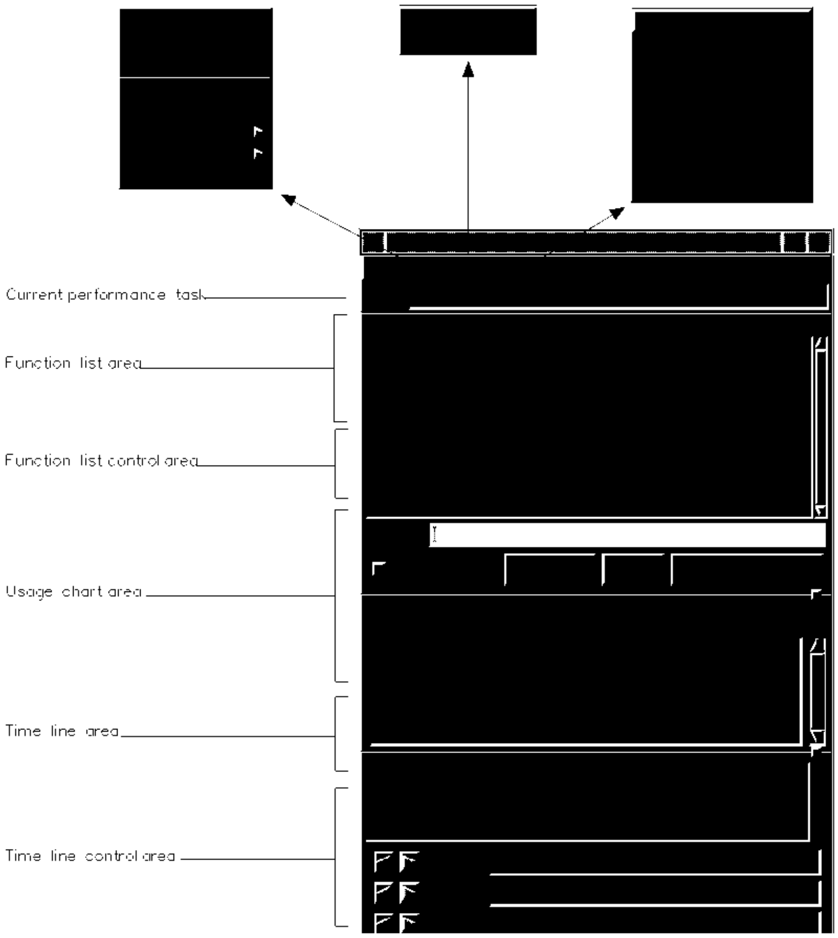


Figure 21−2 Performance Analyzer Main Window with Menus

Task Field

The *Task* field identifies the task for the current experiment and is read−only. See **"Specifying Performance Tasks"** for a summary of the performance tasks. For an in−depth explanation of each task, refer to **Chapter 20, "Setting Up Performance Analysis Experiments"** .

Function List Display and Controls

The function list area displays the program's functions with the associated performance metrics. It also provides buttons for displaying function performance data in other views. See **Figure 21−3** .

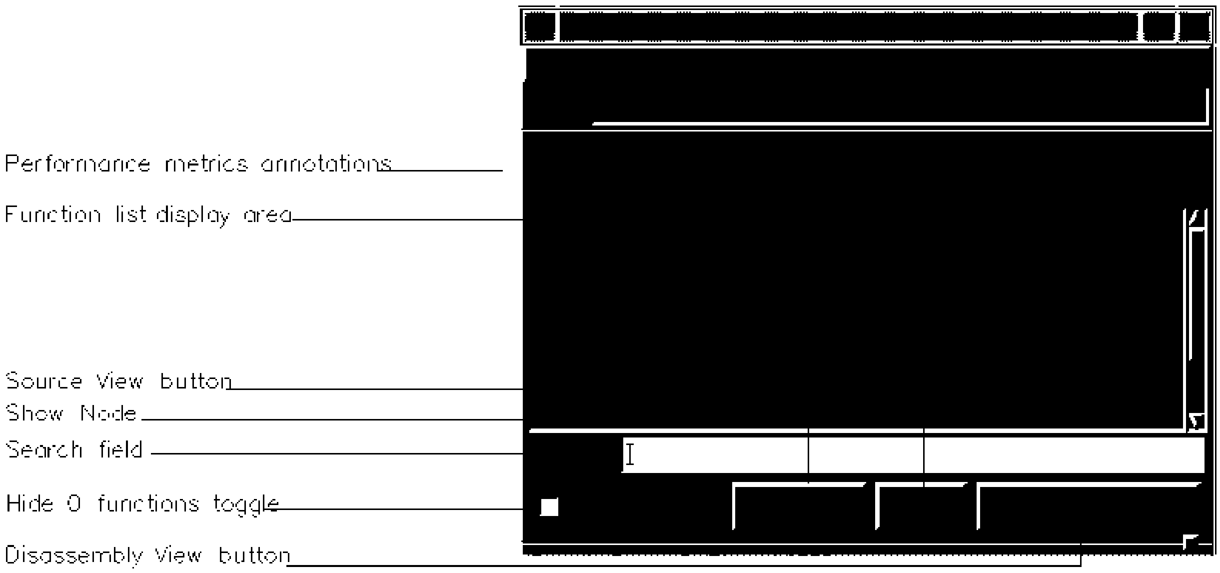


Figure 21−3 Typical Function List Area

The main features of the function list are:

Function list display area

shows all functions in the source code annotated with their associated performance data. The column headings identify the metrics.
You select the performance data to display from the "Preferences..." selection in the Config menu. The order of ranking is set by the "Sort..." selection in the Config menu.
The default order of sorting (depending on availability) is:

1. inclusive time
2. exclusive time
3. counts

Search field

lets you look for a function in the list and in any active views.

Hide 0 Functions toggle

lets you filter functions with 0 counts from the list.

Show Node

causes the specified node to display in the call graph.

Source

lets you display the Source View window corresponding to the selected function. The Source View window displays, with performance metrics in the annotation column. Source View can also be displayed by double−clicking a function in the Function List or a node or arc in the call graph. This is discussed in the next section.

Disassembled Source

Disassembly View displays, annotated with the performance metrics for total (CPU) time.

Usage Chart Area

The usage chart area in the Performance Analyzer main window (see **Figure 21–2**) displays the stripchart most relevant to the current task. The upper subwindow displays the legend for the stripchart and the lower subwindow displays the stripchart itself. This lets you obtain some useful information without having to open the Usage View (Graphs) window. **Table 21–4** shows you the data displayed in the usage chart area for each task.

Table 21–4 Task Display in Usage Chart Area

Task	Data in Usage Chart Area
Determine bottlenecks, identify phases	User versus system time
Get total time per function & source line	User versus system time
Get CPU time per function & source line	User versus system time
Get ideal time per function & source line	User versus system time
Trace I/O activity	*read()*, *write()* system calls
Trace system calls	System call event chart
Trace page faults	Page fault event chart
Find memory leaks	Process Size stripchart
Find floating point exceptions	Floating point exception event chart
Custom task	User versus system time unless tracing data has been selected (see Trace tasks above)

Time Line Area and Controls

The time line shows when each sample event in the experiment occurred. **Figure 21–4** shows the time line portion of the Performance Analyzer window with typical results.

Figure 21–4 Typical Performance Analyzer Time Line

The Time Line Calipers

The calipers let you define an interval for performance analysis. You can set the calipers in the time line to any two sample event points, using the caliper controls or by dragging them directly. The calipers appear solid for the current interval. If you drag them with the mouse (left or middle button), they appear dashed to give you visual feedback. When you stop dragging a caliper, it appears in outlined form denoting a tentative and as yet unconfirmed selection.

Specifying an interval is done as follows:

1. Set the left caliper to the sample event at the beginning of the interval.

   You can drag the left caliper with the left or middle mouse button or by using the left caliper control buttons in the control area. Note that calipers always snap to sample events. (Note that it actually doesn't matter whether you start with the left or right caliper.)

2. Set the right caliper to the sample event at the end of the interval.

   This is similar to setting the left caliper.

3. Confirm the change by clicking the *OK* button in the control area.

   After you confirm the new position, the solid calipers move to the current position of the outlined calipers and change the data in all views to reflect the new interval.

   Clicking *Cancel* or clicking with the right mouse button before the change is confirmed restores the outlined calipers to the solid calipers.

Current Event Selection

If you want to get more information on an event in the time line or in the charts in Usage View (Graphs), you can click an event with the left button. The *Event* field (see **Figure 21–4**) displays

- event number
- description of the trap that triggered the event
- the thread in which it was defined
- whether the sample was taken in all threads or the indicated thread only, in parentheses

In addition, the Call Stack View window updates to the appropriate times, stack frames, and event type for the selected event. A black diamond–shaped icon appears in the time line and charts to indicate the selected event. You can also select an event using the event controls below the caliper controls; they work in similar fashion to the caliper controls.

Time Line Scale Menu

The scale menu lets you change the number of seconds of the experiment displayed in the time line area. The "Full Scale" selection displays the entire experiment on the time line. The other selections are time values; for example, if you select "1 min", the length of the time line displayed will span 1 minute.

Admin Menu

The Admin menu and its options are shown in **Figure 21–5** . The Admin menu has selections common to the other WorkShop tools. There are three selections different in the Performance Analyzer:

"Experiment..."

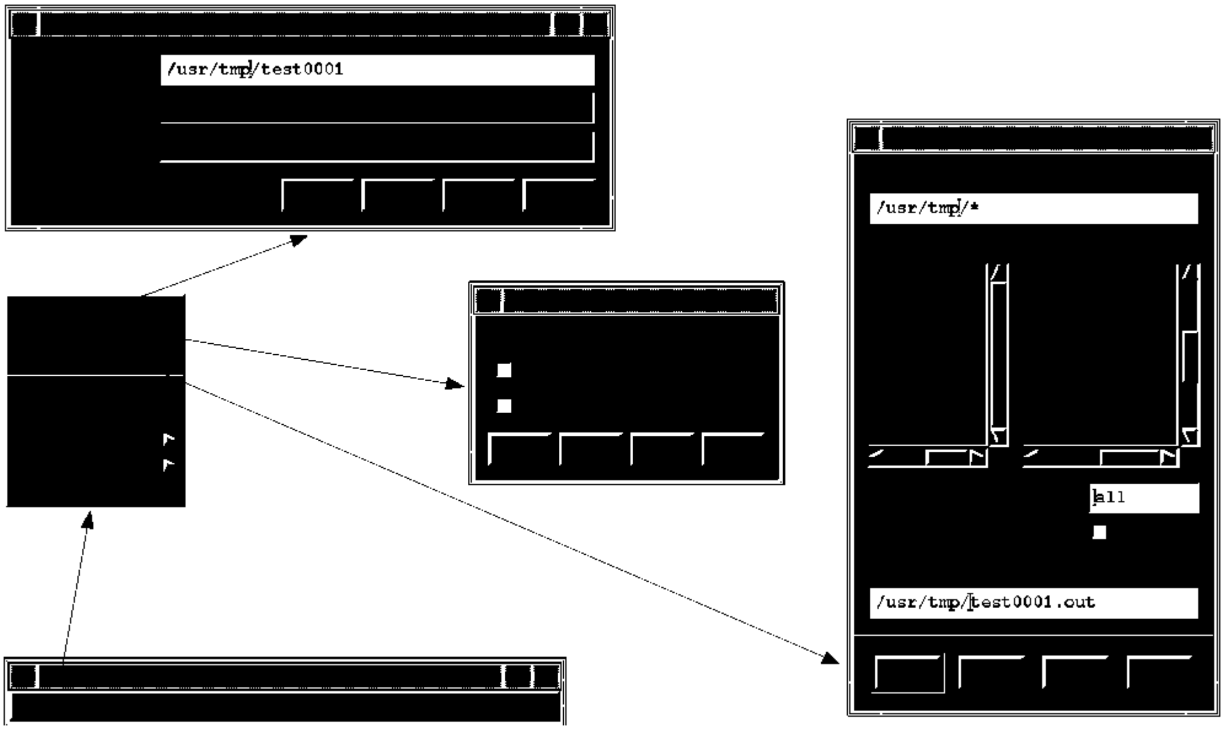
lets you change the experiment directory and displays the dialog box shown in **Figure 21–5** .

"Rerun Experiment"

lets you run another experiment with or without the same Performance Panel settings. A dialog box displays requesting confirmation (see **Figure 21–5**). The Debugger Main View window then displays so that you can start the experiment.

"Save As Text..."

records a text file with preference information selected in the view and displays the dialog box shown in **Figure 21–5** . You can use the default file name or replace it with another name in the File Selection dialog box that displays. You can specify the number of lines to be saved. The data can be saved as a new file or appended to an existing one.



/usr/tmp/test0001

/usr/tmp/*

all

/usr/tmp/test0001.out

Figure 21–5 Performance Analyzer Admin Menu Options

Config Menu

The main purpose of the Config menu in the Performance Analyzer main window is to select the performance metrics for display and for ranking the functions in the Function List. See **Figure 21–6** .

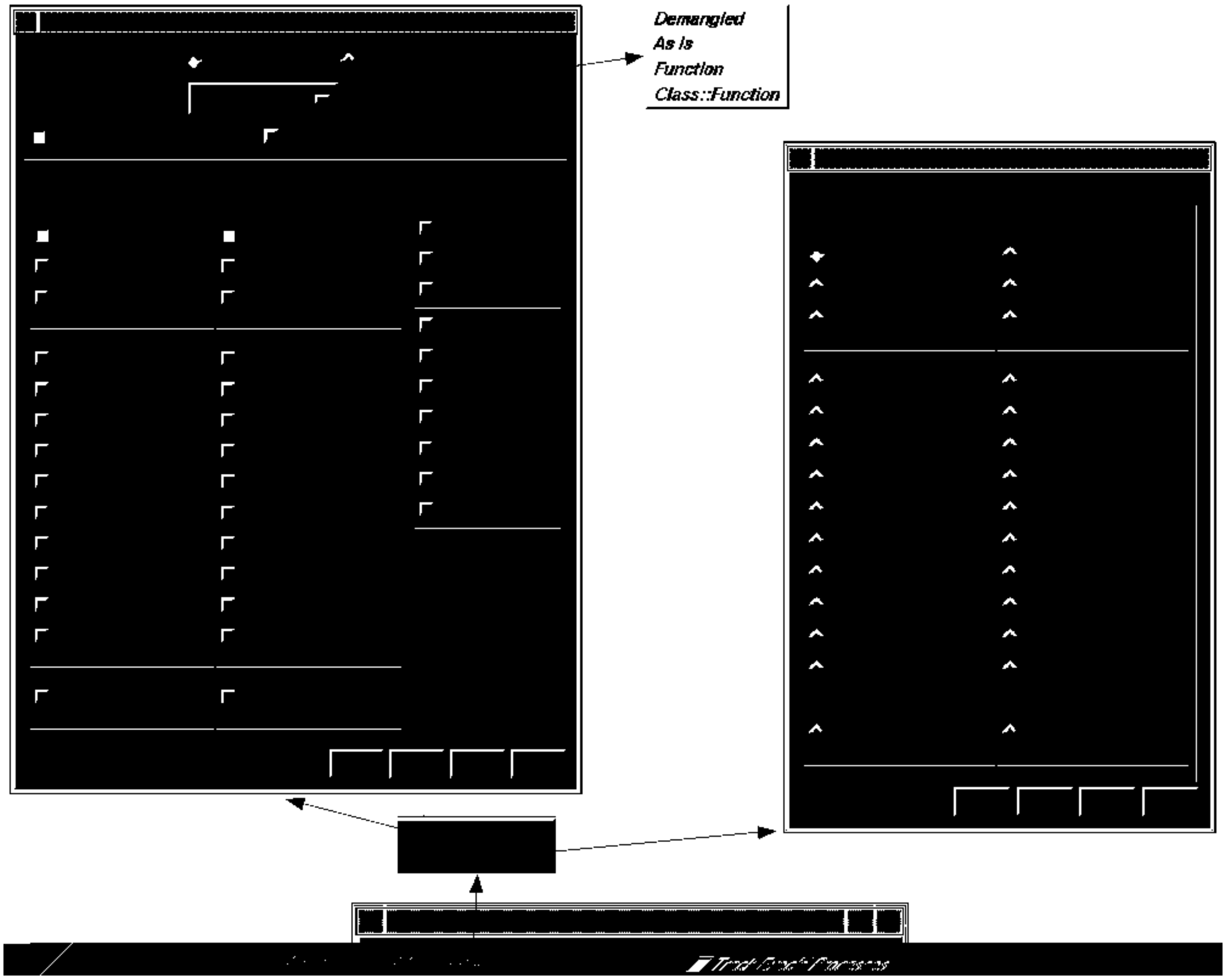


Figure 21–6 Performance Analyzer Config Menu Options

The selections in the Config menu are:

"Preferences..."

lets you select which metrics display and whether they appear as absolute times and counts or percentages. Remember you can only select the types of metrics that were collected in the experiment. You can also specify how C++ file names (if appropriate) are to display:

- "Demangled" shows the function its argument types.
- "As Is" uses the translator–generated "C" style name.
- "Function" shows the function name only.
- "Class::Function" shows the class and function.

"Sort..."

lets you establish the order in which the functions appear; this helps you find questionable functions. The default order of sorting (depending on availability) is:

1. Inclusive Times or counts
2. Exclusive Time or counts
3. Counts

The performance data selections are the same for both the Preferences and Sort dialog boxes. The difference between the inclusive (Incl.) and exclusive (Excl.) metrics is that inclusive data includes a function’s calls and exclusive data does not.

Address

is the address of the function.

Calls

Incl. Total Time, Excl. Total Time

refers to the time spent inside and outside of the CPU (by a function, source line, or instruction). It is calculated by multiplying the number of times the PC appears in any call stack by the average time interval between call stacks.

Incl. CPU Time, Excl. CPU Time

refers to the time spent inside the CPU (by a function, source line, or instruction). It is calculated by multiplying the number of times a PC value appears in the profile by 10 ms.

Incl. Ideal Time, Excl. Ideal Time

refers to the theoretical time spent by a function, source line, or instruction under the assumption of one machine cycle per instruction. It is useful to compare ideal time with actual.

Incl. Malloc counts, Excl. Malloc counts

refers to the number of *malloc*, *realloc*, and *free* operations.

Incl. System calls, Excl. System calls

refers to system calls.

Incl. Page faults, Excl. Page faults

refers to page faults.

Incl. FP operations, Excl. FP operations

refers to floating point operations.

Incl. Load counts, Excl. Load counts

refers to the number of *load* operations.

Incl. Store counts, Excl. Store counts

refers to the number of *store* operations.

Incl. Bytes Read, Excl. Bytes Read

refers to the number of bytes in a *read* operation.

Incl. Bytes Written, Excl. Bytes Written

refers to the number of bytes in a *write* operation.

Incl. FP Exceptions, Excl. FP Exceptions

refers to the number of floating point exceptions.

Incl. Instructions, Excl. Instructions

refers to the number of instructions.

Views Menu

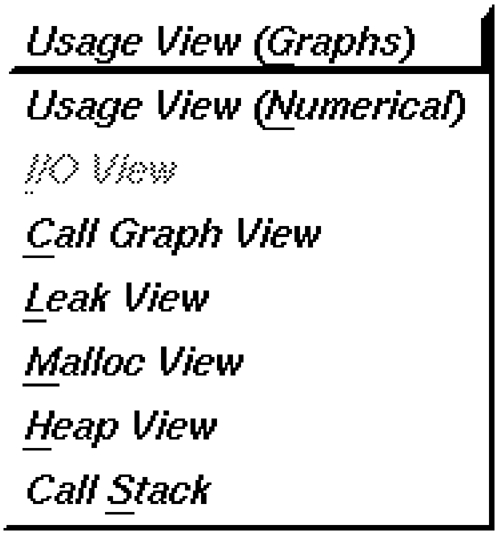


Figure 21–7 Performance Analyzer Views Menu

The Views menu in Performance Analyzer (see **Figure 21–7**) provides these selections for viewing the performance data from an experiment. Each view displays the data for the time interval bracketed by the

Figure 21–7 Performance Analyzer Views Menu

The Views menu in Performance Analyzer (see **Figure 21–7**) provides these selections for viewing the performance data from an experiment. Each view displays the data for the time interval bracketed by the calipers in the time line.

"Usage View (Graphs)"
displays resource usage charts and event charts. Refer to **"Usage View (Graphs)"** .

"Usage View (Numerical)"
displays the aggregate values of resources used. Refer to **"Usage View (Numerical)"** .

"I/O View"
displays I/O events. Refer to **"I/O View"** .

"Call Graph View"
displays a call graph that shows functions and calls and their associated performance metrics. Refer to **"Call Graph View"** .

"Leak View"
displays individual leaks and their associated call stacks.

"Malloc View"
displays individual *malloc*s and their associated call stacks.

"Heap View"
displays a map of heap memory showing *malloc*, *realloc*, *free*, and bad *free* operations. Refer to **"Analyzing the Memory Map with Heap View"** .

"Call Stack"
displays the call stack for the selected event and the corresponding event type. Refer to **"Call Stack"** .

Executable Menu

If you enabled *Track Exec'd Processes* (in the Performance Panel) for the current experiment, the Executable menu will be enabled and will contain selections for any *exec*'d processes. These selections let you see the performance results for the other executables.

Thread Menu

If your process *fork*ed any processes, the Thread menu is activated and contains selections corresponding to the different threads. Selecting a thread displays its performance results.

Usage View (Graphs)

Usage View (Graphs) displays resource usage and event charts containing the performance data from the experiment. These charts show resource usage over time and indicate where sample events took place. Sample events are shown as vertical lines. **Figure 21–8** shows the User vs system time and Page faults graphs; **Figure 21–9** shows the other graphs.

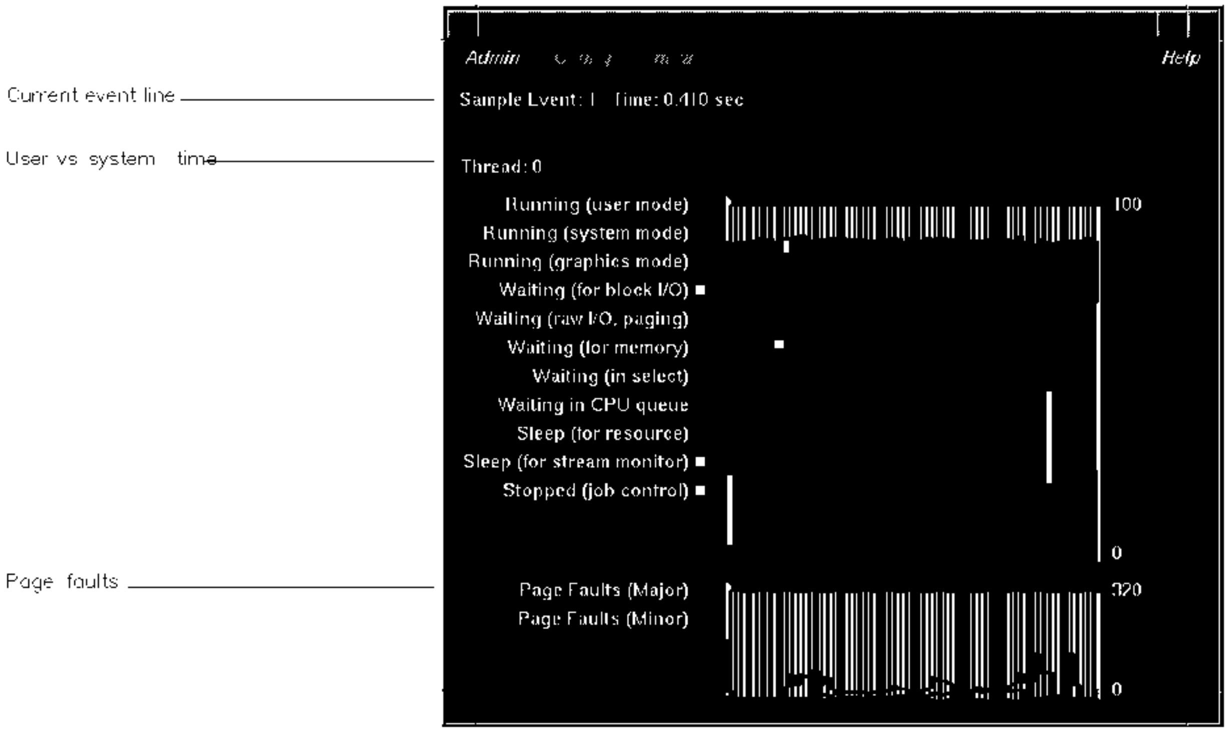


Figure 21–8 Usage View (Graphs) Window: Top Graphs

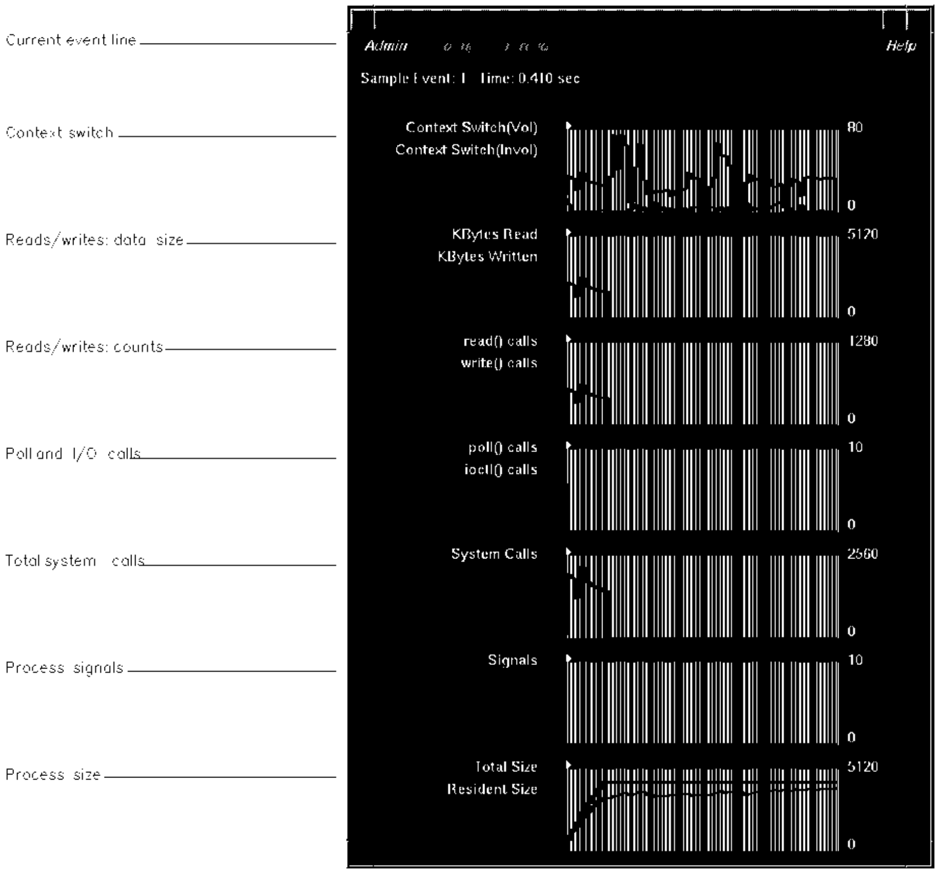


Figure 21–9 Usage View (Graphs) Window: Lower Graphs

Charts in Usage View (Graphs)

The available charts are:

User vs system Time

shows CPU usage. Whenever the system clock ticks, the process occupying the CPU is charged for the entire ten millisecond interval. The time is charged either as user or system time, depending on whether the process is executing in user mode or system mode. The graph provides these annotations to show how time is spent during an experiment's process: *Running (user mode)*, *Running (system mode)*, *Running (graphics mode)*, *Waiting (for block I/O)*, *Waiting (raw I/O, paging)*, *Waiting (for memory)*, *Waiting (in select)*, *Waiting in CPU queue*, *Sleep (for resource)*, *Sleep (for stream monitor)*, and *Stopped (job control)*.

Page faults

shows the number of page faults that occur within a process. *Major faults* are those that require a physical read operation to satisfy; *minor faults* are those where the necessary page is already in memory but not mapped into the process's address space.

Each major fault in a process takes approximately 10–50 ms. A high page fault rate is an indication of a memory–bound situation.

Context Switch

shows the number of voluntary and involuntary context switches in the life of the process.

Voluntary context switches are attributable to an operation caused by the process itself, such as a disk access or waiting for user input. These occur when the process can no longer use the CPU. A high number of voluntary context switches indicates that the process is spending a lot of time waiting for a resource other than the CPU.

Involuntary context switches happen when the system scheduler decides to give the CPU to another process, even if the target process is able to use it. A high number of involuntary context switches indicates a CPU contention problem.

Read/write: data size

shows the number of bytes transferred between the process and the operating system buffers, network connections, or physical devices. *KBytes read* are transferred into the process' address space; *KBytes written* are transferred out of the process' address space.

A high byte transfer rate indicates an I/O–bound process.

Read/write: counts

shows the number of read and write system calls made by the process.

Poll and I/O calls

shows the combined number of poll or select system calls (used in I/O multiplexing) and the number of I/O control system calls made by the process.

Total system calls

shows the total number of system calls made by the process. This includes the counts for the calls shown on the other charts.

Process signals

shows the total number of signals received by the process.

Process Size

shows the total size of the process in pages and the number of pages resident in memory at the end of the time interval when the data is read. It is different from the other charts in that it shows the absolute size measured at the end of the interval and not an incremental count for that interval.

If you see the process total size increasing over time when your program should be in a steady state, the process most likely has leaks and you should analyze it with Leak View and Malloc View.

Getting Event Information from Usage View (Graphs)

The charts indicate trends; to get detailed data, you click the relevant area on the chart and the data displays in the current event line. The left mouse button displays event data; the right displays interval data.

When you click the left mouse button on a sample event in a chart, the following actions take place:

- The point becomes selected, as indicated by the diamond marker above it. The marker appears in the time line, resource usage chart, and Usage View (Graphs) charts if the window is open.
- The current event line identifies the event and displays its time.
- The call stack corresponding to this sample point gets displayed in the Call Stack window (see **"Call Stack"**).

Figure 21–10 illustrates the process of selecting a sample event.

Clicking a graph with the right button displays the values for the fine–grained interval (if collection was specified) or if not, the interval bracketed by the nearest sample events.

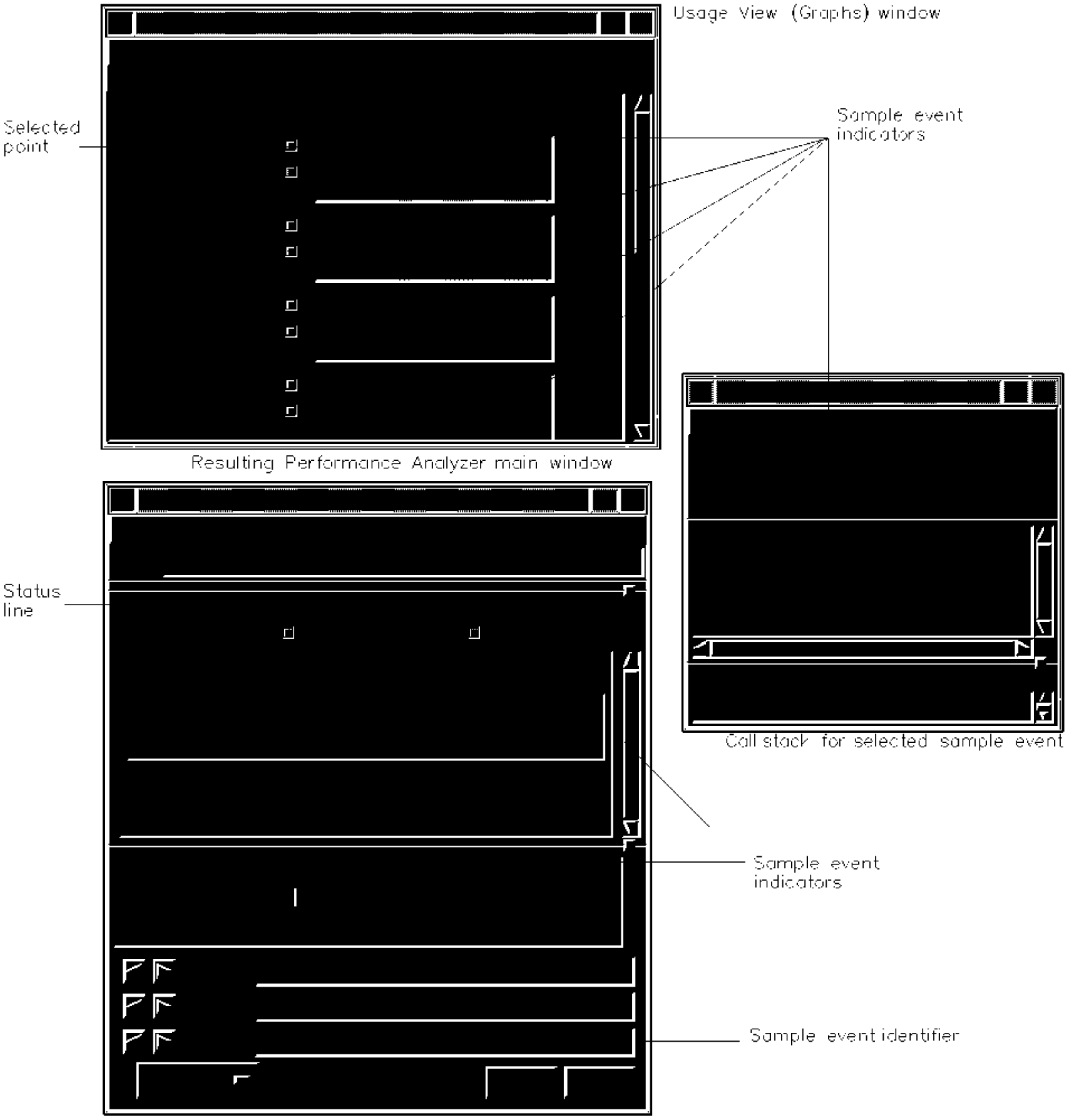


Figure 21–10 Effects of Selecting a Sample Event

Process Meter

Process Meter lets you observe resource usage for a running process without conducting an experiment. To call Process Meter, select "Process Meter" from the Views menu in the Debugger Main View.

A Process Meter window with data and its menus displayed appears in **Figure 21−11** . Process Meter uses the same Admin menu as the WorkShop Debugger tools.

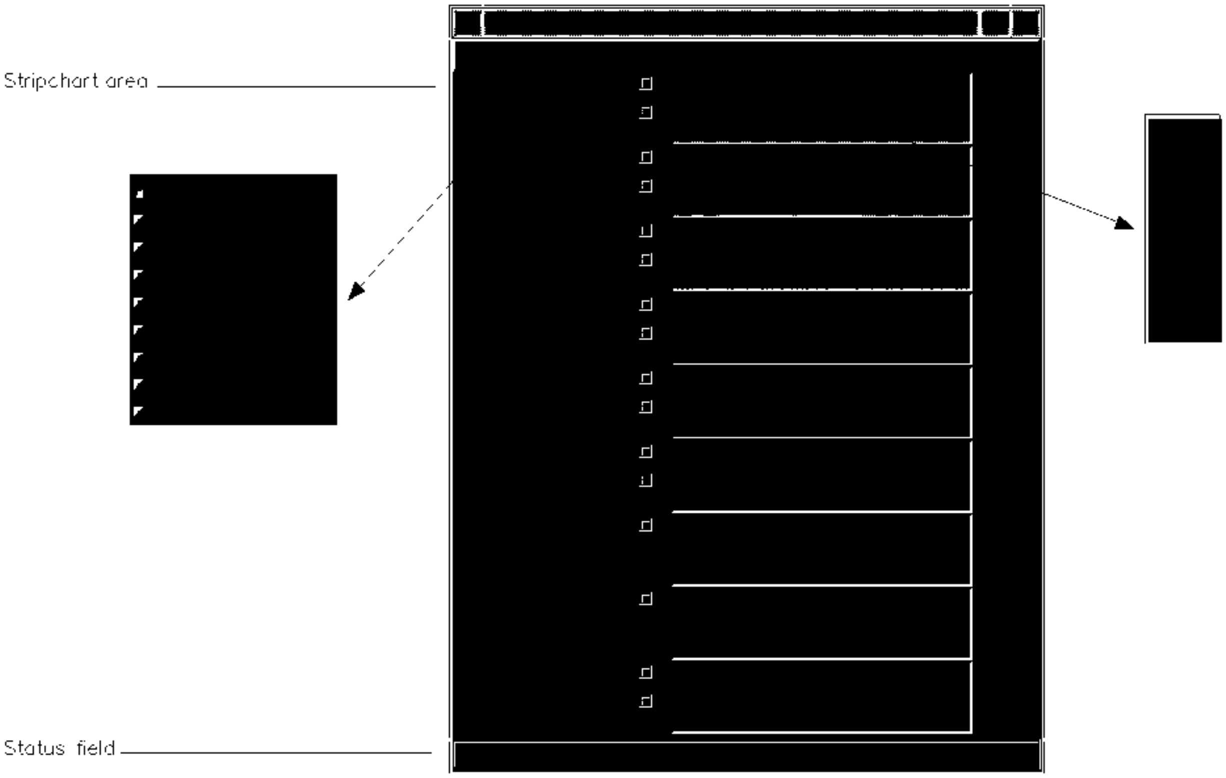


Figure 21−11 The Process Meter with Major Menus Displayed

The Charts menu options display the selected stripcharts in the Process Meter.

The Scale menu adjusts the time scale in the stripchart display area such that the time selected becomes the end value.

You can select which usage charts and event charts display. You can also display sample point information in the Status field by clicking within the charts.

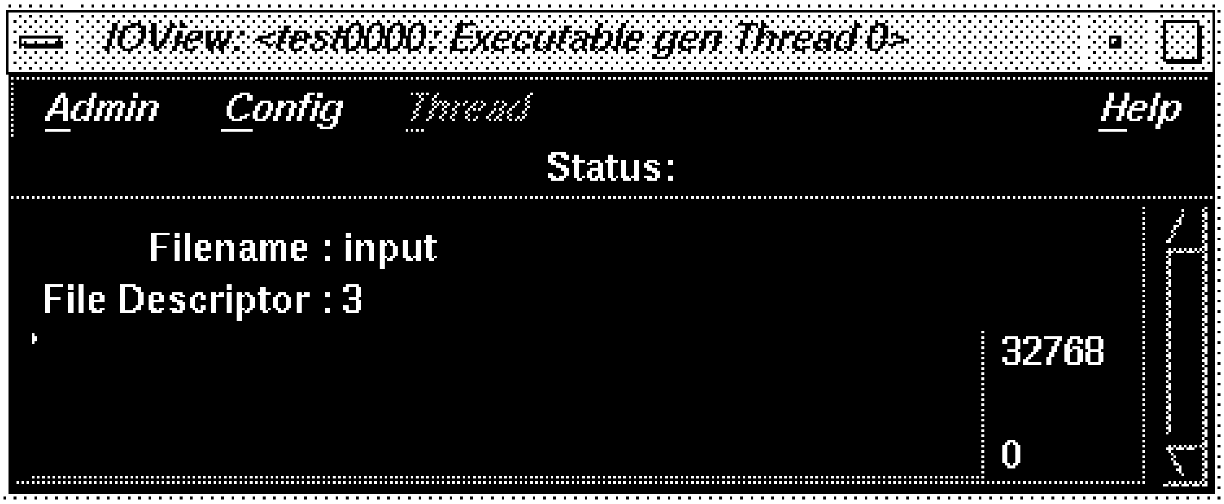
Usage View (Numerical)

The Usage View (Numerical) window (see **Figure 21−13**) shows detailed, process−specific resource usage information in a textual format for the interval defined by the calipers in the time line area of the Performance Analyzer main window. To display the Usage View (Numerical) window, select "Usage View (Numerical)" from the Views menu.

The top of the window identifies the beginning and ending events for the interval. The middle portion of the window shows resource usage for the target executable. The bottom panel shows resource usage on a system−wide basis. Data is shown both as total values and as per−second rates.

I/O View

I/O View helps you determine the problems in an I/O−bound process. It produces a graph of all I/O system calls and identifies up to 10 files involved in I/O. See **Figure 21−12** .



Status:

Filename : input

File Descriptor : 3

32768

0

Figure 21−12 I/O View

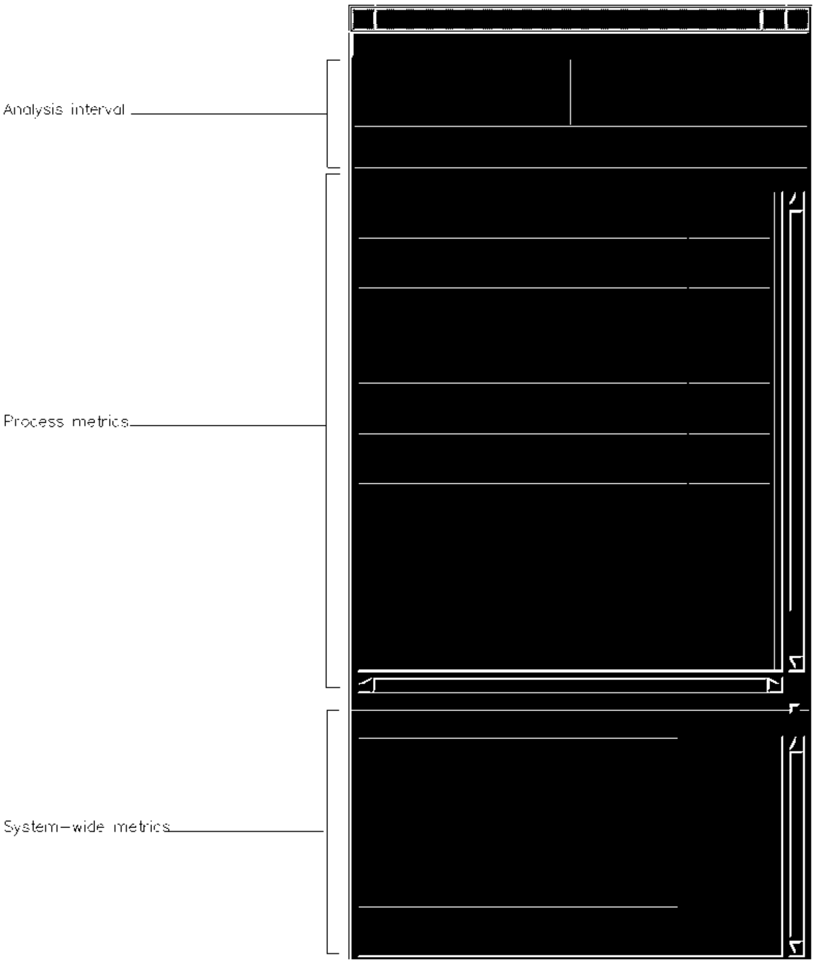


Figure 21−13 Usage View (Numerical)

Call Graph View

The Call Graph View window displays a call graph showing the functions as nodes and their calls as connecting arcs, both annotated with performance metrics (see **Figure 21−14**). You bring up Call Graph View by selecting "Call Graph View" from the Views menu.

