ProDev™ WorkShop:
Static Analyzer User's Guide

007–2580–004

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

Record of Revision

Version	Description
1.0	1991 Original Printing.
2.7	June 1998 Revised for the ProDev WorkShop 2.7 release.
2.8	March 1999 Revised for the ProDev WorkShop 2.8 release.
2.8	August 1999 Document released under new online and print format.
004	November 2001 Revised for ProDev WorkShop 2.9.1 release, including new online and print format.

Contents

Figures

Tables

Procedures

About This Guide

This publication documents the ProDev WorkShop Static Analyzer and Browser for release 2.9.1 running on IRIX systems. The Static Analyzer and Browser help you view and understand the structure of a program and relationships such as call trees, function lists, class hierarchies, and file dependencies.

This manual contains the following chapters:

- Chapter 1, "Introduction to the WorkShop Static Analyzer", page 1, describes the Static Analyzer, which is the WorkShop tool for examining the structure of a program's source code and the relationships between its parts, such as files, functions, and variables.
- Chapter 2, "Tutorials for the Static Analyzer", page 5, provides a sample session to introduce you to some major features in the Static Analyzer.
- Chapter 3, "Creating a Fileset and Generating a Database", page 21, describes the fileset concept. A fileset is a file that contains files you specify for inclusion in the analysis. You also specify whether a file is to be analyzed by the faster scanner mode or the slower, more thorough parser mode.
- Chapter 4, "Queries", page 37, describes how you perform queries using the Static Analyzer.
- Chapter 5, "Views", page 51, describes the text and graphical views that the Static Analyzer uses to present its data.
- Chapter 6, "Working on Large Programming Projects", page 69, presents techniques for applying the Static Analyzer to large projects.
- Chapter 7, "Getting Started with the Browser", page 75, tells you how to start the Browser and describes some of the features common to both the C++ and Ada versions of Browser View.
- Chapter 8, "Browser Tutorial for C++", page 81, provides a short tutorial highlighting the C++ features of Browser View.
- Chapter 9, "Browser Tutorial for Ada", page 97, provides a short tutorial highlighting the Ada features of Browser View.
- Chapter 10, "The Browser Reference", page 107, describes all of the Browser windows, menus, and other features in detail.

Related Publications

The following documents contain additional information that may be helpful:

- *C++ Programmer's Guide*
- *C Language Reference Manual*
- *MIPSpro Fortran 77 Programmer's Guide*
- *MIPSpro Fortran 77 Language Reference Manual*
- *MIPSpro Fortran Language Reference Manual, Volume 1*
- *MIPSpro Fortran Language Reference Manual, Volume 2*
- *MIPSpro Fortran Language Reference Manual, Volume 3*
- *MIPSpro Fortran 90 Commands and Directives Reference Manual*
- *ProDev Workshop: Debugger User's Guide*
- *ProDev Workshop: Performance Analyzer User's Guide*
- *ProDev Workshop: Tester User's Guide*
- *ProDev WorkShop: ProMP User's Guide*
- *ProDev WorkShop: Overview*

Obtaining Publications

To obtain SGI documentation, go to the SGI Technical Publications Library at: `http://techpubs.sgi.com`.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
(1)	User commands

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

Reader Comments

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

You can contact us in any of the following ways:

- Send e-mail to the following address:

  `techpubs@sgi.com`

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

  `http://techpubs.sgi.com`

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

- Send mail to the following address:

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

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

We value your comments and will respond to them promptly.

Chapter 1

Introduction to the WorkShop Static Analyzer

This chapter describes the Static Analyzer, which is the WorkShop tool for examining the structure of a program's source code and the relationships between its parts, such as files, functions, and variables.

Many software projects today contain massive amounts of code that may or may not compile, have few or no comments, and are written by programmers unfamiliar with the original code. The ProDev WorkShop Static Analyzer helps solve problems like these. With the Static Analyzer, you can analyze source code written in C, C++, Fortran 77, Fortran 90, or Ada 95.

The Static Analyzer shows you code structure, including how functions within programs call each other, where and how variables are defined, how files depend on each other, where you can find macros, and other structural details to help you understand the code. It displays answers in text or easily understood graphic form. Because the Static Analyzer is interactive, you can quickly locate the portion of 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 readily trace how a proposed change to one element will affect related elements.

The following topics are covered in this chapter:

How the Static Analyzer Works

The Static Analyzer is basically a database program that reads through one or more source code files and creates a database that includes functions, macros, variables, files, and object-oriented elements for C, C++, and Ada 95 programming languages. The database also includes the interconnections between the elements—which functions call which other functions, which files include which other files, and so on.

Note: Limited support for Fortran 90 is provided for the MIPSPro Fortran 90 compiler, beginning with version 7.1.

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. Scanner mode does not require that your code compile.
- 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. If a program cannot compile, parser mode will not work and you must use scanner mode.

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 the main window). If you have used the UNIX `grep`(1) command, you will 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 more sophisticated queries that follow connections between the following elements of source code: function calls, file includes, class parenthood, and other similar relationships.

When making queries, try not to request too much data. Overly general queries (for example, a query that asks for all functions defined in millions of lines of source code) often return extensive results that are difficult to comprehend. 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 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.

Steps in Static Analysis

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

1. Decide which files to include in your static analysis.

   It is good practice to narrow down the set of files to be analyzed as much as possible. Large static analysis databases are not only difficult to navigate through, but are time-consuming to build. You specify the files to be used in a special file called a *fileset*.

2. Choose how the files will be analyzed: parser mode, scanner mode, a combination, or different modes in multiple passes.

   Scanner mode is good for determining the general structure of a program. It is most appropriate when you are working on uncompilable code, analyzing large filesets, or performing preliminary analysis. 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.

3. 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 either the **Rescan** or the **Force Scan** option from the **Admin** menu of the Static Analyzer main window.

   Generally, you can invoke parser mode 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. For more information on this approach, see "Using the Compiler to Create a Static Analysis Database", page 17.

4. 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, as a list of items and their source lines or graphically as a tree showing relationships between items.

   If you are programming in C++, or Ada you can make object-oriented queries by bringing up the class browser in the **Class View** window. This window lets you view structural and relational information.

5. Once you have isolated an area for analysis, you can edit the source code from the Static Analyzer. Double-clicking an element brings up the corresponding source code in the **Source View** window.

Tutorials for the Static Analyzer

This chapter shows how you might use the Static Analyzer in a typical session. It does not go into full detail, but it does explain the fundamental concepts you will need to use the Static Analyzer. It lists related commands and controls after each tutorial so you can experiment on your own.

This chapter discusses the following topics:

Applying the Static Analyzer to Scanned Files

In this session, 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 command:

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

   This directory contains the C++ source code files for the demo program `bounce`.

2. Use the `ls`(1) command to list the directory's contents to see if the file `cvstatic.fileset` already 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 by entering the following command:

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

   Whenever you run the Static Analyzer, it checks the directory where you invoked it for the `cvstatic.fileset` file and uses the content of that file as its fileset. If it does not find the `cvstatic.fileset` file, it creates and saves its own fileset containing the expression `*.[c|C|f|F]` so that all possible C files (`.c`), Fortran 77 files (`.f`), Fortran 90 (`.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 do not want to use the default fileset, you can create your own or modify the default fileset using the **Edit Fileset** selection on the **Admin** menu. You can also create your own `cvstatic.fileset` file by hand; instructions are found in "Creating a Fileset Manually ", page 28.

3. Start the Static Analyzer by entering the following command:

```
% cvstatic &
```

The **Static Analyzer** window appears (as shown in Figure 2-1).