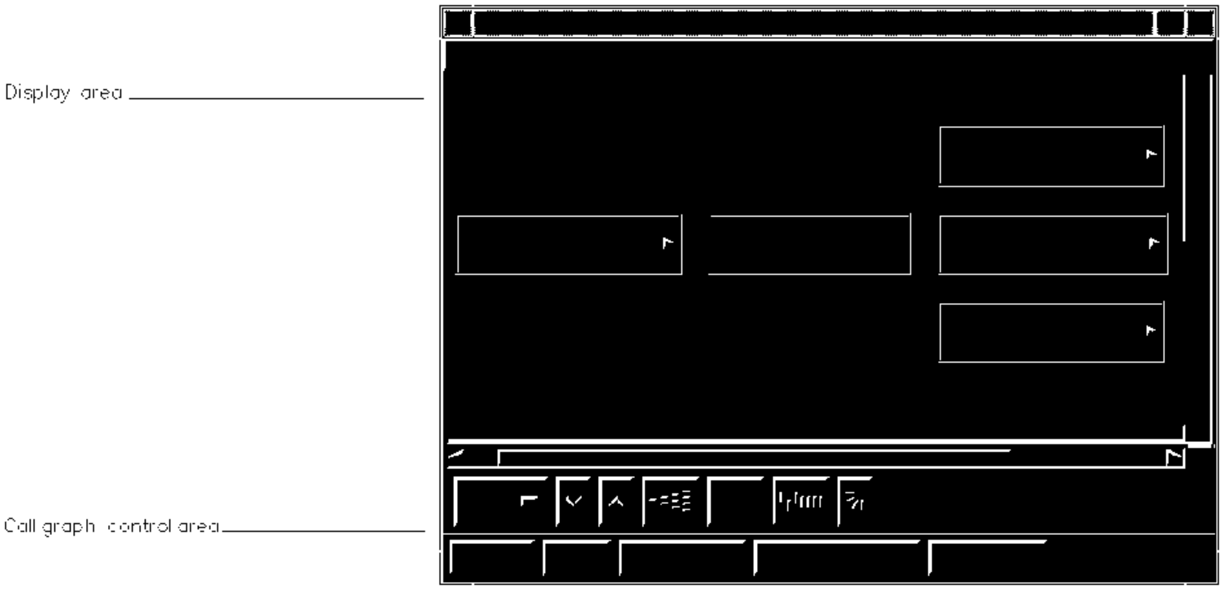


Figure 21−14 Call Graph View with Display Controls

Since a call graph can get quite complicated, Performance Analyzer provides various controls for changing the graph display. The "Preferences" selection in the Config menu lets you specify which performance metrics display and also lets you filter out unused functions and arcs. There are two node menus in the display area; these let you filter nodes individually or as a selected group. The top row of display controls is common to all CASEVision graph displays and let you change scale, alignment, and orientation, or see an overview (see **Appendix A, "Graphical Views in the CASEVision Environment"**). The bottom row of controls lets you define the form of the graph: as a butterfly graph showing the functions that call and are called by a single function or as a chain graph between two functions.

Special Node Icons

Although rare, nodes can be annotated with two types of graphic symbols:

- A right−pointing arrow in a node indicates an indirect call site. It represents a call through a function pointer. In such a case, the called function cannot be determined by the current methods.
- A circle in a node indicates a call to a shared library with a data−space jump table. The node name is the name of the routine called, but the actual target in the shared library cannot be identified. The table might be switched at run time, directing calls to different routines.

Annotating Nodes and Arcs

You can specify which performance metrics appear in the call graph as follows.

Node Annotations

To specify the performance metrics that display inside a node, you need the Preferences dialog box in the Config menu from the Performance Analyzer main view (see **Figure 21−6**).

Arc Annotations

Arc annotations are specified by selecting "Preferences..." from the Config menu in Call Graph View (see **Figure 21–15**). You can display the counts on the arcs. You can also display the percentage of calls to a function broken down by incoming arc. For an explanation of the performance metric items, refer to **"Config Menu"** .

Filtering Nodes and Arcs

You can specify which nodes and arcs appear in the call graph as follows.

Call Graph Preferences Filtering Options

The Call Graph Display Options dialog box accessed from the "Preferences" selection in the Call Graph View Config menu also lets you hide functions and arcs that have 0 (zero) calls. See **Figure 21–15** .

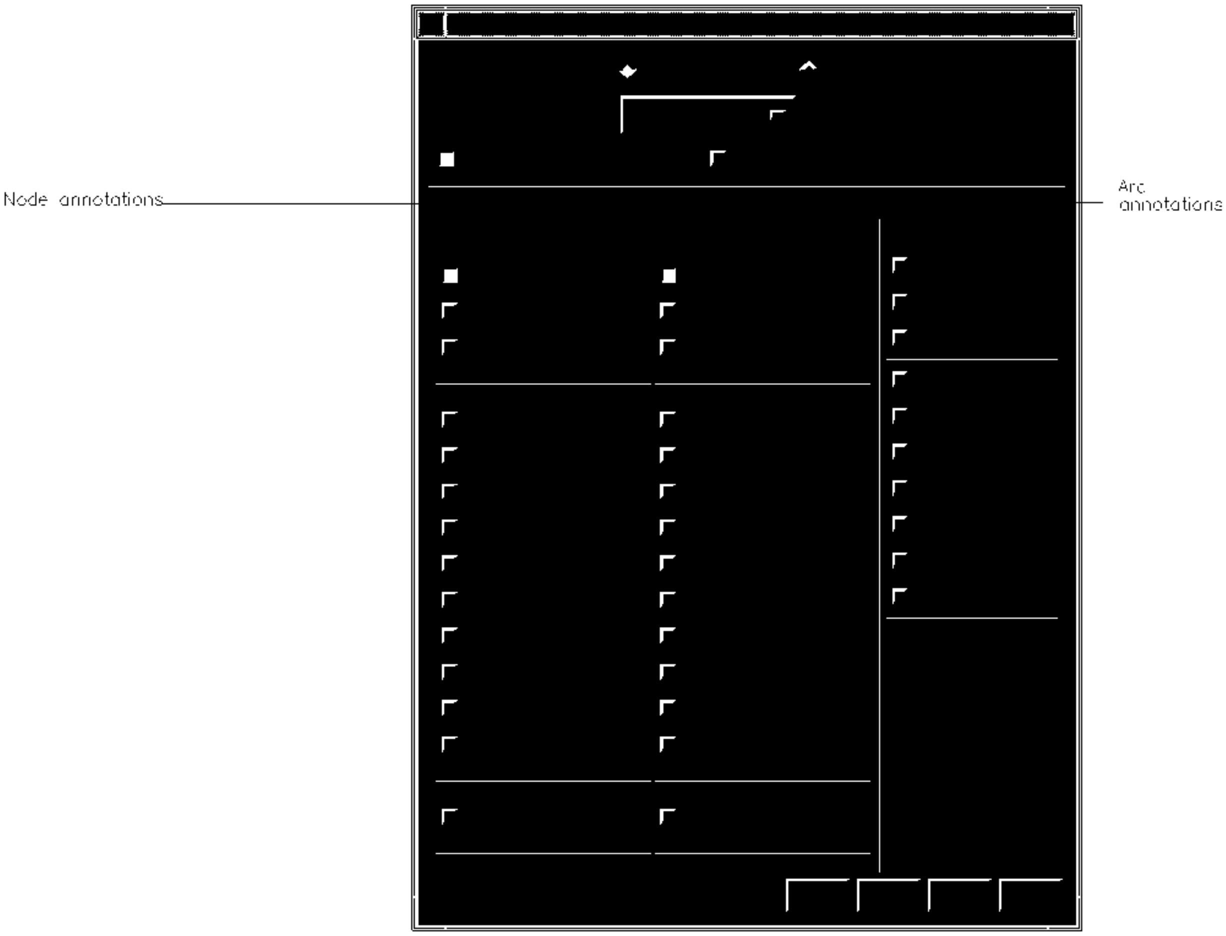


Figure 21–15 Call Graph Preferences

Node Menu

There are two node menus for filtering nodes in the graph: the Node menu and the Selected Nodes menu. Both menus are shown in **Figure 21–16** .

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.

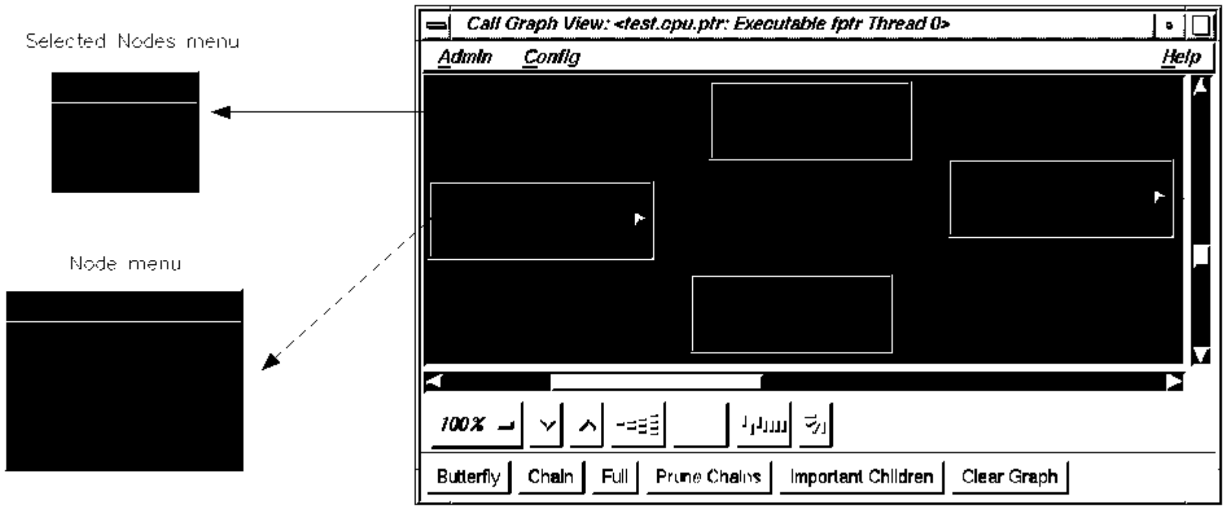


Figure 21−16 Node Menus

The Node menu selections are:

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

"Collapse Subgraph"
removes the nodes called by the selected node (and subsequently called nodes) from the call 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 selections are:

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

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

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

Filtering Nodes through the Display Controls

The lower row of controls in the panel helps you reduce the complexity of a busy call graph (see **Figure 21−17**).

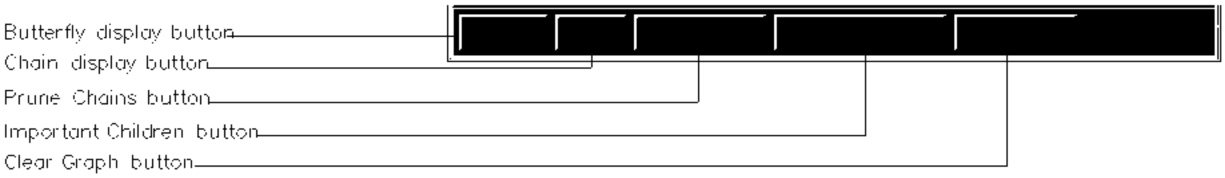


Figure 21−17 Call Graph View Controls for Content Manipulation

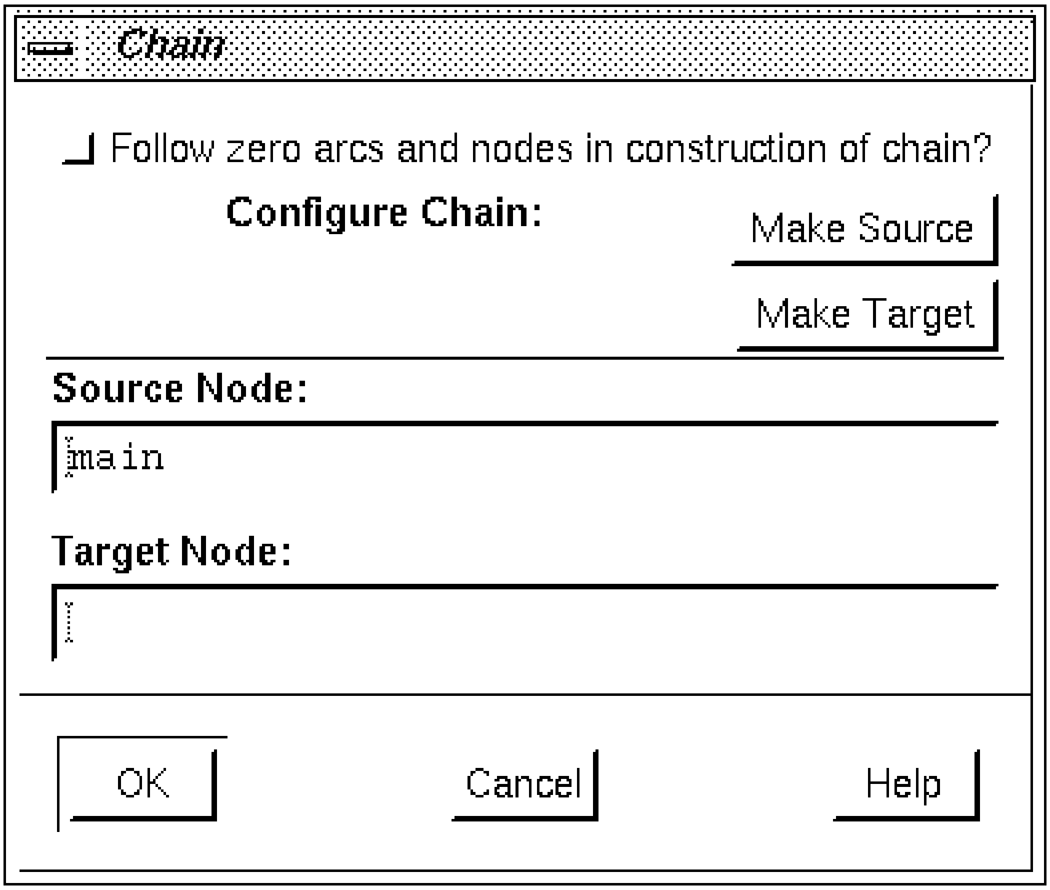
You can perform these display operations:

Butterfly

presents the call graph from the perspective of a single node (the target node), showing only those nodes that call it or are called by it. Functions that call it are displayed to the left and functions it calls are on the right. Selecting any node and clicking *Butterfly* causes the graph to be redrawn with the selected node as the center. The selected node is displayed and highlighted in the function list.

Chain

lets you display all paths between a given source node and target node. The Chain dialog box is shown in **Figure 21−18** . You designate the source function by selecting it or entering it in the *Source Node* field and clicking the *Make Source* button. Similarly, the target function is selected or entered and then established by clicking the *Make Target* button. If you wish to filter out paths that go through nodes and arcs with 0 counts, click the toggle. After these selections are made, click *OK*.



☐ Follow zero arcs and nodes in construction of chain?

Configure Chain:

Source Node:

main

Target Node:

Figure 21−18 Chain Dialog Box

Prune Chains

displays a dialog box that provides two selections for filtering paths from the call graph (see **Figure 21−19**).

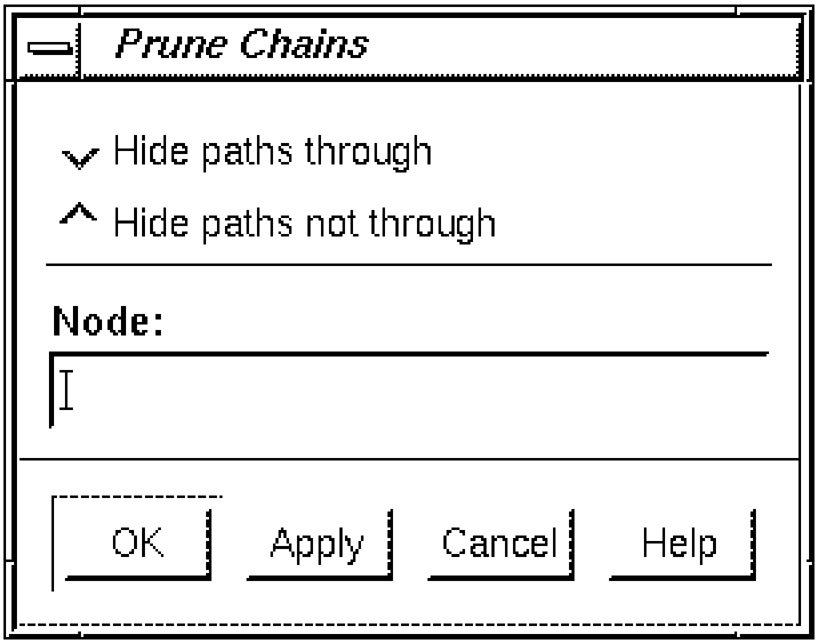


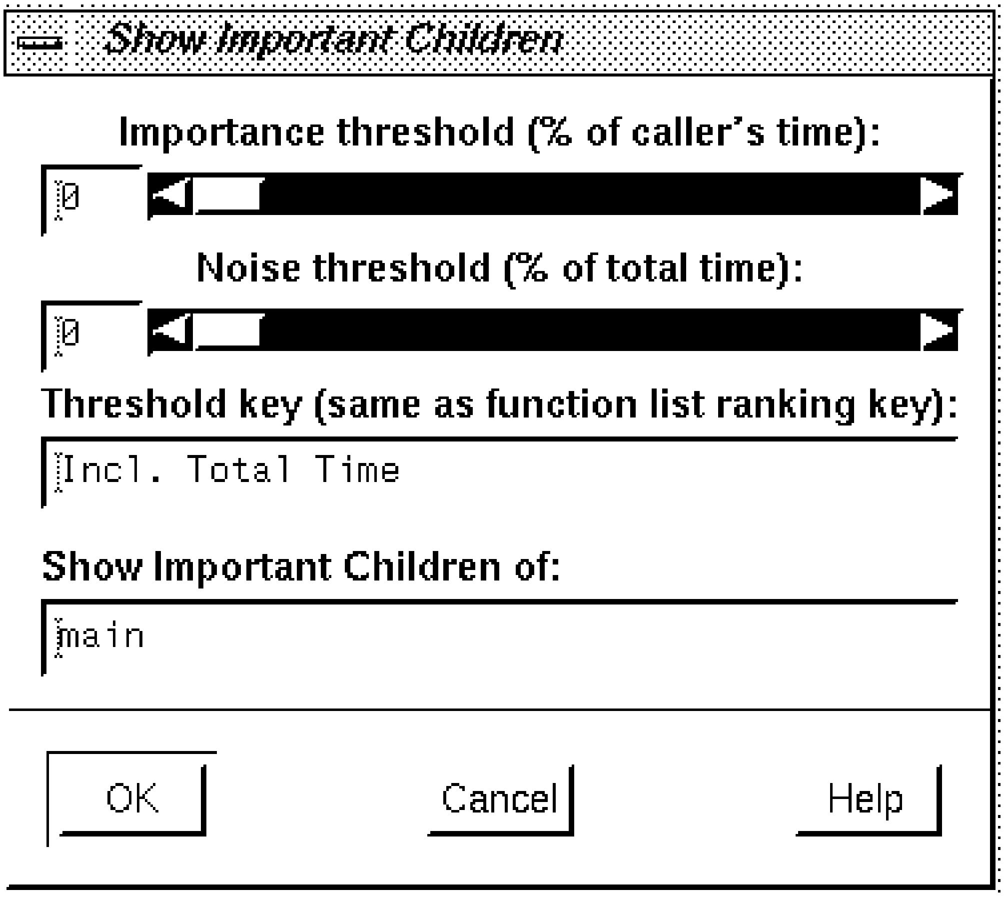
Figure 21−19 Prune Chains Dialog Box

The *Prune Chains* button is only activated when a chain mode operation has been performed. The dialog box selections are:

- The *Hide paths through* toggle removes from view all paths that go through the specified node. You must have a current node specified. Note that this operation is irreversible; you will not be able to re−display the hidden paths unless you perform the chain command again.
- The *Hide paths not through* toggle removes from view all paths except the ones that go through the specified node. This operation is irreversible.

Important Children

lets you focus on a function and its descendants and set thresholds to filter the descendants. You can filter the descendants either by percentage of the caller's time or by percentage of the total time. The *Threshold key* field identifies the type of performance time data used as the threshold. See **Figure 21−20** .



Importance threshold (% of caller's time): 0

Noise threshold (% of total time): 0

Threshold key (same as function list ranking key): Incl. Total Time

Show Important Children of: main

Figure 21–20 Show Important Children Dialog Box

Clear Graph

removes all nodes and arcs from the call graph.

Other Manipulation of the Call Graph

Call Graph View provides facilities for changing the display of the call graph without changing the data content.

Geometric Manipulation through the Control Panel

The controls for changing the display of the call graph are in the upper row of the control panel (see **Figure 21–21**).

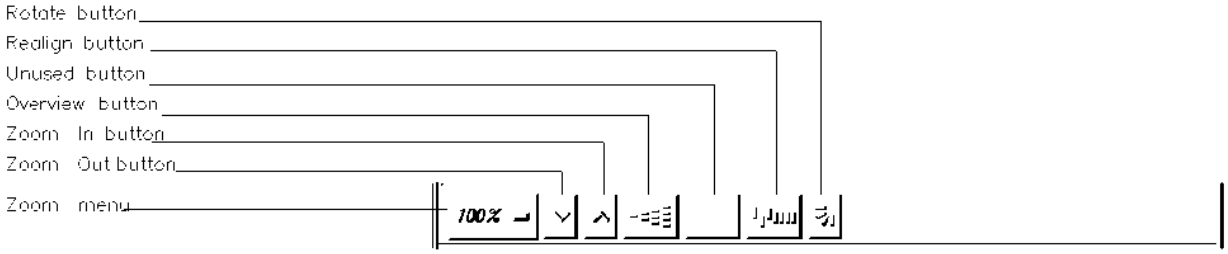


Figure 21–21 Call Graph View Controls for Geometric Manipulation

These facilities are:

Zoom menu button	shows the current scale of the graph. If you click this button, 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.
Overview button	invokes an overview popup display that shows a scaled down representation of the graph. The nodes appear in the analogous places on the overview popup, and a white outline may be used to position the main graph relative to the popup. Alternatively, the main graph may be repositioned with its scroll bars.
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).

For more information on the graphical controls, see **Appendix A, "Graphical Views in the CASEVision Environment."**

Using the Mouse in Call Graph View

You can move an individual node by dragging it using the middle mouse button. This helps reveal obscured arc annotations.

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.

Selecting Nodes from the Function List

You can also select functions from the function list to be highlighted in the call graph. You select a node from the list and then click the *Show Node* button in the Function List window. The node will be highlighted in the graph.

Finding Memory Problems with Leak View, Malloc View, and Heap View

The Performance Analyzer provides three tools for analyzing memory problems: Leak View, Malloc View, and Heap View. Setting up and running a memory analysis experiment is the same for all three tools. After you have conducted the experiment, you can apply any of these tools.

Conducting Memory Leak Experiments

To look for memory leaks or bad frees, or perform other analysis of memory allocation, you need to run a Performance Analyzer experiment with "Find memory leaks" specified as the experiment task. You run a memory corruption experiment like any performance analysis experiment by clicking *Run* in the Debugger Main View. The Performance Analyzer keeps track of each *malloc* (memory allocation), *realloc* (reallocation of memory), and *free.* The general steps in running a memory experiment are:

1. Link your executable with one of the special WorkShop *malloc* libraries (*–lmalloc_cv* or *–lmalloc_cv_d*).

   Note: For a detailed discussion of the *malloc* libraries, see **"The WorkShop Malloc Libraries"** , **"Compiling with the Malloc Libraries"** , and **"Setting the Environment Variables"** . This tutorial assumes that library *–lmalloc_cv* is used.

   Before you even run a memory experiment, you need to relink your executable with the WorkShop *malloc* library (*libmalloc_cv*) instead of the *malloc* library (*libmalloc*). You can compile it from scratch as follows:

   ```
   cc –g –o targetprogram   targetprogram.c   –lmalloc_cv
   ```

   or you can relink it by using:

   ```
   ld –o targetprogram   targetprogram.o   –lmalloc_cv
   ```

2. Display the Performance Panel.

   You can bring up the Performance Panel by selecting "Performance Task..." from the Admin menu in the Debugger Main View or by typing `cvspeed` at the command line.

3. Specify "Find memory leaks" as the experiment task.

   "Find memory leaks" is a selection on the Performance Task menu in the Performance Panel. It ensures that the appropriate performance data is collected. (For more information, see "Find memory leaks" on page 301).

4. Run the memory leak experiment.

   You run experiments by clicking the *Run* button in the Debugger Main View window.

5. Display the Performance Analyzer.

   The Performance Analyzer displays results appropriate to the task selected, in this case, "Find memory leaks". **Figure 21–22** shows the Performance Analyzer window after a memory experiment (after resizing).

   Notice that the Function List displays inclusive and exclusive bytes leaked per function. Double–clicking a function brings up Source View displaying the function's source code annotated with bytes leaked. (To pinpoint the location of a leak more exactly, however, it is better to use Leak View and bring up Source View pointing to the exact location of the leak.) You can set other annotations in Source View and the Function List by choosing "Preferences..." from the Config menu in the Performance Analyzer and selecting the desired items, for example, inclusive and exclusive *malloc* counts.

   The total process size chart displays in the usage chart portion of the window. Leakage and other memory problems cause a process's size to increase over time.

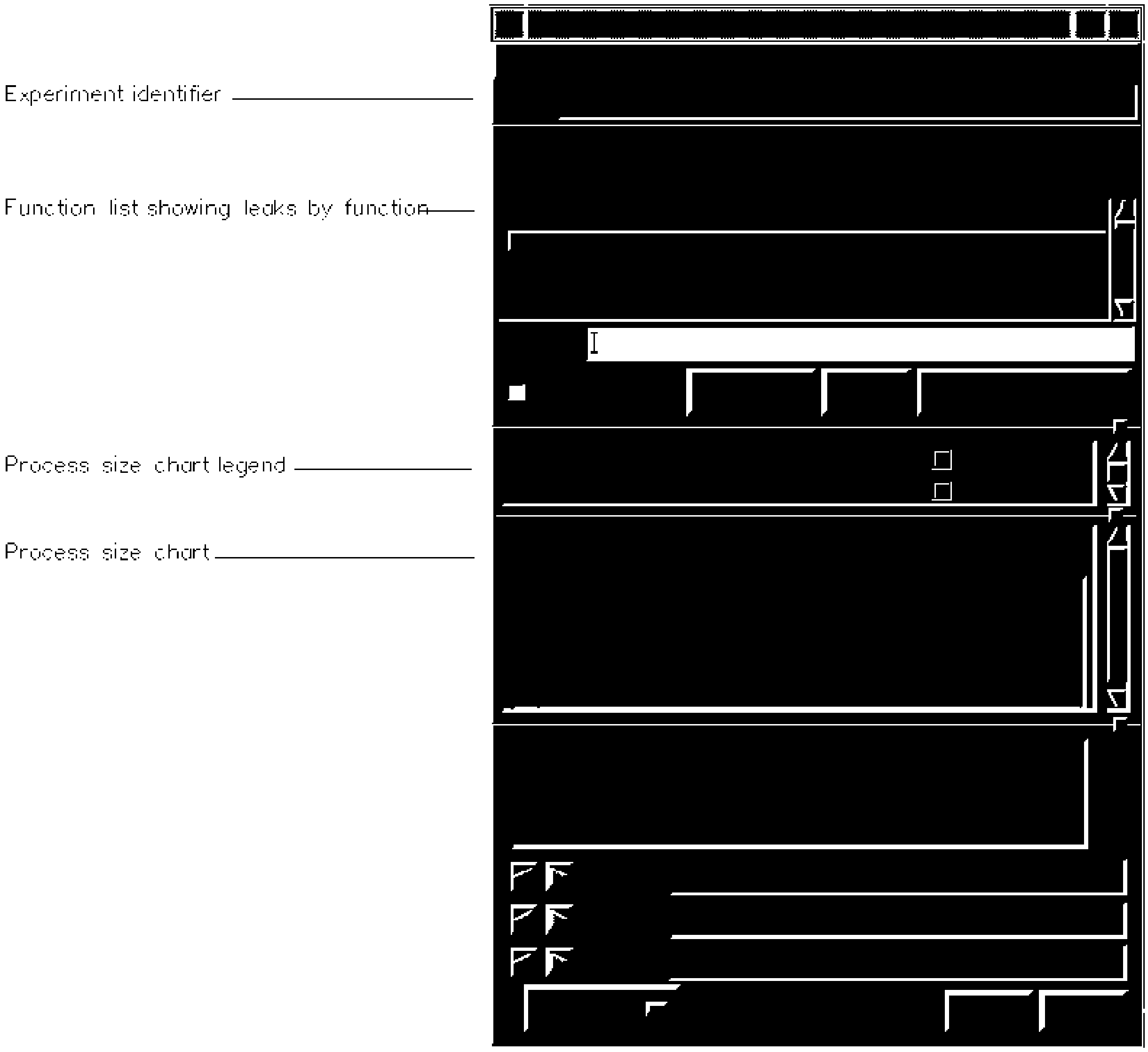
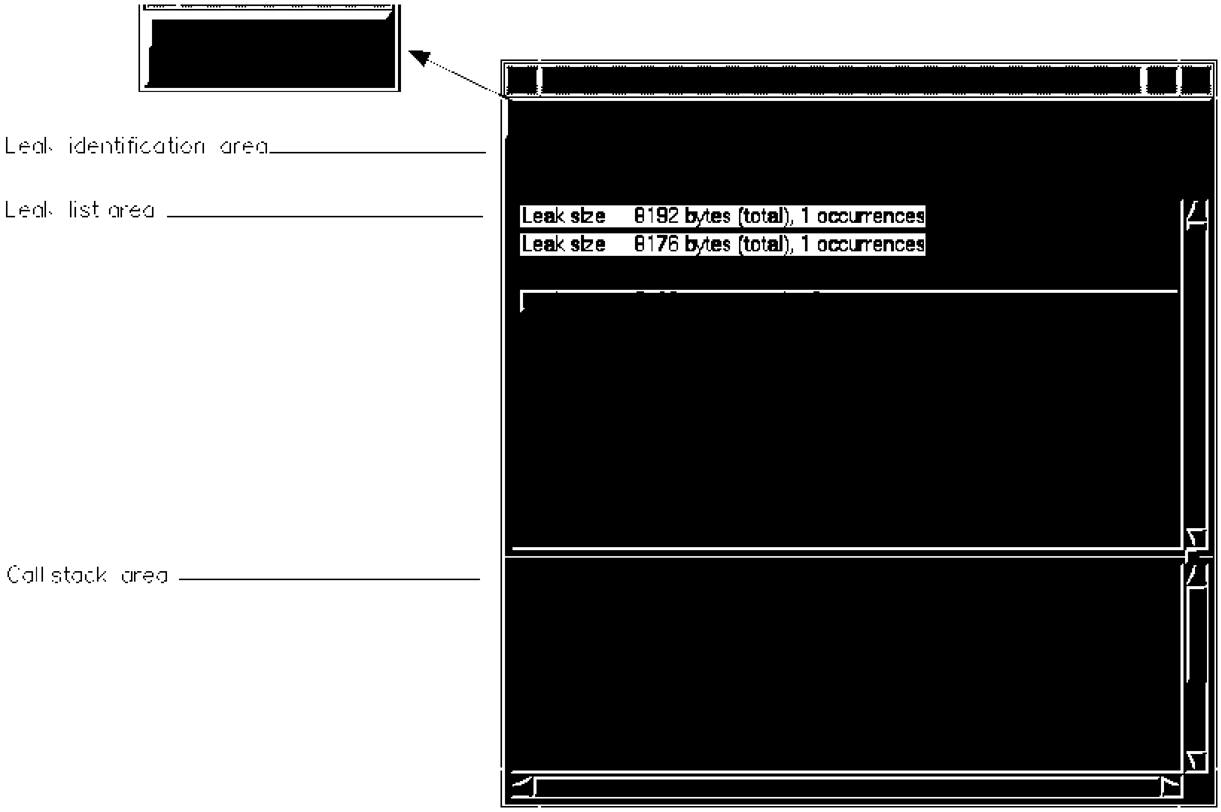


Figure 21–22 Performance Analyzer Displaying Results of a Memory Experiment

6. Analyze the results of the experiment in Leak View when doing leak detection and Malloc View when performing broader memory allocation analysis. To view memory problems within the memory map, use Heap View. To look at the source code annotated with memory problems, bring up Source View from one of the other three tools.

Detecting Memory Leak Problems with Leak View

After you have run a memory experiment using the Performance Analyzer, you can analyze the results using Leak View. Leak View displays each memory leak that occurs in your executable, its size, the number of times the leak occurred at that location during the experiment, and the corresponding call stack (when you select the leak). The list is ordered by the size of the leak. See **Figure 21–23** .



Leak size 8192 bytes (total), 1 occurrences
Leak size 8176 bytes (total), 1 occurrences

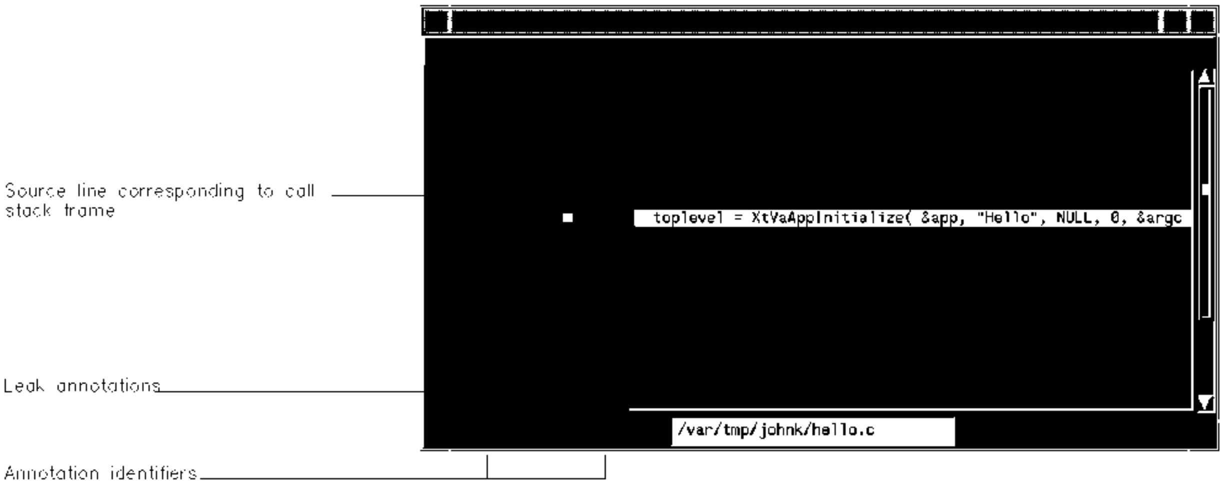
Figure 21–23 Leak View Window with Admin Menu

Note: As an alternative to viewing leaks in Leak View, you can select one or more leaks, choose "Save As Text..." from the Admin menu, and view them separately in a text file along with their call stacks. Multiple items are selected by clicking the first and then either dragging the cursor over the others or shift–clicking the last in the group to be selected.

There are three main areas in the window:

- The leak identification area indicates which leak has been selected, its order in the leak list, and the total amount of memory in bytes lost in leaks.
- The leak list area displays the list of leaks in order of size, largest to smallest. A tabulation is kept for each leak in the experiment and the total quantity of bytes leaked is displayed as the size. For example, the fourth leak in the list shows a total size of 5,498 bytes accumulated over 194 occurrences.

  Note that the currently selected leak is highlighted in pink (in the standard scheme) and that any previously selected leak is highlighted in purple. Leaks are selected by double–clicking.
- The call stack area displays the contents of the call stack when the leak occurred. You can double–click the frame in the call stack to see the source code that caused the leak. **Figure 21–24** shows a typical Source View window with leak annotations (you can change the annotations through the "Preferences..." selection in the Performance Analyzer Config menu).



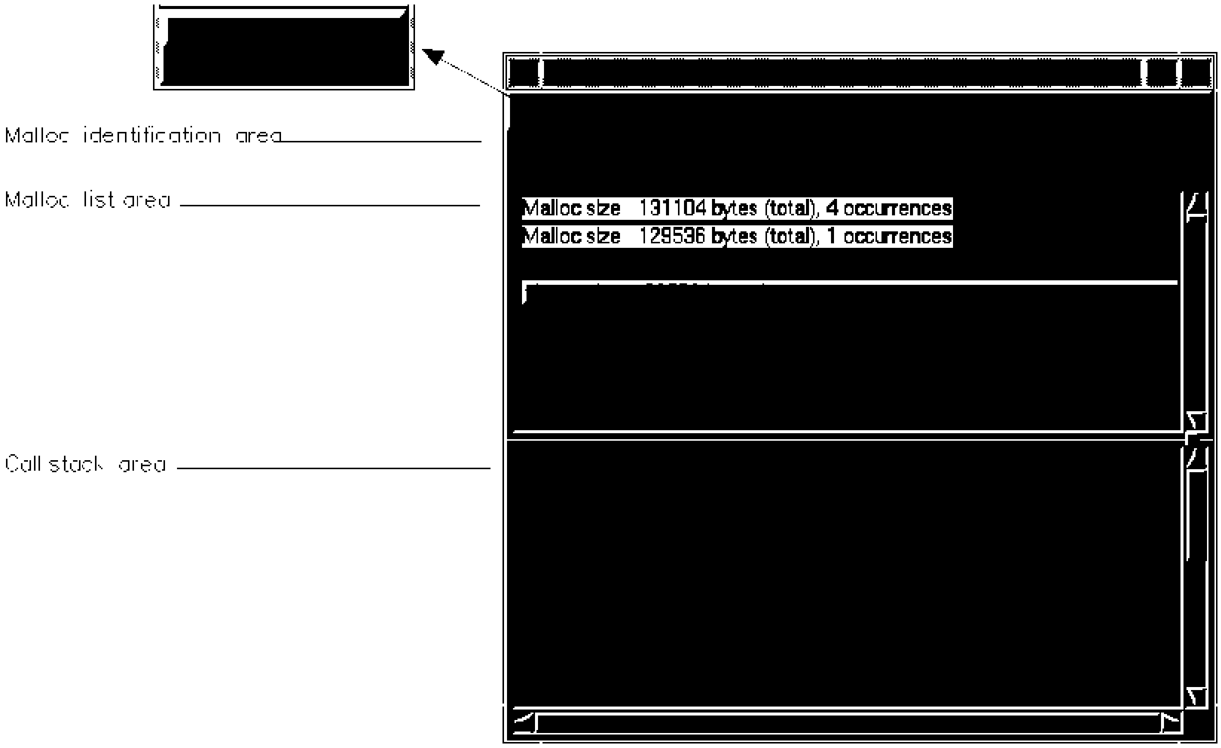
```
toplevel = XtVaAppInitialize( &app, "Hello", NULL, 0, &argc
```

```
/var/tmp/johnk/hello.c
```

Figure 21−24 Source View with Leak Annotations

Analyzing Memory Allocation with Malloc View

One of the reasons that a program’s size increases is that it *mallocs* lots of memory. Malloc View shows you where *mallocs* occur in a Performance Analyzer experiment. Malloc View operates in similar fashion to Leak View. It displays each *malloc* that occurs in your executable, its size, the number of times the *malloc* occurred (a count is kept of *mallocs* with identical call stacks), and the corresponding call stack (when you select the *malloc*). The *mallocs* display in order of size. See **Figure 21−23** . Double−clicking a frame in the call stack displays the corresponding code in Source View with *malloc* size annotations.



Malloc size 131104 bytes (total), 4 occurrences
Malloc size 129536 bytes (total), 1 occurrences

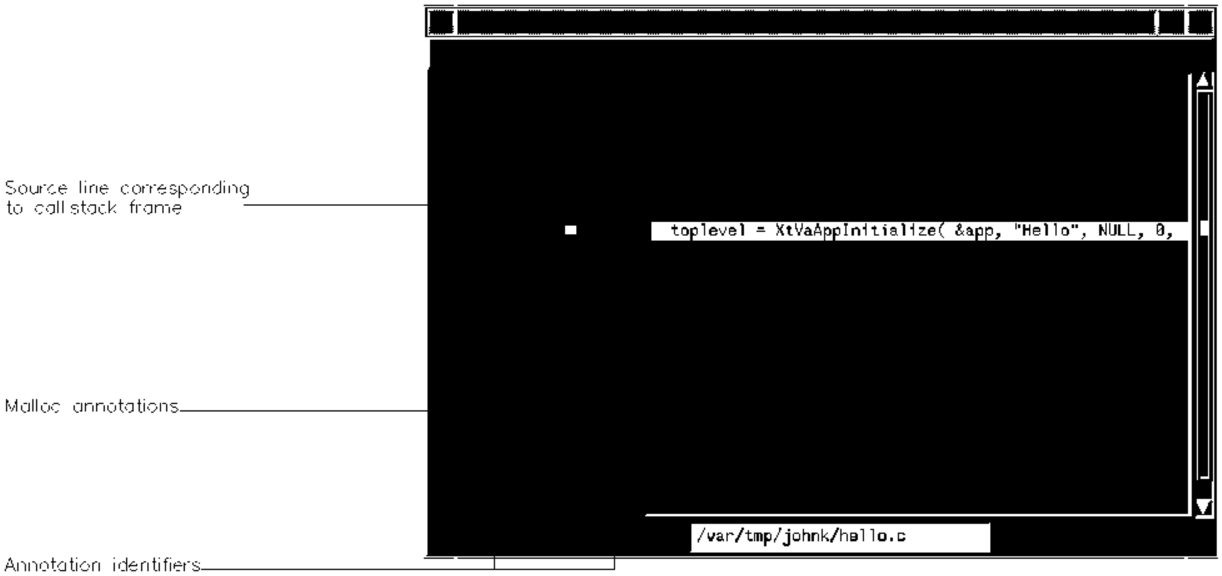
Figure 21–25 Malloc View Window with Admin Menu

Note: In similar fashion to Leak View, you can select one or more *mallocs*, choose "Save As Text..." from the Admin menu, and view them separately in a text file along with their call stacks. Multiple items are selected by clicking the first and then either dragging the cursor over the others or shift–clicking the last in the group to be selected.

There are three main areas in the window:

- The malloc identification area identifies which *malloc* has been selected, its order in the malloc list, and the total size of bytes for all *mallocs*.
- The malloc list area displays the list of *mallocs* in order of size, largest to smallest. The Performance Analyzer accumulates the number of bytes *malloc*ed at each location and displays this number as the total size in bytes, along with the number of times the *malloc* occurred.

  Note that the currently selected *malloc* is highlighted in pink (in the standard scheme) and that any previously selected *malloc* is highlighted in purple. You select *mallocs* by double–clicking.
- The call stack area displays the contents of the call stack when the *malloc* occurred. You can double–click the frame in the call stack to see the source code that caused the *malloc*. **Figure 21–26** shows a typical Source View window with *malloc* annotations (you can change the annotations through the "Preferences..." selection in the Performance Analyzer Config menu).



```
toplevel = XtVaAppInitialize( &app, "Hello", NULL, 0,
```

Figure 21–26 Source View with *malloc* Annotations

Analyzing the Memory Map with Heap View

Heap View lets you use analyze data from an experiment using the "Find Memory Leaks" task. The Heap View window provides a memory map that shows memory problems occurring in the time interval defined by the calipers in the Performance Analyzer window. The map indicates these memory block conditions:

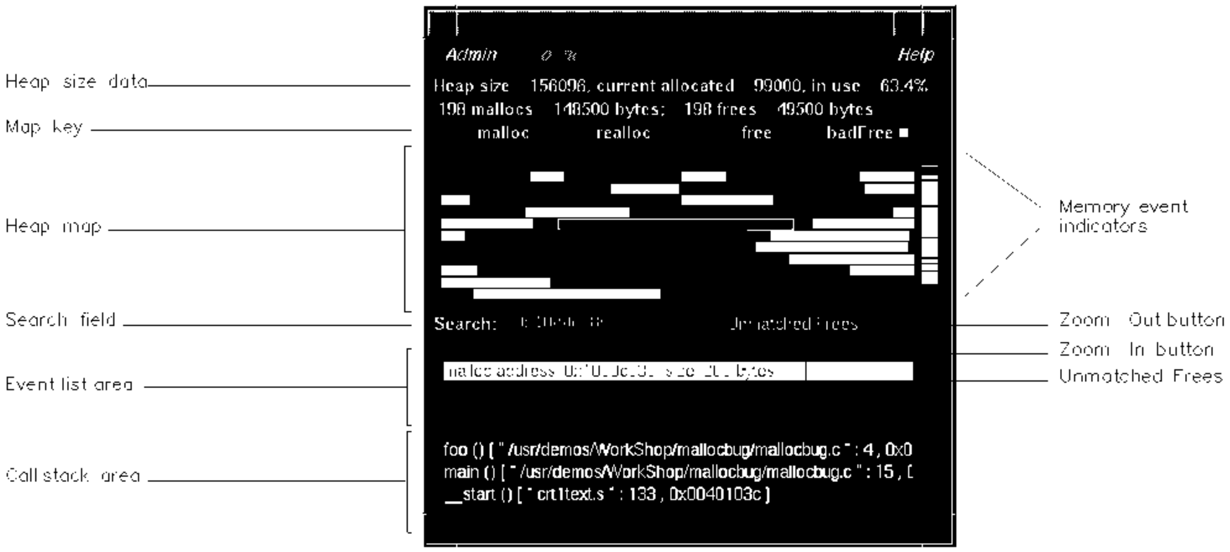
- *malloc*—reserved memory space
- *free*—open space
- *realloc*—reallocated space
- bad *free* space
- unused space

In addition to the Heap View memory map, you can analyze memory leak data using these other tools:

- If you select a memory problem in the map and bring up the Call Stack window, it will show you where the selected problem took place and the state of the call stack at that time.
- The function list in the Performance Analyzer main window shows inclusive *mallocs* and *frees* with bytes used by function for memory leak experiments.
- The Source View window shows inclusive *mallocs* and *frees* and the number of bytes used by source line.

Heap View Window

A typical Heap View window with its parts labeled appears in **Figure 21–27** .



```
Heap size   156096, current allocated   99000, in use   63.4%
 198 mallocs   148500 bytes;   198 frees   49500 bytes
      malloc          realloc              free          badFree ■
```

```
foo () [ " /usr/demos/WorkShop/mallocbug/mallocbug.c " : 4 , 0x0
main () [ " /usr/demos/WorkShop/mallocbug/mallocbug.c " : 15 , 0
__start () [ " crt1text.s " : 133 , 0x0040103c ]
```

Figure 21−27 Heap View Window

The major features of a Heap View window are:

Map key

appears at the top of the heap map area to identify blocks by color. The actual colors depend on your color scheme.

Heap View map area

shows heap memory as a continuous, wrapping horizontal rectangle. The memory addresses begin at the upper left corner and progress from left to right, row by row. The rectangle is broken up into color−coded segments according to memory use status. Clicking a highlighted area in the heap map identifies the type of problem, the memory address where it occurred, and its size in the event list area and the associated call stack in the call stack display area.

Note in **Figure 21−27** that there are only a few problems in the memory at the lower addresses and many more at the higher addresses.

Memory event indicators

appear color−coded in the scroll bar. Clicking an indicator with the middle button scrolls the display to the selected problem.

Search field

provides two functions:

If you enter a memory address in the field, the corresponding position will be highlighted in the heap map. If there was a problem at that location, it will be identified in the event list area. If there was no problem, the address at the beginning of the memory block and its size display.

If you hold down the left mouse button and position the cursor in the heap map, the corresponding address will display in the *Search* field.

Unmatched Frees button

displays a list of bad *free* problems *(frees* that are not matched in the current interval) and their sizes in the event list area (see **Figure 21−28**). You can then enter the address of a bad *free* in the *Search* field, press `<Enter>`, and see the associated call stack. Note that moving the calipers may prove that these are not bad *frees* after all.

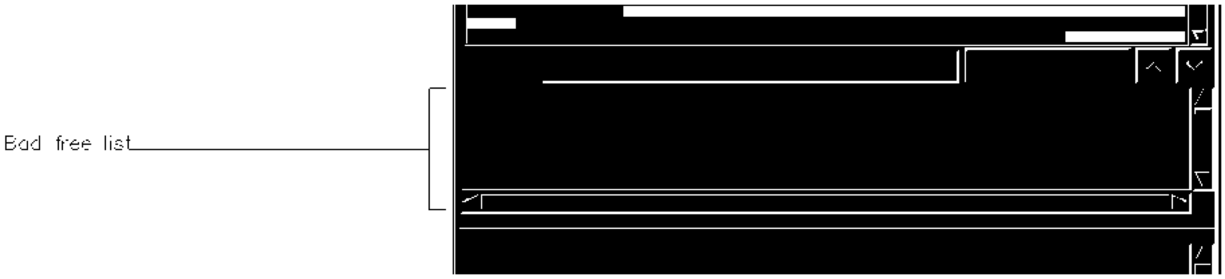


Figure 21−28 Bad Free List in the Heap View Window

Zoom in button

(the upward−pointing arrow) redisplays the heap area at twice the current size of the display. If you reach the limit, an error message displays.

Zoom out button

(the downward−pointing arrow) redisplays the heap area at half the current size (to a limit of one pixel per byte). If you reach the limit, an error message displays.

Event list area

displays the events occurring in the selected block. If only one event was received at the given address, its call stack is shown by default. If more than one event is shown, double−clicking an event will display its corresponding call stack.

Call stack area

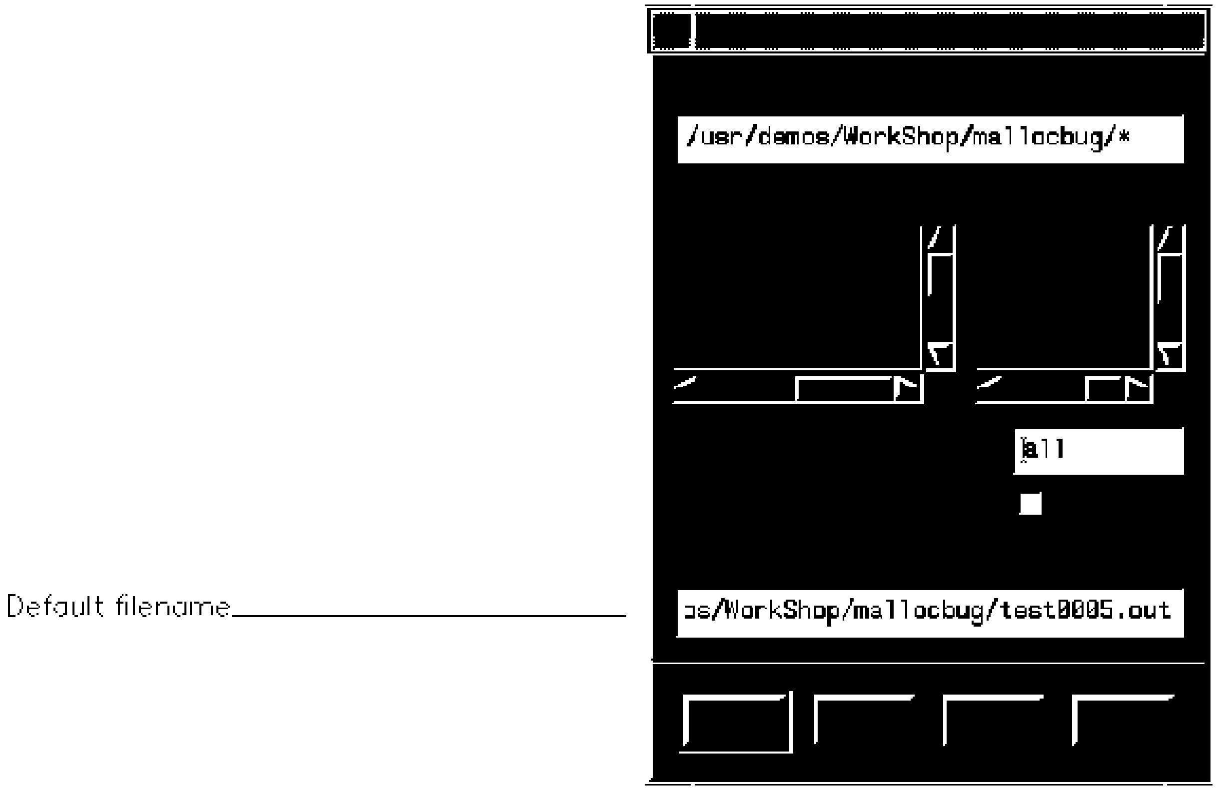
displays the call stack corresponding to the event highlighted in the event list area.

Source View *malloc* Annotations

Like Malloc View, if you double−click a line in the call stack area of the Heap View window, the Source View window displays the portion of code containing the corresponding line, which is highlighted and indicated by a caret (^) with the number of bytes used by *malloc* in the annotation column. See **Figure 21−26** .

Saving Heap View Data as Text

Selecting "Save As Text..." from the Admin menu in Heap View opens the Save Text dialog box, shown in **Figure 21−29** . This lets you select the file name for saving the Heap View data. The default file name suggested is *<experiment−directory>.out*. When you click *OK*, the data for the current caliper setting and the list of unmatched *frees*, if any, is appended to the specified file.



/usr/demos/WorkShop/mallocbug/*

all

Default filename

s/WorkShop/mallocbug/test0005.out

Figure 21–29 Heap View Save Text Dialog Box

The "Save As Text..." selection in the File menu for the Source View from Heap View saves the current file. No file name default is provided, and the file that you name will be overwritten.

Memory Experiment Tutorial

In this tutorial, you will run an experiment to analyze memory usage. The short program below generates memory problems useful that demonstrate how you can use the Performance Analyzer to detect memory problems.

1. Go to the */usr/demos/WorkShop/mallocbug* directory. Note the executable *mallocbug_cv*. This was compiled as follows:

   ```
   cc -g -o mallocbug_cv mallocbug.c -lmalloc_cv -lc
   ```

2. Invoke the Debugger by typing

   ```
   cvd mallocbug_cv
   ```

3. Bring up the Performance Panel by selecting "Performance Task..." from the Admin menu in Main View.
4. Select "Find memory leaks" from the Task menu and click the *OK* button. Then click *Run* to begin the experiment.

   The program runs quickly and terminates.

5. Select "Performance Analyzer" from the "Launch Tool" submenu in the Debugger Admin menu.
6. Select "Leak View..." from the Performance Analyzer Views menu.

   The Leak View window displays, showing one leak and its associated call stack. This leak occurred 99 times at 1,000 bytes each occurrence.

7. Select "Malloc View..." from the Performance Analyzer Views menu.

The Malloc View window displays, indicating two *malloc* locations.

8. Double−click the function *foo* in the call stack area.

   Source View displays showing the function’s code, annotated by the exclusive and inclusive leaks.

9. Select "Heap View..." from the Performance Analyzer Views menu.

   The Heap View window displays. The heap size and percentage used is shown at the top. The heap map area of the window shows the heap map as a continuous, wrapping horizontal rectangle. The rectangle is broken up into color−coded segments, according to memory use status.The color key at the top of the heap map area identifies memory usage as *malloc*, *realloc*, *free*, or bad *free*. Notice also that color−coded indicators showing *mallocs*, *reallocs*, and bad *frees* are displayed in the scroll bar trough. At the bottom of the heap map area are: the *Search:* field for identifying or finding memory locations; the *Unmatched Frees* button for finding unmatched *frees*; a zoom−in control (upwards pointing arrow) and a zoom−out control (downwards arrow).

   The event display area and the call stack are at the bottom of the window. Clicking any event in the heap area displays the appropriate information in these fields.

10. Click on any memory block in the heap map.

    The beginning memory address appears in the *Search:* field. The event information displays in the event field. The call stack information for the last event appears in the call stack area.

11. Select other memory blocks to try out this feature.

    As you select other blocks, the data at the bottom of the Heap View window changes.

12. Double−click on a frame in the call stack.

    A Source View window comes up with the corresponding source code displayed.

13. Close the Source View.

14. Click the *Unmatched Frees* button.

    The data in the Heap View information window changes to display unmatched *frees*. Note that a *free* may be unmatched within the analysis interval, yet it may have a corresponding *free* outside of the interval.

15. Click *Close* to leave the Heap View information window.

16. Select "Exit" from the Admin menu in any open window to end the experiment.

This ends the tutorial.

Call Stack

The Call Stack window accessed from the Performance Analyzer Views menu lets you get call stack information for a sample event selected from one of the Performance Analyzer views. See **Figure 21−23** .

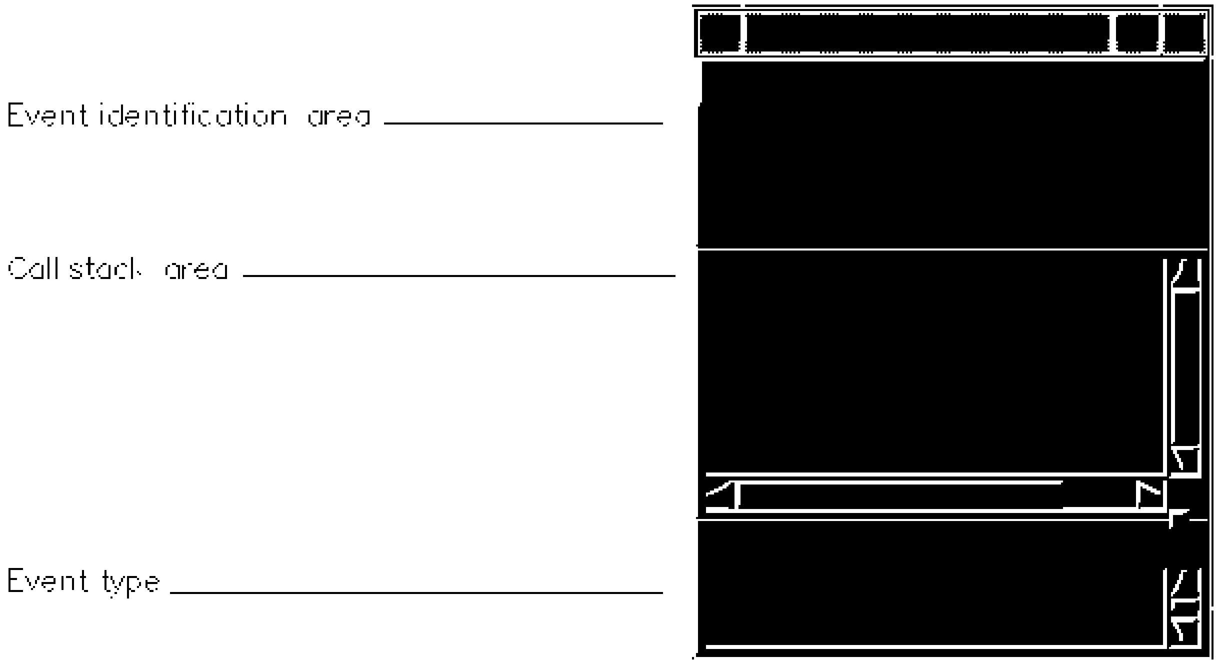


Figure 21–30 Performance Analyzer Call Stack

There are three main areas in the window:

- The event identification area displays the number of the event, its time stamp, and the time within the experiment. If you have a multi–process experiment, the thread will be indicated here.
- The call stack area displays the contents of the call stack when the sample event took place.
- The event type area highlights the type of event and shows the thread in which it was defined and whether the sample was taken in all threads or the indicated thread only, in parentheses.

Chapter 22

Using Tester

This chapter describes the Tester usage model. It shows the general approach of applying Tester for coverage analysis. It contains these sections:

- **"Tester Overview"**
- **"Usage Model"**

Tester Overview

WorkShop Tester is a UNIX™–based software quality assurance toolset for dynamic test coverage over any set of tests. The term *covered* means the test has executed a particular unit of source code. In this product, units are functions, individual source lines, arcs, blocks, or branches. If the unit is a branch, covered means it has been executed under both true and false conditions. This product is intended for software and test engineers and their managers involved in the development, test, and maintenance of long–lived software projects.

WorkShop Tester provides these general benefits:

- Provides visualization of coverage data, which yields immediate insight into quality issues at both engineering and management levels.
- Provides useful measures of test coverage over a set of tests/experiments.
- Lets you view the coverage results of a dynamically shared object (DSO) by executables that use it.
- Provides comparison of coverage over different program versions.

- Provides tracing capabilities for arguments and function arcs that go beyond traditional test coverage tools.
- Supports programs written in C, C++, and Fortran.
- Is integrated into the CASEVision family of products.
- Allows users to build and maintain higher quality software products.

There are two versions of Tester:
- *cvcov* is the command line version of the test coverage program.
- *cvxcov* is the GUI version of the test coverage program.

Most of the functionality is available from either program, although the graphical representations of the data are available only from *cvxcov*, the GUI tool.

Test Coverage Data

Tester provides the following basic coverage:
- Basic block—how many times was this basic block executed?
- Function—how many times was this function executed?
- Branch—did this condition take on both TRUE and FALSE values?

You can also request the following coverage information:
- Arc—was function *F* called by function *A* and function *B*? Which arcs for function *F* were NOT taken?
- Source line coverage—how many times has this source line been executed and what percentage of source lines is covered?
- Argument—what were the maximum and minimum values for argument *X* in function *F* over all tests?
- When the target program *execs*, *forks*, or *sprocs* another program, only the main target is tested, unless you specify which executables are to be tested, the parent and/or child programs.

Note: When you compile with the `-g` flag, you may create assembly blocks and branches that can never be executed, thus preventing "full" coverage from being achieved. These are usually negligible. However, if you compile with the `01` flag (the default), you can increase the number of executable blocks and branches.

Types of Experiments

You can conduct Tester coverage experiments for:
- Separate tests.
- A set of tests operating on the same executable.
- A list of executables related by *fork*, *exec*, or *sproc* commands.
- A test group of executables sharing a common dynamically shared object (DSO).

Experiment Results

Tester presents the experiment results in these reports:
- Summary of test coverage, including user parameterized dynamic coverage metric.
- List of functions, which can be sorted by count, file, or function name and filtered by percentage of block, branch, or function covered.
- Comparison of test coverage between different versions of the same program.
- Source or assembly code listing annotated with coverage data.
- Breakdown of coverage according to contribution by tests within a test set or test group.

The graphical user interface lets you view test results in different contexts to make them more meaningful. It provides:
- Annotated function call graph highlighting coverage by counts and percentage (ASCII function call graph supported as well).
- Annotated Source View showing coverage at the source language level.
- Annotated Disassembly View showing coverage at the assembly language level.
- Bar chart summary showing coverage by functions, lines, blocks, branches, and arcs.

Multiple Tests

Tester supports multiple tests. You can:

- define and run a test set to cover the same program.
- define and run a test group to cover programs sharing a common DSO. This approach is useful if you want to test different client programs that bind with the same libraries.
- automate test execution via command line interface as well as GUI mode.

Test Components

Each test is a named object containing the following:

- instrumentation file—This describes the data to be collected.
- executable—This is the program being instrumented for coverage analysis.
- executable list—If the program you are testing can *fork*, *exec*, or *sproc* other executables and you want these other executables included in the test, then you can specify a list of executables for this purpose.
- command—This defines the program and command line arguments.
- instrumentation directory—The instrumentation directory contains directories representing different versions of the instrumented program and related data. Instrumentation directories are named *ver##<n>* where *n* is the version number. Several tests can share the same instrumentation directory. This is true for tests with the same instrumentation file and program version. The instrumentation directory contains the following files, which are automatically generated:

  ```
  <program|DSO>.Arg     optional arg trace file
  <program|DSO>.Binmap  basic block & branches bitmap file
  <program|DSO>.Graph   arc data
  <program|DSO>.Log     instrumentation log file (cvinstr)
  <program|DSO>.Map     function map file
  <program|DSO>_Instr   instrumented executable
  ```

  As part of instrumentation, you can filter the functions to be included or excluded in your test, through the directives INCLUDE, EXCLUDE, and CONSTRAIN.
- experiment results—Test run coverage results are deposited in a results directory. Results directories are named *exp##<n>* where *n* corresponds to the instrumentation directory used in the experiment. There is one results directory for each version of the program in the instrumentation directory for this test. Note that results are not deposited in the instrumentation directory because the instrumentation directory may be shared by other tests. The results directory is different when you run the test with or without the `-keep` option.

  When you run your test without the `-keep` option the results directory contains the following files:

COV_DESC	description file of experiment
COUNTS_<exe>	counts file for each executable; <exe> is an executable file name
USER_SELECTIONS	instrumentation criteria

  When you run your test with the `-keep` option the results directory contains the following files:

COV_DESC	description file of experiment
COUNTS_ <exe>	counts file for each executable; <exe> is an executable file name.
USER_SELECTIONS	instrumentation criteria
ARGTRACE_<n>	argument trace database; <n> is a unique number for each process
COUNTS_<n>	basic block and branch counts database
DESC	experiment description file
FPTRACE_<n>	function pointer tracing database
LOG	experiment log file (*cvmon*)
TRAP	N/A
USAGE_<n>	N/A

  There are also soft links of the instrumentation data files in the results directory to the instrumentation directory described above.

Usage Model

This section is divided into three parts:

- **"Single Test Analysis Process"** shows the general steps in conducting a test.
- **"Automated Testing"** discusses using scripts to automate your testing.
- **"Additional Coverage Testing"** describes strategies using multiple tests.

Single Test Analysis Process

In performing coverage analysis for a single test, you typically go through the following steps:

1. Plan your test.

   Test tools are only as good as the quality and completeness of the tests themselves.

2. Create (or reuse) an instrumentation file.

   The instrumentation file defines the coverage data you wish to collect in this test. You can define:

   - COUNTS—three types of count items perform tracking. `bbcounts` tracks execution of basic blocks. `fpcounts` counts calls to functions through function pointers. `branchcounts` tracks branches at the assembly language level.
   - INCLUDE/EXCLUDE—lets you define a subset of functions to be covered. INCLUDE adds the named functions to the current set of functions. EXCLUDE removes the named functions from the set of functions. Simple pattern matching is supported for pathnames and function names. The basic component for inclusion/exclusion is of the form:

     ```
     <shared library | program name>:<functionlist>
     ```

     INCLUDE, EXCLUDE, and CONSTRAIN (see below) play a major role in working with DSOs. Tester instruments all DSOs in an executable whether you are testing them or not, so it is necessary to restrict your coverage accordingly. By default, the directory */usr/tmp/cvinstrlib/CacheExclude* is used as the excluded DSOs cache and */usr/tmp/cvinstrlib/CacheInclude* as the included DSOs cache. If you wish to override these defaults, set the CVINSTRLIB environment variable to the desired cache directory.
   - CONSTRAIN—equivalent to EXCLUDE *, INCLUDE <*subset*>. Thus, the only functions in the test will be those named in the CONSTRAIN subset. You can constrain the set of functions in the program to either a list of functions or a file containing the functions to be constrained. The function list file format is:

     ```
     function_1
     function_2
     function_3
     ...
     ```

     You can use the `-file` option to include an ASCII file containing all the functions as follows:

     ```
     CONSTRAIN -file filename
     ```
   - TRACE—lets you monitor argument values in the functions over all experiments. The only restriction is that the arguments must be of the following basic types: *int*, *char*, *long*, *float*, *double*, or *pointer* (treated as a 4-byte *unsigned int*). MAX monitors the maximum value of an argument. MIN monitors the minimum value of an argument. BOUNDS monitors both the minimum and maximum values. RETURN monitors the function return values.

   The default instrumentation file */usr/WorkShop/usr/lib/WorkShop/Tester/default_instr_file* contains:

   ```
   COUNTS -bbcounts -fpcounts -branchcounts
   EXCLUDE libc.so.1:*
   EXCLUDE libC.so:*
   EXCLUDE libInventor.so:*
   EXCLUDE libMrm.so.1:*
   EXCLUDE libUil.so.1:*
   EXCLUDE libX11.so.1:*
   ```

```
EXCLUDE libXaw.so:*
EXCLUDE libXawI18n.so:*
EXCLUDE libXext.so:*
EXCLUDE libXi.so:*
EXCLUDE libXm.so.1:*
EXCLUDE libXmu.so:*
EXCLUDE libXt.so:*
EXCLUDE libcrypt.so:*
EXCLUDE libcurses.so:*
EXCLUDE libdl.so:*
EXCLUDE libfm.so:*
EXCLUDE libgen.so:*
EXCLUDE libgl.so:*
EXCLUDE libil.so:*
EXCLUDE libks.so:*
EXCLUDE libmf.so:*
EXCLUDE libmls.so:*
EXCLUDE libmutex.so:*
EXCLUDE libnsl.so:*
EXCLUDE librpcsvc.so:*
EXCLUDE libsocket.so:*
EXCLUDE libtbs.so:*
EXCLUDE libtermcap.so:*
EXCLUDE libtermlib.so:*
EXCLUDE libtt.so:*
EXCLUDE libview.so:*
EXCLUDE libw.so:*
EXCLUDE nis.so:*
EXCLUDE resolv.so:*
EXCLUDE straddr.so:*
EXCLUDE tcpip.so:*
```

The excluded items are all dynamically shared objects that might interfere with the testing of your main program.

Note: If you do not use the *default_instr_file* file, functions in shared libraries will be included by default, unless your instrumentation file excludes them.

The minimum instrumentation file contains the line:

```
COUNTS -bbcounts
```

You create an instrumentation file using your preferred text editor. Comments are allowed only at the beginning of a new line and are designated by the "#" character. Lines can be continued using a back slash (\) for lists separated with commas. White space is ignored. Keywords are case insensitive. Options and user−supplied names are case sensitive. All lines are additive to the overall experiment description.

Here is a typical instrument file:

```
COUNTS -bbcounts -fpcounts -branchcounts
# defines the counting options, in this case,
# basic blocks, function pointers, and branches.
CONSTRAIN program:abc, xdr*, functionF, \
 classX::methodY, *::methodM, functionG
# constrains the set of functions in the
# "program" to the list of user specified functions
TRACE BOUNDS functionF(argA)
# traces the upper and lower values of argA
TRACE MAX classX::methodY(argZ)
# traces the maximum value of argZ
```

```
EXCLUDE libc.so.1:*
...
```

Note: Instrumentation can increase the size of a program two to five times. Using DSO caching and sharing can alleviate this problem.

3. Apply the instrument file to the target executable(s).

   This is the instrumentation process. You can specify a single executable or more than one if you are creating other processes through *fork*, *exec*, or *sproc*.

   The command line interface command is *runinstr*. The graphical user interface equivalent is the "Run Instrumentation" selection in the Test menu.

   The effect of performing a run instrument operation is shown in **Figure 22–1** . An instrumentation directory is created (*.../ver##<n>*). It contains the instrumented executable and other files used in instrumentation.

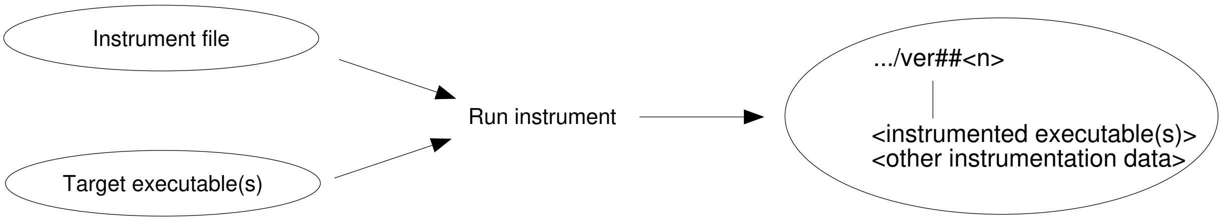


Figure 22–1 Instrumentation Process

4. Create the test directory.

   This part of the process creates a test data directory (*test0000*) containing a test description file named *TDF*. See **Figure 22–2** .

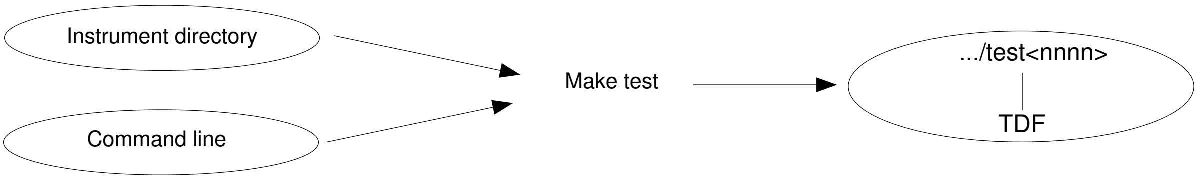


Figure 22–2 Make Test Process

   Tester names the test directory *test0000* by default and increments it automatically for subsequent *make test* operations. You can supply your own name for the test directory if you prefer.

   The TDF file contains information necessary for running the test. A typical *TDF* file contains the test name, type, command–line arguments, instrument directory, description, and list of executables. In addition, for a test set or test group, the TDF file contains a list of subtests.

   Note that *Instrument Directory* can be either the instrumentation directory itself (such as ver##0) or a directory containing one or more instrumentation subdirectories.

   The command line interface command is *mktest*. The graphical user interface equivalent is the "Make Test" selection in the Test menu.

5. Run the instrumented version of the executable to collect the coverage data.

   This creates a subdirectory (*exp##0*) under the test directory in which results from the current experiment will be placed. See **Figure 22–3** . The commands to run a test use the most recent instrumentation directory version unless you specify a different directory.

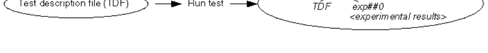


Figure 22–3 Run Test Process

The command line interface command is *runtest*. The graphical user interface equivalent is the "Run Test" selection in the Test menu.

6. Analyze the results.

Tester provides a variety of column–based presentations for analyzing the results. The data can be sorted by a number of criteria. In addition, the graphical user interface can display a call graph indicating coverage by function and call.

The Tester interface provides many kinds of queries for performing analysis on a single test. **Table 22–1** shows query commands for a single test that are available either from the command line or the graphical user interface Queries menu.

Table 22–1 Common Queries for a Single Test

Command Line	Graphical User Interface	Description
lsarc	List Arcs	Shows the function arc coverage. An arc is a call from one function to another.
lsblock	List Blocks	Shows basic block count information.
lsbranch	List Branches	Shows the count information for assembly language branches.
lsfun	List Functions	Shows coverage by function.
lssum	List Summary	Provides a summary of overall coverage.
lstrace	List Argument Traces	Shows the results of argument tracing, including argument, type, and range.
lsline	List Line Coverage	Shows coverage for native source lines.
cattest	Describe Test	Describes the test details.
diff	Compare Test	Shows the difference in coverage between programs.
lsinstr	List Instrumentation	Show instrumentation details for a test.

Other queries are accessed differently from either interface.

- *lscall*—shows a function graph indicating caller and callee functions and their counts. From the graphical user interface, function graphs are accessed from a Call Tree View (Views menu selection).
- *lssource*—displays the source or assembly code annotated with the execution count by line. From the graphical user interface, you access source or assembly code from a Source View (using the *Source* button) or a Disassembly View (using the *Disassembly* button), respectively.

The queries available in the graphical user interface are shown in **Figure 22–4** .

Figure 22−4 The Queries Menu from Main Tester Window

Automated Testing

Tester is best suited to automated testing of command−line programs, where the test behavior can be completely specified at the invocation. Command−line programs let you incorporate contextual information, such as environment variables and current working directory.

Automated testing of server processes in a client−server application proceeds basically the same as single−program cases except that startup time introduces a new factor. Tester can substantially increase the startup time of your target process so that the instrumented target process will run somewhat slower than the standard, uninstrumented one. Tests which start a server, wait a while for it to be ready, and then start the client will have to wait considerably longer. The additional time depends on the size and complexity of the server process itself and on how much and what kind of data you have asked Tester to collect. You will have to experiment to see how long to wait.

Automated testing of interactive or nondeterministic tests is somewhat harder. These tests are not completely determined by their command line; they can produce different results (and display different coverage) from the same command line, depending upon other factors, such as user input or the timing of events. For tests such as these, Tester provides a `-sum` argument to the runtest command. Normally each test run is treated as an independent event, but when you use "runtest −sum," the coverage from each run is added to the coverage from previous runs of the same test case. Other details of the coverage measurement process are identical to the first case.

In each case, you first need to instrument your target program, then run the test, sum the test results if desired, and finally analyze the results. There are two general approaches to applying *cvcov* in automated testing

- If you have not yet created any test scripts or have a small number of tests, you should create a script that makes each test individually and then runs the complete test set. See **Example 22−1** , a script that automates a test program called *target* with different arguments:

Example 22−1 Making Tests and Then Running Them

```
# instrument program
cvcov runinstr -instr_file instrfile mypath/target
# test machinery
# make all tests
cvcov mktest -cmd "target A B C" -testname test0001
cvcov mktest -cmd "target D E F" -testname test0002
...
# define testset to include all tests
cvcov lstest > mytest_list
cvcov mktset -list mytest_list -testname mytestset
# run all tests in testset and sum up results
cvcov runtest mytestset
```

- If you have existing test scripts of substantial size or an automated test machinery setup, then you may find it straightforward to embed Tester by replacing each test line with a script containing two Tester command lines for making and running the test and then accumulating the results in a testset, such as in **Example 22−2** . Of course, you can also rewrite the whole test machinery as described in **Example 22−1** .

Example 22−2 Applying a Make−and−Run Script

```
# instrument program
cvcov runinstr -instr_file instrfile mypath/target
# test machinery
# make and run all tests
make_and_run "target A B C"
make_and_run "target D E F"
...
# make testset
```

```
cvcov lstest > mytestlist
cvcov mktset -list mytestlist -testname mytestset
# accumulate results
cvcov runtest mytestset
```

where the *make_and_run* script is:

```
#!/bin/sh
testname=`cvcov mktest -instr_dir /usr/tmp -cmd "$*"`
testname=`expr "$testname" : ".*Made test directory: `.*'"`
cvcov runtest $testname
```

Note that both examples use simple testset structures—these could have been nested hierarchically if desired.

After running your test machinery, you can use *cvcov* or *cvxcov* to analyze your results. Make sure that your test machinery does not remove the products of the test run (even if the test succeeds), or it may destroy the test coverage data.

Additional Coverage Testing

After you have created and run your first test, you typically need additional testing. Here are some scenarios.

- You can define a test set so that you can vary your coverage using the same instrumentation. You can analyze the new tests singly or you can combine them in a set and look at the cumulative results. If the tests are based on the same executable, they can share the same instrumentation file. You can also have a test set with tests based on different executables but they should have the same instrumentation file.
- You can change the instrumentation criteria to gather different counts, examine a different set of functions, or perform argument tracing differently.
- You can create a script to run tests in batch mode (command line interface only).
- You can run different programs that use a common dynamically shared object (DSO) and accumulate test coverage for a test group containing the DSO.
- You can run the same tests using the same instrumentation criteria for two versions of the same program and compare the coverage differences.
- You can run a test multiple times and sum the result over the runs. This is typically used for GUI-based applications.

As you conduct more tests, you will be creating more directories. A typical coverage testing hierarchy is shown in **Figure 22-5** .

There are two different instrumentation directories, *ver##0* and *ver##1*. The test directory *test0000* contains results for a single experiment that uses the instrumentation from *ver##0*. (Note that the number in the name of the experiment results directory corresponds to the number of the instrumentation directory.) Test directory *test0001* has results for two experiments corresponding to both instrumentation directories, *ver##0* and *ver##1*.

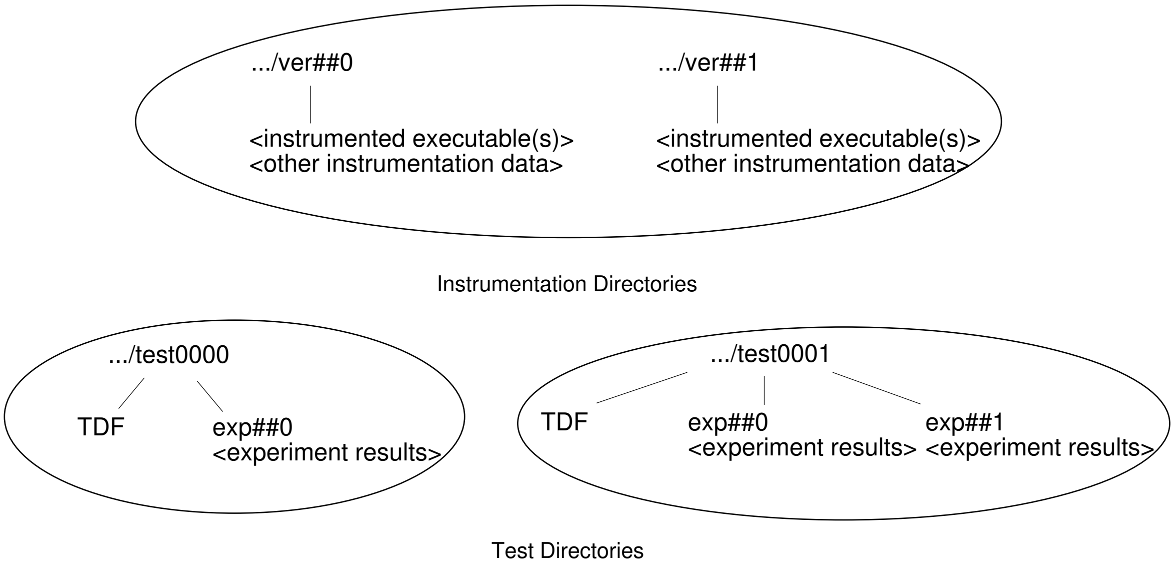


Figure 22–5 Typical Coverage Testing Hierarchy

Chapter 23

Tester Command Line Interface Tutorial

The tutorials in this chapter are based on simple programs written in C. To run them, you need the C compiler. The chapter is broken down into these sections:

- **"Setting Up the Tutorials"** shows you how to run the script that creates the files needed for the tutorials.
- **"Tutorial #1: Analyzing a Single Test"** takes you through the steps of performing coverage analysis for a single test.
- **"Tutorial #2: Analyzing a Test Set"** discusses creating additional tests to achieve full coverage.
- **"Tutorial #3: Optimizing a Test Set"** explains how to fine−tune a test set to eliminate redundant tests.
- **"Tutorial #4: Analyzing a Test Group"** explains how you would use a test group to analyze the coverage of a dynamically shared object (DSO) in different executables sharing the DSO.

Note that if you are going to run these tutorials, you must run them in order; each tutorial builds on the results of previous tutorials.

If at any time a command syntax is not clear, enter:

```
% cvcov help < commandname   >
```

Setting Up the Tutorials

1. Enter the following to set up the tutorials:

   ```
   % cp −r /usr/demos/WorkShop/Tester /usr/tmp/tutorial
   % cd /usr/tmp/tutorial
   % echo ABCDEFGHIJKLMNOPQRSTUVWXYZ > alphabet
   % make −f Makefile.tutorial copyn
   ```

   This moves some scripts and source files used in the tutorial to */usr/tmp/tutorial*, creates a test file named *alphabet*, and makes a simple program, *copyn*, which copies *n* bytes from a source file to a target file.

2. To see how the program works, try a simple test by typing:

   ```
   % copyn alphabet targetfile 10
   % cat targetfile
   ```

```
ABCDEFGHIJ
```

You should see the first 10 bytes of *alphabet* copied to *targetfile*.

Tutorial #1: Analyzing a Single Test

Tutorial #1 discusses the following topics:

- Instrumenting an executable
- Making a test
- Running a test
- Analyzing test coverage data

Instrumenting an Executable

This is the first step in providing test coverage. The user defines the instrumentation criteria in an instrumentation file.

1. Enter the following to see the instrumentation directives in the file *tut_instr_file* used in the tutorials:

   ```
   % cat tut_instr_file
   COUNTS -bbcounts -fpcounts -branchcounts
   CONSTRAIN main, copy_file
   TRACE BOUNDS copy_file(size)
   ```

   We will be getting all counting information (blocks, functions, branches, and arcs) for the two functions specified in the CONSTRAIN directive, *main* and *copy_file*. We will also be tracing the *size* argument for the *copy_file* function.

2. Enter the following command to instrument *copyn*:

   ```
   % cvcov runinstr -instr_file tut_instr_file copyn
   cvcov: Instrument "copyn" of version "0" succeeded.
   ```

   Directory *ver##0* has been created by default. This contains the instrumented executable, *copyn_Instr*, and other instrumentation data.

Making a Test

A *test* defines the program and arguments to be run, instrument directory, executables, and descriptive information about the test.

1. Enter the following to make a test:

   ```
   %   cvcov mktest -cmd "copyn alphabet targetfile 20"
   ```

   You will see the message:

   ```
   cvcov: Made test directory: "/usr/var/tmp/tutorial/test0000"
   ```

   Directory *test0000* has been created by default. It contains a single file, *TDF*, the test description file.

   Note: The directory */usr/var/tmp* is linked to */usr/tmp*.

2. Enter the following to get a textual listing of the test:

   ```
   %   cvcov cattest test0000
   Test Info               Settings
   -----------------------------------------------------
   Test                    /usr/var/tmp/tutorial/test0000
   Type                    single
   Description
   Command Line            copyn alphabet targetfile 20
   Number of Exes          1
   ```

```
Exe List                copyn
Instrument Directory    /usr/var/tmp/tutorial
Experiment List
```

Running a Test

To run a test, we use technology from the WorkShop Performance Analyzer. The instrumented process is set to run, and a monitor process (*cvmon*) captures test coverage data by interacting with the WorkShop process control server (*cvpcs*).