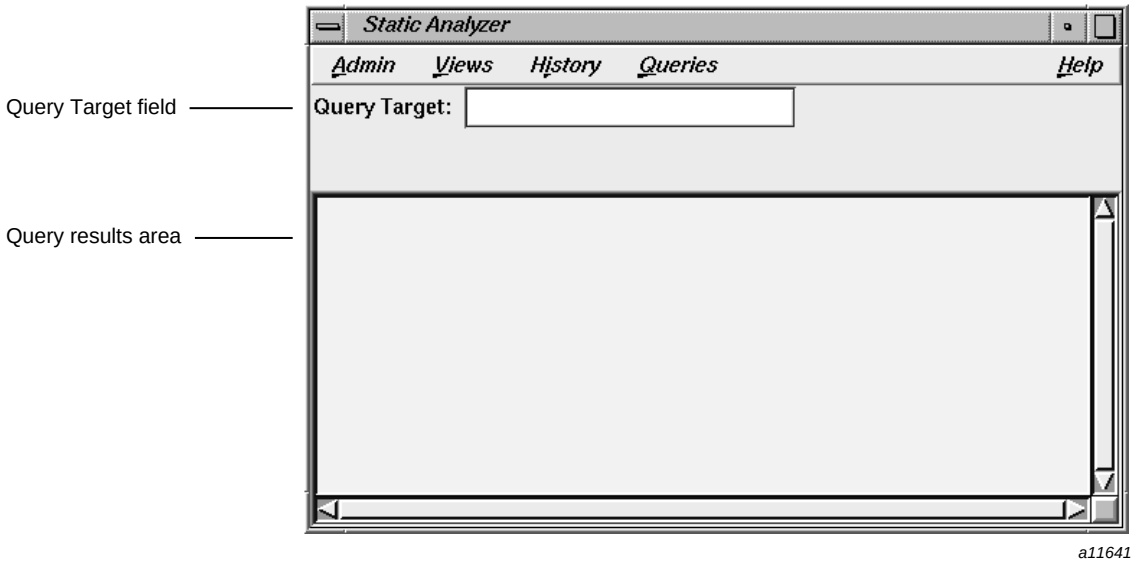

Figure 2-1 The **Static Analyzer** Window

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

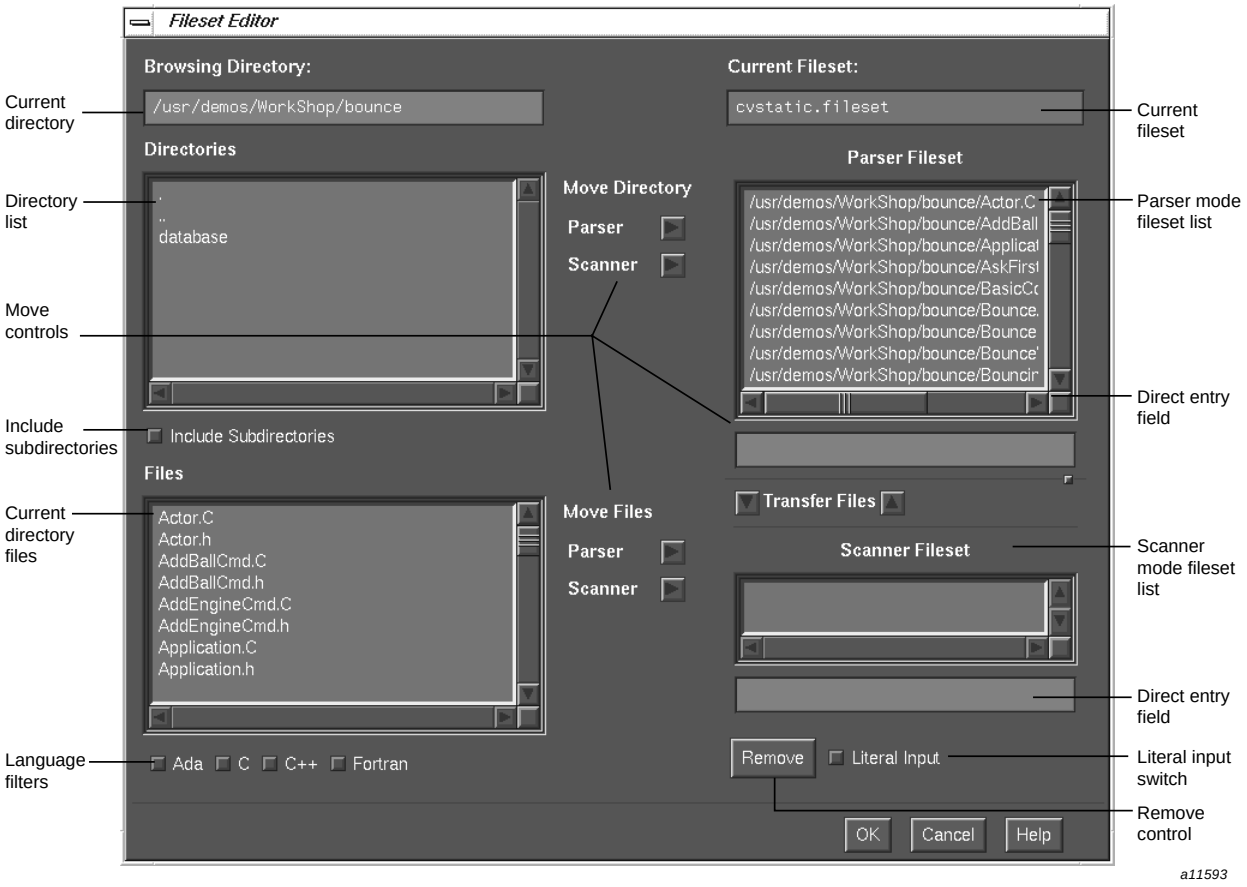

Figure 2-2 The **Fileset Editor** Window

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

5. Select the expression `*.[c|C|f|F]` 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 **Scanner** button from the **Move Files** set of buttons.

   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.

   After you have 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 (using scanner mode). This occurs before 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.xref`; the accompanying files are named `cvstatic.index` and `cvstatic.posting.` These files are stored in the same directory as the fileset with which they are 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 selected files for parsing, the files will have compiler flags following their path names.

10. Click the **Queries** menu to open it.

    To query the database, choose a command from the **Queries** menu. You'll find the commands grouped in submenus according to the type of query (see Figure 2-3).

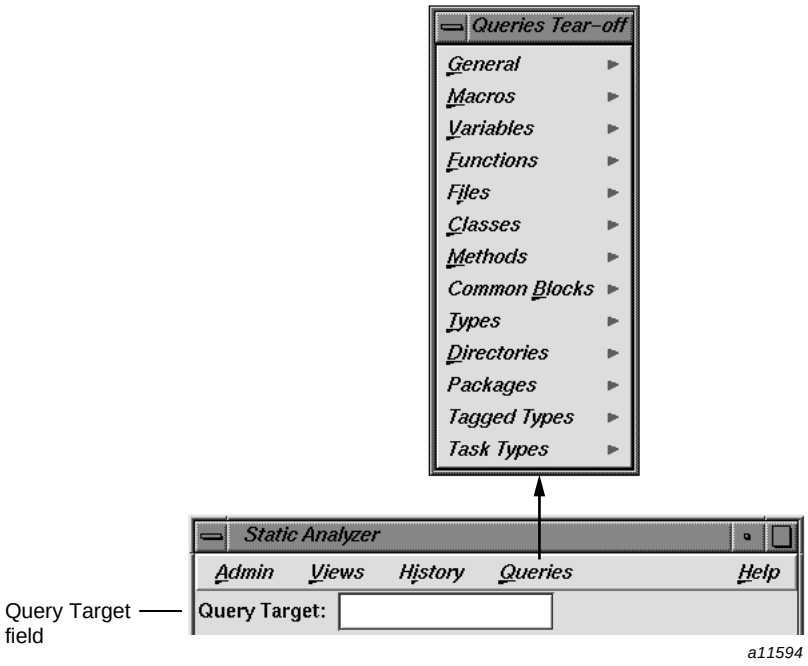

Figure 2-3 Static Analyzer **Queries** Menu and **Query Target** Field

The **Query** submenus let you perform the following types of different searches:

- The **General** submenu searches for text strings, regular expressions, and symbols.
- The **Macros** submenu searches for locations of macro definitions and places where macros are used.
- The **Variables** submenu searches for global and local variables and shows where they are defined and who references and sets the variables.
- The **Functions** submenu searches for functions, shows where they are defined, and shows who calls them and whom they in turn call.
- The **Files** submenu searches for files in the fileset (including headers and libraries) and shows which files are included by which other files.

- The **Classes** submenu searches C and C++ files for classes and shows where they are defined. It also shows subclass and superclass relationships and lists the methods defined within classes.
- The **Methods** submenu searches C/C++ files for methods and shows where they are defined and declared.
- The **Common Blocks** submenu searches Fortran 77 and Fortran 90 files for common blocks.
- The **Types** submenu searches C and C++ files for type information.
- The **Directories** submenu lets you list directories or the files in a directory.
- The **Packages** submenu lets you search for Ada packages.
- The **Tagged Types** submenu lets you search for Ada tagged types.
- The **Task Types** submenu lets you search for Ada task types.

To start a query, choose the type of query you want from the **Queries** menu. The Static Analyzer 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, enter the target text in the **Query Target** field above the query results area (as shown in Figure 2-1, page 6).

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

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

    The Static Analyzer builds its cross-reference database and notifies you that it is doing so. When it is finished, the Static Analyzer displays a list of all functions found in the fileset (as shown in Figure 2-4), 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.