1. Enter the following command:

   ```
   % cvcov runtest test0000
   ```

   You will see the message:

   ```
   cvcov: Running test "/usr/var/tmp/tutorial/test0000" ...
   ```

   Now the directory *test0000* contains the directory *exp##0,* which contains the results of the first test experiment.

Analyzing Test Coverage Data

You can analyze test coverage data many ways. In this tutorial, we will illustrate a simple top–down approach. We'll start at the top to get a summary of overall coverage, proceed to the function level, and go finally to the actual source lines.

1. Enter the following to get the summary:

   ```
   % cvcov lssum test0000
   ```

   You will see the display shown in **Example 23–1** .

Example 23–1 `lssum` Example

```
% cvcov lssum test0000
Coverages                   Covered     Total       % Coverage    Weight
--------------------------------------------------------------------------
Function                    2           2           100.00%       0.400
Source Line                 17          35          48.57%        0.200
Branch                      0           10          0.00%         0.200
Arc                         8           18          44.44%        0.200
Block                       19          42          45.24%        0.000
Weighted Sum                                        58.60%        1.000
```

Notice that although both functions have been covered, we have incomplete coverage for source lines, branches, arcs, and blocks.

Note: Items are highlighted on your screen to emphasize null coverage. As a convention in this manual, we're showing highlighting or user input in boldface.

2. Enter the following to look at the line count information for the *main* function:

   ```
   % cvcov lssource main test0000
   ```

   This produces a source listing annotated with counts, shown in **Example 23–2** .

Example 23–2 `lssource` Example

```
% cvcov lssource main test0000
Counts  Source
---------------------------------------------------------------------
          #include <stdio.h>
          #include <sys/types.h>
```

```
          #include <sys/stat.h>
          #include <fcntl.h>
          #define OPEN_ERR         1
          #define NOT_ENOUGH_BYTES 2
          #define SIZE_0           3

          int copy_file();
          main (int argc, char *argv[])
1       {
              int bytes, status;
1           if( argc < 4){
0               printf("copyn: Insufficient arguments.\n");
0               printf("Usage: copyn f1 f2 bytes\n");
0               exit(1);
              }
1           if( argc > 4 ) {
0               printf("Error: Too many arguments\n");
0               printf("Usage: copyn f1 f2 bytes\n");
0               exit(1);
              }
1           bytes = atoi(argv[3]);
1           if(( status = copy_file(argv[1], argv[2], bytes)) >0){
0              switch ( status) {
                     case SIZE_0:
0                      printf("Nothing to copy\n");
0                      break;
                     case NOT_ENOUGH_BYTES:
0                      printf("Not enough bytes\n");
0                      break;
                     case OPEN_ERR:
0                      printf("File open error\n");
0                      break;
                 }
0              exit(1);
              }
1       }

   int copy_file( source, destn, size)
          char *source, *destn;
          int size;
1       {
              char *buf;
              int fd1, fd2;
              struct stat fstat;
1           if( (fd1 = open( source, O_RDONLY)) <= 0){
0               return OPEN_ERR;
              }
1           stat( source, &fstat);
1           if( size <= 0){
0               return SIZE_0;
              }
1           if( fstat.st_size < size){
0               return NOT_ENOUGH_BYTES;
              }
1           if( (fd2 = creat( destn, 00777)) <= 0){
0               return OPEN_ERR;
              }
1           buf = (char *)malloc(size);
```

```
1          read( fd1, buf, size);
1          write( fd2, buf, size);
1          return 0;
0       }
```

Notice that the 0−counted lines appear in a highlight color. In this example, the lines with 0 counts occur where there is an error condition. This is our first good look at branch and block coverage at the source line level. The branch and block coverage in the summary are at the assembly language level.

Tutorial #2: Analyzing a Test Set

In the second tutorial, we are going to create additional tests with the objective of achieving 100% overall coverage. From examining the source code in **Example 23−2** , it seems that the 0−count lines in *main* and *copy_file* are due to error−checking code that is not tested by *test0000*.

Note: This tutorial needs *test0000*, which was created in the previous tutorial.

The script *tut_make_testset* is supplied to demonstrate how to set up this test set.

1. Enter `sh -x tut_make_testset` to run the script.

   Example 23−3 shows the first portion of the script (as it runs), in which the individual tests are created. The *tut_make_testset* script uses *mktest* to create eight additional tests. The tests *test0001* and *test0002* pass too few and too many arguments, respectively. *test0003* attempts to copy from a nonexistent file named *no_file*. *test0004* attempts to pass 0 bytes, which is illegal. *test0005* attempts to copy 20 bytes from a file called *not_enough*, which contains only one byte. In *test0006*, we attempt to write to a directory without proper permission. *test0007* tries to copy too many bytes. In *test0008*, we attempt to copy from a file without read permission.

Example 23−3 *tut_make_testset* Script: Making Individual Tests

```
% sh -x tut_make_testset
+ cvcov mktest -cmd copyn alphabet target -des not enough arguments
cvcov: Made test directory: "/usr/var/tmp/tutorial/test0001"
+ cvcov mktest -cmd copyn alphabet target 20 extra_arg \
-des too many arguments
cvcov: Made test directory: "/usr/var/tmp/tutorial/test0002"
+ cvcov mktest -cmd copyn no_file target 20 -des cannot access file
cvcov: Made test directory: "/usr/var/tmp/tutorial/test0003"
+ cvcov mktest -cmd copyn alphabet target 0 -des pass bad size arg
cvcov: Made test directory: "/usr/var/tmp/tutorial/test0004"
+ echo a
+ cvcov mktest -cmd copyn not_enough target 20 -des not enough data \
(less bytes than requested) in original file
cvcov: Made test directory: "/usr/var/tmp/tutorial/test0005"
+ cvcov mktest -cmd copyn alphabet /usr/bin/target 20 \
-des cannot create target executable due to permission problems
cvcov: Made test directory: "/usr/var/tmp/tutorial/test0006"
+ ls -ld /usr/bin
drwxr-xr-x    3 root     sys         3584 May 12 18:25 /usr/bin
+ cvcov mktest -cmd copyn alphabet targetfile 200
-des size arg too big
cvcov: Made test directory: "/usr/var/tmp/tutorial/test0007"
+ cvcov mktest -cmd copyn /usr/adm/sulog targetfile 20 \
-des no read permission on source file
cvcov: Made test directory: "/usr/var/tmp/tutorial/test0008"
```

After the individual tests are created, the script uses *mktset* to make a new test set and *addtest* to include the new tests in the set. **Example 23–4** shows the portion of the script in which the test set is created and the individual tests are added to the test set.

Example 23–4 *tut_make_testset* Script: Making and Adding to the Test Set

```
+ cvcov mktset -des full coverage testset -testname tut_testset
cvcov: Made test directory: "/usr/var/tmp/tutorial/tut_testset"
+ cvcov addtest test0000 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0000" to "tut_testset"
+ cvcov addtest test0001 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0001" to "tut_testset"
+ cvcov addtest test0002 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0002" to "tut_testset"
+ cvcov addtest test0003 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0003" to "tut_testset"
+ cvcov addtest test0004 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0004" to "tut_testset"
+ cvcov addtest test0005 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0005" to "tut_testset"
+ cvcov addtest test0006 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0006" to "tut_testset"
+ cvcov addtest test0007 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0007" to "tut_testset"
+ cvcov addtest test0008 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0008" to "tut_testset"
```

2. Enter **`cvcov cattest tut_testset`** to check that the new test set was created correctly.

   This is shown in **Example 23–5**. The index numbers in brackets in the subtest list are used to identify the individual tests as part of a test set. This index is used to list the contribution of each test.

Example 23–5 Contents of the New Test Set

```
% cvcov cattest tut_testset
Test Info                Settings
----------------------------------------------------------
Test                    /usr/var/tmp/tutorial/tut_testset
Type                    set
Description             full coverage testset
Number of Exes          1
Exe List                copyn
Number of Subtests      9
Subtest List
                          [0] /usr/var/tmp/tutorial/test0000
                          [1] /usr/var/tmp/tutorial/test0001
                          [2] /usr/var/tmp/tutorial/test0002
                          [3] /usr/var/tmp/tutorial/test0003
                          [4] /usr/var/tmp/tutorial/test0004
                          [5] /usr/var/tmp/tutorial/test0005
                          [6] /usr/var/tmp/tutorial/test0006
                          [7] /usr/var/tmp/tutorial/test0007
                          [8] /usr/var/tmp/tutorial/test0008
Experiment List
```

3. Enter the following to run the tests in the test set:

   ```
   % cvcov runtest tut_testset
   ```

   By applying the *runtest* command to the test set, we can run all the tests together. See **Example 23–6**. Note that when you run a test set, only tests without results are run; tests that already have results will not be run again. In this case, *test0000* has already been run. If you need to rerun a test, you can do so using the `-force` flag.

Example 23–6 Running the New Test Set

```
% cvcov runtest tut_testset
cvcov: Running test "/usr/var/tmp/tutorial/test0000" ...
cvcov: Running test "/usr/var/tmp/tutorial/test0001" ...
copyn: Insufficient arguments.
Usage: copyn f1 f2 bytes
cvcov: Running test "/usr/var/tmp/tutorial/test0002" ...
Error: Too many arguments
Usage: copyn f1 f2 bytes
cvcov: Running test "/usr/var/tmp/tutorial/test0003" ...
File open error
cvcov: Running test "/usr/var/tmp/tutorial/test0004" ...
Nothing to copy
cvcov: Running test "/usr/var/tmp/tutorial/test0005" ...
Not enough bytes
cvcov: Running test "/usr/var/tmp/tutorial/test0006" ...
File open error
cvcov: Running test "/usr/var/tmp/tutorial/test0007" ...
Not enough bytes
cvcov: Running test "/usr/var/tmp/tutorial/test0008" ...
File open error
```

4. Enter **`cvcov lssum tut_testset`** to list the summary for the test set.

   Example 23–7 shows the results of the tests in the new test set with *lssum*.

Example 23–7 Examining the Results of the New Test Set

```
% cvcov lssum tut_testset
Coverages                    Covered     Total       % Coverage    Weight
--------------------------------------------------------------------------
Function                     2           2           100.00%       0.400
Source Line                  35          35          100.00%       0.200
Branch                       9           10          90.00%        0.200
Arc                          18          18          100.00%       0.200
Block                        39          42          92.86%        0.000
Weighted Sum                                         98.00%        1.000
```

5. Enter **`cvcov lssource main tut_testset`** to see the coverage for the individual source lines as shown in **Example 23–8** .

Example 23–8 Source with Counts

```
% cvcov lssource main tut_testset
Counts  Source
--------------------------------------------------------------------
          #include <stdio.h>
          #include <sys/types.h>
          #include <sys/stat.h>
          #include <fcntl.h>
          #define OPEN_ERR        1
          #define NOT_ENOUGH_BYTES 2
          #define SIZE_0          3

          int copy_file();

          main (int argc, char *argv[])
9       {
              int bytes, status;
```

```
9     if( argc < 4){
1       printf("copyn: Insufficient arguments.\n");
1       printf("Usage: copyn f1 f2 bytes\n");
1       exit(1);
       }
8     if( argc > 4 ) {
1       printf("Error: Too many arguments\n");
1       printf("Usage: copyn f1 f2 bytes\n");
1       exit(1);
       }
7     bytes = atoi(argv[3]);
7     if(( status = copy_file(argv[1], argv[2], bytes)) >0){
6      switch ( status) {
          case SIZE_0:
1          printf("Nothing to copy\n");
1          break;
          case NOT_ENOUGH_BYTES:
2          printf("Not enough bytes\n");
2          break;
          case OPEN_ERR:
3          printf("File open error\n");
3          break;
        }
6      exit(1);
    }
1   }

     int copy_file( source, destn, size)
     char *source, *destn;
     int size;
7   {
       char *buf;
       int fd1, fd2;
       struct stat fstat;
7     if( (fd1 = open( source, O_RDONLY)) <= 0){
2       return OPEN_ERR;
       }
5     stat( source, &fstat);
5     if( size <= 0){
1       return SIZE_0;
       }
4     if( fstat.st_size < size){
2       return NOT_ENOUGH_BYTES;
       }
2     if( (fd2 = creat( destn, 00777)) <= 0){
1       return OPEN_ERR;
       }
1     buf = (char *)malloc(size);

1    read( fd1, buf, size);
1    write( fd2, buf, size);
1    return 0;
0   }
```

As you look at the source code, notice that all lines are covered.

6. Enter **`cvcov lssource -asm main tut_testset`** to see the coverage for the individual assembly lines.

When we list the assembly code using *lssource −asm*, we find that not all blocks and branches are covered at the assembly level. This is due to compilation with the `−g` flag, which adds debugging code that can never be executed.

Enter **`cvcov lsline tut_testset`** to see the coverage at the source line level. Notice that 100% of the lines have been covered.

Tutorial #3: Optimizing a Test Set

Tester lets you look at the individual test coverages in a test set. When you put together a set of tests, you may wish to improve the efficiency of your coverage by eliminating redundant tests. The *lsfun*, *lsblock*, and *lsarc* commands all have the `−contrib` option, which displays coverage result contributions by individual tests. Let's look at the contributions by tests for the test set we just ran, *tut_testset*.

Note: This tutorial needs *tut_testset* and all its subtests; these were created in the previous tutorial.

1. Enter **`cvcov lsfun −contrib −pretty tut_testset`** to see the function coverage test contribution.

   Example 23−9 shows how the test set covers functions. Note that the subtests are identified by index numbers; use *cattest* if you need to map these results back to the test directories.

Example 23−9 Test Contributions by Function

```
% cvcov lsfun −contrib −pretty tut_testset
Functions     Files        Counts
----------------------------------------
main          copyn.c       9
copy_file     copyn.c       7

Functions    Files       [0]    [1]    [2]    [3]    [4]    [5]
------------------------------------------------------------------
main         copyn.c     1      1      1      1      1      1
copy_file    copyn.c     1      0      0      1      1      1

Functions    Files       [6]    [7]    [8]
----------------------------------------------
main         copyn.c     1      1      1
copy_file    copyn.c     1      1      1
```

At the function level, each test covers both functions except for Tests [1] and [2]. The information here is not sufficient to tell us if we have optimized the test set. To do this, we must look at contributions at the arc and block levels. Tester shows arc and block coverage information by test when you apply the `−contrib` flag to *lsarc* and *lsblock*, respectively.

2. Enter the following to see the arc coverage test contribution.

```
% cvcov lsarc −contrib −pretty tut_testset
```

   Example 23−10 shows the individual test contributions. Notice that Tests [5] and [7] have identical coverage to each other; so do Tests [3] and [8].

   We can get additional information by looking at block coverage, confirming our hypothesis about redundant tests.

Example 23−10 Arc Coverage Test Contribution Portion of Report

```
Callers      Callees      Line        Files       [0]     [1]     [2]     [3]
    [4]     [5]
--------------------------------------------------------------------------
----------------
```

```
main        copy_file   27          copyn.c     1      0      0      1
      1      1
main        printf      17          copyn.c     0      1      0      0
      0      0
main        printf      18          copyn.c     0      1      0      0
      0      0
main        exit        19          copyn.c     0      1      0      0
      0      0
main        printf      22          copyn.c     0      0      1      0
      0      0
main        printf      23          copyn.c     0      0      1      0
      0      0
main        exit        24          copyn.c     0      0      1      0
      0      0
main        atoi        26          copyn.c     1      0      0      1
      1      1
main        printf      30          copyn.c     0      0      0      0
      1      0
main        printf      33          copyn.c     0      0      0      0
      0      1
main        printf      36          copyn.c     0      0      0      1
      0      0
main        exit        39          copyn.c     0      0      0      1
      1      1
copy_file   _open       50          copyn.c     1      0      0      1
      1      1
copy_file   _stat       53          copyn.c     1      0      0      0
      1      1
copy_file   _creat      60          copyn.c     1      0      0      0
      0      0
copy_file   _malloc     63          copyn.c     1      0      0      0
      0      0
copy_file   _read       65          copyn.c     1      0      0      0
      0      0
copy_file   _write      66          copyn.c     1      0      0      0
      0      0

Callers     Callees     Line        Files       [6]    [7]    [8]
---------------------------------------------------------------------
main        copy_file   27          copyn.c     1      1      1
main        printf      17          copyn.c     0      0      0
main        printf      18          copyn.c     0      0      0
main        exit        19          copyn.c     0      0      0
main        printf      22          copyn.c     0      0      0
main        printf      23          copyn.c     0      0      0
main        exit        24          copyn.c     0      0      0
main        atoi        26          copyn.c     1      1      1
main        printf      30          copyn.c     0      0      0
main        printf      33          copyn.c     0      1      0
main        printf      36          copyn.c     1      0      1
main        exit        39          copyn.c     1      1      1
copy_file   _open       50          copyn.c     1      1      1
copy_file   _stat       53          copyn.c     1      1      0
```

3. Enter the following to see the test contribution to block coverage:

   ```
   % cvcov lsblock -contrib -pretty tut_testset
   ```

   If you examine the results, you'll see that Tests [5] and [7] and Tests [3] and [8] are identical.

Now we can try to tune the test set. If we can remove tests with redundant coverage and still achieve the equivalent overall coverage, then we have tuned our test set successfully. Since the arcs and blocks covered by Test [7] are also covered by Test [5], we can remove either one of them without affecting the overall coverage. The same analysis holds true for Tests [3] and [8].

4. Delete *test0007* and *test0008* as shown in **Example 23–11** . Then rerun the test set and look at its summary.

   Note that the coverage is retabulated without actually rerunning the tests. The test summary shows that overall coverage is unchanged, thus confirming our hypothesis.

Example 23–11 Test Set Summary after Removing Tests [8] and [7]

```
% cvcov deltest test0008 tut_testset
cvcov: Deleted "/usr/var/tmp/tutorial/test0008" from "tut_testset"
% cvcov deltest test0007 tut_testset
cvcov: Deleted "/usr/var/tmp/tutorial/test0007" from "tut_testset"
% cvcov runtest tut_testset
cvcov: Running test "/usr/var/tmp/tutorial/test0000" ...
cvcov: Running test "/usr/var/tmp/tutorial/test0001" ...
cvcov: Running test "/usr/var/tmp/tutorial/test0002" ...
cvcov: Running test "/usr/var/tmp/tutorial/test0003" ...
cvcov: Running test "/usr/var/tmp/tutorial/test0004" ...
cvcov: Running test "/usr/var/tmp/tutorial/test0005" ...
cvcov: Running test "/usr/var/tmp/tutorial/test0006" ...
% cvcov lssum tut_testset
Coverages                   Covered     Total       % Coverage    Weight
---------------------------------------------------------------------------
----
Function                    2           2           100.00%       0.400
Source Line                 35          35          100.00%       0.200
Branch                      9           10          90.00%        0.200
Arc                         18          18          100.00%       0.200
Block                       39          42          92.86%        0.000
Weighted Sum                                        98.00%        1.000
```

Tutorial #4: Analyzing a Test Group

Test groups are used when you are conducting tests on executables that use a common dynamically shared object (DSO). The results will be limited to whatever constraints you set on the DSO and thus will not include branches, arcs, and other code that lie outside the executables.

Note: This tutorial may be run independently of the previous tutorials. However, it does use *copyn*. If you have run the other tutorials previously, the instrumentation directory *ver##1* will be created for the new executable; otherwise, *ver##0* is created when *copyn* is compiled.

In this tutorial, we will test coverage for a DSO called *libc.so.1*, which is shared by *copyn*, the executable from the previous tutorials, and a simple application called *printtest*. The script *tut_make_testgroup* is provided to run this tutorial.

1. Run the script by typing `tut_make_testgroup`

   The *tut_make_testgroup* script creates the test group and its subtests. **Example 23–12** shows the results of running the initial preparation part of the script using `sh -x`.

   First, the script makes the two applications, *printtest* and *copyn*. The next step is to instrument the programs. The script stores the instrumentation data for *printtest* in a subdirectory called *print_instr_dir* and the *copyn* data in *copyn_instr_dir.*

   The script then makes test directories for the applications and names them *print_test0000* and *copyn_test0000*, respectively. It makes a test group called *tut_testgroup* and adds both tests to it.

mktgroup is the only command that we haven't used previously in the tutorials. *mktgroup* creates the test group. As a final part of the preparation, the script performs a *cattest* to show the contents of the test group.

Example 23–12 Setting up a Test Group

```
% sh -x tut_make_testgroup
+ make -f Makefile.tutorial all
           /usr/bin/cc -g -o printtest printtest.c  -lc
+ cvcov runinstr -instr_dir print_instr_dir -instr_file
 tut_group_instr_file printtest
cvcov: Instrument "printtest" of version "0" succeeded.
+ cvcov runinstr -instr_dir copyn_instr_dir -instr_file
 tut_group_instr_file copyn
cvcov: Instrument "copyn" of version "0" succeeded.
+ cvcov mktest -cmd printtest 10 2 3 -instr_dir print_instr_dir -testname
 print_test0000
cvcov: Made test directory: "/usr/var/tmp/tutorial4/print_test0000"
+ cvcov mktest -cmd copyn tut4_instr_file targetfile -instr_dir
 copyn_instr_dir -testname
 copyn_test0000
cvcov: Made test directory: "/usr/var/tmp/tutorial4/copyn_test0000"
+ cvcov mktgroup -des Group sharing libc.so.1 -testname tut_testgroup
 libc.so.1
cvcov: Made test directory: "/usr/var/tmp/tutorial4/tut_testgroup"
+ cvcov addtest print_test0000 tut_testgroup
cvcov: Added "/usr/var/tmp/tutorial4/print_test0000" to "tut_testgroup"
+ cvcov addtest copyn_test0000 testgroup
cvcov: Added "/usr/var/tmp/tutorial4/copyn_test0000" to "tut_testgroup"
+ cvcov cattest tut_testgroup
Test Info               Settings
----------------------------------------------------------------
Test                    /usr/var/tmp/tutorial4/tut_testgroup
Type                    group
Description             Group sharing libc.so.1
Number of Objects       1
Object List             libc.so.1
Number of Subtests      2
Subtest List
                          [0] /usr/var/tmp/tutorial4/print_test0000
                          [1] /usr/var/tmp/tutorial4/copyn_test0000
Experiment List
```

Finally, the script runs the test group and performs the queries shown in **Example 23–13** .

Example 23–13 Examining Test Group Results

```
+ cvcov runtest tut_testgroup
cvcov: Running test "/usr/var/tmp/tutorial4/print_test0000" ...
2
3
10
cvcov: Running test "/usr/var/tmp/tutorial4/copyn_test0000" ...
copyn: Insufficient arguments.
Usage: copyn f1 f2 bytes
+ cvcov lssum tut_testgroup
Coverages                   Covered     Total       % Coverage    Weight
------------------------------------------------------------------------
-
Function                    33          1777        1.86%         0.400
Source Line                 438         25525       1.72          0.200
```

```
Source Line                  438        25525      1.72          0.200
Branch                       27         10017      0.27%         0.200
Arc                          31         6470       0.48%         0.200
Block                        363        27379      1.33%         0.200
Weighted Sum                                       1.24%         1.000
+ cvcov lsfun -pretty -contrib -pat printf tut_testgroup
Functions    Files       Counts
-------------------------------------
printf       doprnt.c    5

Functions    Files       [0]    [1]
---------------------------------------
printf       doprnt.c    3      2
+ cvcov lsfun -pretty -contrib -pat sscanf tut_testgroup
Functions    Files       Counts
-------------------------------------
sscanf       scanf.c     3

Functions    Files       [0]    [1]
---------------------------------------
sscanf       scanf.c     3      0
```

Chapter 24

Tester Command Line Reference

This chapter describes the *cvcov* commands. It contains two parts:

- **"Common cvcov Options"**— the command arguments that are common to more than one command
- **"cvcov Command Syntax and Description"**— the specifications with descriptions for each command

A complete description of the *cvcov* commands, including individual arguments, is available in the man pages by typing:

`man cvcov.`

Common *cvcov* Options

This section contains descriptions of some *cvcov* flags and variables that are common to more than one command.

`[-ver]`

displays the version of *cvcov*. Note that there are no other arguments permitted; you enter:

`cvcov -ver`

`[-v` *versionnumber***]**

allows you to specify a version of the instrumentation or experiment directory other than the most recent, which is the default.

`[-contrib]`

shows the list of tests that contributed to coverage for the particular query.

`[-exe` *exe_name***]**

lets you specify an executable for coverage testing. This is used when there are multiple executables involved, as in testing processes created by the *fork*, *exec*, or *sproc* command.

`[-instr_dir` *instr_dir***]**

allows you to specify an instrumentation directory other than the current working directory, which is the default.

`[-instr_file` *instr_file***]**

criteria you have selected.

[-list *list_file***]**
specifies a file containing a list of test names to be made part of a test set or group. If no `-list` option is specified, an empty test set will be created.

[-r]
is a mnemonic for recursion. It lets you specify tests in a hierarchy of subdirectories.

[-arg]
displays functions with their arguments.

[-pretty]
displays output aligned in columns. Without `-pretty`, the output is in columns but more condensed.

[-sort]
sorts the output by the specified criteria, as follows:
function—alphabetically by function
diff—by differences in the counting information for coverage type
caller—alphabetically by calling function
callee—alphabetically by called function
count—by counts for current coverage type
file—alphabetically by file name
type—alphabetically by argument type

[-functions]
displays list of constrained functions.

[-pat *func_pattern***]**
lets you enter a pattern instead of a complete function name. The pattern can be of the form **func_name**, **dso_:func_name**, or **'dso:*'**.

experiment **|** *test_name*
lets you specify either the experiment subdirectory or the test directory. The test directory is typically of the form *test<nnnn>*, where *<nnnn>* is a number in a sequence counting from 0000. You can specify your own name. The test directory contains all information about a test including the experiment directory. The experiment directory is typically of the form *exp##<n>*, where *<n>* is a sequential number, counting from 0.

cvcov Command Syntax and Description

This section contains the syntax and description for all *cvcov* commands in the command line interface. If you need information on command arguments that are not described in this section, please refer back to **"Common cvcov Options"** .

The most general command is the help command.

cvcov help *command_name*
prints help on the specified command. If the optional command name is not specified, it prints help for all the commands.

The rest of the commands are divided up into these categories:

- general test commands
  - *cvcov cattest*
  - *cvcov lsinstr*
  - *cvcov lstest*
  - *cvcov mktest*
  - *cvcov rmtest*
  - *cvcov runinstr*
  - *cvcov runtest*
- coverage analysis commands
  - *cvcov lssum*
  - *cvcov lsfun*
  - *cvcov lsblock*
  - *cvcov lsbranch*
  - *cvcov lsarc*
  - *cvcov lscall*

  - *cvcov lsline*
  - *cvcov lssource*
  - *cvcov lstrace*
  - *cvcov diff*
- test set commands
  - *cvcov mktset*
  - *cvcov addtest*
  - *cvcov deltest*
- test group command
  - *cvcov mktgroup*

General Test Commands

The following commands support the creation, inspection, modification, and deletion of tests.

`cvcov cattest [-r]` *test_name*

describes the test details for a test, test set, or test group. **Example 24–1** shows the ASCII display for a single test.

Example 24–1 cattest Example

```
% cvcov cattest test0000
Test Info                Settings
-------------------------------------------------------------
Test                     /disk2/tutorial/tutorial/test0000
Type                     single
Description
Command Line             copyn alphabet targetfile 20
Number of Exes           1
Exe List                 copyn
Instrument Directory     /disk2/tutorial/tutorial/
Experiment List
                           exp##0
                           exp##1
```

Example 24–2 shows the ASCII report for a test set without recursion.

Example 24–2 cattest Example without `-r`

```
% cvcov cattest tut_testset
Test Info                Settings
----------------------------------------------------------
Test                    /disk2/tutorial/tutorial/tut_testset
Type                    set
Description             full coverage testset
Number of Exes          1
Exe List                copyn
Number of Subtests      9
Subtest List
                          [0] /disk2/tutorial/tutorial/test0000
                          [1] /disk2/tutorial/tutorial/test0001
                          [2] /disk2/tutorial/tutorial/test0002
                          [3] /disk2/tutorial/tutorial/test0003
                          [4] /disk2/tutorial/tutorial/test0004
                          [5] /disk2/tutorial/tutorial/test0005
                          [6] /disk2/tutorial/tutorial/test0006
                          [7] /disk2/tutorial/tutorial/test0007
                          [8] /disk2/tutorial/tutorial/test0008
Experiment List
                          exp##0
```

Example 24–3 shows the ASCII report for a nested test set.

Example 24–3 cattest Example with `-r`

```
% cvcov cattest -r tut_testset
Test Info                Settings
---------------------------------------------------------
Test                     /disk2/tutorial/tutorial/tut_testset
Type                     set
Description              full coverage testset
Number of Exes           1
Exe List                 copyn
Number of Subtests       9
Subtest List
                           /disk2/tutorial/tutorial/test0000
                           /disk2/tutorial/tutorial/test0001
                           /disk2/tutorial/tutorial/test0002
                           /disk2/tutorial/tutorial/test0003
                           /disk2/tutorial/tutorial/test0004
                           /disk2/tutorial/tutorial/test0005
                           /disk2/tutorial/tutorial/test0006
                           /disk2/tutorial/tutorial/test0007
                           /disk2/tutorial/tutorial/test0008
Experiment List
                           exp##0
```

cvcov lsinstr [-exe *exe_name*] **[-functions]**
[-v *versionnumber*] *test_name*

displays the instrumentation information for a particular test. *exe_name* is the executable targeted for query. The main program is the default if no executable is specified. The `-functions` parameter shows the functions that are included in the coverage experiment. The **versionnumber** parameter allows you to specify the version of the program that was instrumented. You can specify the test directory using the *test_name* parameter. See **Example 24–4** .

Example 24–4 lsinstr Example

```
% cvcov lsinstr test0000
Instrumentation          Info
---------------------------------------------------------
Executable               copyn
Version                  0
Instrument Directory     /x/tmp/carol/
Instrument File          tut_instr_file
Criteria                 RBPA
Instrumented Objects     copyn_Instr(2.57X)
                           libc.so.1_RBP_Instr(1.07X)
Argument Tracing         copy_file(size[bounds]) [copyn.c]
```

cvcov lstest [-r] [*test_name...***]**

lists the test directories in the current working directory. Note that the *test_name* parameter will accept regular expressions for *lstest*.

cvcov mktest -cmd *cmd_line*
[-des *description***]**
[-instr_dir *directoryname***]**
[-testname *test***]**
[*exe1 exe2* **...]**

creates a test directory. You specify the program and command line options for the program to be tested. This includes any redirection for *stdin*, *stderr*, or *stdout* as run from the Bourne shell. The `-cmd` qualifier is mandatory, even if it only includes the program name. If no executables are specified, only the main program is tested.

Example 24–5 shows an example of *mktest*, followed by *cattest* to display the contents of the Test Description File (*TDF*).

Example 24–5 Test Description File Examples

```
% cvcov mktest -cmd "copyn tut_instr_file targetfile"
cvcov: Made test directory: /d/Tester/tutorial/test0002

% cvcov cattest test0002
Test Info               Settings
---------------------------------------------------------
Test                    /d/Tester/tutorial/test0002
Type                    single
Description
Command Line            copyn tut_instr_file targetfile
Number of Exes          1
Exe List                copyn
Instrument Directory    /d/Tester/tutorial
Experiment List
```

cvcov rmtest [-r] *test_name* **...**

removes tests and test sets. Note that the *test_name* parameter will accept regular expressions for *rmtest*. It is recommended to separate the test set directory from its test subdirectories and the instrument directory. In this way, *rmtest* will not remove instrumentation data or subtests if you choose to remove the test set only.

cvcov runinstr [-instr_dir *instr_dir***]**
[-instr_file *instr_file***]**
[-v *versionnumber***]** *executable*

adds code to the target executable to enable you to capture coverage data, according to the criteria you specify. The instrument file is an ASCII description of the instrumentation criteria for the experiment.You can also specify the version of the executable and instrument directory.

You can capture basic block counts, function pointer counts, and branch counts (at the assembly language level). You can use INCLUDE, EXCLUDE, or CONSTRAIN to modify the set of functions covered. CONSTRAIN lets you define a set of functions for the test.

cvcov runtest [-force] [-keep] [-sum] [-v *versionnumber***]**
[-noarc] [-rmsub] *test_name*

runs a test or a set of tests. The `-force` flag forces the test to be run again even if an experiment is present. It uses WorkShop performance tool technology to set up the instrumented process, run the process, and monitor the run, collecting counting information upon exit. The `-keep` flag retains all performance data collected in the experiment. By default, the performance data is not retained, because it is not required by the coverage tool. The `-sum` flag accumulates (sum over) the coverage data into the existing experiment results. This allows users to run and rerun the same test and accumulate the results in one place.

The `-noarc` flag prevents arc information from being saved in the test database. With the `-noarc` flag, all arc–related queries will not work (for example, *lsarc* and *lscall*). The `-rmsub` flag removes results for individual subtests for a test set or test group. There will be no data to query if you are querying a subtest. `-noarc` and `-rmsub` save disk space.

Coverage Analysis Commands

Once the data has been collected from the test experiments, the user can analyze the data. There are special commands for the various types of coverage available.

cvcov lssum [-exe *exe_name***] [-weight** *func_factor* **:**
line_factor **:** *branch_factor* **:** *arc_factor* **:** *block_factor***]** *experiment* **|**
test_name

shows the overall coverage based on the user–defined weighted average over report.

Example 24–6 lssum Example

```
% cvcov lssum test0000
Coverages        Covered     Total       % Coverage    Weight
-------------------------------------------------------------
Function         2           2           100.00%       0.400
Source Line      17          35          48.57%        0.200
Branch           0            10           0.00%        0.200
Arc              8           18          44.44%        0.200
Block            19          42          45.24%        0.000
Weighted Sum                             58.60%        1.000
```

cvcov lsfun [-arg]
[-bf *filter_type block_filter_value***] [-blocks] [-branches] [-contrib]**
[-exe *exe_name***]**
[-ff *filter_type func_filter_value***]**
[-pat *func_pattern***] [-pretty]**
[-rf *filter_type branch_filter_value***] [-sort** *count* **|** *file* **|** *function***]**
experiment **|** *test_name*

lists coverage information for the specified functions in the program that was tested. Several sorting, matching, and filtering techniques are available. For example, you can show the list of functions that have 0 counts (were not covered) in alphabetical order. You can display arguments with the `-arg` flag. **Example 24–7** shows a typical *lsfun* ASCII report.

Example 24–7 lsfun Example

```
% cvcov lsfun -pretty -sort function test0000
Functions    Files       Counts
-------------------------------------
copy_file    copyn.c     1
main         copyn.c     1
```

Note: C++ in–line functions are not counted as functions.

cvcov lsblock [-arg] [-contrib] [-exe *exe_name***]**
[-pat *func_pattern***] [-pretty]**
[-sort *count* **|** *file* **|** *function***]** *experiment* **|** *test_name*

displays a list of blocks for one or more functions and the count information associated with each block. Blocks are identified by the line numbers in which they occur. If there are multiple blocks in a line, blocks subsequent to the first are shown in order with an index number in parentheses. Be careful before listing all blocks in the program, since this can produce a lot of data.

Example 24–8 shows a typical *lsblock* ASCII report.

Example 24–8 lsblock Example

```
% cvcov lsblock -pat main -pretty test0000
Blocks      Functions    Files       Counts
-------------------------------------------------
13~16       main         copyn.c     1
17~17       main         copyn.c     0
18~18       main         copyn.c     0
19~19       main         copyn.c     0
21~21       main         copyn.c     1
22~22       main         copyn.c     0
23~23       main         copyn.c     0
24~24       main         copyn.c     0
26~26       main         copyn.c     1
```

```
26~27       main        copyn.c     1
27~27       main        copyn.c     1
28~28       main        copyn.c     0
28~28(2)    main        copyn.c     0
28~28(3)    main        copyn.c     0
28~28(4)    main        copyn.c     0
30~30       main        copyn.c     0
31~31       main        copyn.c     0
33~33       main        copyn.c     0
34~34       main        copyn.c     0
36~36       main        copyn.c     0
37~37       main        copyn.c     0
39~39       main        copyn.c     0
41~41       main        copyn.c     0
43~43       main        copyn.c     1
43~43(2)    main        copyn.c     0
43~43(3)    main        copyn.c     1
```

cvcov lsbranch [-arg] [-exe *exe_name***]**
[-pat *func_pattern***] [-pretty]**
[-sort function | file]
experiment | *test_name*

lists coverage information for branches in the program, including the line number at which the branch occurs. Branch coverage counts assembly language branch instructions that are both taken and not taken.

Example 24–9 shows a typical branch coverage ASCII report. Note that branches with incomplete or null coverage are highlighted (**boldfaced**).

Example 24–9 lsbranch Example

```
% cvcov lsbranch -pretty -sort function test0000
Line        Functions    Files       Taken       Not Taken
-------------------------------------------------------------
50          copy_file    copyn.c     1           0
54          copy_file    copyn.c     1           0
57          copy_file    copyn.c     1           0
60          copy_file    copyn.c     1           0
16          main         copyn.c     1           0
21          main         copyn.c     1           0
27          main         copyn.c     1           0
28          main         copyn.c     0           0
28(2)       main         copyn.c     0           0
28(3)       main         copyn.c     0           0
```

cvcov lsarc [-arg] [-callee *callee_pattern***]**
[-caller *caller_pattern***] [-contrib]**
[-exe *exe_name***] [-pretty]**
[-sort *caller* **|** *callee* **|** *count* **|** *file***]** *experiment* **|** *test_name*

shows arc coverage, that is, the number of arcs taken out of the total possible arcs. An arc is a function caller–callee pair. Both *callee_pattern* and *caller_pattern* can be specified in the same way as *func_pattern* (used with the `-pat` option) as shown under **"Common cvcov Options"** .

Example 24–10 shows a typical *lsarc* ASCII report.

Example 24–10 lsarc Example

```
% cvcov lsarc -callee printf -pretty test0001
Callers     Callees      Line        Files       Counts
-------------------------------------------------------
main        printf        17          copyn.c     1
main        printf        18          copyn.c     1
```

```
main        printf      22          copyn.c     0
main        printf      23          copyn.c     0
main        printf      30          copyn.c     0
main        printf      33          copyn.c     0
main        printf      36          copyn.c     0
```

cvcov lscall [-arg] [-exe *exe_name***]**
[-node *func_name***] [-pretty] [-r] experiment |**
test_name
lists the call graph for the executable with counts for each function. The contribution to this coverage by each test is shown in a separate column. **Example 24–11** shows a typical *lscall* ASCII report. `N/A` means the node is excluded.

Example 24–11 lscall Example

```
% cvcov lscall -pretty test0000
Graph            Counts
---------------------------------
main             1
     copy_file     1
        _open      N/A
        _stat...   N/A
        _creat     N/A
       _malloc... N/A
        _read      N/A
        _write     N/A
     printf...     N/A
     exit...       N/A
     atoi          N/A
```

A function that has more than one parent and has children is called a *subnode*. Using `-r` will display the subnodes. Subnodes are given their own starting point in the textual call graph. They are identified by a trailing ellipsis (...). For example, see *printf*, *exit*, and *malloc* in **Example 24–11** .

cvcov lsline [-arg] [-exe *exe_name***] [-pat** *func_pattern***]**
[-pretty] [-sort function | file]
experiment | test_name
lists the coverage for native source lines. Use **`arg`** to show arguments for functions. If no executable is specified, the main program is the default. Use **`pretty`** to provide column–aligned output. See **Example 24–12** .

Example 24–12 lsline Example

```
% cvcov lsline -pretty -pat main test0000
Functions   Files      Covered   Total     % Coverage
---------------------------------------------------------
main        copyn.c    6         20        30.00%
```

cvcov lssource [-asm] [-exe *exe_name***]** *function* *experiment*
test_name
displays the source annotated with line counts. The `-asm` switch displays the assembly level source code annotated with line counts. Lines with 0 counts are highlighted to show the absence of coverage. This is useful for mapping to the source level blocks and branches that were not covered. Lines in functions that were not included in the test appear without count annotations.
Example 24–13 shows a segment of a typical *lssource* ASCII report.

Note: *lssource* requires the code to be compiled with the `-g` option.

Example 24–13 lssource Example

```
% cvcov lssource main test0000
```

```
Counts  Source
------------------------------------------------------------
          #include <stdio.h>
          #include <sys/stat.h>
          #include <sys/types.h>
          #include <fcntl.h>

          #define OPEN_ERR        1
          #define NOT_ENOUGH_BYTES 2
          #define SIZE_0          3

          int copy_file();

          main (int argc, char *argv[])
1       {
              int bytes, status;

1           if( argc < 4){
0               printf("copyn: Insufficient arguments.\n");
0               printf("Usage: copyn f1 f2 bytes\n");
0               exit(1);
              }
1           if( argc > 4 ) {
0               printf("Error: Too many arguments\n");
0               printf("Usage: copyn f1 f2 bytes\n");
0               exit(1);
              }
1           bytes = atoi(argv[3]);
```

cvcov lstrace [-exe *exe_name***] [-pat** *func_pattern***] [-pretty]**
[-sort *function* | *type***]**
experiment | *test_name*

shows the argument tracing information. **Example 24-14** shows a typical *lstrace* ASCII report. The Range column shows upper and lower bounds. A dash (-) means that side of the bound has not been counted.

Example 24-14 lstrace Example

```
% cvcov lstrace -pretty testtrace
Arguments       Type    Range
---------------------------------
main(argc)      int     -, 4
copy_file(size) int     1, 20
```

Note: *lstrace* requires the code to be compiled with the `-g` option.

cvcov diff [-arg] [-exe *exe_name***] [-functions] [-pretty]**
[-sort diff | function**]**
experiment1 *experiment2*

shows the difference in coverage for different versions of the same program. **Example 24-15** shows an example of the *diff* command applied to two tests (although you should make sure that the comparison is relevant). **Example 24-16** shows *diff* applied to different instrumentations of the same test.