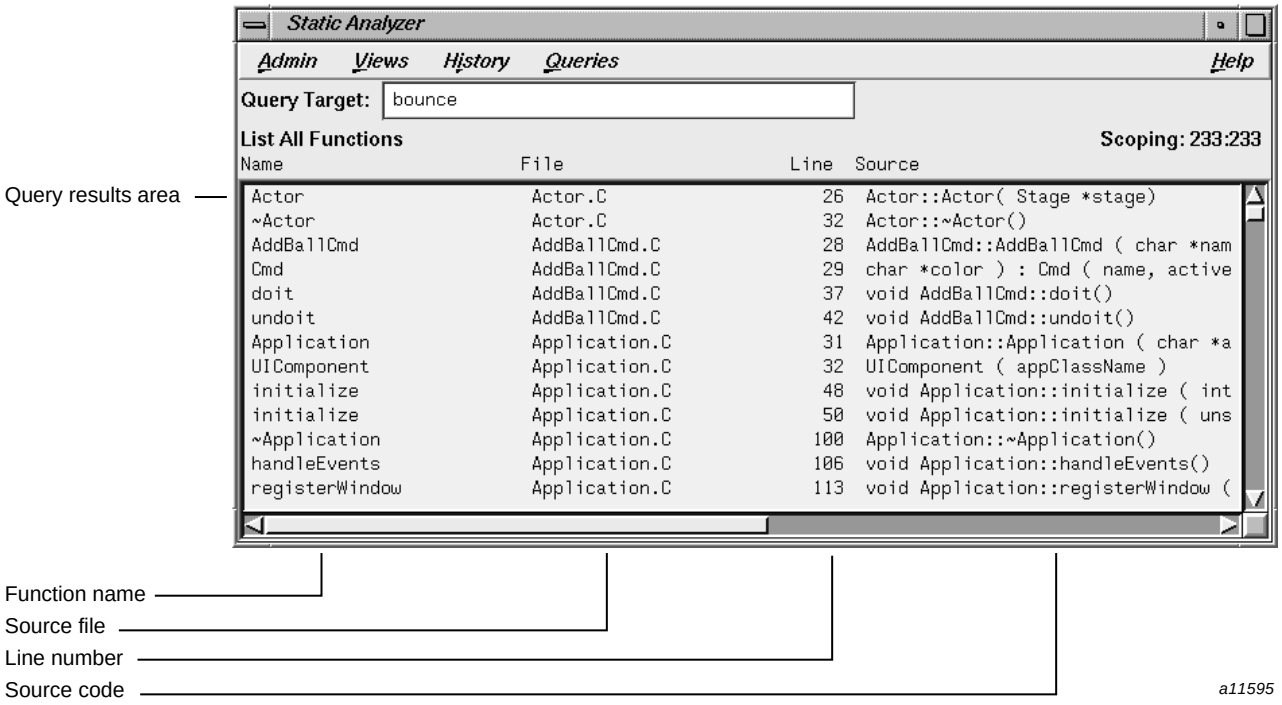

Figure 2-4 The Results of a **List Functions** Query

The Static Analyzer returns the results of all queries in the 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 want to determine what functions a particular function calls, click

on the function name to put it into the **Query Target** field and then choose the **Who Is Called By** selection 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, 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. 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.

    In the next steps, you are going to search for any occurrences of the text string `buffer` that might lead 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. Click on the **History** menu to open it.

    It displays the queries you have made so far.

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.

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

19. Choose **Close** from the **Source View File** menu to close it.

20. Choose **Exit** from the Static Analyzer **Admin** menu to end this tutorial.

Applying the Static Analyzer to Parsed C++ Files

In this session, you will create a fileset for the `bounce` demo program by 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 command:

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

2. Use the `ls`(1) command to list the directory's contents to see if the file `cvstatic.fileset` already 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 by entering the following command:

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

   Whenever you run the Static Analyzer, it checks the directory where you invoked it for the `cvstatic.fileset` file and uses the content of that file as its fileset. If it does not find the `cvstatic.fileset` file, it creates and saves its own fileset containing the expression `*.[c|C|f|F]` 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 do not want to use the default fileset, you can create your own or modify the default fileset using the **Edit Fileset** selection on the **Admin** menu. You can also create your own `cvstatic.fileset` file by hand; instructions are found in "Creating a Fileset Manually ", page 28.

3. Start the Static Analyzer by entering the following command:

   ```
   % 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. Choose **Edit Fileset** from the **Admin** menu.

   This will allow you to use parser mode through the **Fileset Editor** window.

5. Select the expression `*.[c|C|f|F]` in both the **Parser Fileset** and **Scanner Fileset** list fields (if it appears), and click the **Remove** button.

6. Select the `BouncingBall.C` file in the **File** list at the lower left of the **Fileset Editor** window and click the **Parser** button from the **Move Files** set of buttons to transfer the file to the **Parser Fileset** list.

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

7. Click the **OK** button at the bottom of the **Fileset Editor** window to save the new fileset. From a shell window, enter the following command to display the contents of `cvstatic.fileset`:

   ```
   % cat cvstatic.fileset
   ```

   After you have 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 (using scanner mode). This occurs before 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.xref`; the accompanying files are named `cvstatic.index` and `cvstatic.posting.` These files are stored in the same directory as the fileset with which they are 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.

8. Choose **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. Choose **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

you 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** that displays C/C++ classes and their hierarchy and **File Dependency View** that displays 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 do not 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. Click 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 2-5.

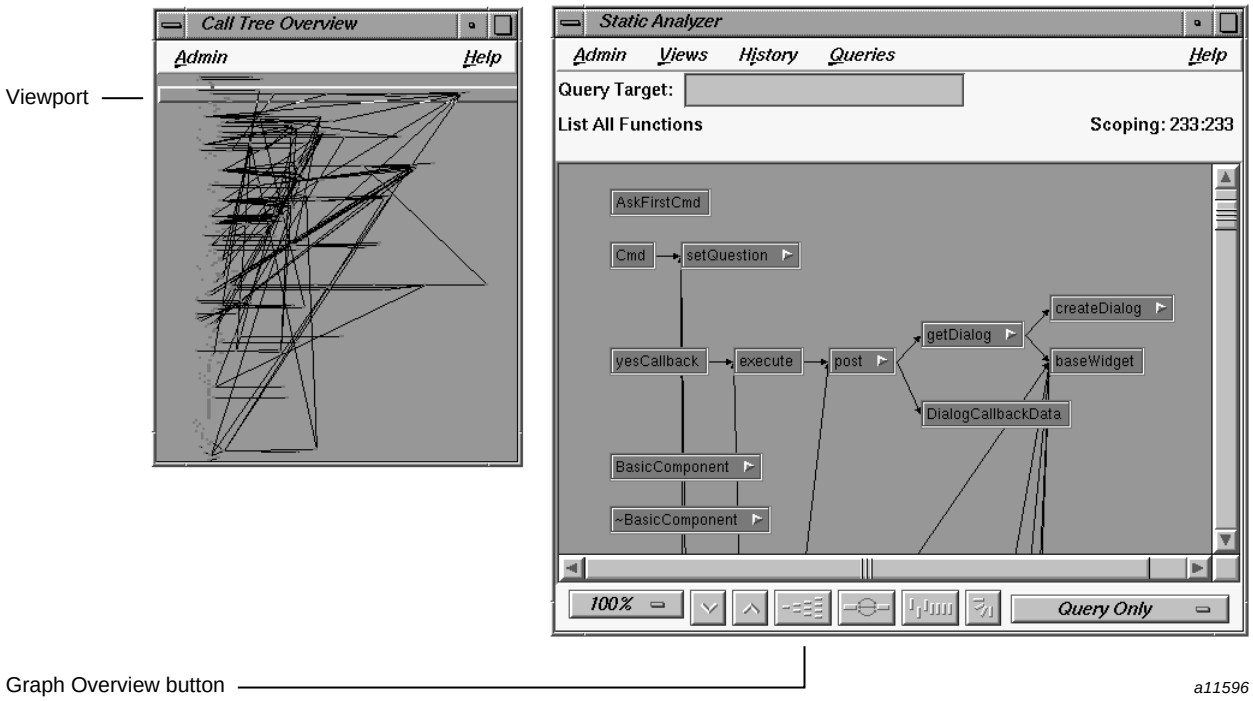

Figure 2-5 Typical Static Analyzer Call Tree

11. 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. Finish 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. Choose **Close** from the **Admin** menu in the **Call Tree Overview** window to close it.

14. Type **colorSelected** in the **Query Target** field and choose **Who Calls?** from the **Functions** submenu in the **Queries** menu. This reduces the graph to three nodes.
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 arrow at the right of the **colorSelected** node indicates that it has undisplayed child nodes. Therefore, **Show Immediate Children** is enabled. Because 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 providing additional selections.
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 in the *ProDev WorkShop: Overview*. 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 files generated by the Static Analyzer from the directory by entering the following command:

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

Using the Compiler to Create a Static Analysis Database

In this session, you will create a static analysis database by using parser mode.

Caution: The steps listed in this section will not work in all cases. You should be aware of the following limitations:

- Using the C compiler with the `-sa` flag does not work for `o32` programs.
- Using the C++ compiler with the `-sa` flag for `o32` programs may produce error messages and possibily a core dump file.
- Templates are not supported when using the C++ compiler with the `-sa` flag for `n32` programs.

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

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

   We will analyze the `bounce` demonstration program.

2. Create a new subdirectory, by entering the following command:

   ```
   % mkdir staticdir
   ```

   This creates the subdirectory in which you 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 type **`initcvdb.sh`**.

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

4. 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 by entering the following command:

   ```
   -sa,staticdir -nocode
   ```

   The `-sa` flag 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 used. The `-nocode` flag saves time by telling the compiler not to create object code.

Caution: The `-sa` flag should be added only to a Makefile that does a sequential build. Adding the `-sa` flag to a Makefile that does a parallel build causes multiple copies of `cc` or `CC` to try to write to the same database. However, the database accepts only one writer at a time.

5. Enter the following command:

   ```
   % make -k
   ```

   This runs the compiler as you have specified in the Makefile. The `-k` option instructs the `make` command to abandon work on the current entry if it fails, but to continue on other branches that do not depend on the failed entry. This may take a while. Running parser mode performs all major operations of compiling, short of creating the object code.

6. Go to the `staticdir` subdirectory and enter the following command:

   ```
   % 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 shown in the previous tutorials.