Example 24-15 *diff* between Two Tests

```
% cvcov diff test0000/exp##0 test0001/exp##0

Experiment 1:    test0000/exp##0
Experiment 2:    test0001/exp##0
```

```
Coverages            Exp 1       Exp 2        Differences
----------------------------------------------------------
Function Coverage    2(100.00%)  1(50.00%)    1(50.00%)
Source Line Coverage 17(48.57%)  5(14.29%)    12(34.29%)
Branch Coverage      0(0.00%)    0(0.00%)     0(0.00%)
Arc Coverage         8(44.44%)   3(16.67%)    5(27.78%)
Block Coverage       19(45.24%)  4(9.52%)     15(35.71%)
```

Example 24–16 *diff* between Different Instrumentations of the Same Test

```
% cvcov diff test0000/exp##0 test0000/exp##1

Experiment 1:    test0000/exp##0
Experiment 2:    test0000/exp##1

Coverages            Exp 1        Exp 2       Differences
----------------------------------------------------------
Function Coverage    2(100.00%)    2(100.00%)    0(0.00%)
Source Line Coverage 17(48.57%)    17(47.22%)    0(1.35%)
Branch Coverage      0(0.00%)      0(0.00%)      0(0.00%)
Arc Coverage         8(44.44%)     8(44.44%)     0(0.00%)
Block Coverage       19(45.24%)    19(44.19%)    0(-1.05%)
```

Test Set Commands

A test set is a named collection of tests and other test sets. Test sets can be hierarchical. For example, *compiler_language_suite* might include *C++_suite*, *C_suite*, and *Fortran_suite*, where *Fortran_suite* is a test set with subdirectories. The following commands support creation, inspection, modification, and deletion of test sets. Both *addtest* and *deltest* are also used with test groups, described in the next section.

`cvcov mktset [-des` *description*`] [-list` *list_file*`]`
`[-testname` *test*`]`

makes a test set. If no test name is specified, the command assigns one automatically.

`cvcov addtest` *test_name* *test_set_name* **|** *test_group*

adds a test or test set to a test set or test group.

`cvcov deltest` *test_name* *test_set_name* **|** *test_group*

removes a test or test set from a test set or test group.

Note: Don't use UNIX commands *mv* and *cp* to rename or copy test sets because they are constructed with absolute files paths.

Test Group Commands

A *test group* is a collection of programs to be tested that have a common dynamically shared object (DSO). The coverage testing is limited to activity with the DSO so that the arcs and branches that terminate outside of the DSO will not be included. See descriptions of *addtest* and *deltest* in the previous section as well as the following command.

`cvcov mktgroup [-des` *description*`] [-list` *list_file*`]`
`[-testname` *test*`]` *target1* *target2* `...`

creates a test group that can contain other tests or test groups. The targets are either the target libraries or DSOs.

Note: Don't use UNIX commands *mv* and *cp* to rename or copy test groups because they are constructed with absolute files paths.

Chapter 25

Tester Graphical User Interface Tutorial

This chapter provides a tutorial for the Tester graphical user interface. It covers these topics:

- **"Setting Up the Tutorial"**
- **"Tutorial #1 — Analyzing a Single Test"**
- **"Tutorial #2 — Analyzing a Test Set"**
- **"Tutorial #3 — Exploring the Graphical User Interface"**

Setting Up the Tutorial

If you have already set up a tutorial directory for the command line interface tutorial, you can continue to use it. If you remove the subdirectories, your directory names will match exactly; if you leave the subdirectories in, you can add new ones as part of this tutorial.

To set up a tutorial directory from scratch, do the following; otherwise you can skip the rest of this section.

1. Enter the following:

   ```
   % cp -r /usr/demos/WorkShop/Tester /usr/tmp/tutorial
   % cd /usr/tmp/tutorial
   % echo ABCDEFGHIJKLMNOPQRSTUVWXYZ > alphabet
   % make -f Makefile.tutorial copyn
   ```

   This moves some scripts and source files used in the tutorial to */usr/tmp/tutorial*, creates a test file named alphabet, and makes a simple program, *copyn*, which copies *n* bytes from a source file to a target file.

2. To see how the program works, try a simple test by typing:

   ```
   % copyn alphabet targetfile 10
   % cat targetfile
   ABCDEFGHIJ
   ```

   You should see the first 10 bytes of *alphabet* copied to *targetfile*.

Tutorial #1 — Analyzing a Single Test

Tutorial #1 discusses the following topics:

- **"Invoking the Graphical User Interface"**
- **"Instrumenting an Executable"**
- **"Making a Test"**
- **"Running a Test"**
- **"Analyzing the Results of a Coverage Test"**

Invoking the Graphical User Interface

You typically call up the graphical user interface from the directory that will contain your test subdirectories. This section tells you how to invoke the Tester graphical user interface and describes the main window.

1. Enter `cvxcov` from the tutorial directory to bring up the Tester main window.

   Figure 25–1 shows the main Tester window with all its menus displayed.

   Note: You can also access Tester from the Admin menu in other WorkShop tools.

2. Observe the features of the Tester window.

   The *Test Name* field is used to display the current test. You can switch to different tests through this field.

Test results display in the *coverage display area*. You display the results by choosing an item from the Queries menu. You also can select the format of the data from the Views menu.

The *Source* button lets you bring up the standard CASEVision Source View window with Tester annotations. Source View shows the counts for each line included in the test and highlights lines with 0 counts. Lines from excluded functions display but without count annotations.

The *Disassembly* button brings up the CASEVision Disassembly View window for assembly language source. It operates in a similar fashion to the *Source* button.

The *Contribution* button displays a separate window with the contributions to the coverage made by each test in a test set or test group.

A sort button lets you sort the test results by such criteria as function, count, file, type, difference, caller, or callee. The criteria available (shown by the name of the button) depend on the current query.

The status area displays status messages regarding the test.

The area below the status area labeled "Query−specific fields area" contains special fields depending on the query you select.

You can launch other WorkShop applications from the Launch Tool submenu of the Admin menu. The applications include the Build Analyzer, Debugger, Parallel Analyzer, Performance Analyzer, and Static Analyzer.

You'll find an iconized version of Execution View labeled *cvxcovExec*. It is a shell window for viewing test results as they would appear on the command line.

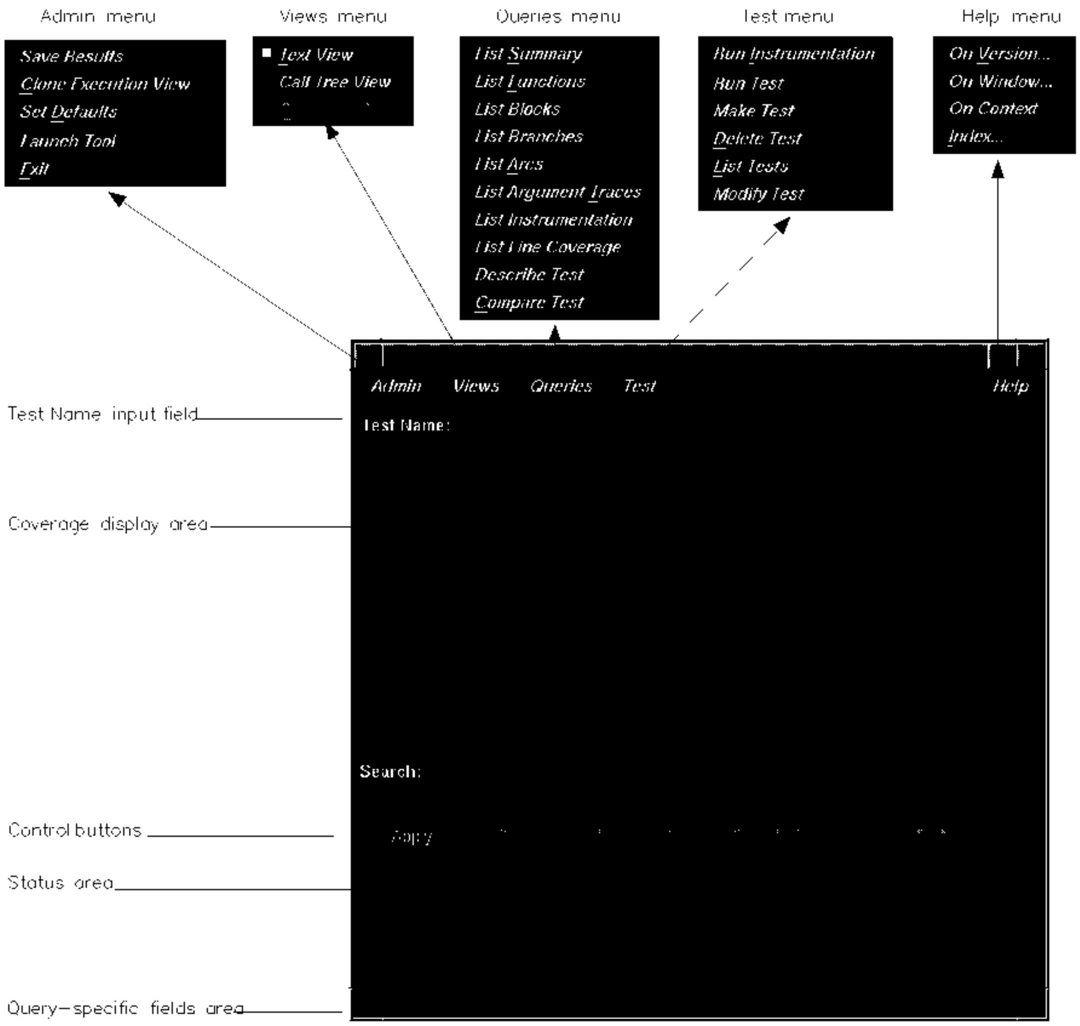


Figure 25–1 Main Tester Window

Instrumenting an Executable

The first step in providing test coverage is to define the instrumentation criteria in an instrumentation file.

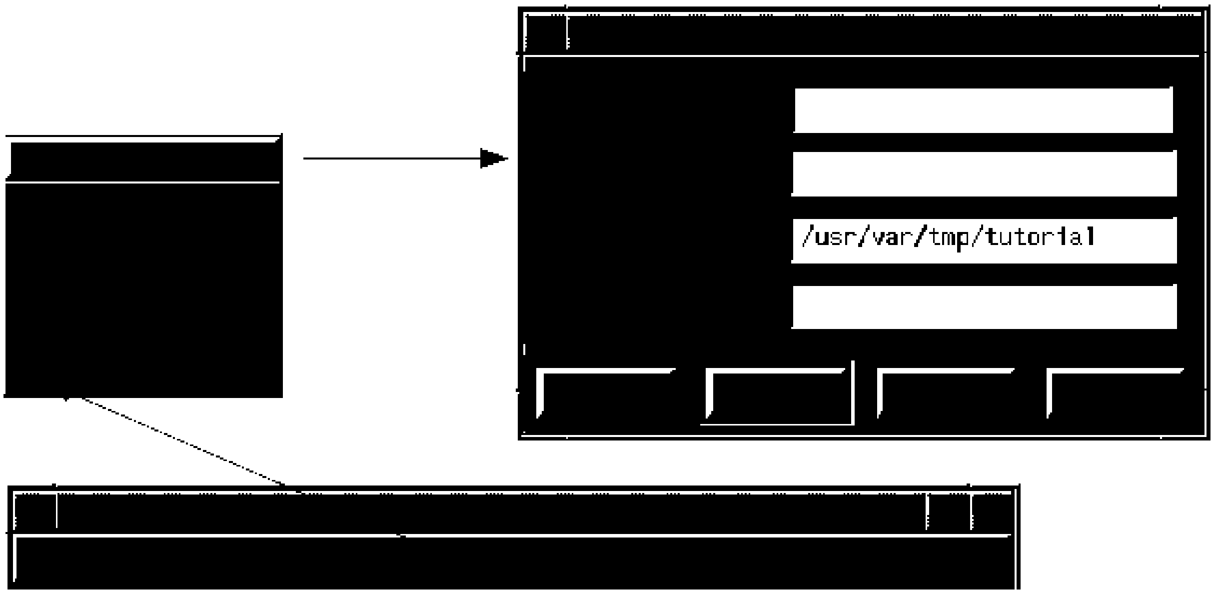
1. On the command line or from Execution View, enter the following to see the instrumentation directives in the file *tut_instr_file* used in the tutorials:

   ```
   % cat tut_instr_file
   COUNTS -bbcounts -fpcounts -branchcounts
   CONSTRAIN main, copy_file
   TRACE BOUNDS copy_file(size)
   ```

   We will be getting all counting information (blocks, functions, source lines, branches, and arcs) for the two functions specified in the CONSTRAIN directive, *main* and *copy_file*. We will also be tracing the *size* argument for the *copy_file* function.

2. Select "Run Instrumentation" from the Test menu.

   This process inserts code into the target executable that enables coverage data to be captured. The dialog box shown in **Figure 25–2** displays when "Run Instrumentation" is selected from the Test menu.



/usr/var/tmp/tutorial

Figure 25–2 Running Instrumentation

3. Enter **`copyn`** in the *Executable* field.

   The *Executable* field is required, as indicated by the red highlight. You enter the executable in this field.

4. Enter **`tut_instr_file`** in the *Instrument File* field.

   The *Instrument File* field lets you specify an instrumentation file containing the criteria for instrumenting the executable. In this tutorial, we use the file *tut_instr_file*, which was described earlier.

5. Leave the *Instrument Dir* and *Version Number* fields as is.

   The *Instrument Dir* field indicates the directory in which the instrumented programs are stored. A versioned directory is created (the default is *ver##n*, where *n* is 0 the first time and is incremented automatically if you subsequently change the instrumentation). The version number *n* helps you identify the instrumentation version you use in an experiment. The experiment results directory will have a matching version number. The instrument directory is the current working directory; it can be set from the Admin menu.

6. Click *OK*.

   This executes the instrumentation process. If there are no problems, the dialog box closes and the message `Instrumentation succeeded` displays in the status area with the version number created.

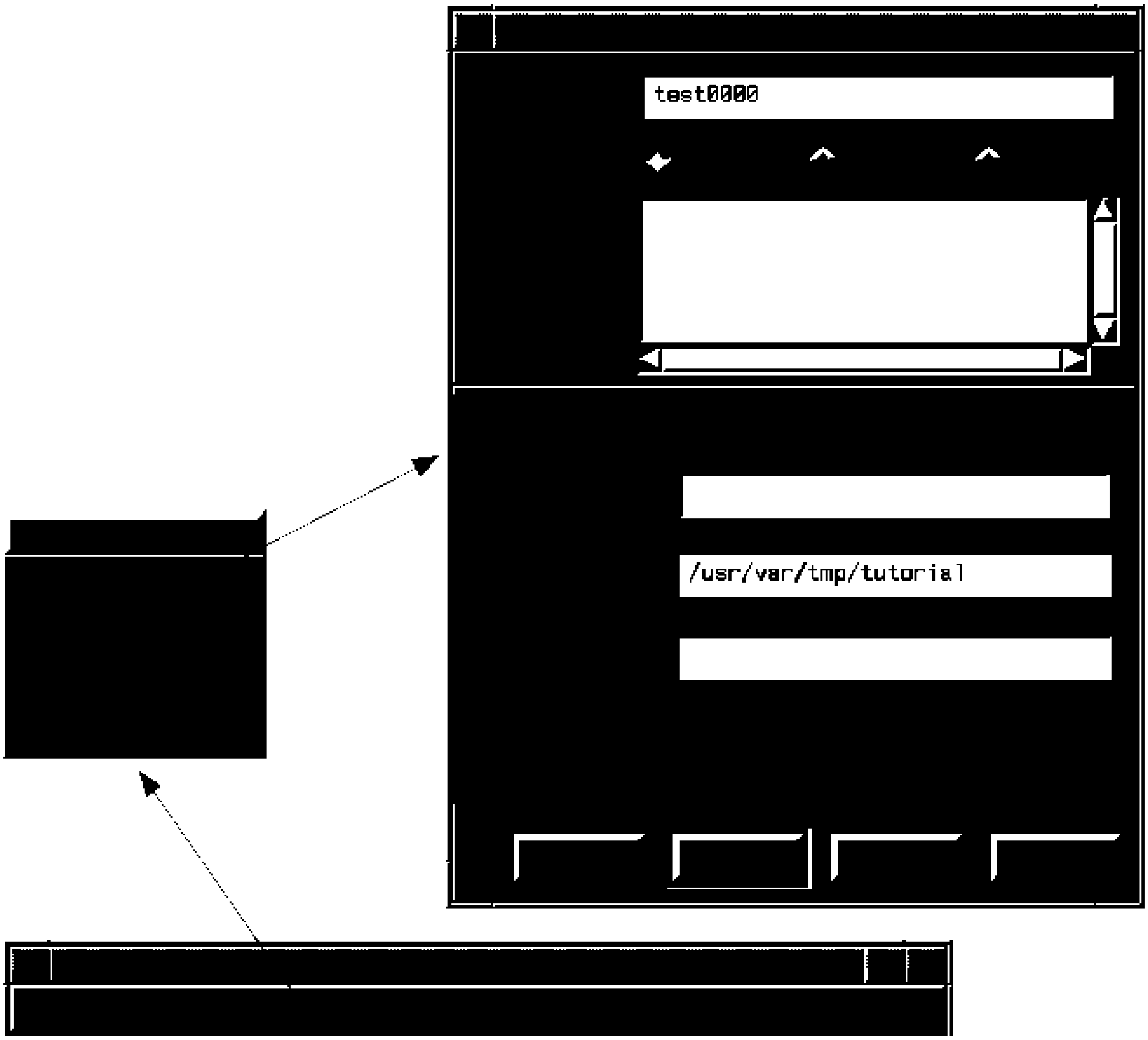
Making a Test

A *test* defines the program and arguments to be run, the instrumentation criteria, and descriptive information about the test.

1. Select "Make Test" from the Test menu.

   This creates a test directory. **Figure 25–3** shows the Make Test window.

   You specify the name of the test directory in the *Test Name* field, in this case **`test0000`**. The field displays a default directory *test<nnnn>*, where *nnnn* is 0000 the first time and incremented for subsequent tests. You can edit this field if necessary.



test0000

/usr/var/tmp/tutorial

Figure 25-3 Selecting "Make Test"

2. Enter a description of the test in the *Description* field.

   This is optional, but can help you differentiate between tests you have created.

3. Enter the executable to be tested with its arguments in the *Command Line* field, in this example:

   `copyn alphabet targetfile 20`

   This field is mandatory, as indicated by its highlighting.

4. Leave the remaining fields as is.

   Tester supplies a default instrumentation directory in the *Instrument Dir* field. The *Executable List* field lets you specify multiple executables when your main program *fork*s, *exec*s, or *sproc*s other processes.

5. Click *OK* to perform the make test operation with your selections.

   The results of the make test operation display in the status area of the main Tester window.

Running a Test

To run a test, we use technology from the WorkShop Performance Analyzer. The instrumented process is set to run, and a monitor process (*cvmon*) captures test coverage data by interacting with the WorkShop process control server (*cvpcs*).

1. Select "Run Test" from the Test menu.

   The dialog box shown in **Figure 25-4** displays. You enter the test directory in the *Test Name* field. You can also specify a version of the executable in the *Version Number* field if you don't wish to use the latest, which is the default. The *Force Run* toggle forces the test to be run again even if a test result already exists. The *Keep Performance Data* toggle retains all the performance data collected in the experiment. The *Accumulate Results* toggle sums over the coverage data into the existing experiment results. Both *No Arc Data* and *Remove Subtest Expt* toggles retain less data in the experiments and are designed to save disk space.

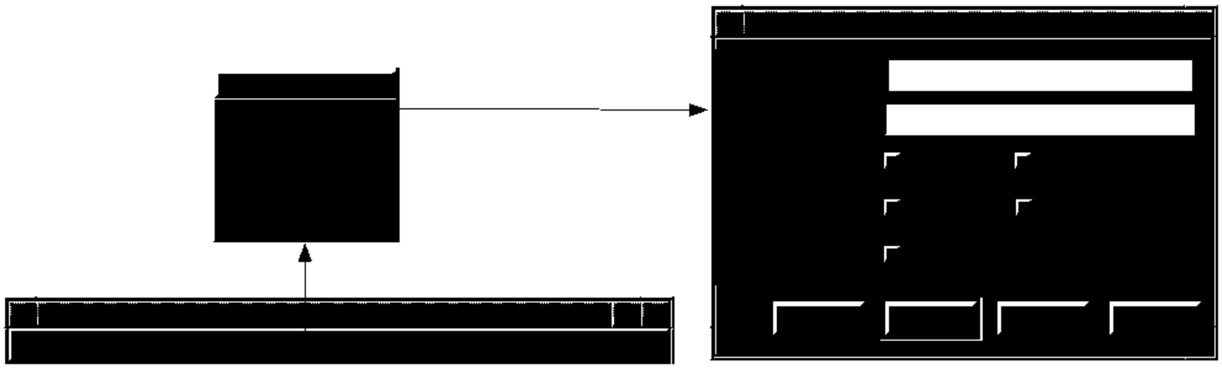


Figure 25-4 "Run Test" Dialog Box

2. Enter **`test0000`** in the *Test Name* field.

3. Click *OK* to run the test with your selections.

   When the test completes, a status message showing completion displays and you will have data to be analyzed. You can observe the test as it runs in Execution View.

Analyzing the Results of a Coverage Test

You can analyze test coverage data in many ways. In this tutorial, we will illustrate a simple top-down approach. We will start at the top to get a summary of overall coverage, proceed to the function level, and finally go to the actual source lines.

Having collected all the coverage data, now you can analyze it. You do this through the Queries menu in the main Tester window.

1. Enter **`test0000`** in the *Test Name* field in the main window and select "List Summary" from the Queries menu.

   This loads the test and changes the main window display as shown in **Figure 25–5** . The query type (in this case, "List Summary") is indicated above the display area. Column headings identify the data, which displays in columns in the coverage display area. The status area is shortened. The query–specific fields (in this case, coverage weighting factors) that appear below the control buttons and status area are different for each query type. You can change the numbers and click *Apply* to weight the factors differently. The *Executable List* button brings up the Target List dialog box. It displays a list of executables used in the experiment and lets you select different executables for analysis. You can select other experiments from the experiment menu (*Expt*).

   "List Summary" shows the coverage data (number of coverage hits, total possible hits, percentage, and weighting factor) for functions, source lines, branches, arcs, and blocks. The last coverage item is the weighted average, obtained by multiplying individual coverage averages by the weighting factors and summing the products.

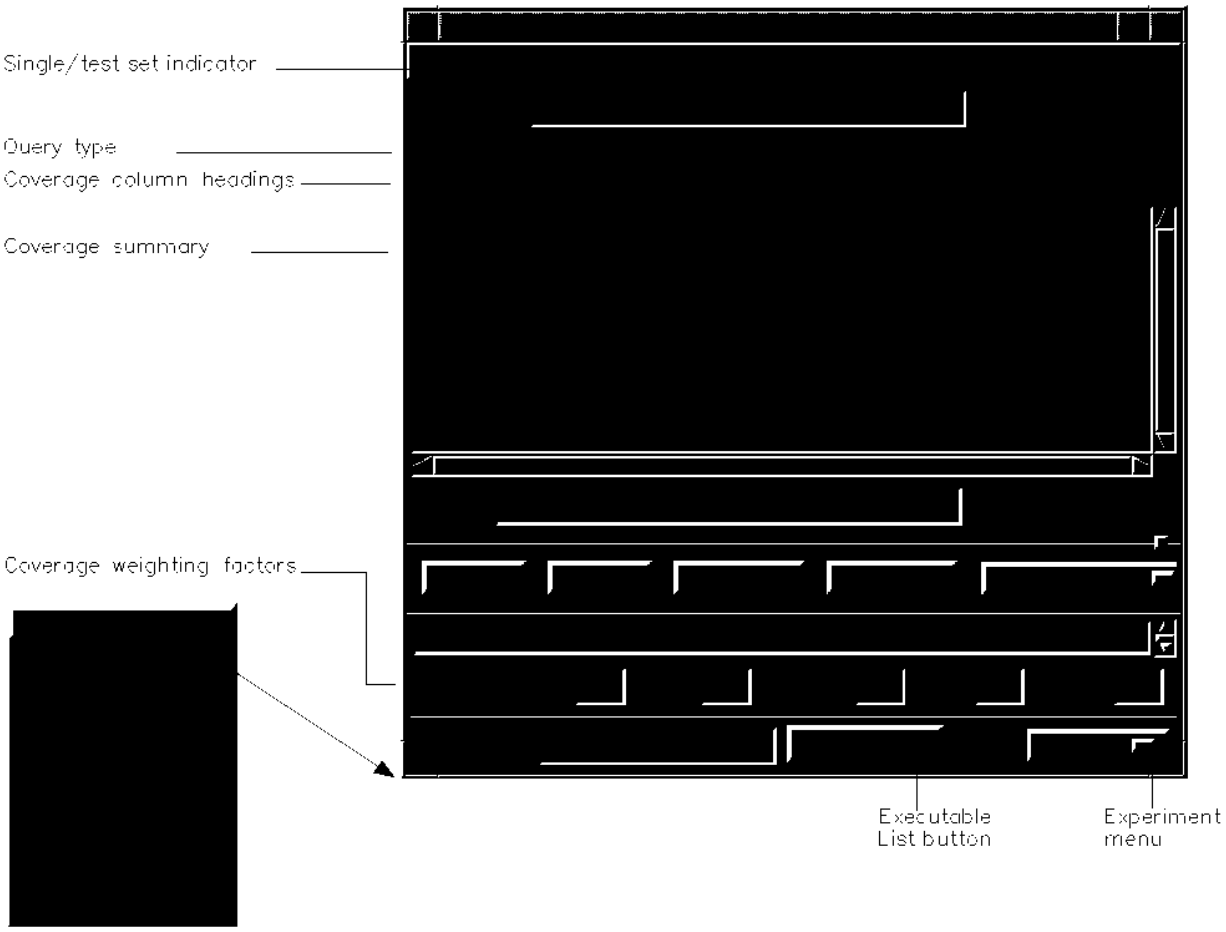


Figure 25–5 "List Summary" Query Window

2. Select "List Functions" from the Queries menu.

   This query lists the coverage data for functions specified for inclusion in this test. The default version is shown in **Figure 25–6** , with the available options.

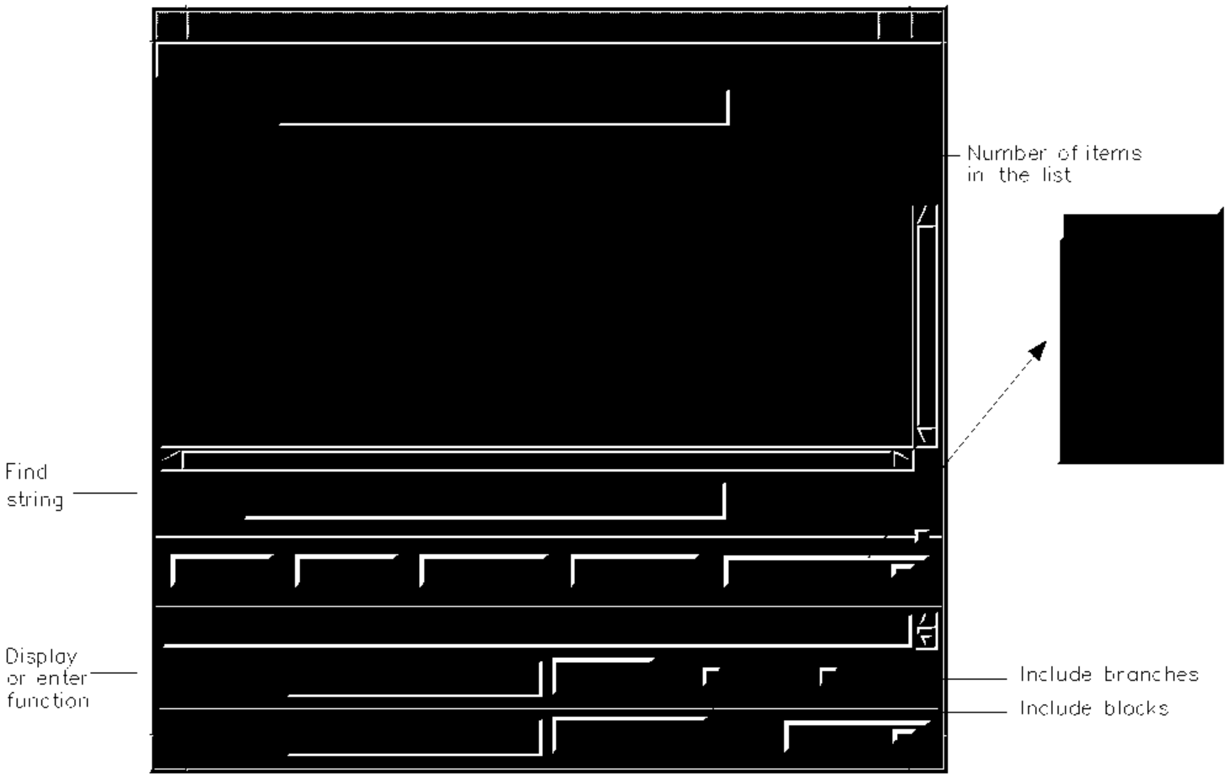
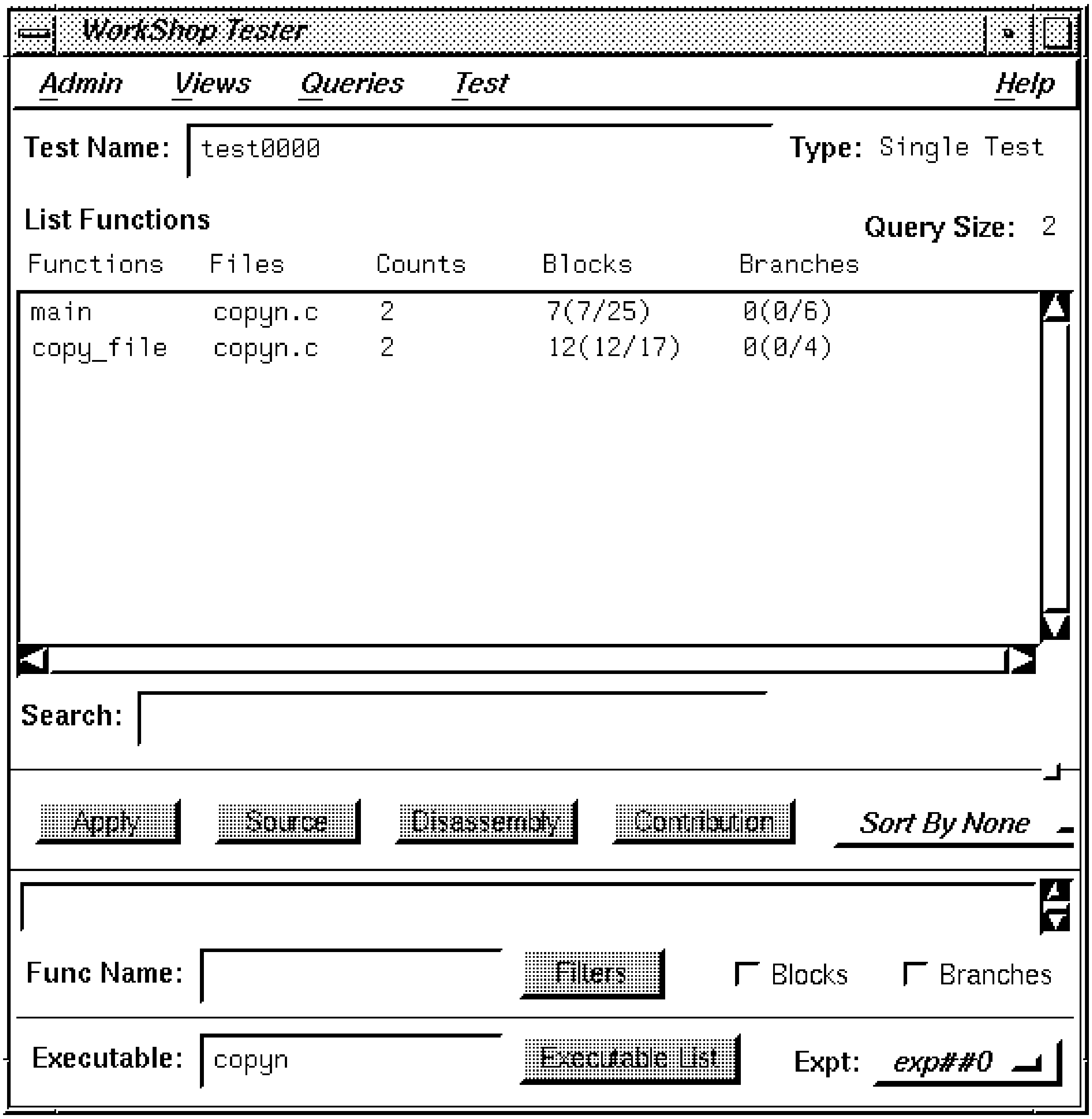


Figure 25–6 "List Functions" Query with Options

If there are functions with 0 counts, they will be highlighted. The default column headings are *Functions*, *Files*, and *Counts*.

3. Click the *Blocks* and *Branches* toggles.

   The *Blocks* and *Branches* toggle buttons let you display these items in the function list. **Figure 25–7** shows the display area with *Blocks* and *Branches* enabled.



Test Name: test0000

Type: Single Test

Query Size: 2

Functions	Files	Counts	Blocks	Branches
main	copyn.c	2	7(7/25)	0(0/6)
copy_file	copyn.c	2	12(12/17)	0(0/4)

Executable: copyn

Expt: exp##0

Figure 25–7 "List Functions" Display Area with Blocks and Branches

The *Blocks* column shows three values. The number of blocks executed within the function is shown first. The number of blocks covered out of the total possible for that function is shown inside the parentheses. If you divide these numbers, you'll arrive at the percentage of coverage.

Similarly, the *Branches* column shows the number of branches covered, followed by the number covered out of the total possible branches. The term *covered* means that the branch has been executed under both true and false conditions.

4. Select the function *main* in the display area and click *Source*.

   The Source View window displays with count annotations as shown in **Figure 25–8** . Lines with 0 counts are highlighted in the display area and in the vertical scroll bar area. Lines in excluded functions display with no count annotations.

5. Click the *Disassembly* button in the main window.

   The Disassembly View window displays with count annotations as shown in **Figure 25–9** . Lines with

0 counts are highlighted in the display area and in the vertical scroll bar area.

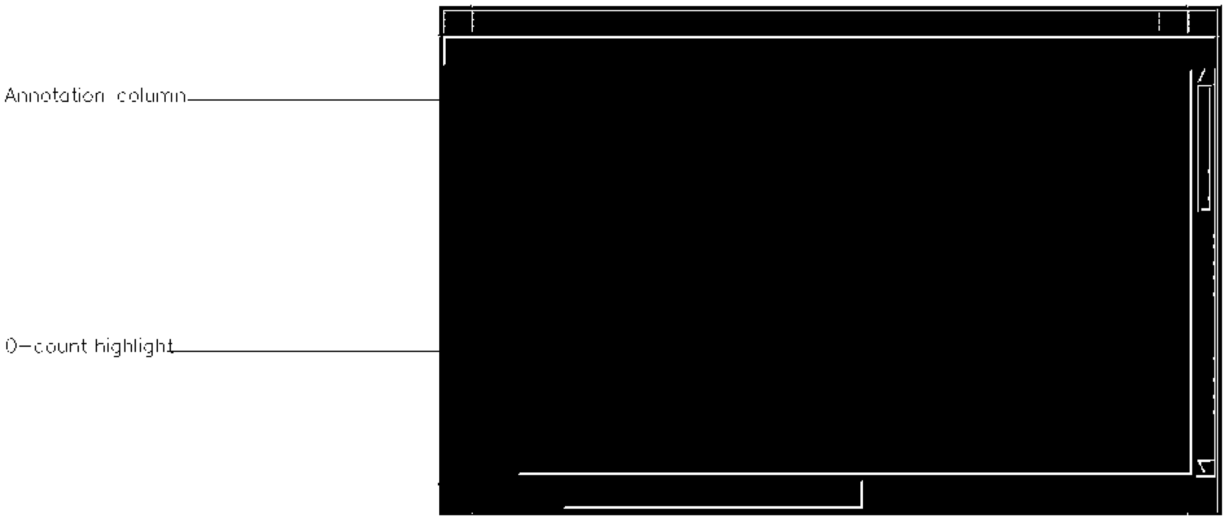


Figure 25–8 Source View with Count Annotations

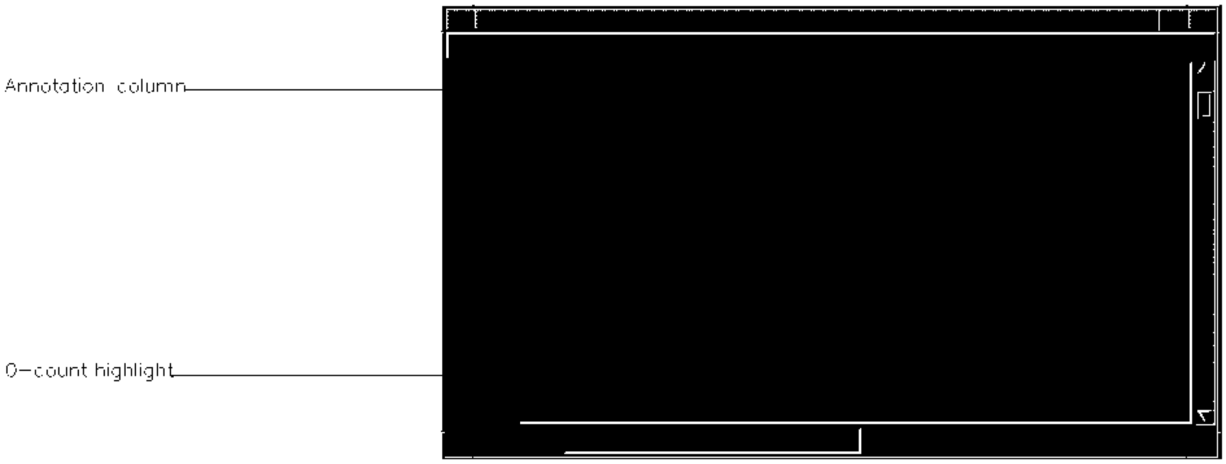


Figure 25–9 Disassembly View with Count Annotations

Tutorial #2 — Analyzing a Test Set

In the second tutorial, we are going to create additional tests with the objective of achieving 100% overall coverage. From examining the source code, it seems that the 0–count lines in *main* and *copy_file* are due to error–checking code that is not tested by *test0000*.

Note: This tutorial needs *test0000*, which was created in the previous tutorial.

1. Select "Make Test" from the Test menu.

   This displays the Make Test dialog box. It is easy to enter a series of tests. As is standard in CASEVision, using the *Apply* button in the dialog box instead of the *OK* button completes the task without closing the dialog box. The *Test Name* field supplies an incremented default test name after each test is created.

We are going to create a test set named *tut_testset* and add to it 8 tests in addition to *test0000* from the previous tutorial. The tests *test0001* and *test0002* pass too few and too many arguments, respectively. *test0003* attempts to copy from a file named *no_file* that doesn't exist. *test0004* attempts to pass 0 bytes, which is illegal. *test0005* attempts to copy 20 bytes from a file called *not_enough*, which contains only one byte. In *test0006*, we attempt to write to a directory without proper permission. *test0007* tries to pass too many bytes. In *test0008*, we attempt to copy from a file without read permission.

The following steps show the command line target and arguments and description for the tests we will use in the tutorial. The descriptions are helpful but optional. **Figure 25–10** shows the features of the dialog box you'll need for creating these tests.

2. Enter `copyn alphabet target` in the *Command Line* field, `not enough arguments` in the *Description* field, and click *Apply* (or simply press `<return>`) to make *test0001*.
3. Enter `copyn alphabet target 20 extra_arg` in the *Command Line* field, `too many arguments` in the *Description* field, and click *Apply* to make *test0002*.

test0000

not enough arguments

copyn alphabet target

/usr/var/tmp/tutorial

Figure 25–10 "Make Test" Dialog Box with Features Used in Tutorial

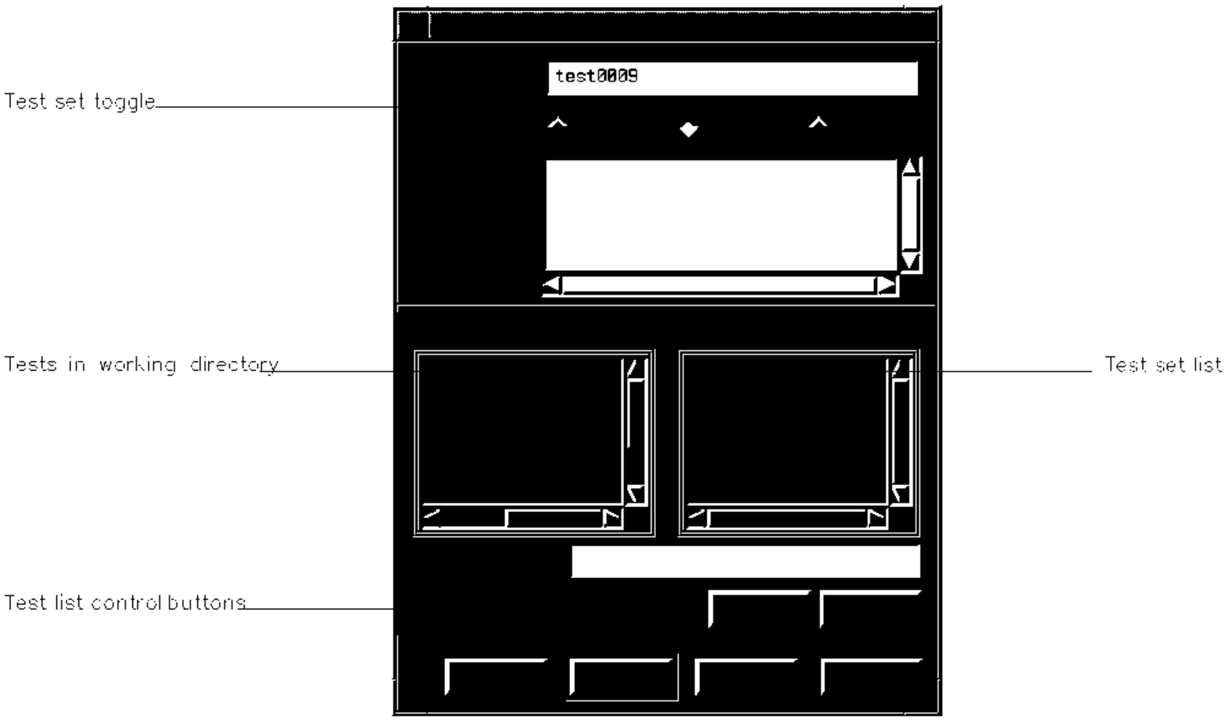
4. Enter `copyn no_file target 20` in the *Command Line* field, `cannot access file` in the *Description* field, and click *Apply* to make *test0003*.
5. Enter `copyn alphabet target 0` in the *Command Line* field, `pass bad size arg` in the *Description* field, and click *Apply* to make *test0004*.
6. Enter `copyn not_enough target 20` in the *Command Line* field, `not enough data` in the *Description* field, and click *Apply* to make *test0005*.
7. Enter `copyn alphabet /usr/bin/target 20` in the *Command Line* field, `cannot create target executable due to permission problems` in the *Description* field, and

click *Apply* to make *test0006*.

8. Enter `copyn alphabet targetfile 200` in the *Command Line* field, `size arg too big` in the *Description* field, and click *Apply* to make *test0007*.
9. Enter `copyn /usr/etc/snmpd.auth targetfile 20` in the *Command Line* field, `no read permission on source file` in the *Description* field, and click *Apply* to make *test0008*.

   We now need to create the test set that will contain these tests.
10. Click the *Test Set* toggle in the *Test Type* field.

    This changes the dialog box as shown in **Figure 25–11** .



test0009

Figure 25–11 "Make Test" Dialog Box for Test Set Type

11. Change the default in the *Test Name* field to `tut_testset`.

    This is the name of the new test set. Now we have to add the tests to the test set.
12. Select the first test in the *Test List* field and click *Add*.

    This displays the selected test in the *Test Include List* field, indicating that it will be part of the test set after you click *OK* (or *Apply* and *Close*).
13. Repeat the process of selecting a test and clicking *Add* for each test in the *Test List* field. When all tests have been added to the test set, click *OK*.

    This saves the test set as specified and closes the "Make Test" dialog box.
14. Enter `tut_testset` in the *Test Name* field and select "Describe Test" from the Queries menu.

    This displays the test set information in the display area of the main window.
15. Select "Run Test" from the Test menu, enter `tut_testset` in the *Test Name* field in the "Run Test" dialog box.

    This runs all the tests in the test set.
16. Enter `tut_testset` in the *Test Name* field in the main Tester window and select "List Summary" from the Queries menu.

    This displays a summary of the results for the entire test set.

17. Select "List Functions" from the Queries menu.

    This step serves two purposes. It enables the *Source* button so that we can look at counts by source line. It displays the list of functions included in the test, from which we can select functions to analyze.

18. Click the *main* function, which is displayed in the function list, and click the *Source* button.

    This displays the source code, with the counts for each line shown in the annotations column. Note that the counts are higher now and full coverage has been achieved at the source level (although not necessarily at the assembly level).

Tutorial #3 — Exploring the Graphical User Interface

The rest of this chapter shows you how to use the graphical user interface (GUI) to analyze test data. The GUI has all the functionality of the command line interface and in addition shows the function calls, blocks, branches, and arcs graphically.

For a discussion of applying Tester to test set optimization, refer to **"Tutorial #3 — Optimizing a Test Set"** . To learn more about test groups, see **"Tutorial #4 — Analyzing a Test Group"** . Although these are written for the command line interface, you can use the graphical interface to follow both tutorials.

1. Enter **`test0000`** in the *Test Name* field of the main window and press **`<return>`**.

   Since *test0000* has incomplete coverage, it is more useful for illustrating how uncovered items appear.

2. Select "List Functions" from the Queries menu.

   The list of functions displays in the text view format.

3. Select "Call Tree View" from the Views menu.

   The Tester main window changes to call graph format. **Figure 25–12** shows a typical call graph. Initially, the call graph displays the *main* function and its immediate callees.

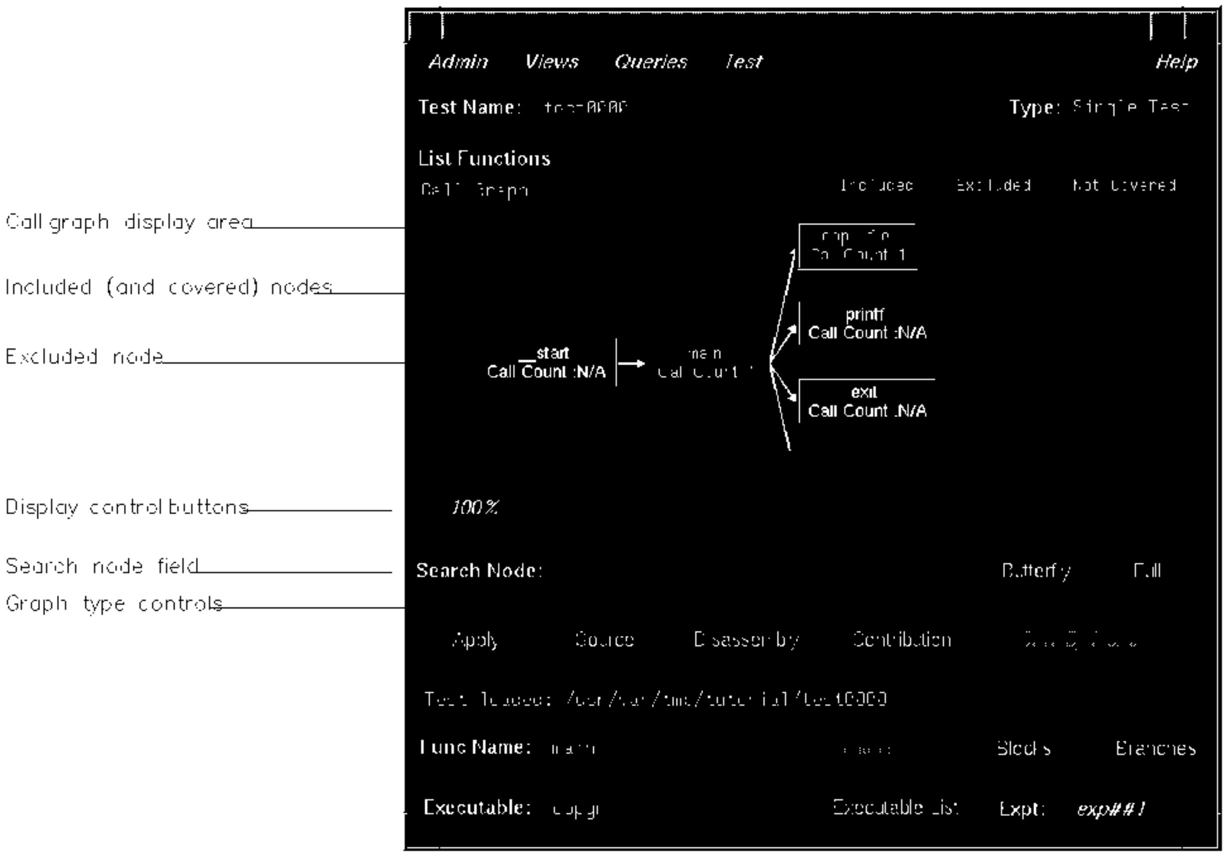


Figure 25–12 Call Graph for "List Functions" Query

The call graph displays functions as nodes and calls as connecting arrows. The nodes are annotated by call count information. Functions with 0 counts are highlighted. Excluded functions when visible appear in the background color.

The controls for changing the display of the call graph are just below the display area (see **Figure 25–13**).

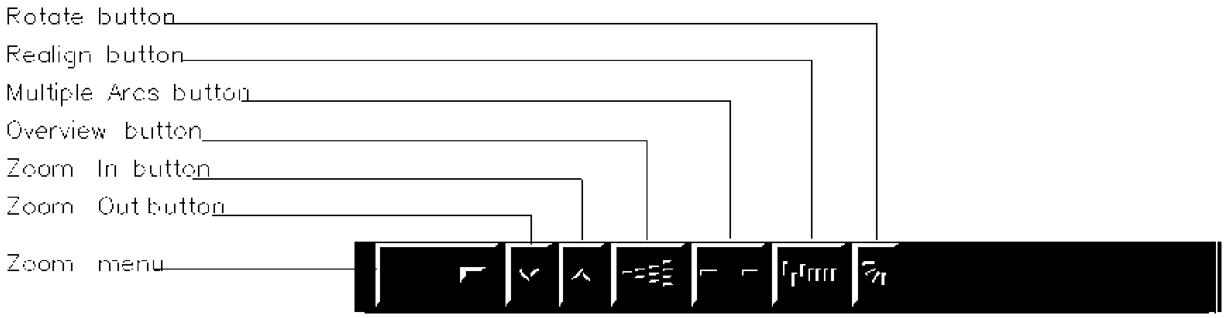


Figure 25–13 Call Graph Display Controls

These facilities are:

Zoom menu icon

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 nominal (100%) size.

Zoom Out icon

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

Zoom In icon

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

Overview icon

invokes an overview pop−up display that 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.

Multiple Arcs icon

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 icon

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

Rotate icon

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

Entering a function in the *Search Node* field scrolls the display to the portion of the graph in which the function is located.

There are two buttons controlling the type of graph. Entering a node in the *Func Name* field and clicking *Butterfly* displays the calling and called functions for that node only (*Butterfly* mode is the default). Selecting *Full* displays the entire call graph (although not all portions may be visible in the display area).

4. Select "List Arcs" from the Queries menu.

   The "List Arcs" query displays coverage data for calls made in the test. Because we were just in call graph mode for the "List Functions" query, "List Arcs" comes up in call graph rather than text mode.

   See **Figure 25−14** . To improve legibility, this figure has been scaled up to 150% and the nodes moved by middle−click−dragging the outlines. Arcs with 0 counts are highlighted in color. Notice that in "List Arcs", the arcs rather than the nodes are annotated.

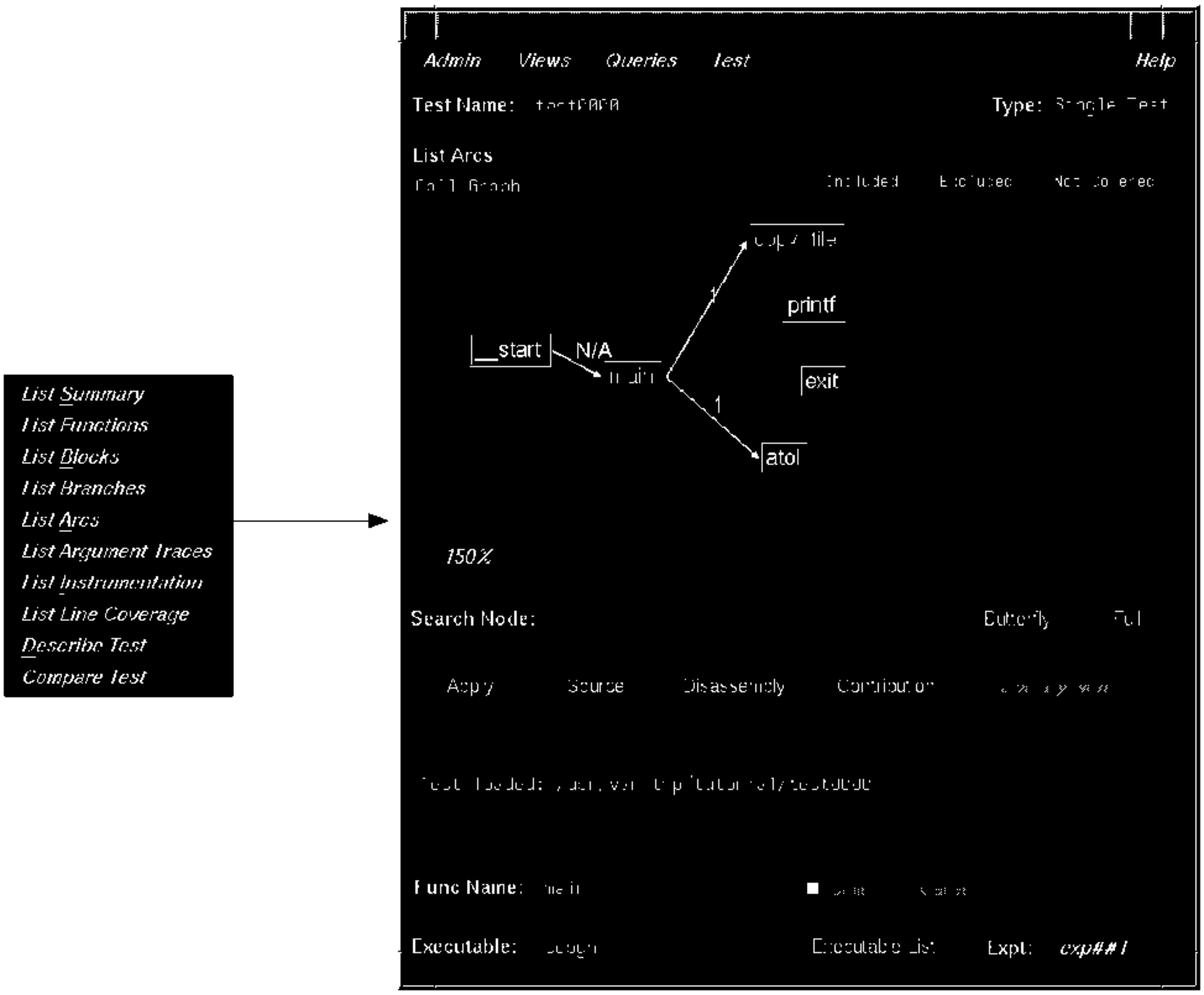


Figure 25–14 Call Graph for "List Arcs" Query

5. Click the *Multiple Arcs* button (the third button from the right in the row of display controls).

   This displays each of the potential arcs between the nodes. See **Figure 25–15** . Arcs labeled N/A connect excluded functions and do not have call counts.

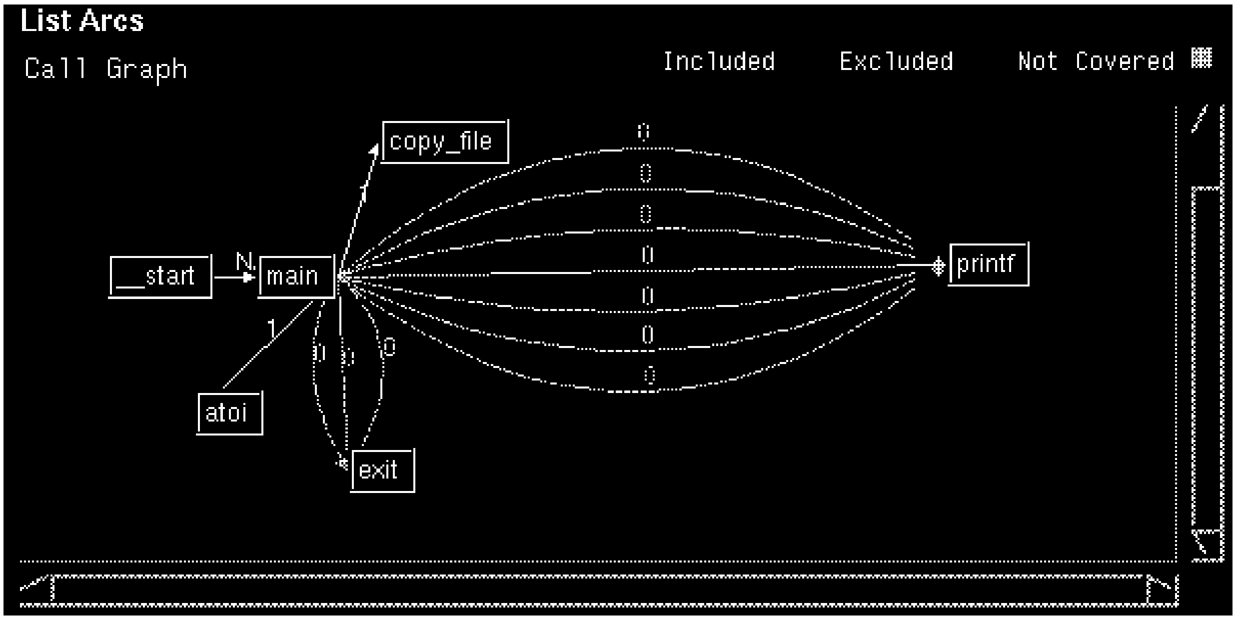


Figure 25–15 Call Graph for "List Arcs" Query — Multiple Arcs

6. Select "Text View" from the Views menu.

   This returns the display area to text mode from call graph mode. See **Figure 25–16** .

   The *Callers* column lists the calling functions. The *Callees* column lists the functions called. *Line* provides the line number where the call occurred; this is particularly useful if there are multiple arcs between the caller and callee. The *Files* column identifies the source code file. *Counts* shows the number of times the call was made.

   You can sort the data in the "List Arcs" query by count, file, caller, or callee.

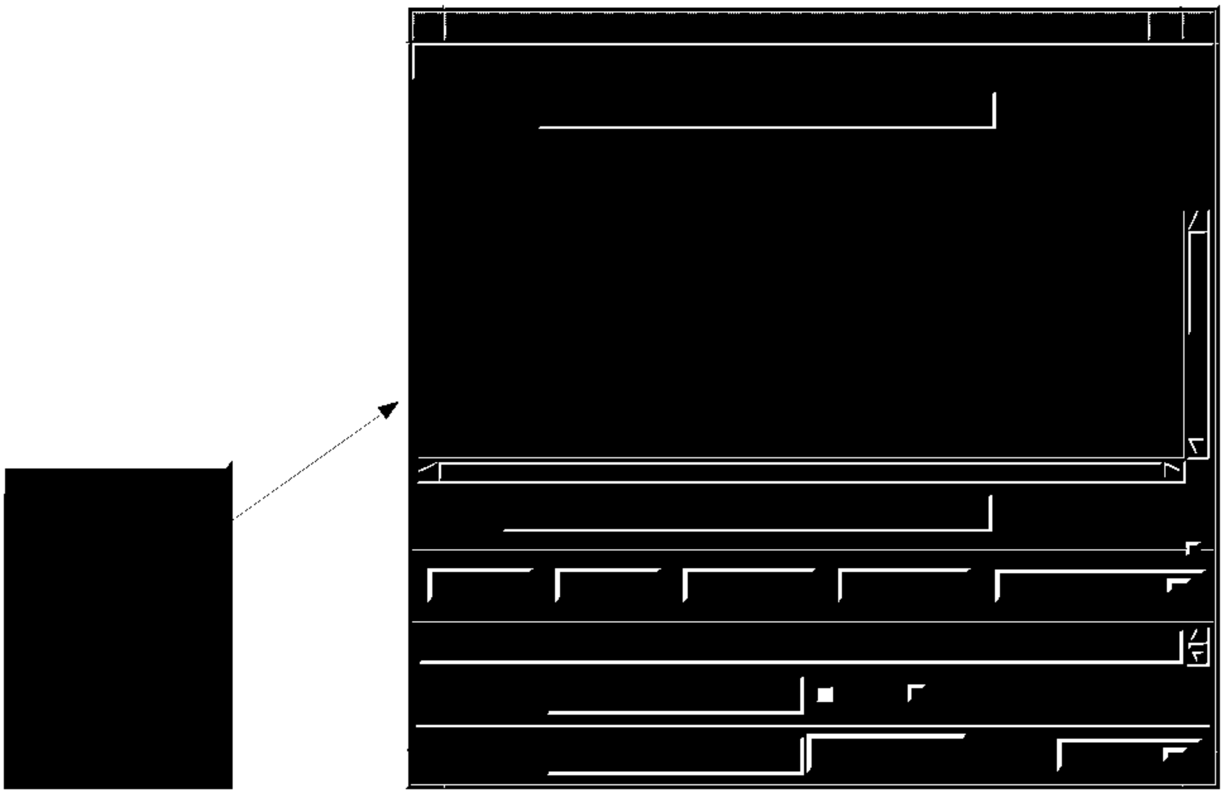


Figure 25–16 Test Analyzer Queries: "List Arcs"

7. Select "List Blocks" from the Queries menu.

   The window should be similar to **Figure 25–17** . The data displays in order of blocks, with the starting and ending line numbers of the block indicated. Blocks that span multiple lines are labeled sequentially in parentheses. The count for each block is shown with 0–count blocks highlighted.

   Caution: Listing all blocks in a program may be very slow for large programs. To avoid this problem, limit your "List Blocks" operation to a single function.

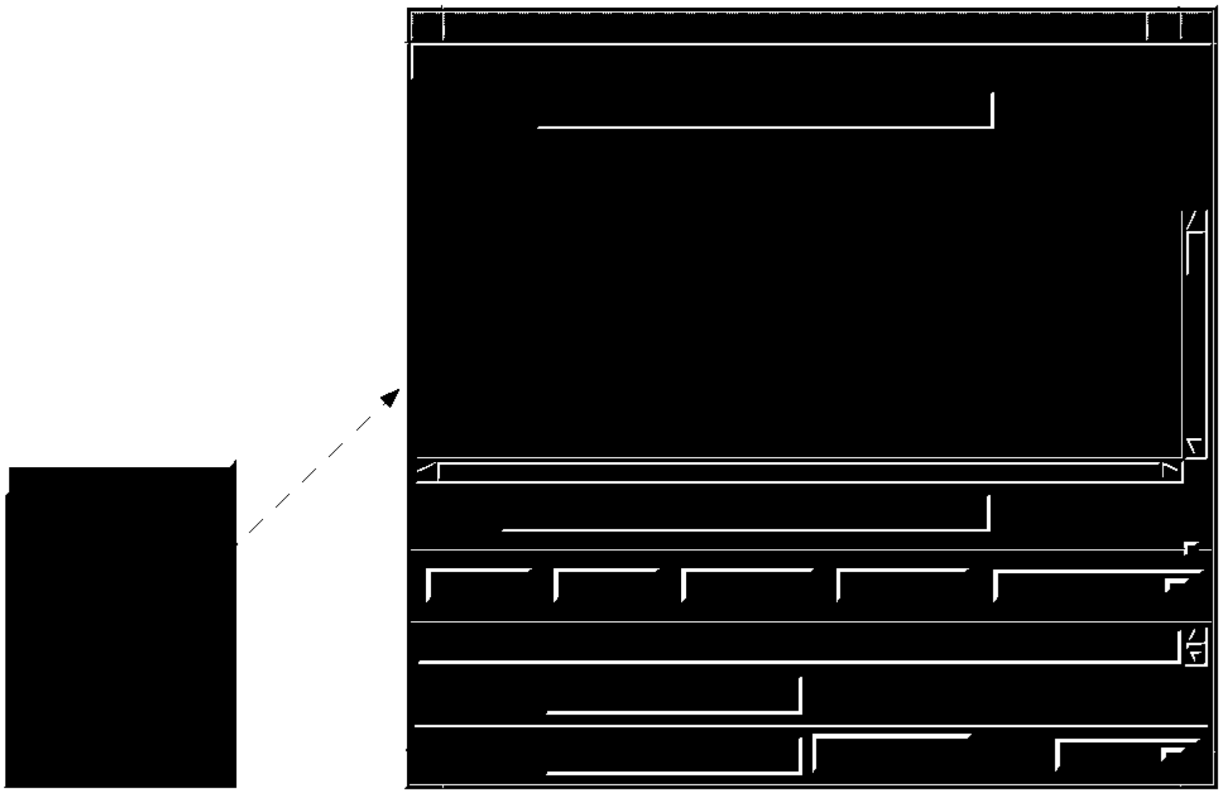


Figure 25–17 Test Analyzer Queries: "List Blocks"

You can sort the data for "List Blocks" by count, file, or function.

8. Select "List Branches" from the Queries menu.

   The "List Branches" query displays a window similar to **Figure 25–18** .

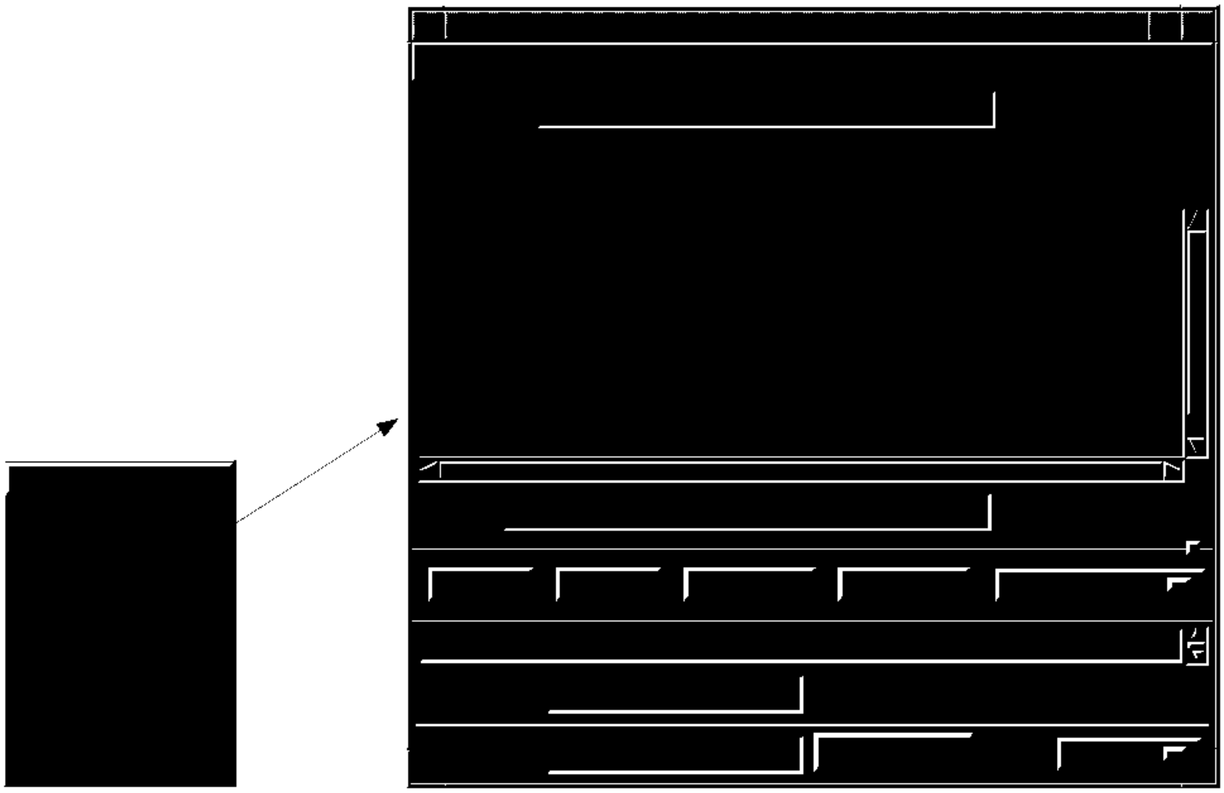


Figure 25–18 Test Analyzer Queries: "List Branches"

The first column shows the line number in which the branch occurs. If there are multiple branches in a line, they are labeled by order of appearance within trailing parentheses. The next two columns indicate the function containing the branch and the file. A branch is considered covered if it has been executed under both true and false conditions. The *Taken* column indicates the number of branches that were executed only under the true condition. The *Not Taken* column indicates the number of branches that were executed only under the false condition.

The "List Branches" query permits sorting by function or file.

Chapter 26

Tester Graphical User Interface Reference

This chapter describes the Tester graphical user interface. It contains these sections:

- **"Accessing the Tester Graphical Interface"**
- **"Main Window and Menus"**
- **"Test Menu Operations"**
- **"Views Menu Operations"**
- **"Queries Menu Operations"**
- **"Admin Menu Operations"**

When you run *cvxcov*, the main Tester window opens and an iconized version of the Execution View appears on your screen. It displays the output and status of a running program and accepts input. To open a closed Execution View, see "Clone Execution View" in **"Admin Menu Operations"** .

Accessing the Tester Graphical Interface

There are two methods of accessing the Tester graphical user interface:

- Type **cvxcov** at the command line with these optional arguments: `-testname` *test* to load the test ; `-ver` to show the Tester release version; and `-scheme` *schemename* to set a predefined CASEVision color scheme.
- Select "Tester" from the "Launch Tool" submenu in a WorkShop Admin menu (see **Figure 26–1**). The major WorkShop tools, the Debugger, Static Analyzer, and Build Manager provide Admin menus from which you can access Tester.

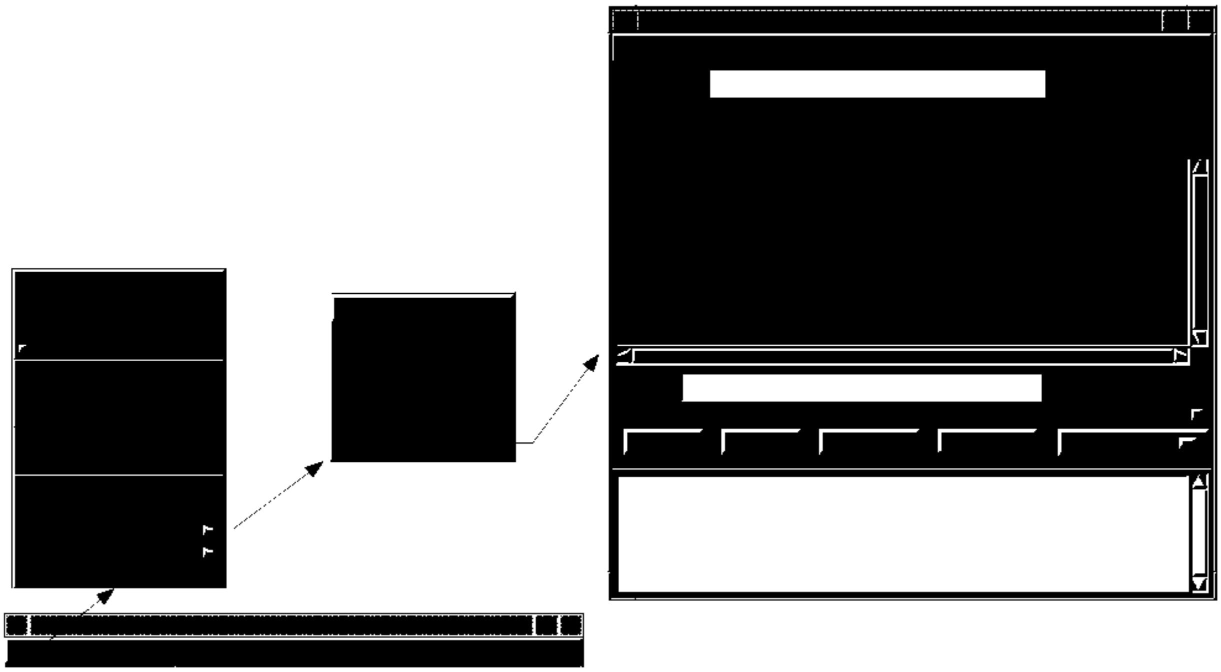


Figure 26–1 Accessing Tester from the WorkShop Debugger

Main Window and Menus

The main window and its menus are shown in **Figure 26–2** .

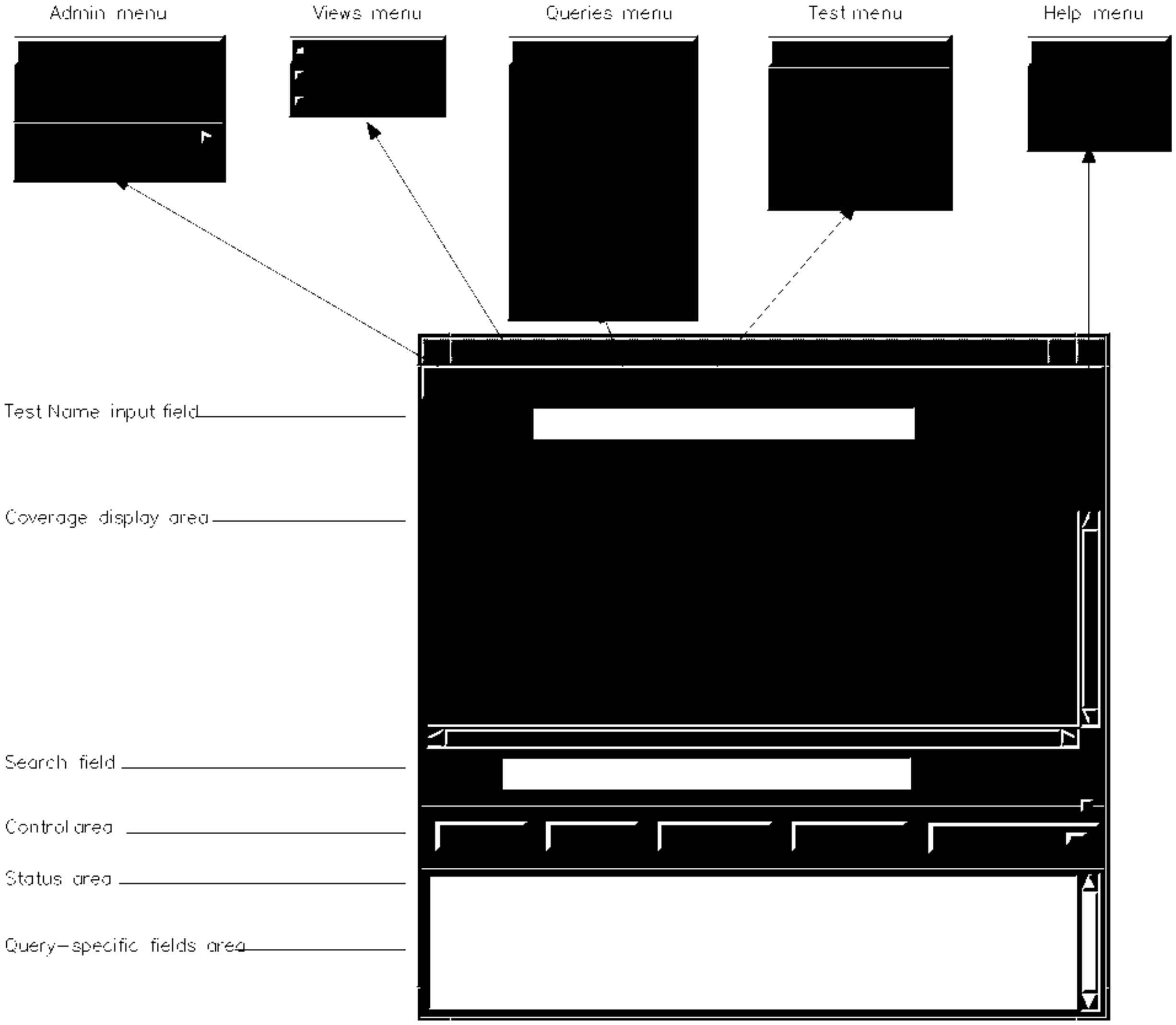


Figure 26–2 Main Test Analyzer Window

Test Name Input Field

The current test is entered (and displayed) in the *Test Name* field. You can switch to a different test, test set, or test group through this field. To the right, the *Type* field indicates whether it is a Single Test, Test Set, or Test Group. You can select a test (test set or test group) from the "List Tests" dialog box under the Test menu, to appear in the *Test Name* field in the main window.

Coverage Display Area

Test results display in the *coverage display area*. You select the results by choosing an item from the Queries menu. You can select the format of the data—text, call tree, or bar chart— from the Views menu. (Note that the Text View format is available for all queries, whereas the other two views are limited.)

The *Query Type* displays under the *Test Name* field, just over the display. It is followed on the far right of the window by the *Query Size* (number of items in the list). Headings above the display are specific to each query.

Search Field

The *Search* field lets you look for strings in the coverage data. It uses an incremental search, that is, as you enter characters, the highlight moves to the first matching target. When you press **`<return>`**, the highlight moves to the next occurrence.

Control Area Buttons

Apply is a general–purpose button for terminating data entry in text fields; you can use <**`return`**> equivalently. Both start the query.

Source lets you bring up the standard CASEVision Source View window with Tester annotations. Source View shows the counts for each line and highlights lines with 0 counts. By default Source View is shared with other applications. For example, if *cvstatic* performs a search for function *A*, the results of the query overwrite Tester query results that are in the shared Source View. To stop sharing Source View with other applications, set the following resource:

```
cvsourceNoShare:  True
```

Disassembly brings up the CASEVision Disassembly View window, called "Assembly Source Coverage", which operates at the machine level in a similar fashion to the Source View. This view is not shared with other applications.

Note: If a test has very large counts, there may not be enough space in the Source View and Disassembly View windows to display them. To make more room, increase the *canvasWidth* resource in the *Cvxcov* app–defaults file, *Cvxcov*test*testdata*canvasWidth*.

Contribution brings up the Test Contribution window with the contributions made by each test so that you can compare the results. It is available for the queries "List Functions", "List Arcs", and "List Blocks". When the tests do not fit on one page, multiple pages are used. Use the Previous Page and Next Page buttons to display all the tests.

Sort lets you sort the test results by criteria such as function, count, file, type, difference, caller, or callee. The criteria available depend on the current query.

Status Area and Query–Specific Fields

The *status area* displays status messages that confirm commands, issue warnings, and indicate error conditions. When you enter a test name in the *Test Name* field, the *Func Name* field appears (along with other items) in the status area for use with queries. Entering a function in this field displays the coverage results limited to that function only.

Additional items display in the area below the status area that change when you select commands from the Queries menu. These items are specific to the query selected. Some of these items can be used as defaults (see **"Queries Menu Operations"**).

Main Window Menus

The Admin menu lets you perform general housekeeping concerning saving files, setting defaults, changing directories, launching other WorkShop applications, and exiting.

The Test menu lets you create, modify, and run tests, test sets, and test groups.

The Views menu lets you choose one of the following modes:

- text mode, which displays results numerically in columns
- graphical mode, which displays
  - functions as nodes (rectangles) annotated by results
  - calls as arcs (connecting arrows)
- bar graph mode, which displays the summary of a test as a bar graph.

The Queries menu lets you analyze the results of tests. The Help menu is standard in all CASEVision tools.

Test Menu Operations

All operations for running tests are accessed from the Test menu in the main Tester window. **Figure 26–3** shows the dialog boxes used to perform test operations.

The Test menu provides these selections:

"Run Instrumentation"

instruments the target executable. Instrumentation adds code to the executable to collect coverage data. For a more detailed discussion of instrumentation and instrument files, see **"Single Test Analysis Process"** .

The "Run Instrumentation" dialog box (see **Figure 26–4**) provides these fields:

- *Executable* lets you enter the name of the target.
- *Instrumentation File* is for entering the instrumentation file, which is an ASCII description of the instrumentation criteria for the experiment.
- *Instrumentation Dir* lets you enter the directory in which the instrumentation file is stored (not necessary if you're using the current working directory).

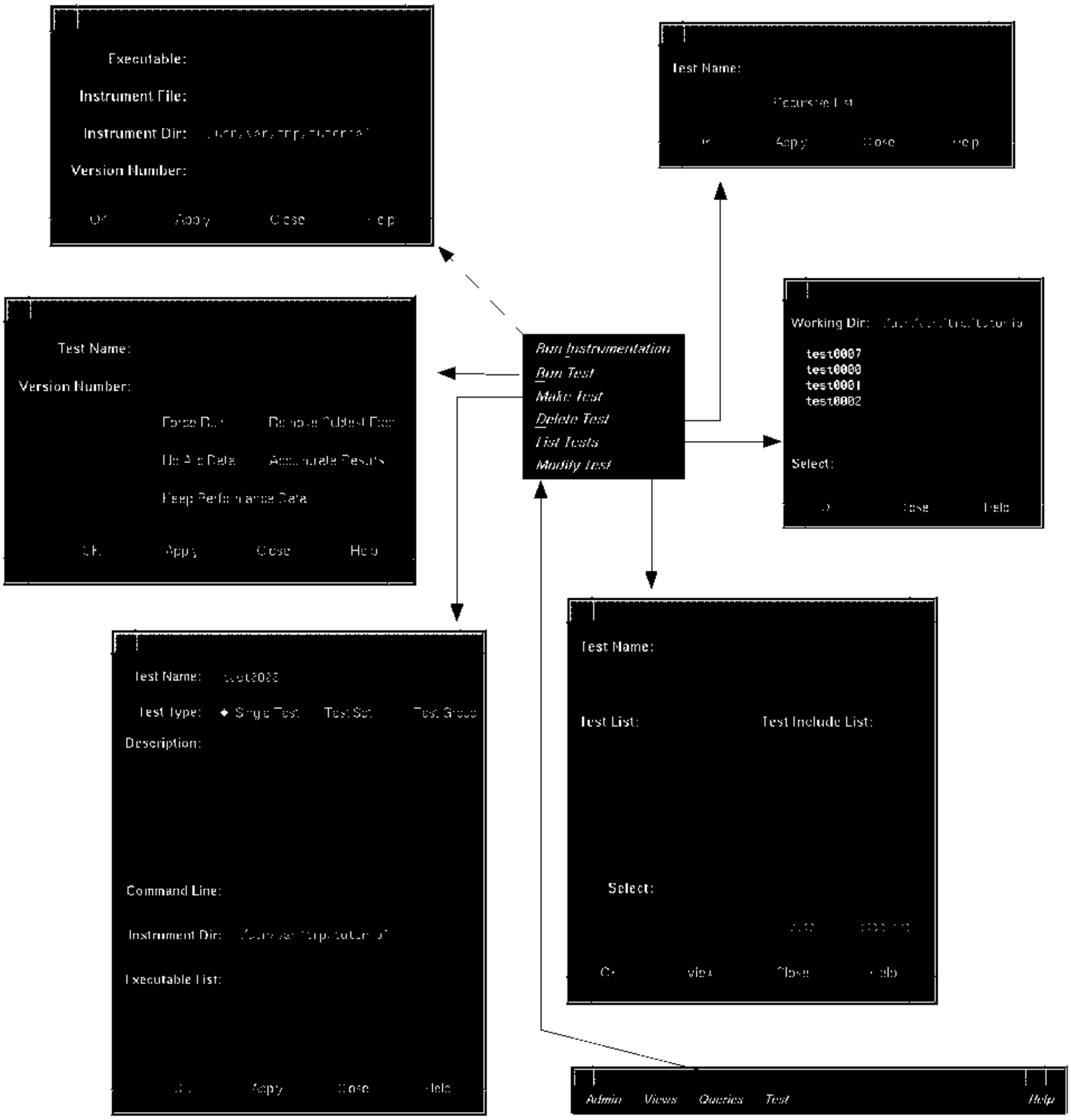
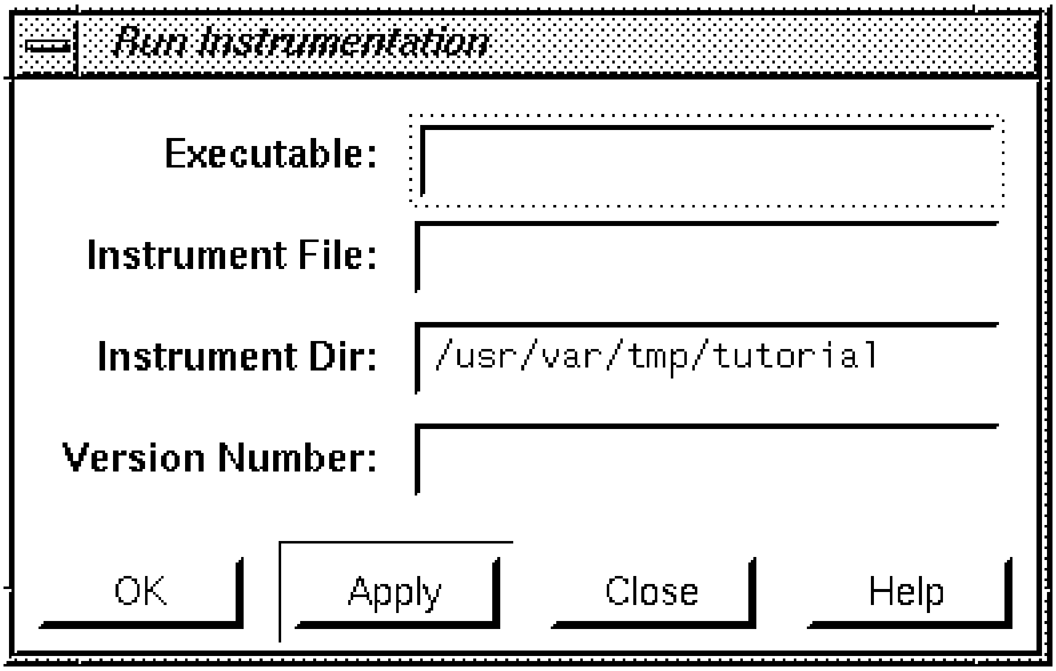


Figure 26–3 Test Menu Commands

- *Version Number* lets you specify the version number of the instrumentation directory (*ver##<versionnumber>*). If this field is left blank, the version number increments automatically.