Other Static Analyzer Features

You can find complete information about querying in Chapter 4, "Queries". To explore on your own, try these commands in the **Admin** menu that 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 regardless of whether they have changed.
- **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. Not only will you have to wait hours for your results, but the results will probably be so complicated that they will be meaningless to you.

Creating a Fileset and Generating a Database

This chapter describes the fileset concept. A *fileset* is a file that contains the names of the files you want included in the analysis. You also specify whether these files are to be analyzed by the faster scanner mode or the slower, more thorough, parser mode.

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

Fileset Specifications

A Static Analyzer *fileset* is a single file used to specify the source code files to be analyzed. There are several 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 `*.[c|C|f|F]`
- Letting `cvstatic` do it automatically at startup by designating an executable file
- Using the compiler to create a fileset (and database) by adding the `-sa,`*dbdirectory* option to your Makefile

Caution: Information in this section will not work in all cases. You should be aware of the following limitations:

- For C and C++ files, the only set of compiler options that works is the following, where *cvstatic.fileset* is the name of the fileset if you do not use `-sa_fs` (`cc -o32` rejects the `-sa` option):

  `CC -o32 -sa [-sa_fs |` *cvstatic.fileset*`]`

- `CC -n32 -sa` and `cc -n32 -sa` both produce a fileset but do not produce a database.
- The `-sa` flag should be added only to a Makefile that does a sequential build. Adding the `-sa` flag to a Makefile that does a parallel build causes multiple copies of `cc` or `CC` to try to write to the same database. However, the database accepts only one writer at a time.

A fileset is a regular ASCII file with a format of one entry per line, each line is separated from the next by a carriage return. The fileset always begins with the following line:

```
-cvstatic
```

The other entries can be a mixture of the following entities:

- Regular expressions
- File names
- Included directories preceded by the `-I` dwsignator

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", page 30).

Using Regular Expressions

Each line in the fileset can use shell expansion characters, a wild card system in standard use for specifying file names 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:

```
*.[c|C|f|F]
```

The asterisk specifies any number of characters (zero or greater) before a period, and the bracketed set of characters specifies any of following single characters: `c`, `C`, `f`, or `F`, after the period. The result is that the line specifies any file names in the current directory that use one of these extensions.

Note: If you are analyzing Ada files, then the default expression `*.[c|C|f|F]` is not appropriate. You may wish to substitute an expression like `*.adb` for Ada bodies or `*.adb` for Ada specifications.

Do not confuse the shell expansion characters used here with the regular expressions used in the **Fileset Selection Browser** window; they are different systems.

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 file names, 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 a space) by the pathname. For example:

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

This pathname listed in a fileset requests that 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 want, but by convention you should use the `.fileset` extension.

Defining Macros 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 does not 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. Macros are specified at the end of a fileset by appending a line in the following format for each preprocessor symbol you want to define:

`-D` *symbolname*

or

`-D` *symbolname=value*

For example, to set the macros `DEBUG` and `BUFFERSIZE`, you would append two lines like the following to the end of the fileset:

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

In a similar manner, `-U` undefines macros. These symbol definitions are used for processing all files in the fileset.

Note: Using the `-preprocess` option increases the scanning time tremendously (scanner mode only). Use it only when absolutely necessary.

Using the Default Fileset

When you start the Static Analyzer in a directory that does not 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:

```
*.[c|C|f|F]
```

This line specifies any C, C++, Fortran 77, or Fortran 90 files in the working directory.

Note: This line assumes that C++ files have a `.C` extension, which may not be the case for all C++ files because there is not 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 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 cannot change either the fileset or the working directory here; to do so, 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", page 29.) The top list area shows files in the fileset to be parsed, and the lower area shows 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 file names.

Adding Lines to the Fileset Contents List

Both fileset list areas have 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 a literal filename 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 toggle button just below the line entry area. When this button is on, the Fileset Editor treats all 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.**Literal Input**

Removing Lines from the Fileset Lists

To remove a line from a fileset list, click on it to select it and then 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, drag the cursor over a range of files or hold down the `Control` key while clicking, then click the **Remove** button.

Browsing for Fileset Contents

You can use the following 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. You can double-click on 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. 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 are browsing. You can use it to type an absolute pathname to a new directory. First, put the pointer in the area to type and then press `Enter`. The contents of the **Directories** list changes 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 language filter buttons below the list. 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 Ada, 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.
- The **Ada** button restricts files shown to those with `.adb` and `.ads` extensions.

  By default when you click on the **Ada** button, only those Ada files with `.adb` extension are displayed. If you want to view files with both `.adb` and `.ads` extensions, set the `*suffixSource` resource as shown in the following sample setting:

  ```
  *suffixSource: C++.c++ C++.C C++.cxx Fortran.f Ada.adb Ada.ads
  ```

  You cannot override only the Ada part of this resource. You must list all the languages you might want to browse by using the Static Analyzer.

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

Adding File Names from Lists

If you want to add one or more file names from the **Files** list to one of the fileset lists, select the file name and 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, select the directory name (or directory names if you want more than one) in the **Directories** list, then 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 click on 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 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. You can then transfer the file to scanner mode, which is not sensitive to programming languages.

Leaving the Fileset Editor Window

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

Creating a Fileset Manually

You can create a fileset, 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`(1) command useful for returning all specified filenames within a directory tree. For example, the following command creates a fileset of all Fortran 77 files (those with a `.f` extension) found within the current directory and all of its subdirectories:

```
% find . -name ‘‘*.f’’ -print > cvstatic.fileset
```

You can pipe the output of the `find`(1) command through filtering commands such as `sed`(1) to further modify the fileset created. For example, the following command finds C files within a directory tree and strips out any `.c` files left by the C++ compiler:

```
% find . -name "*.c" -print | sed’/\.\.c/d > cvstatic.fileset
```

Using Command-Line Options to Create and Use a Fileset

The Static Analyzer provides the following special options when you invoke `cvstatic` from the command line:

- The `-executable` option followed by the file name of an executable file instructs the Static Analyzer to create a fileset that contains the absolute pathname of every file used to compile that executable. For example, entering the following command finds C files within a directory tree and strips out any `.c` files left by the C++ compiler:

  ```
  % cvstatic -executable jello
  ```

  The executable file must not be stripped because stripped files do not contain the names of their source files. When using the `-executable` option, it is a good idea to use the Fileset Editor to exclude files with incomplete names that can occur with files compiled into `lib` using compilers prior to 4.0.1 or nonsupported languages like Assembler or Pascal. The `-executable` option requires that the executable file be built on the same system as that performing the static analysis.

- The `-fileset` option followed by the file name of a fileset instructs the Static Analyzer to start using a fileset other than `cvstatic.fileset`.

- The `-mode` flag takes the options `SCANNER` or `COMPILER` to indicate the types of files in the fileset to be used in queries. If you do not use the `-mode` flag, then scanner mode will be assumed for those files in the fileset without compiler driver specifications.

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. The following two methods are available 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 do not 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 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.

Scanner mode is based on a sophisticated pattern matcher. It works by searching for and identifying common patterns that occur in programs. Both philosophically, and in terms of the actual implementation, `cvstatic`(blank) is most closely related to the `grep`(1) command. If you expect `cvstatic` to produce the type of results that can be accomplished only with a full-compilation type of analysis, you should use the compiler-based parser mode. If you think of scanner mode as a sort of “super grep” command and use scanner mode as most programmers use the `grep` command to explore a new program, you can get a quick, high-level look at your code.

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, C++, and Fortran 77 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.

Preparing the Fileset for Parser Mode

File entries for parser mode take the following 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`.

The Static Analyzer recognizes the type of language by the file extension. 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 need to set the include paths before the Static Analyzer scans the files, so do this before performing any queries or choosing **Force Scan** from the **Admin** menu.

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** window.
- 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 `-sa[,`*databasedirectory*`]` and the `-nocode` flags. Without arguments, the `-sa` flag stores the static analysis database in the current directory. If you enter a comma (`,`) and a database directory name, the static analysis database will be stored in the specified directory. If you specify the `-nocode` flag, the database will be built without creating new object files.

Caution: Information in this section will not work in all cases. You should be aware of the following limitations:

- For C and C++ files, the only set of compiler options that works is the following, where *cvstatic.fileset* is the name of the fileset if you do not use `-sa_fs` (`cc -o32` rejects the `-sa` option):

  `CC -o32 -sa [-sa_fs |` *cvstatic.fileset*`]`

- `CC -n32 -sa` and `cc -n32 -sa` both produce a fileset but do not produce a database.
- The `-sa` flag should be added only to a Makefile that does a sequential build. Adding the `-sa` flag to a Makefile that does a parallel build causes multiple copies of `cc` or `CC` to try to write to the same database. However, the database accepts only one writer at a time.

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 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 rebuilds pieces only when they are out-of-date.

For more information on creating a database in parser mode, see "Using the Compiler to Create a Static Analysis Database", page 17.

Parser Mode Shortcuts

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

- 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.

Size Limitations

The following limitations and shortcomings are largely a consequence of the `grep`(1)-like model supported by scanner mode. Still, `cvstatic` does provide a

more powerful way to approach understanding a set of source files than using the `grep`(1) command.

When you use the Fileset Editor to add entire directories of files, you cannot enter more than 10,000 files. This limit exists to prevent someone from inadvertently starting at the root of a file system and trying to add all files. Note that there is no limitation on the number of files that can be added to the fileset when the fileset file is constructed in other ways, such as compiling source files with the `-sa` flag, or emitting a fileset from a Makefile rule.

The Static Analyzer displays a maximum of 20,000 lines of unfiltered results from a query in the **Text View** window. Larger results can, however, be saved to a file or reduced to a more manageable size by using the Results Filter.

The Static Analyzer displays no more than 5,000 functions in the **Call Tree View**, 10,000 files in the **File Dependency View**, or 10,000 classes in the **Class Tree View**. These are absolute maximum limits, and the actual limits may be much lower depending on characteristics of the graph being displayed. In particular, all graph views are displayed in a scrolled X Window System window, which is sized to accommodate the graph. The X Window System imposes a maximum size on windows that graphs cannot exceed. To get around this limitation, you can use one of the following methods:

- Use more specific queries to focus on the part of the program that is of the most interest.
- Reduce the scale used to view the graph.
- Use the Results Filter to trim query results.
- Use the **Incremental Mode** setting in the various graph views or the pop-up menus on nodes of the graph to follow a specific path through a large tree.

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 have 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 does not open the Static Analyzer's main window, however, and it quits the Static Analyzer after 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 will not work.
- `-noindex`: stops creation of the `.index` and `.posting` files. Therefore, the Static Analyzer does not create an inverted index for the cross-reference database. This speeds database creation but slows database query response.

Note: This works in scanner mode only.

Setting the 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 does not find the included file in either of these directories, it displays 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 and Flags** from the **Admin** menu to open the **Scanning Options** dialog box.

The **Include Directories** list at the top of the box lists all directories that the Static Analyzer searches 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 the `Enter` key (or click on the **Add Directory** button). The path should be relative to the directory in which `cvstatic` is running. To delete a directory, click its name in the **Include Directories** list (this puts it in the **Directories** field), then click the **Remove Directory** button. You can also add flags such as `-I` for including files, `-D` for defining macros, or `-U` for undefining macros, as described in "Defining Macros in the Fileset", page 24.

To exclude `/usr/include` from the Static Analyzer's search path, click the **No Standard Includes** button to turn on the option. Turn on this option whenever you do not 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. However, you will not be able to find data in the header files normally found in `/usr/include`.

To close the **Scanning Options** dialog box, click the **Close** button. Any directories you added to the search path are stored as part of the fileset. You will not see the directories listed if you open the **Fileset Editor** window, but you will see them if you examine the fileset file directly because 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 3-1 by choosing **Change Fileset** from the **Admin** menu.

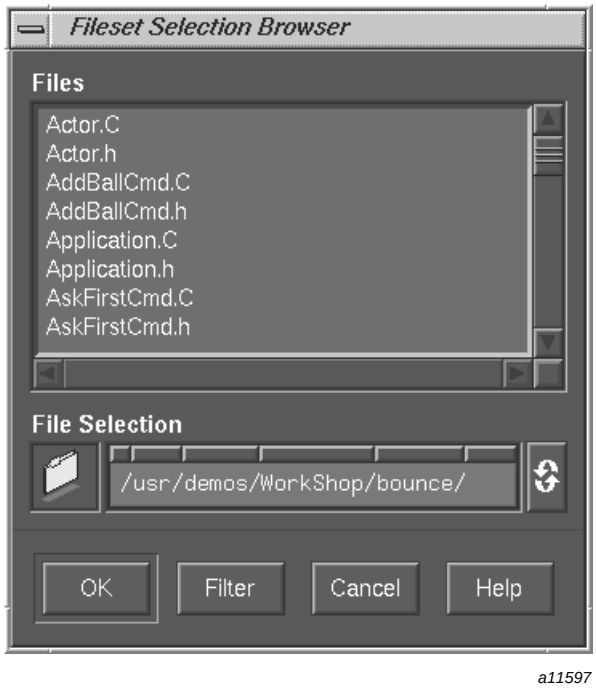

Figure 3-1 The **Fileset Selection Browser** Window

To load a new fileset, change to the directory in which the fileset is located by using the **File Selection** field (either by dragging a folder icon into it or by typing directly). Then select the fileset in the **Files** list. Once you change to a new fileset, the directory where it is 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 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.

Queries

This chapter describes how you perform queries, which ask the Static Analyzer for specific information about the source code files included in the fileset. This chapter covers the following topics:

For examples of using queries, refer to "Applying the Static Analyzer to Scanned Files", page 5, and "Applying the Static Analyzer to Parsed C++ Files", page 13.

Defining the Scope of a Query

The Static Analyzer has two types of queries: comprehensive queries (such as **List All Functions** and **List Global Symbols**) that do not require a query target and specific queries (such as **Who Is Called By?** and **List Methods In Class**) that do require a query target. Specific query selections in the **Queries** menu are grayed 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 click an element in the query results area and the Static Analyzer pastes it into the text area. For example, you can 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 matching the query parameters and the target text. You can also make a query by pressing the `Enter` key 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.

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`(1) command to construct literal strings or wild card strings. Regular expression syntax is described in the man page for `ed`(1).

If you enter target text without using any of the following special characters, the Static Analyzer reads the text as a literal string 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: Do not confuse regular expressions with the shell expressions you use to create a fileset. They are different systems.

Case Sensitivity

The Static Analyzer is case-sensitive and recognizes the difference between uppercase and lowercase characters in target text during queries. However, if you want to ignore case in target text during a query (useful for case-insensitive Fortran code), choose **General Options** from the **Admin** menu to open the **General Options** dialog box. Click the **Ignore Case In Searches** button to turn it on, then click the **Close** button to close the dialog box.

Making a Query

To make a query, choose a query type from the **Queries** menu as shown in Figure 4-1, page 39.