  If you are testing multiple executables (that is, testing coverage of an executable that *forks*, *execs*, or *sprocs* other processes), then you need to store these in the same instrumentation directory. You do this by entering the same number in the *Version Number* field.



Instrument Dir: /usr/var/tmp/tutorial

Figure 26–4 "Run Instrumentation" Dialog Box

"Run Test"

invokes the executable with selected arguments and collects the coverage data. The "Run Test" dialog box (see **Figure 26–5**) provides these fields and buttons:

- *Test Name* is for entering the test name.
- *Version Number* is for entering the version number of the directory (*ver## <number>*) containing the instrumented executable. If you are using the most current (highest) version number, then you can leave the field blank; otherwise, you need to enter the desired number.
- *Force Run* is a toggle that when turned on causes the test to be run even if results already exist.
- *Keep Performance Data* is a toggle that when turned on retains all the performance data collected in the experiment.
- *Accumulate Results* is a toggle that when turned on accumulates (sums over) the coverage data into the existing experiment results.
- *No Arc Data* prevents arc information from being collected in the experiment. It can't be used with "List Arcs" or a Call Tree View. "List Summary" and "Compare Test" will have 0% coverage on arc items. Use it to save space if you don't need arc data.
- *Remove Subtest Expt* removes results for individual subtests for test sets or test groups, letting you see the top level and taking less space. There will be no data to query if you are querying a subtest.

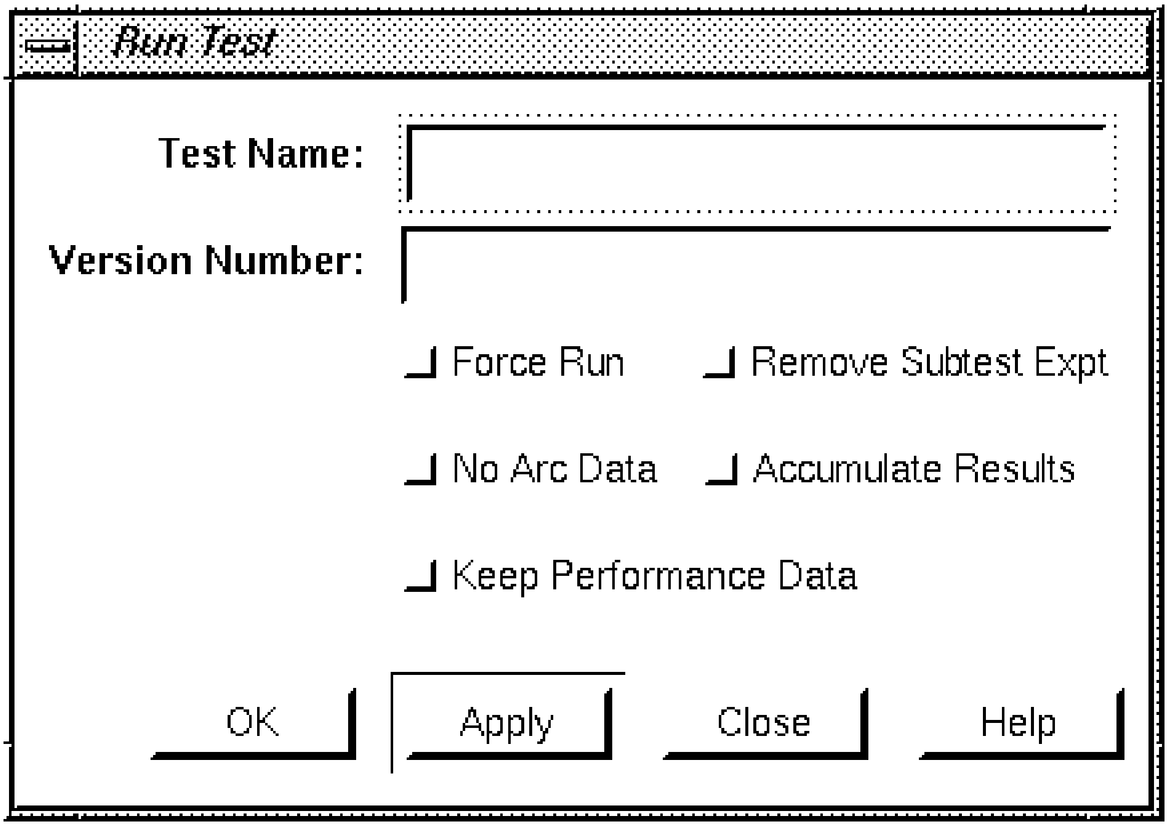


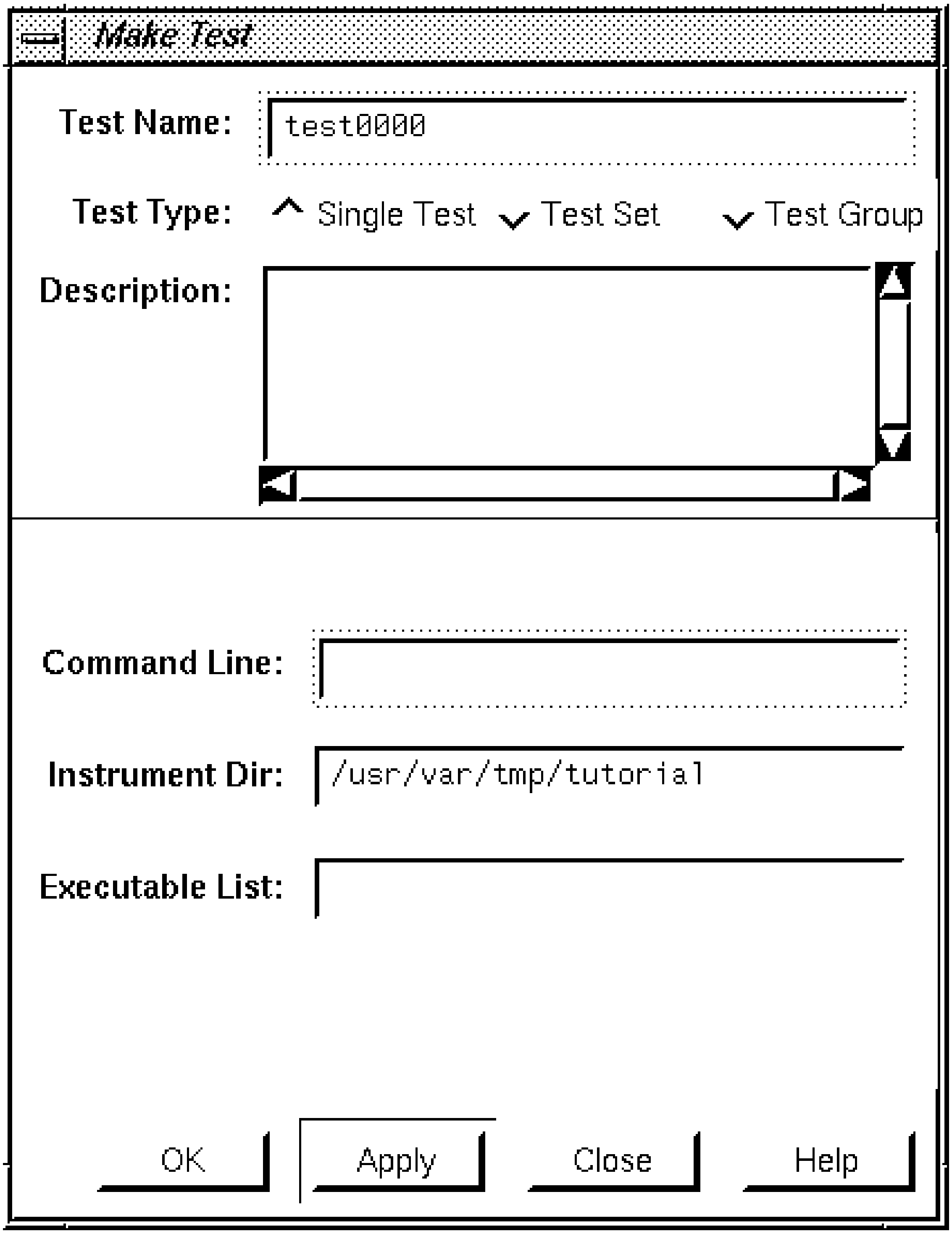
Figure 26–5 "Run Test" Dialog Box

"Make Test"

creates a test directory where the coverage data is to be stored and stores a TDF (test description file). For more information on this process, see **"Single Test Analysis Process"** .

The "Make Test" dialog box (see **Figure 26–6**) provides these fields for tests, test sets, and test groups:

- *Test Name* is for entering the test name.
- *Test Type* is a toggle for indicating the type of test: single, test set, or test group (for dynamically shared objects).
- *Description* lets you enter a description to document the test.



Test Name: test0000

Instrument Dir: /usr/var/tmp/tutorial

Figure 26–6 "Make Test" Dialog Box

If you select *Single Test*, the following fields are provided:

- *Command Line* lets you enter the target and any arguments to be used in the test.
- *Instrument Dir* is the directory in which the instrumentation file and related data are stored (not necessary if current working directory).
- *Executable List* is used if you are testing coverage of an executable that *forks*, *execs*, or *sprocs* other processes and want to include those processes. You must specify these executables in the *Executable List* field.

If you select *Test Set*, the following fields and buttons are provided:

- *Test List* contains all the tests in the working directory.
- *Test Include List* (to the right) displays tests included in the test set or test group.
- *Add* looks at the selected item in the *Test List* or *Select* field and adds it to the *Test Include List*.
- *Remove* looks at the selected item in the *Test Include List* and removes it.
- *Select* displays the currently selected test.

For a test group (see **Figure 26–7**), the following field is added to the same fields and buttons used for a test set:

- *Targets* lets you enter a list of target DSOs or shared libraries, separated by spaces.

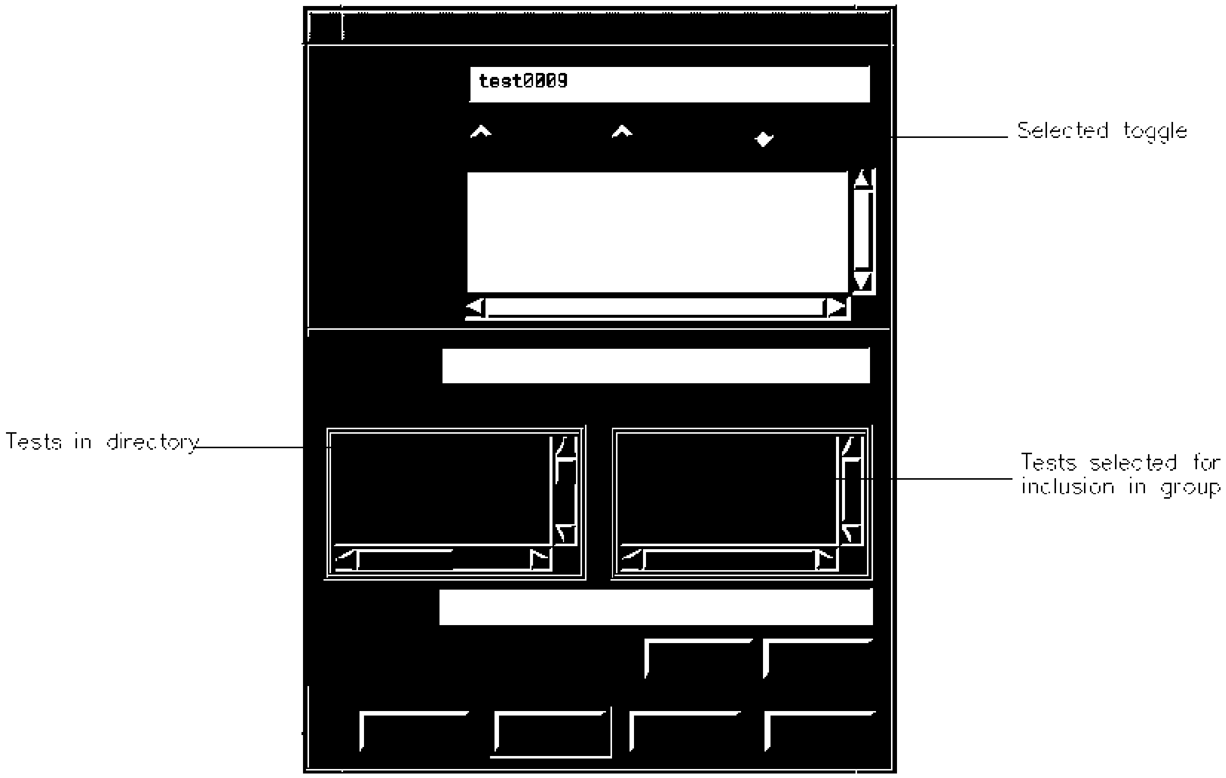

test0009

Figure 26–7 "Make Test" Dialog Box with Test Group Selected

"Delete Test"

removes the specified test directory and its contents. The "Delete Test" dialog box (see **Figure 26–8**) provides these fields:

- *Test Name* is for entering the test name.
- *Recursive List* is a toggle that when turned on includes all subtests in the removal of test sets and test groups.

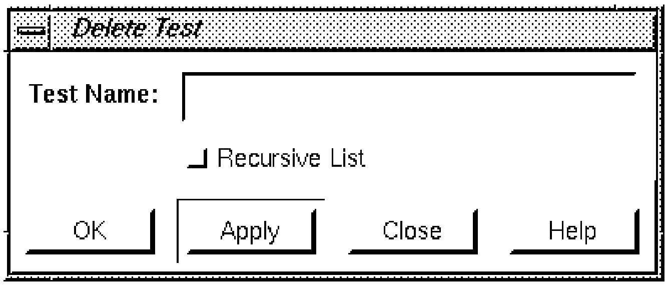
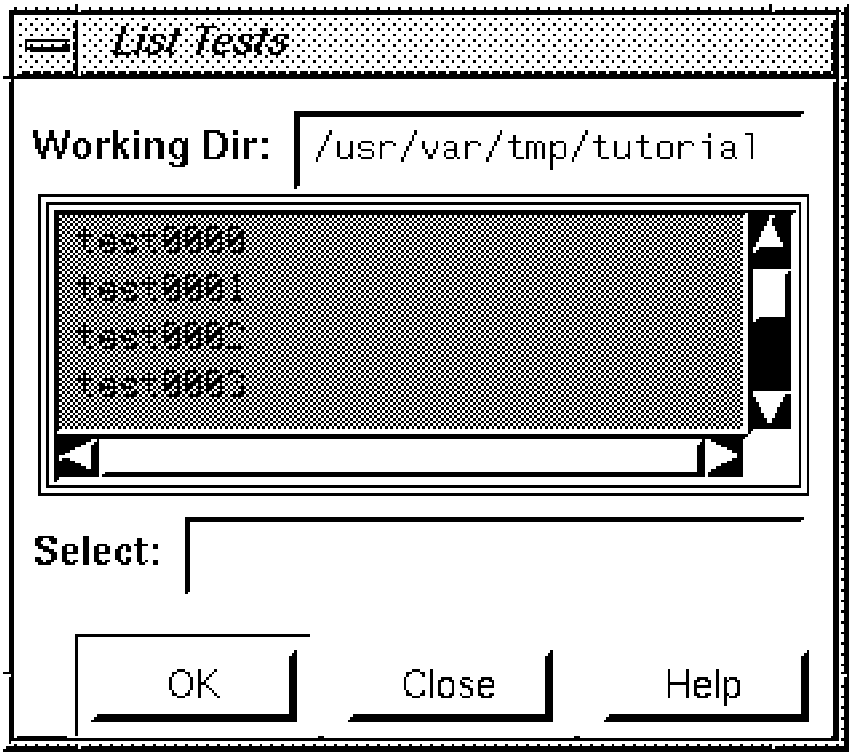


Figure 26–8 "Delete Test" Dialog Box

"List Tests"

shows you the tests in the current working directory. The "List Tests" dialog box (see **Figure 26–9**) provides these fields:

- *Working Dir* shows the directory containing the tests.
- A scrollable list field displays the tests present in the specified directory. The scroll bars let you navigate through the tests if they don't fit completely in the field. Clicking an item places it in the *Select* field. Double–clicking on a test selects and loads it.
- *Select* displays the test name you type in or that you clicked in the list. Click *OK* to load your selection into the *Test Name* field of the main Tester window.
- *Close* lets you exit without loading a selection.



Working Dir: /usr/var/tmp/tutorial

test000
test001
test002
test003

Select:

Figure 26–9 "List Tests" Dialog Box

"Modify Test"

lets you modify a test set or test group. You enter the test name in the *Test Name* field and press **`<return>`** or click the *View* button to load it. The *View* button changes to *Apply*, the *Test List* field displays tests in the current working directory, and the *Test Include List* field displays the contents of the test set or test group. You can then add or delete tests, test sets, or test groups in the current test set or test group, respectively. The "Modify Test" dialog box (see **Figure 26–10**) has these fields:

- *Test Name* is for entering the test name.
- *Test List* displays the tests in the current directory.
- *Test Include List* displays the subtests for the test specified in the *Test Name* field.
- *Select* displays the test currently selected for adding or removing. You can enter the test directly in this field instead of selecting it from the *Test List* or *Test Include List*.
- The *Add* button lets you add the selected test to the *Test Include List*.
- The *Remove* button lets you delete the selected test from the *Test Include List*.
- The *Apply* button applies the changes you have selected. (The button name is *View* until you load something.)

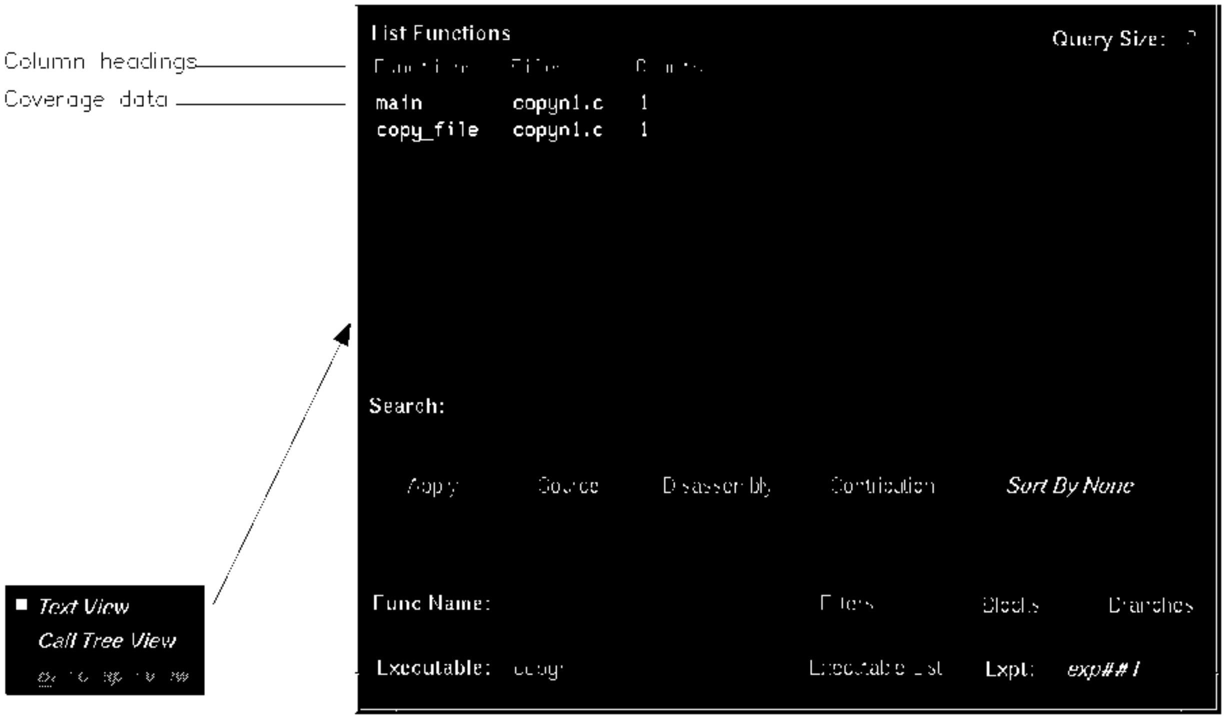
Figure 26–10 "Modify Test" Dialog Box After Loading Tests

Views Menu Operations

The Views menu has three selections that let you view coverage data in different forms. The selections are:

"Text View"

displays the coverage data in text form. The information displayed depends on which query you have selected. See **Figure 26–11** .



```
main        copyn1.c   1
copy_file   copyn1.c   1
```

Figure 26–11 "List Functions" Query in "Text View" Format

"Call Tree View"

displays coverage data graphically, with functions as nodes (rectangles) and calls as arcs (connecting arrows). This view is only valid for "List Functions", "List Blocks", "List Branches", and "List Arcs". See **Figure 26–12** . It is not available if you run a test with No Arc Data on.

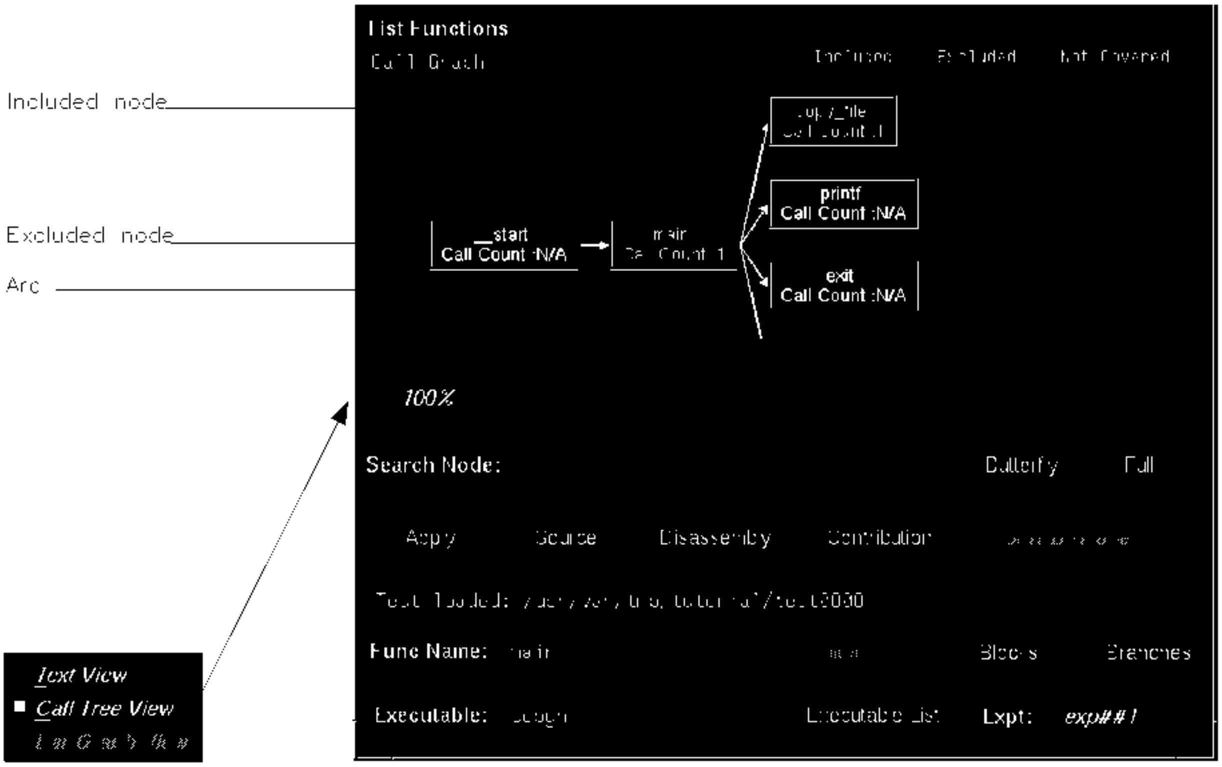


Figure 26−12 "List Functions" Query in "Call Tree View" Format

"Bar Graph View"

displays a bar chart showing the percentage covered for functions, lines, blocks, branches, and arcs. See **Figure 26−13** . This view is only valid for "List Summary", which is described in detail in **"Queries Menu Operations"** .

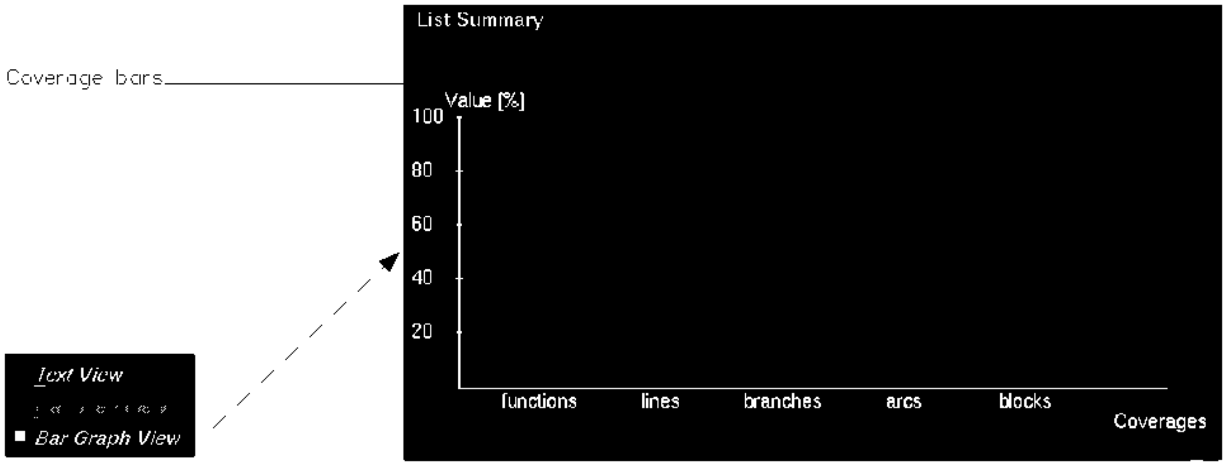


Figure 26–13 "List Summary" Query in "Bar Graph View" Format

Queries Menu Operations

The Queries menu provides different methods for analyzing the results of coverage tests. Each type of query displays the coverage data in the coverage display area in the main Tester window and displays items that are specific to the query in the area below the status area. When you set these items for a query, the same values are used by default for subsequent queries until you change them. You can set these defaults before the first query or as part of any query. For a single test or test set, all queries except "Describe Test" have the fields shown in **Figure 26–14** .

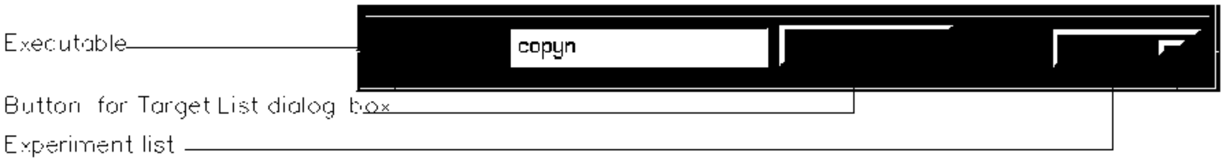


Figure 26–14 Query–Specific Default Fields for a Test or Test Set

The *Executable* field displays the executable associated with the current coverage data. You can switch to a different executable by entering it directly in this field. You can also switch executables by clicking the *Executable List* button, selecting from the list in the Target List dialog box and clicking *Apply* in the dialog box.

The experiment menu (*Expt*) lets you see the results for a different experiment that uses the same test criteria.

Note: When you are performing queries on a test group, the *Executable* field changes to *Object* field and the *Executable List* button changes to *Object List* as shown in **Figure 26–14** . These items act analogously except that they operate on dynamically shared objects (DSOs). Refer to **"Tutorial #4 — Analyzing a Test Group"** for more information on test groups.

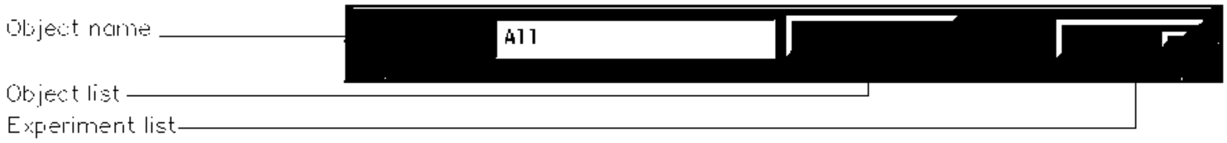


Figure 26−15 Query−Specific Default Fields for a DSO Test Group

The Queries menu (see **Figure 26−16**) provides these selections:

Figure 26−16 Queries Menu

"List Summary"

shows the overall coverage based on the user−defined weighted average over function, source line, branch, arc, and block coverage. The coverage data appears in the coverage display area. A typical summary appears in **Figure 26−17** .

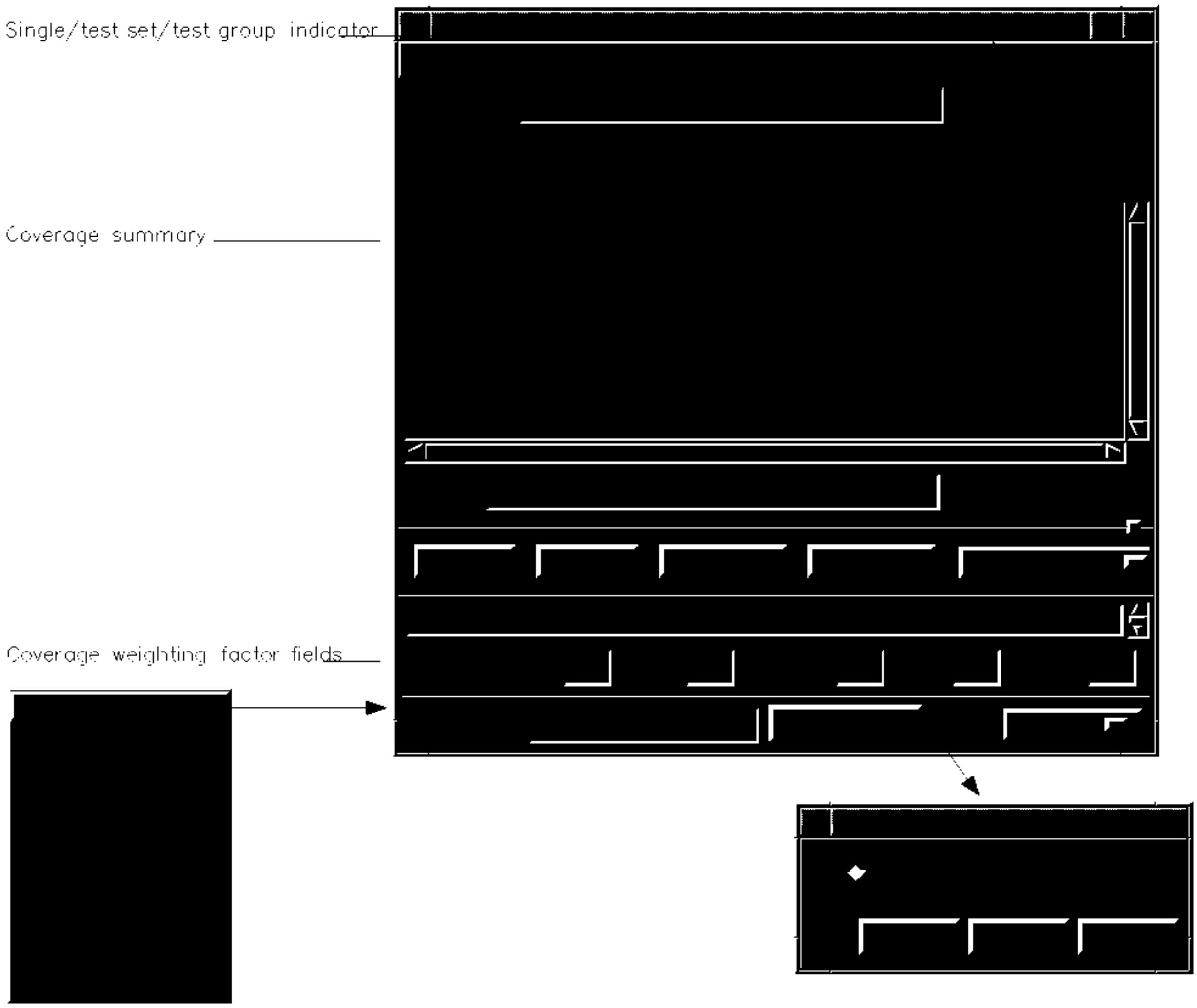


Figure 26–17 "List Summary" Query

The *Coverages* column indicates the type of coverage. The *Covered* column shows the number of functions, source lines, branches, arcs, and blocks that were executed in this test (or test set or test group). The *Total* column indicates the total number of items that could be executed for each type of coverage. The *% Coverage* column is simply the *Covered* value divided by the *Total* value in each category. The *Weight* column indicates the weighting assigned to each type of coverage. It is used to compute the *Weighted Sum*, a user–defined factor that can be used to judge the effectiveness of the test. The *Weighted Sum* is obtained by first multiplying the individual coverage percentages by the weighting factors and then summing the products.

The "List Summary" command causes the coverage weighting factor fields to display below the status area. Use these to adjust the factor values as desired. They should add up to 1.0.

If you select "Bar Graph View" from the Views menu, the summary will be shown in bar graph format as shown in **Figure 26–18** . The percentage covered is shown along the vertical axis; the types of coverage are indicated along the horizontal axis.

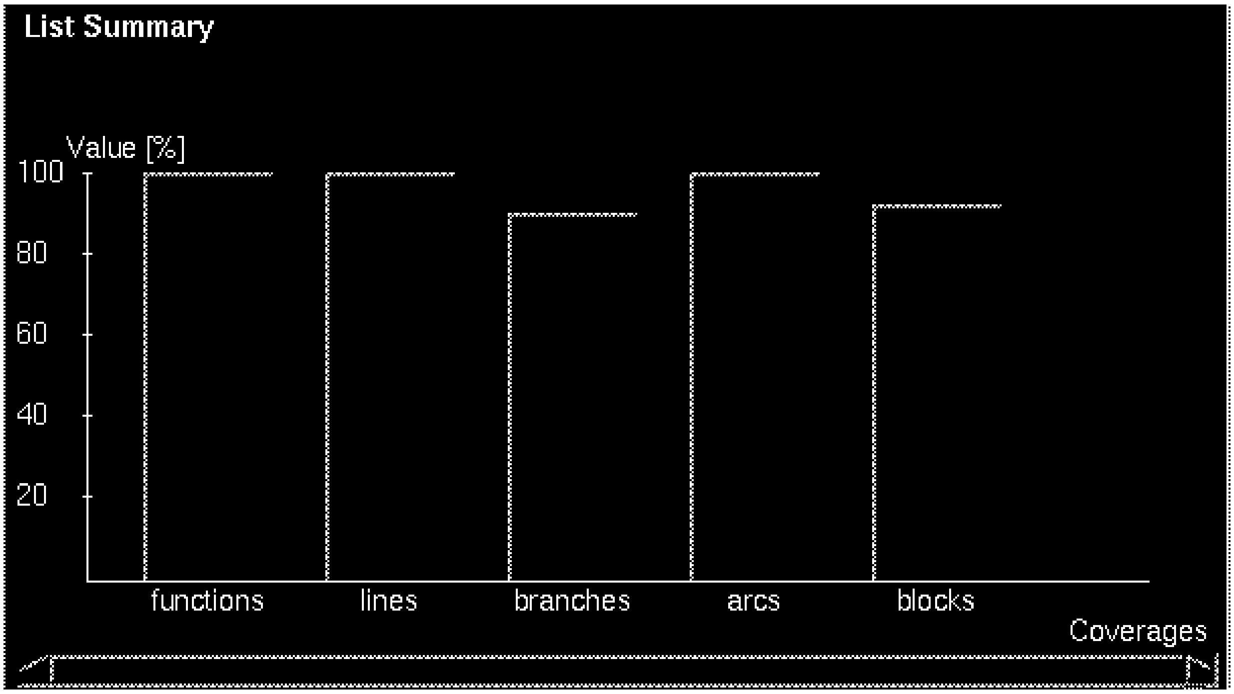


Figure 26–18 "List Summary" Bar Graph Example

"List Functions"

displays the coverage data for functions in the specified test. The *Functions* column heading identifies the function, *Files* shows the source file containing the function, and *Counts* displays the number of times the function was executed in the test.

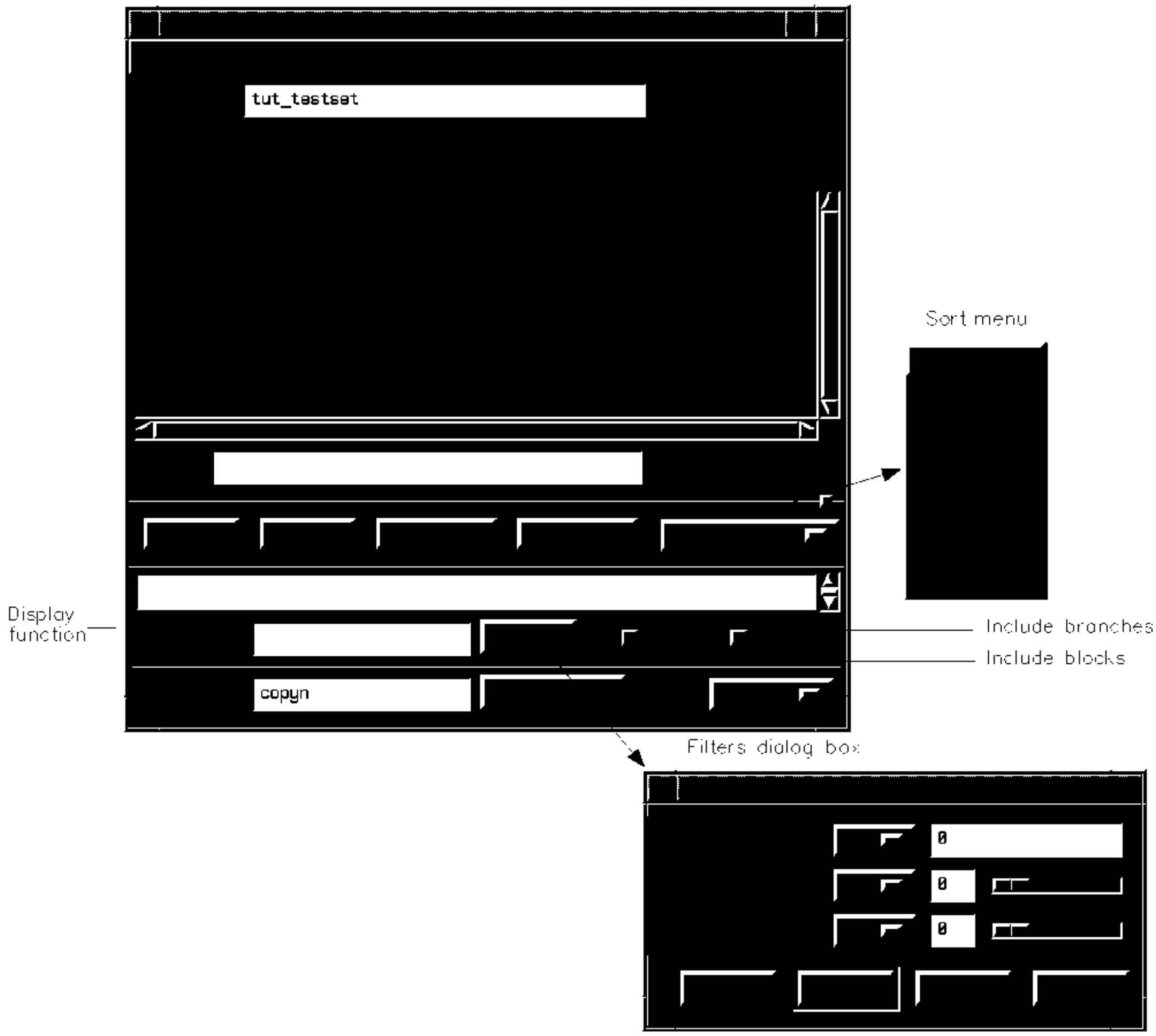
"List Functions" enables the sort menu that lets you determine the order in which the functions display. Only the sort criteria appropriate for the current query are enabled, in this case, "Sort By Func", "Sort By Count", and "Sort By File" as shown in **Figure 26–19** .

The *Search* field scrolls the list to the string entered. The string may occur in any of the columns. This is an incremental search and is activated as you enter characters, scrolling to the first matching occurrence.

Entering a function in the *Func Name* field displays the coverage results limited to that function only in the display area.

The *Filters* button displays the Filters dialog box, which lets you enter filter criteria to display a subset of the coverage results. There are three types of filters: *Function Count*, *Block Count (%)*, and *Branch Count (%)*. For blocks or branch coverage, use the toggles described below. Following each label is an operator menu to define the relationship to the limit quantity entered. Each filter type has a text field for entering the desired limit. The limits for *Block Count* and *Branch Count* are percentages (of coverage) and can also be entered using sliders.

Two toggles are available for including branch and block counts. Both appear as actual counts followed by parentheses containing the ratio of counts to total possible.



tut_testset

copyn

0

0

0

Figure 26–19 "List Functions" Query with Options

If you select "Call Tree View" from the Views menu with a "List Functions" query, a call graph displays (see **Figure 26–20**). The call graph displays coverage data graphically, with functions as nodes (rectangles) and calls as arcs (connecting arrows). The nodes are color–coded according to whether the function was included and covered in the test, included and not covered, or excluded from the test. Arcs labeled N/A connect excluded functions and do not have call counts.

If you hold down the right mouse button over a node, the node menu displays, including the function name, coverage statistics, and standard node manipulation commands. If you have a particularly large graph, you may find it useful to zoom to 15% or 40% and look at the coverage statistics through the node menu.