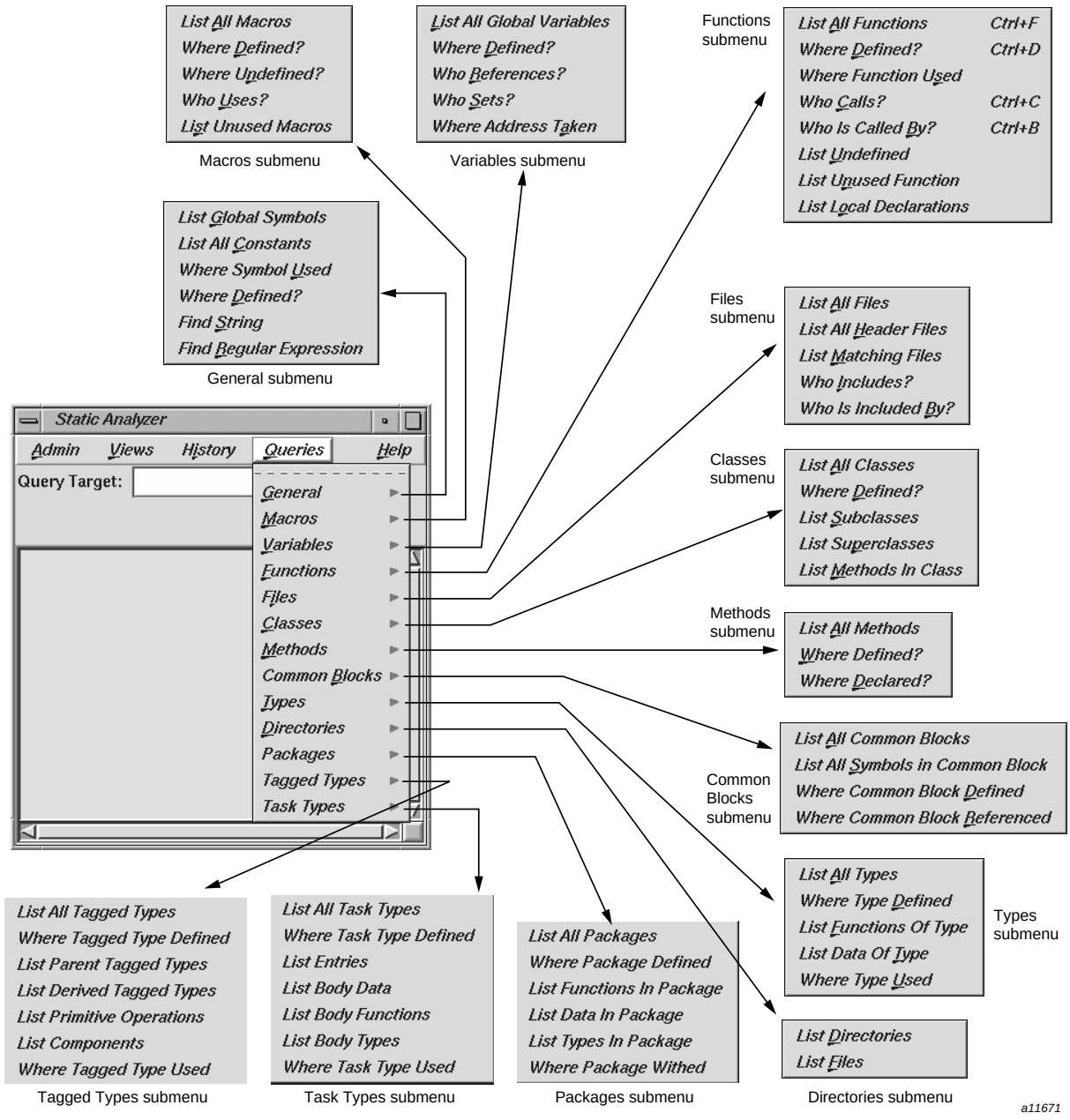

Figure 4-1 Static Analyzer **Queries** Menu with Submenus

The Static Analyzer displays the results in the query results area of the main window. The following sections describe the queries that you can make from the submenus of the **Queries** menu.

General Queries

The **General** submenu 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 such as macro names, functions, variables, and other source code elements. The following general queries are available:

- **List Global Symbols**: returns all global symbols found in the files defined by the fileset and ignores any target text. Global symbols are standard elements of code including functions, macros, variables, classes, and so forth.
- **List All Constants**: returns all constants in the source code including `enums`,named constants, and Fortran 77 parameters.
- **Where Symbol Used**: expects a symbol name in the **Query Target** field. Returns the source code locations of all references to the symbol.
- **Where Defined?**: expects a symbol name in the **Query Target** field. 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. Returns 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. Returns source code locations of all strings that match the target text.

Macro Queries

The **Macros** submenu contains queries that deal with macros. The following queries are available:

- **List All Macros**: returns all macros found in files defined by the fileset. Ignores any target text.

- **Where Defined?**: expects a macro name in the **Query Target** field. Finds all macros that match the target text and returns the source code locations where the macros are defined.
- **Where Undefined?**: expects a macro name in the **Query Target** field. Finds all macros that match the target text and returns source code locations where the macros are undefined (by using `#undef`).
- **Who Uses?**: 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 dealing with variables. In performing a variable query, you typically list variables first and then select an individual variable for further information. Figure 4-2 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 the notation of `global`, the file in which the variable is defined or declared, the line number at which it is first defined or declared, and the actual source line.

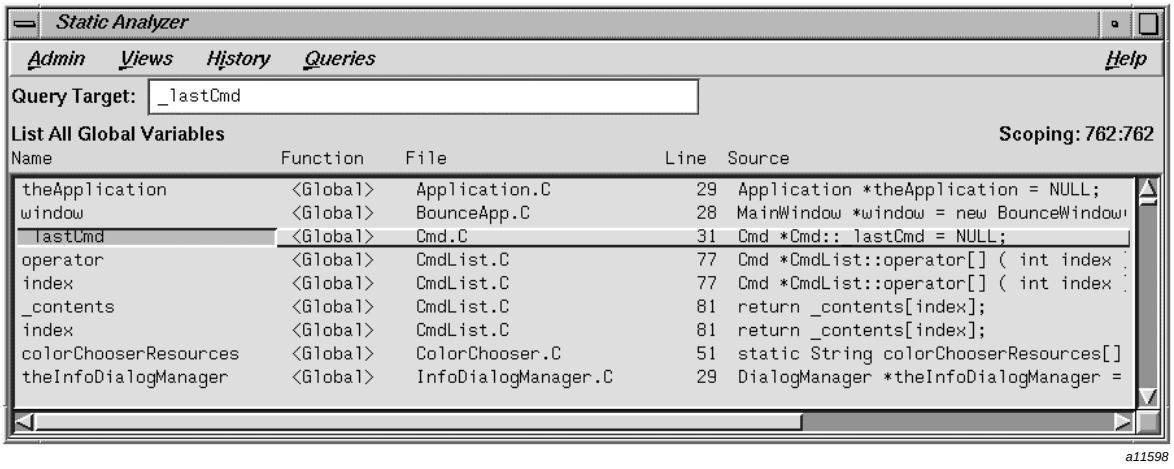

Figure 4-2 List All Global Variables Results

From the list resulting from **List All Global Variables**, you can select individual variables for specific queries. You do this by clicking the variable name. Figure 4-3 shows the results of a **Who References?** query.

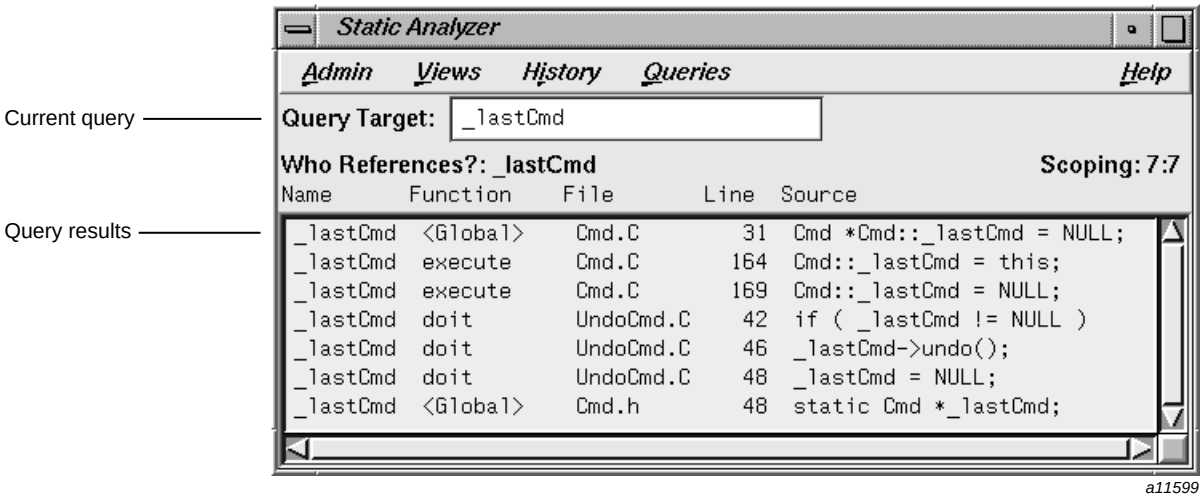

Figure 4-3 Who References? Results

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

The **Variables** submenu offers the following types of queries:

- **List All Global Variables**: returns all global variables found in files defined by the fileset. Ignores any target text.
- **Where Defined?**: finds the locations where the variable was defined.
- **Who References?**: expects a variable name in the **Query Target** field. 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. Finds all variables that match the target text and returns all source code locations where the values of the 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

The **Functions** submenu contains queries that deal with functions. It operates in similar fashion to the variable queries; that is, you create a list of functions and select individual functions for detailed queries. The following selections are available:

- **List All Functions**: returns all functions it finds implemented in the fileset. Ignores any target text.
- **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 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

The **Files** submenu contains queries that deal with files. The following selections are available:

- **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). Ignores any target text.
- **List All Header Files**: returns all header (*filename*`.h`) files in the fileset.
- **List Matching Files**: expects either a file name in the **Query Target** field or no target text at all. If it finds target text, it returns all file names 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. Returns the names of all files that include the files specified by the target text.
- **Who is Included By?**: expects a file name in the **Query Target** field or a selected file name. Returns the names of all files that are included by the specified files.

Class Queries

The **Classes** submenu contains queries that deal with C++ classes. The following queries are available:

- **List All Classes**: returns all classes it finds in files defined by the fileset. Ignores any target text.
- **Where Defined?**: expects a class name in the **Query Target** field. 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. Returns the immediate subclasses of the classes matching the target text.
- **List Superclasses**: expects a class name in the **Query Target** field. 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. Returns those methods defined within the classes that match the target text.

Method Queries

The **Methods** submenu contains queries that deal with C++ member functions, also called methods. The following queries are available:

- **List All Methods**: returns all methods in the fileset. Ignores any target text.
- **Where Defined?**: expects a method name in the **Query Target** field. 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. Returns source code locations of all class declarations that include methods that match the target text.

Common Blocks Queries

The **Common Blocks** submenu applies to Fortran source code only. The following queries are available:

- **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. Finds all common blocks that match the target text and returns the source code locations where the common blocks are defined.
- **Where Common Block Referenced**: returns all source code locations where the common block appears.

Types Queries

The **Types** submenu helps you get type information. The following queries are available:

- **List All Types**: returns all types used in the source code.
- **Where Type Defined**: expects a type in the **Query Target** field. 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 where functions and data items using the type appear.

Directories Queries

The **Directories** submenu helps you determine the organization of the current fileset. The following queries are available:

- **List Directories**: lists all directories in the fileset.
- **List Files**: lists all files in the fileset.

Packages Queries

The **Packages** submenu helps you get package information when you are analyzing programs written in Ada. The following queries are available:

- **List All Packages**: lists all packages in the fileset.
- **Which Package Defined**: expects a package name in the **Query Target** field. Finds all packages that match the target text and returns all source code locations where those packages are defined.
- **List Functions in Package**: expects a package name in the **Query Target** field and returns all functions declared in the package spec and the body.
- **List Data in Package**: expects a package name in the **Query Target** field and returns data declared in the package spec and body.
- **List Types in Package**: expects a package name in the **Query Target** field and returns all types declared in the package spec and body.
- **Where Package Withed**: expects a package name in the **Query Target** field and returns all packages that with the given package.

Tagged Types Queries

The **Tagged Types** submenu helps you get tagged type information when you are analyzing programs written in Ada. The following queries are available:

- **List All Tagged Types**: lists all tagged types in the fileset.
- **Where Tagged Type Defined**: expects a tagged type name in the **Query Target** field and returns all tagged types that match the target text and returns all source code locations where those tagged types are defined.
- **List Parent Tagged Types**: lists parent types for the tagged type entered in the **Query Target** field.
- **List Derived Tagged Types**: lists derived types for the tagged type entered in the **Query Target** field.
- **List Primitive Operations**: lists primitive operations for the tagged type entered in the **Query Target** field.
- **List Components**: lists parent types for the tagged type entered in the **Query Target** field.
- **Where Tagged Type Used**: returns all declarations of functions and data of this type as well as sites where other types derive from this one or refer to it.

Task Types Queries

The **Task Types** submenu helps you get task type information when you are analyzing programs written in Ada. The following queries are available:

- **List All Task Types**: lists all task types in the fileset.
- **Where Task Type Defined**: expects a task type name in the **Query Target** field and returns all task types that match the target text and returns all source code locations where those task types are defined.
- **List Entries**: lists all entries for the given task type.
- **List Body Data**: lists data local to the body for the given task type.
- **List Body Functions**: lists all nonentry functions local to the task body.
- **List Body Types**: lists all declared types that are local to the task body.

- **Where Task Type Used**: lists all tasks of this type, as well as other types that derive from this type or refer to it.

Viewing Source Code

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

The **Source View** window opens the file containing the element and highlights the source line. Although this 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.

If you prefer to view source code in a text editor window, choose **General Options** from the **Admin** menu to open the **General Options** dialog box, which offers the **Use Source View** selection. Turn this option off to select `vi` as your text editor for source code. To set a different alternate text editor, add the following line to your `.Xdefaults` file, where *editor* is the command for the editor you want to use:

`*editorCommand:` *editor*

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 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 does not 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 4-4.

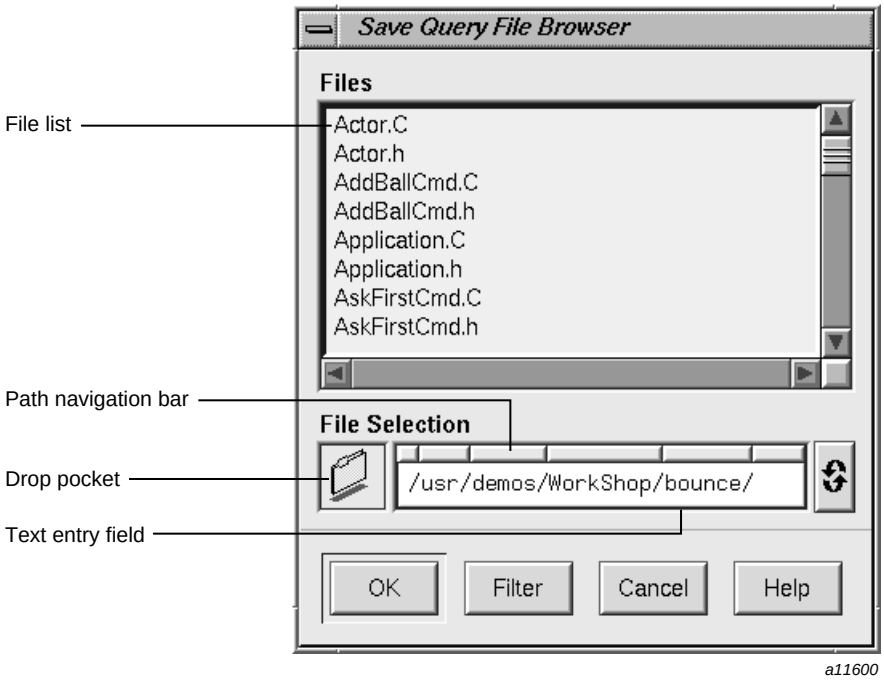

Figure 4-4 The **Save Query File Browser** Window

To save query results, move to the directory in which you want to make the save. To specify a directory, you can use the path navigation bar, enter a path in the text field, or drag a folder into the drop pocket. Then 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 are in Text View, the Static Analyzer saves the results in text format. If you are looking at a graphical view, the graph is saved in PostScript format. The Static Analyzer adds a heading to the text that lists the query type and the target text that specified the query. It also includes field headings that match those at the top of the query results area in the main window.

Views

This chapter discusses the different views available to display your query results. The Static Analyzer **Views** menu contains the following selections: **Text View**, **Call Tree View**, **Class Tree View**, **File Dependency View**. The **Results Filter** selection can be accessed from the Static Analyzer **Admin** menu. This chapter covers the following topics:

Text View

Text View is the Static Analyzer's default display for query results. Because this view is limited to text, it 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 5-1) that identify the query type, show the extent of Results Filter reductions (called the **Scoping** field), 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.