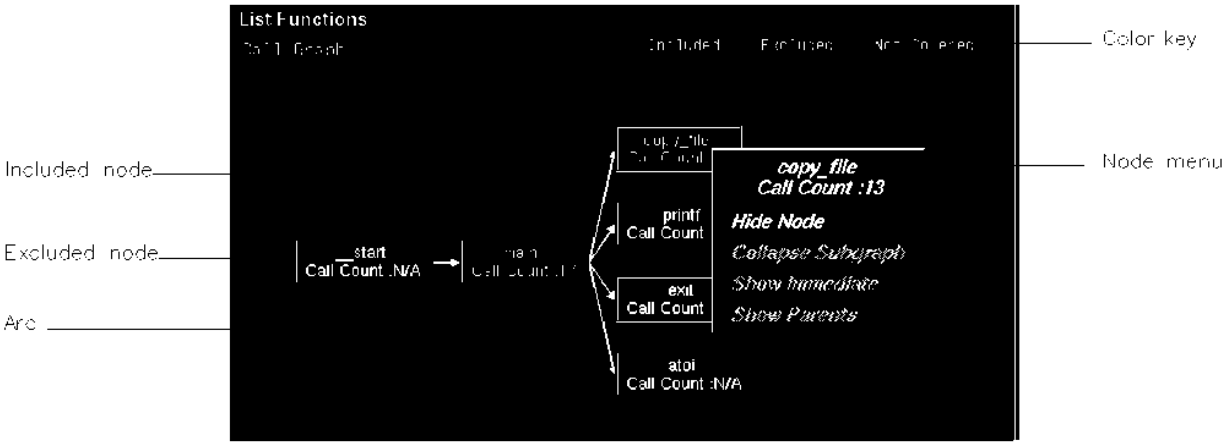


Figure 26–20 "List Functions" Example in "Call Tree View" Format

"List Blocks"

displays a list of blocks for one or more functions and the count information associated with each block (see **Figure 26–21**). The *Blocks* column displays the line number in which the block occurs. If there are multiple blocks in a line, blocks subsequent to the first are shown in order with an index number in parentheses. The other three columns show the function and file containing the block and the count, that is, the number of times the block was executed in the test. Uncovered blocks (those containing 0 counts) are highlighted. Block data can be sorted by function, file, or count.

Be careful before listing all blocks in the program, since this can produce a lot of data. Entering a function in the *Func Name* field displays the coverage results limited to that function only in the display area.

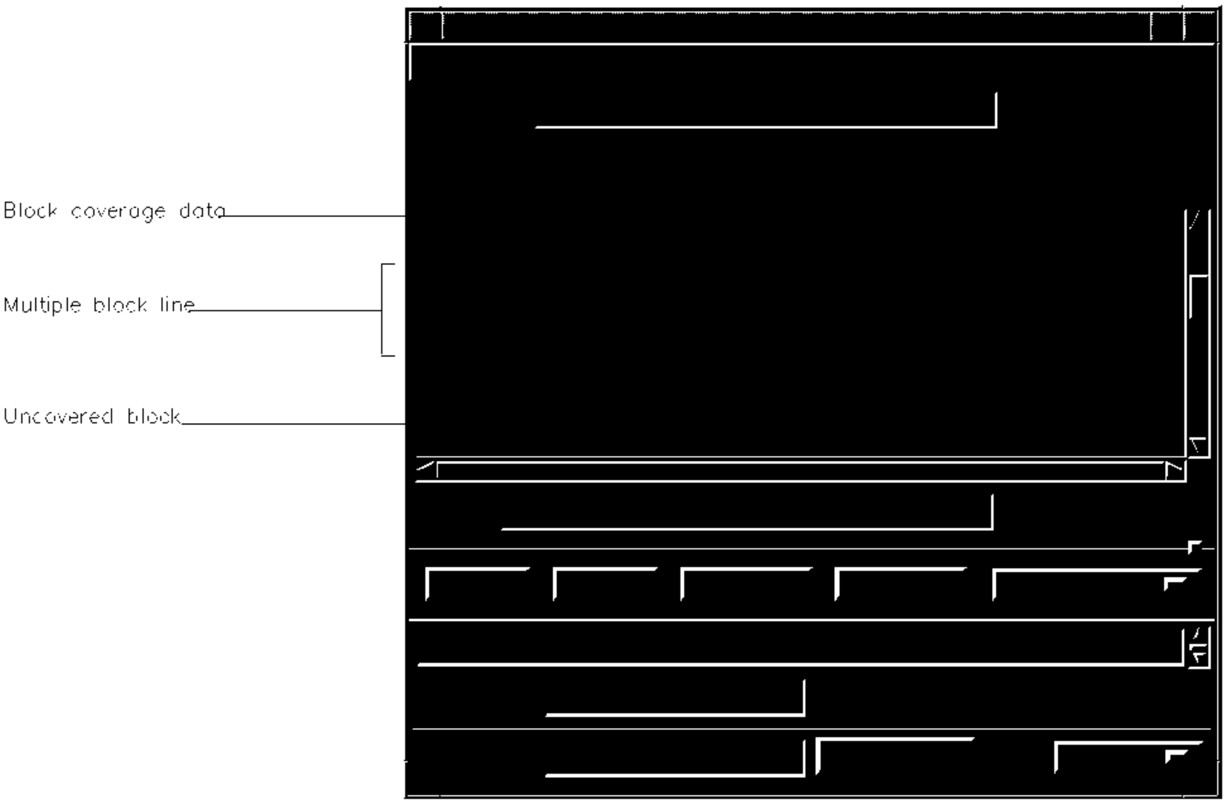


Figure 26–21 "List Blocks" Example

"List Branches"

lists coverage information for branches in the program. Branch coverage counts assembly language branch instructions that are taken and not taken. See **Figure 26–22**.

The first column shows the line number in which the branch occurs. If there are multiple branches in a line, they are labeled by order of appearance within trailing parentheses. The next two columns indicate the function containing the branch and the file. A branch is considered covered if it has been executed under both true and false conditions. The *Taken* column indicates the number of branches that were executed only under the true condition. The *Not Taken* column indicates the number of branches that were executed only under the false condition. Branch coverage can be sorted only by function and file. Entering a function in the *Func Name* field displays the coverage results limited to that function only in the display area.

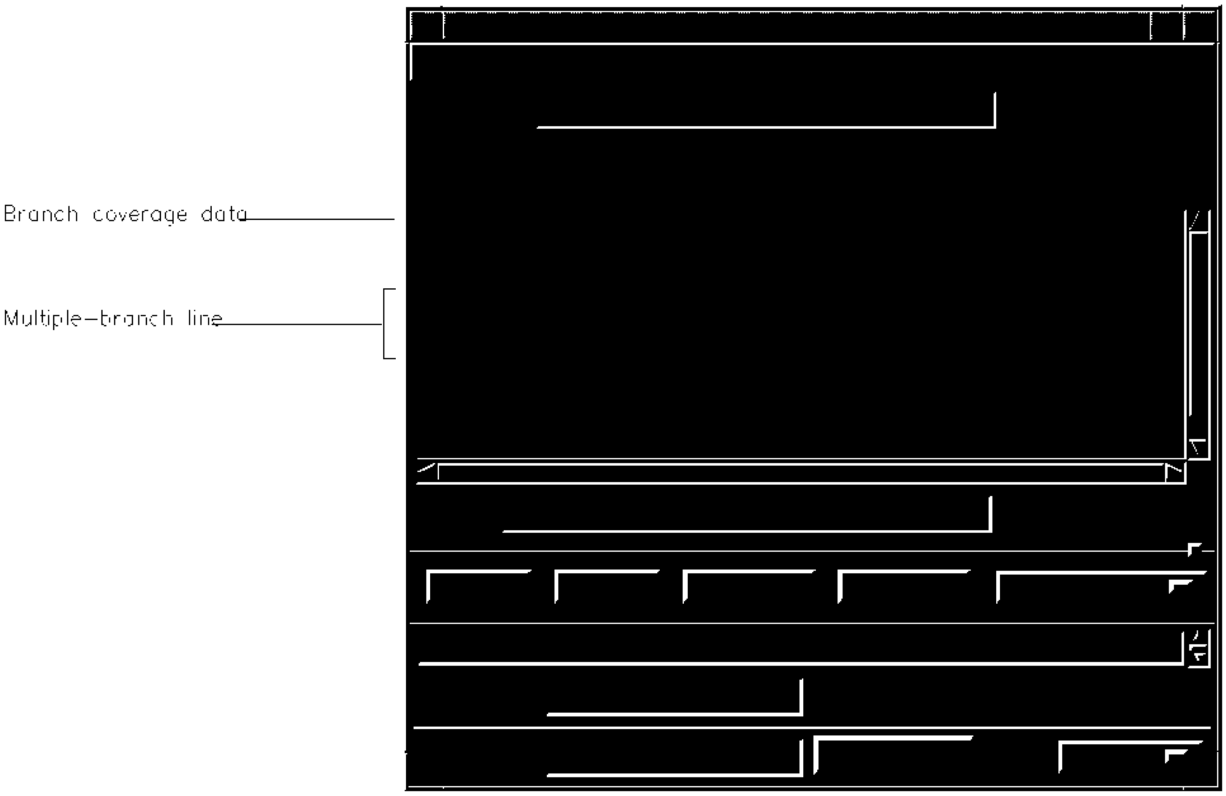


Figure 26–22 "List Branches" Example

"List Arcs"

shows arc coverage, that is, the number of arcs taken out of the total possible arcs. An arc is a call from one function (caller) to another (callee). See **Figure 26–23** . The caller and callee functions are identified in the first two columns. The *Line* column identifies the line in the caller function where the call occurs. The file and arc execution count display in the last two columns.

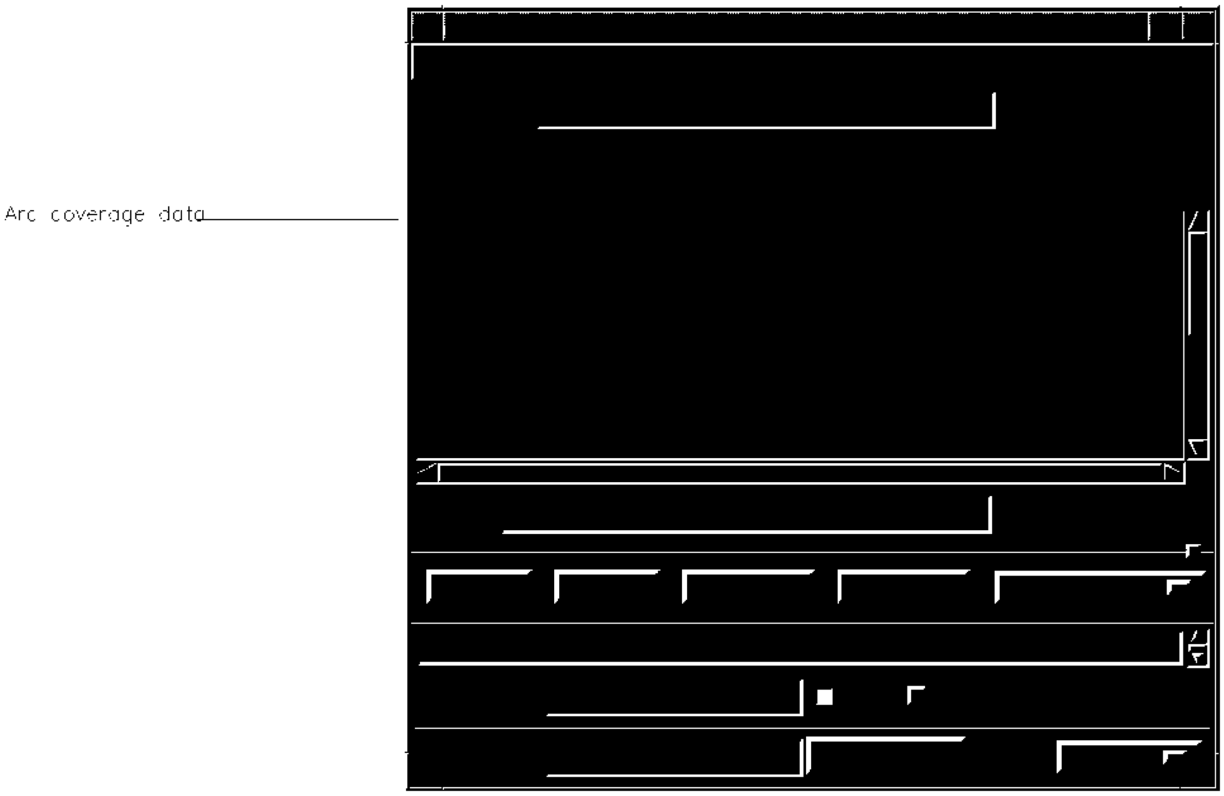


Figure 26–23 "List Arcs" Example

Entering a function in the *Func Name* field displays the coverage results limited to that function only.

The *Caller* and *Callee* toggles let you view the arcs for a single function either as a caller or callee. You do this by entering the function name in the *Func Name* field and then clicking the appropriate toggle, *Caller* or *Callee*.

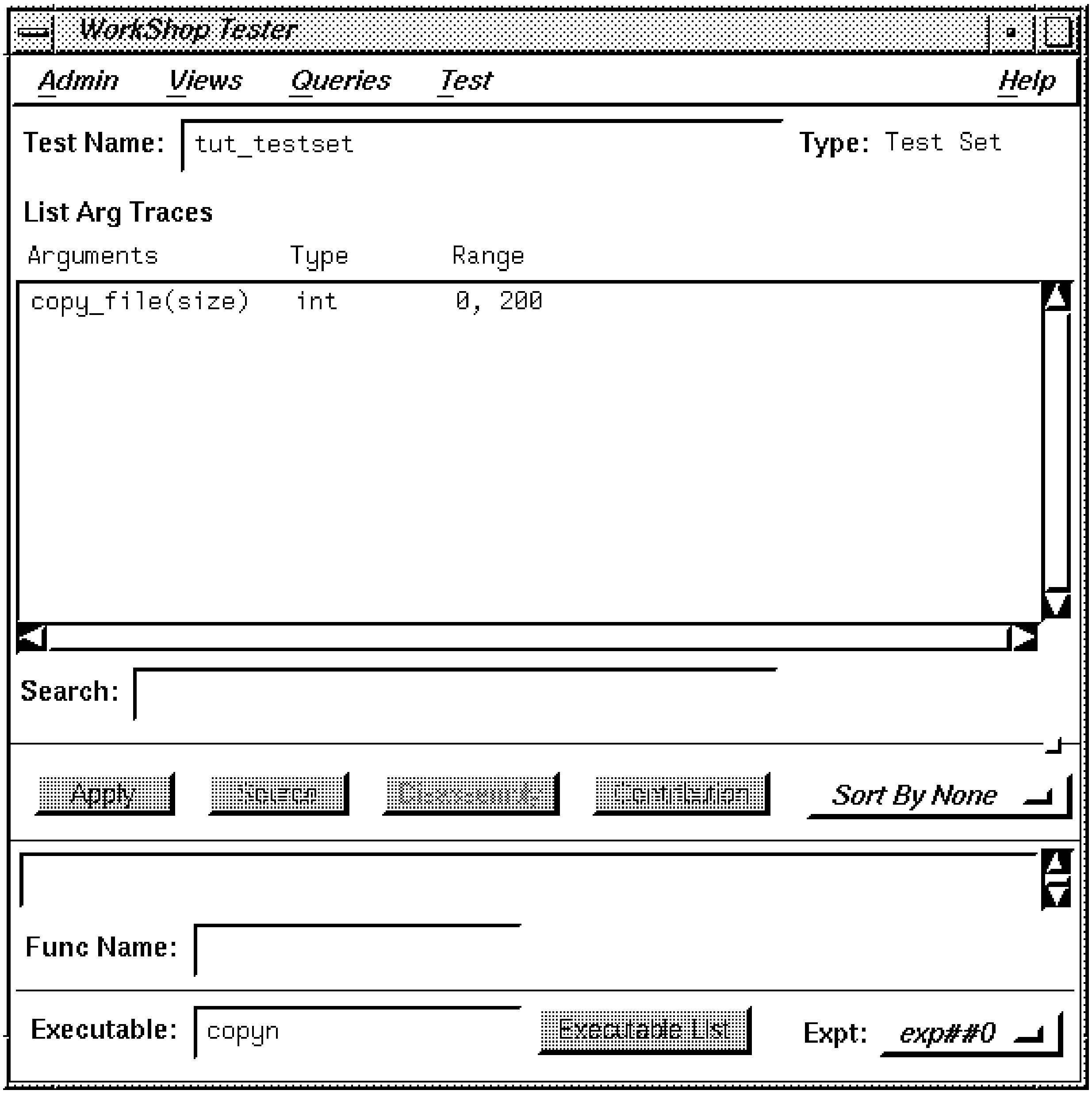
"List Argument Traces"

shows argument tracing information (see **Figure 26–24**). Argument tracing is enabled in the instrumentation file using the TRACE command with the MAX, MIN, BOUNDS, and RETURN options. TRACE lets you monitor argument values in the functions over all experiments. The syntax in the file is:

```
TRACE [RETURN] MAX|MIN|BOUNDS function(arg)
```

where:

- MAX monitors the maximum value of an argument.
- MIN monitors the minimum value of an argument.
- BOUNDS monitors both the minimum and maximum values.
- RETURN monitors the function return values. For more information on the instrumentation file, see **"Single Test Analysis Process"** .



Test Name: tut_testset

Type: Test Set

List Arg Traces

Arguments	Type	Range
copy_file(size)	int	0, 200

Search:

Func Name:

Executable: copyn

Expt: exp##0

Figure 26–24 "List Argument Traces" Example

The *Arguments* column shows the calling function with its argument. *Type* indicates the type of the argument. *Range* shows the minimum and maximum values if TRACE bounds was selected; otherwise, it shows the end of the range selected with a short line (–) substituted for the opposite end of the range.

Entering a function in the *Func Name* field displays the coverage results limited to that function only in the display area.

"List Instrumentation"

displays the instrumentation information for a particular test. See **Figure 26–25** .

Function List toggle shows the functions that are included in the coverage experiment.

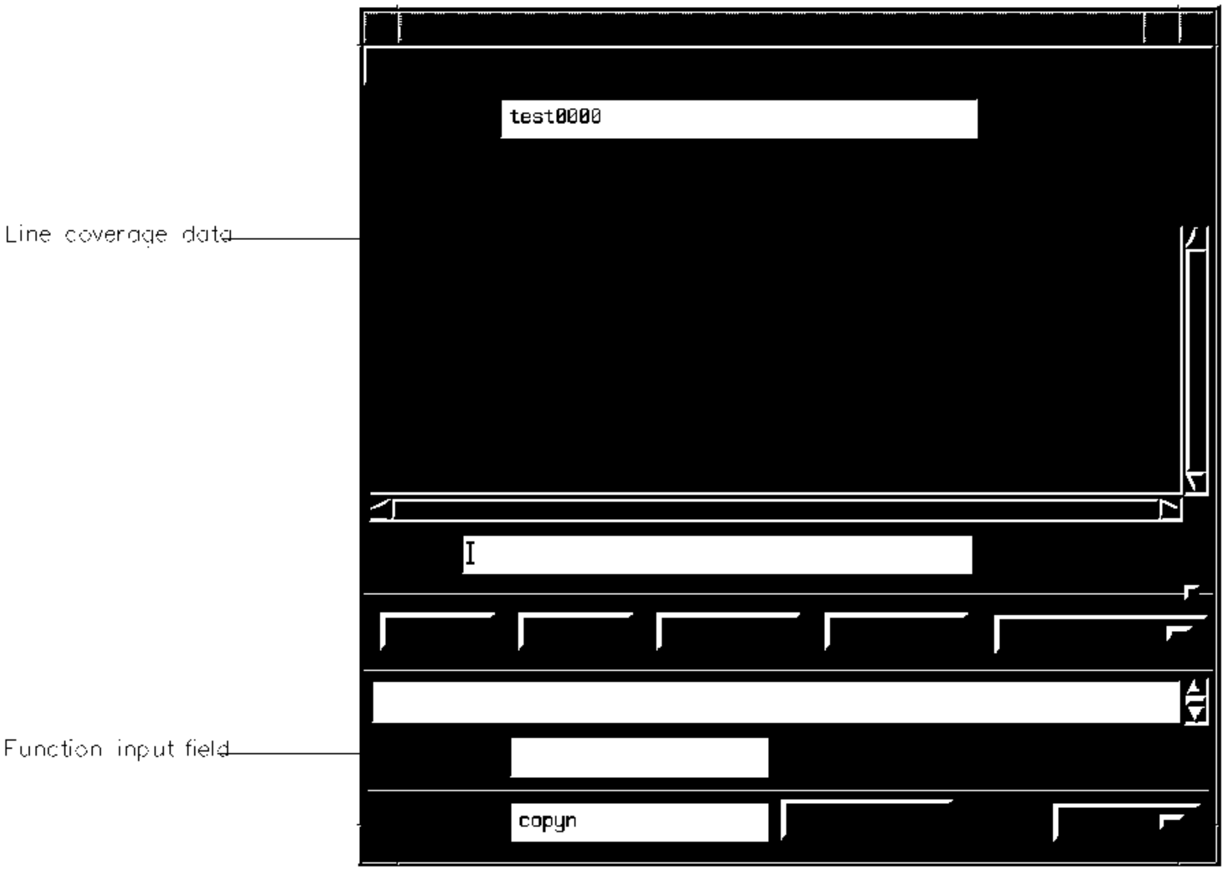
Ver allows you to specify the version of the program that was instrumented. The latest version is used by default.

Executable displays the executable associated with the current coverage data. You can switch to a different executable by entering it directly in this field. You can also switch executables by clicking the *Executable List* button, selecting from the list in the dialog box, and clicking *Apply* in the dialog box.

Figure 26–25 "List Instrumentation" Example

"List Line Coverage"

lists the coverage for each function for native source lines. Entering a function in the *Func Name* field displays the coverage results limited to that function only in the display area. See **Figure 26–26** .



test0000

copyn

Line coverage data

Function input field

Figure 26–26 "List Line Coverage" Example

"Describe Test"

describes the details of the test, test set, or test group. When working with test sets and test groups, it is useful to select the *Recursive List* toggle, because it describes the details for all subtests. See **Figure 26–27** .

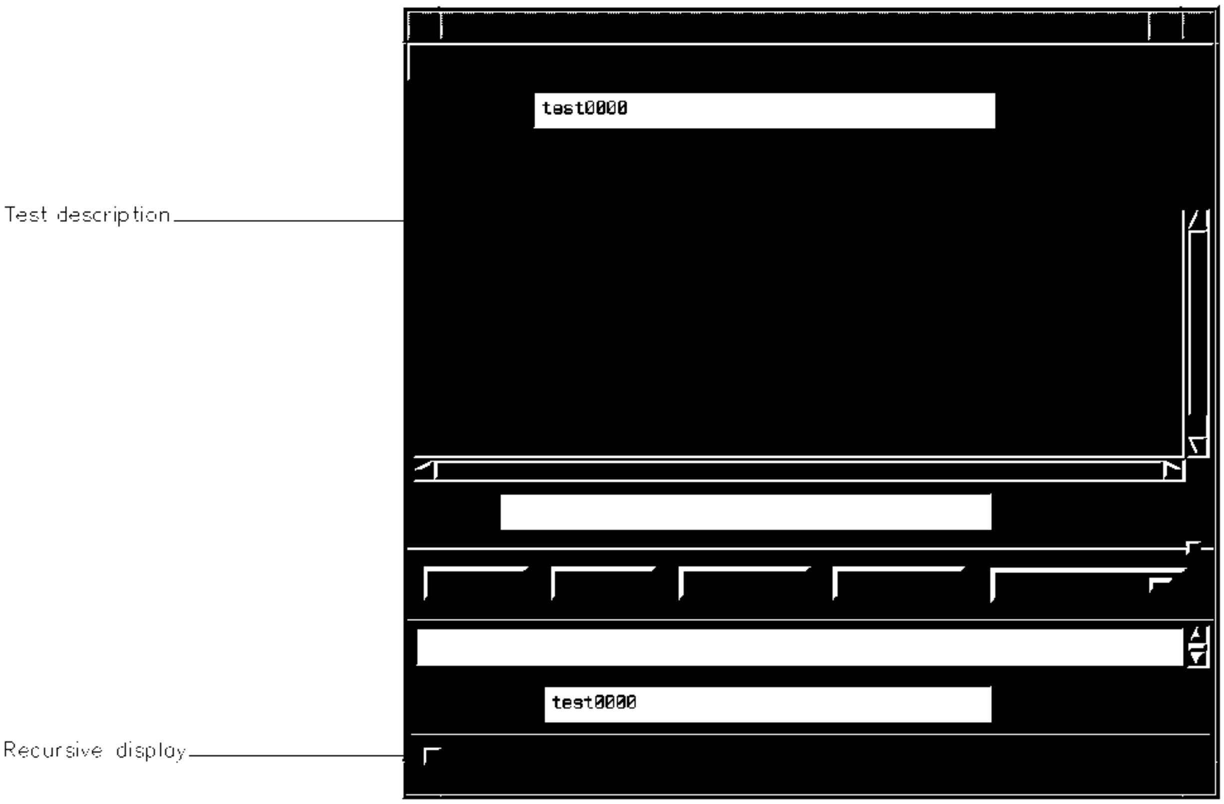
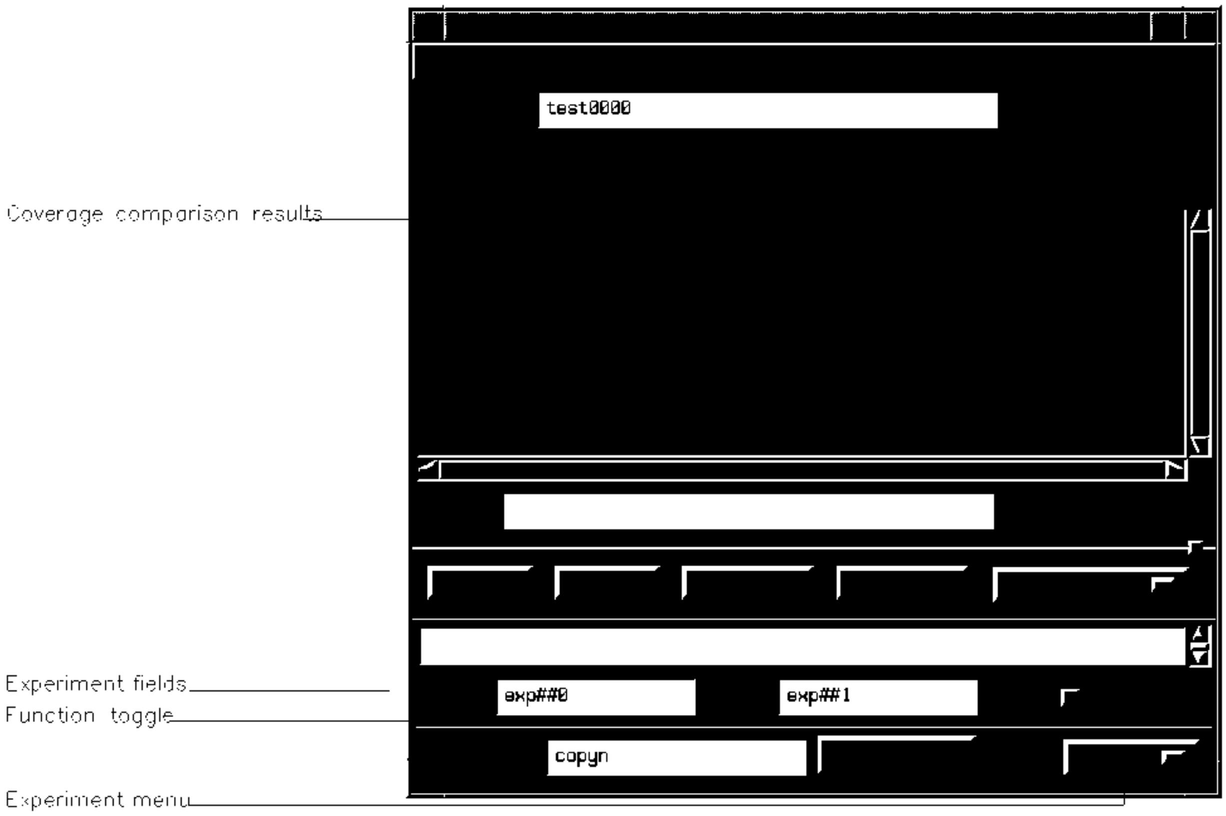


Figure 26–27 "Describe Test" Example

"Compare Test"

shows the difference in coverage for the same test applied to different versions of the same program. To perform a comparison, you need to select "Compare Test" from the Queries menu, enter experiment directories in the experiment fields, and click *Apply* or press **`<return>`**. The experiments are entered in the form *exp##<n>* if in the same test or in the form *test<nnnn>/exp##<n>* when comparing the results of different tests. See **Figure 26–28** .



test0000

exp##0

exp##1

copyn

Figure 26–28 "Compare Test" Example— Coverage Differences

The comparison data displays in the coverage display area. The basic types of coverage display in the *Coverages* column. *Result 1* and *Result 2* display the results of the experiments specified in the *Expt1* and *Expt2* fields, respectively. Results are shown as the counts followed by the coverage percentage in parentheses. The values in the *Result 2* column are subtracted from those in *Result 1* and the differences are shown in the *Differences* column. If you want to view the available experiments, click the *Expt*: menu.

You can also compare the differences in function coverage by clicking the *Diff Functions* toggle. **Figure 26–29** shows a typical function difference example.

Differences column

Function toggle

test0000

exp##0

exp##1

copyn

Figure 26–29 "Compare Test" Example — Function Differences

Admin Menu Operations

The Admin menu is shown in **Figure 26–30** .

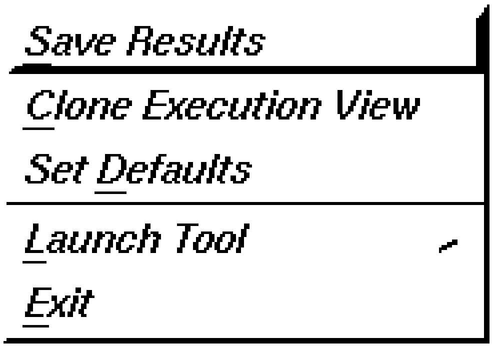


Figure 26–30 Admin Menu

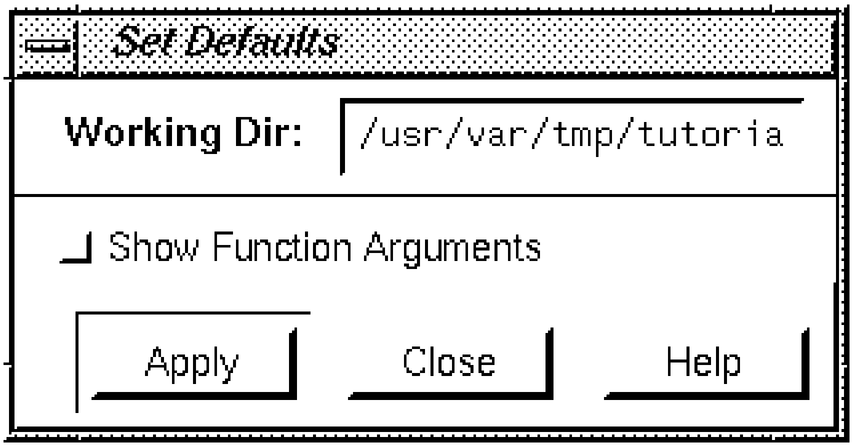
The Admin menu provides these selections:

"Save Results"
brings up the standard CASEVision File Browser dialog box so that you can specify a file in which to save the results.

"Clone Execution View"
displays an Execution View window. Use this if you have closed the initial Execution View window and need a new one. (You need this window to see the results of "Run

"Set Defaults"

allows you to change the working directory for work on tests in other directories. Also, you can select whether or not to show function arguments. This is useful when distinguishing functions that have the same name but different arguments (for example, C++ constructors and overloaded functions). See **Figure 26–31** .



Working Dir: /usr/var/tmp/tutoria

☐ Show Function Arguments

Figure 26–31 "Set Defaults" Dialog Box

"Launch Tool"

The Launch Tool submenu contains commands for launching other WorkShop applications (see **Figure 26–32**). You can launch the following applications from the menu:

- The Build Manager (see Chapter 1, "Using the Build Manager," in the *CASEVision/WorkShop User's Guide*)
- The WorkShop Debugger (see Chapter 8, "Getting Started with the WorkShop Debugger," in the *CASEVision/WorkShop User's Guide*)
- The Parallel Analyzer (see the *CASEVision/WorkShopProMPF User's Guide*)
- The Performance Analyzer (see Chapter 18, "Introduction to the Performance Analyzer," in the *CASEVision/WorkShop User's Guide*)
- The Static Analyzer (see Chapter 2, "Introduction to the WorkShop Static Analyzer," in the *CASEVision/WorkShop User's Guide*)

If any of these tools is not installed on your system, the corresponding menu item will be grayed out.

"Exit" closes all Tester windows.

Appendix A

Graphical Views in the CASEVision Environment

Many tools in the CASEVision environment provide graphical views. The CASEVision 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.

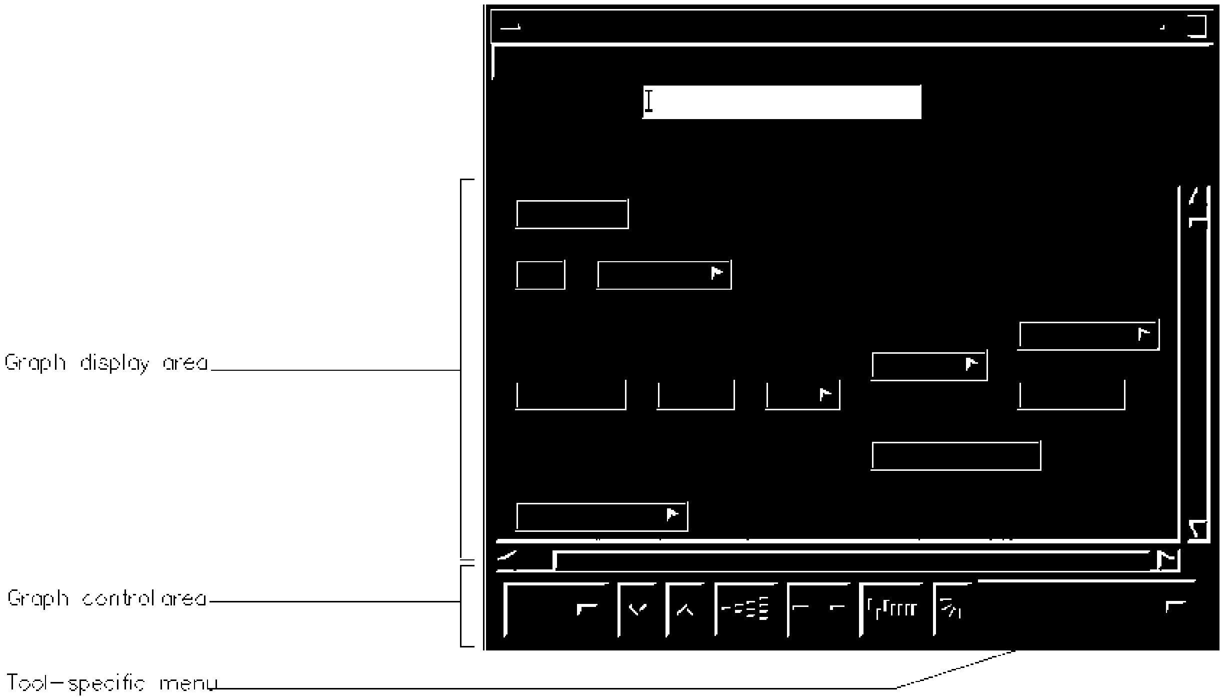


Figure A–1 Typical Graphical View

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** .

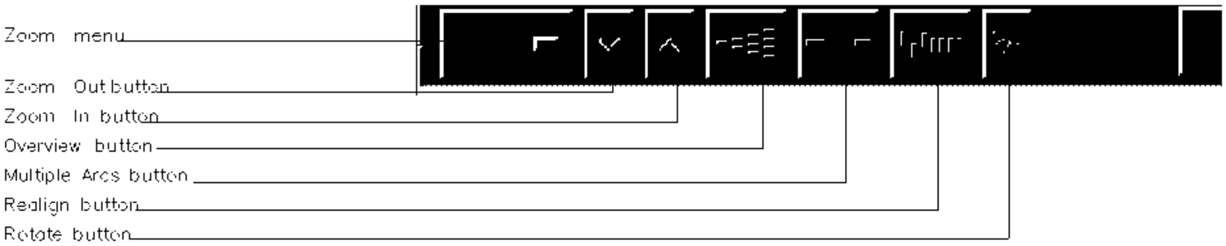


Figure A–2 Graph Display Controls

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.

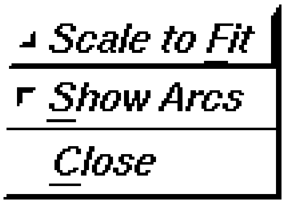


Figure A–3 Admin Menu in the Overview Window

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.

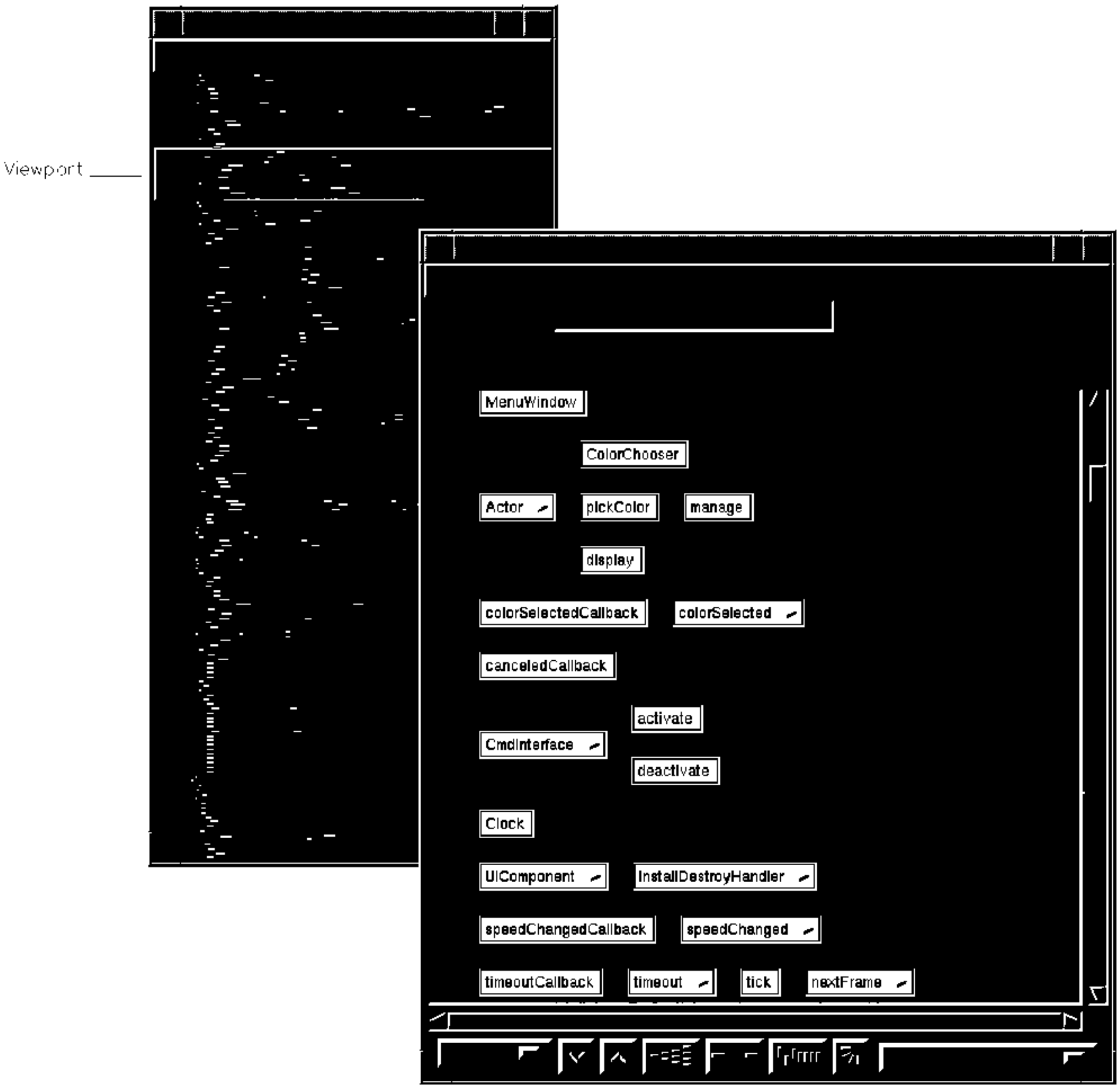


Figure A–4 Overview Window with Resulting Graph

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** .

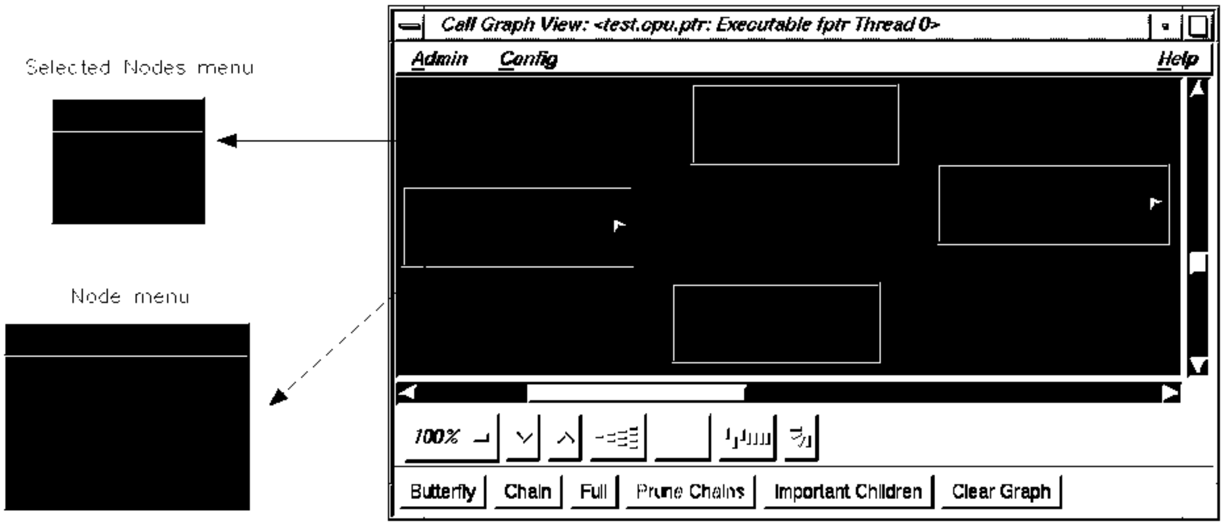


Figure A–5 Node Pop–up Menus

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 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 WorkShop"**
- **"Changing X Resources"**

Customizing Within WorkShop

If you want to change the appearance of the 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.

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.

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.