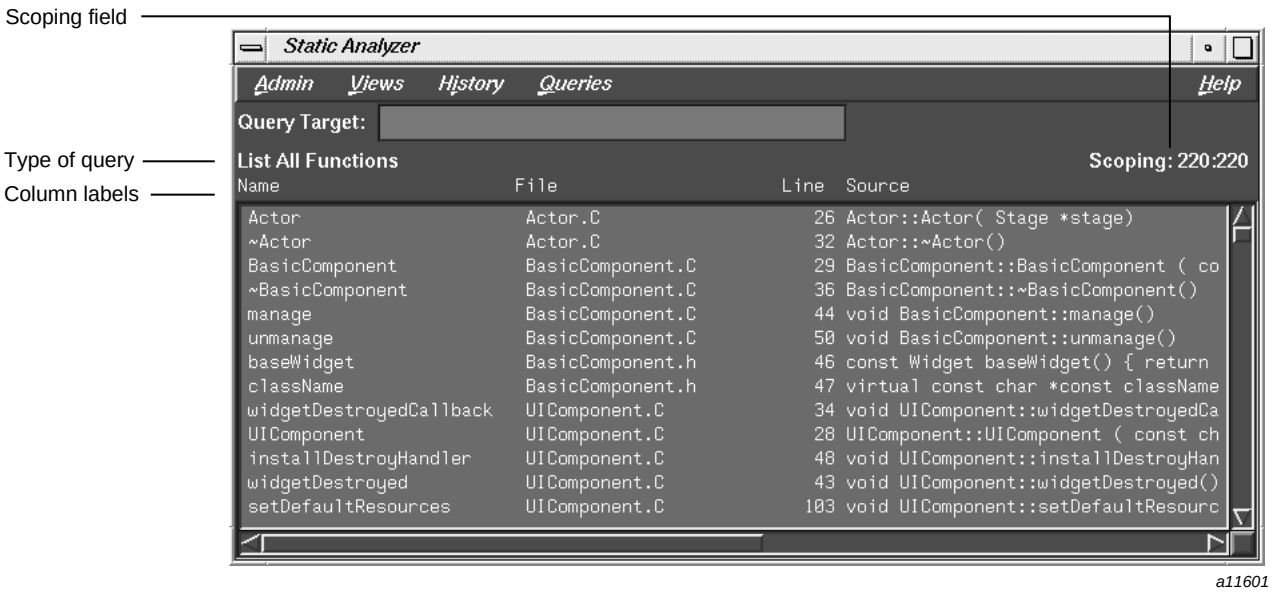

Figure 5-1 Sample Text View

Text View's arrangement of information within each element line depends on the query type. The left field always lists the type of element for which you have searched. Fields to its right show the location of that element and, if applicable, the content of the source code line where the element is located. For example, Text View shows the results of a function query with the function name in the first field, the file name 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 dragging a corner.

To see the source code listing where an element occurs, 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.

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 file name (or at least as full a pathname as the Static Analyzer can find), use the **Full Pathnames** option from the **General Options** selection of the **Admin** menu.

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, 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 is 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 5-2), 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 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.

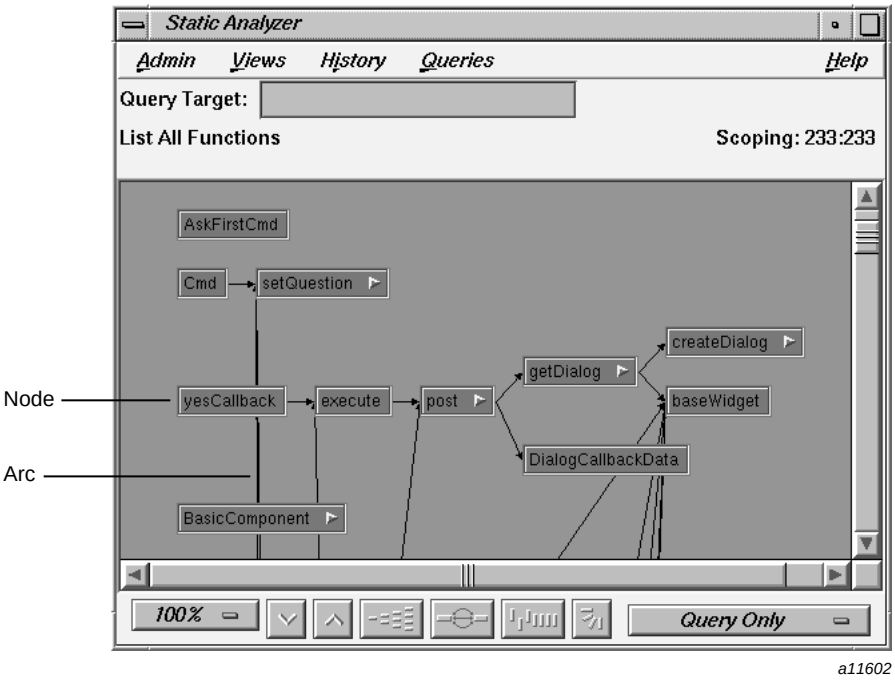

Figure 5-2 Call Tree View Displaying Functions and Function Calls

The Static Analyzer Control Panel

The **Static Analyzer** view control panel (shown in Figure 5-3) below the query results area offers a set of controls you can use to change the view. They help you see query results in the format most useful to you.

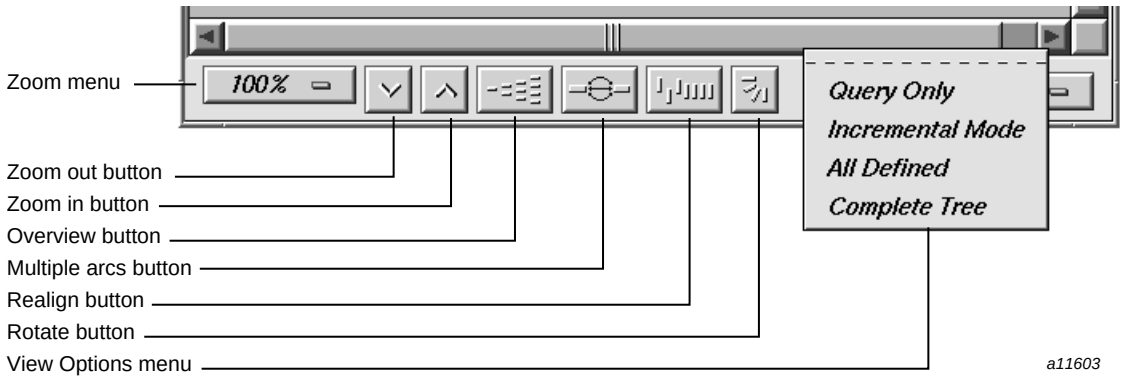

Figure 5-3 The View Control Panel

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 are viewing does not fit 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 to toggle back to horizontal), click the **Rotate** button.

The 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 the *ProDev WorkShop: Overview*.

Setting View Options

The **View Options** menu (at the lower right of the 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. 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 following selections are availabe:

- **Query Only**: shows only the target and results of each query 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. This is the default selection.
- **Incremental Mode**: leaves results of the previous query in the query results area and 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.

  Shows the target and the results of the last query in target-and-result colors. All other nodes are shown in a different 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, it shows the query target-and-result nodes in target and result colors. All other nodes appear in the nonquery 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 functions known within the fileset, regardless of whether they are defined. The display includes all the defined functions shown in **All Defined** display mode and adds any functions called but not defined. Because these include calls to external libraries, even a small program can generate a very large tree. The **Complete Tree** selection, like the **All Defined** selection, shows the results of any queries you make by highlighting in target-and-result colors, leaving all other nodes in nonquery 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 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. With Call Tree View you can view a function call by double-clicking 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 also 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 by using the **Node** menu.

Note: To compile the `jello` demo (from `WorkShop.sw.demos`), the `gl_dev.sw.gldev` subsystem must be installed.

---

1. Move to the demo directory `jello` by entering the following command:

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

2. Enter the following command to 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 by entering the following command:

   ```
   % cvstatic &
   ```

4. Select **Edit Fileset** from the **Admin** menu and move the `jello.c` file into the **Scanner Fileset** field by 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 is 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 click on it to select it.

   The `draw_everything` node 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` nodes to its right. The nodes from the previous query, `main` and its other children, still appear in the query results area in a nonquery color.

11. Select the result node `draw_jello` by moving the pointer into the **Query Target** field and pressing `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 5-4. The nodes from the two previous queries are still in the query results area.

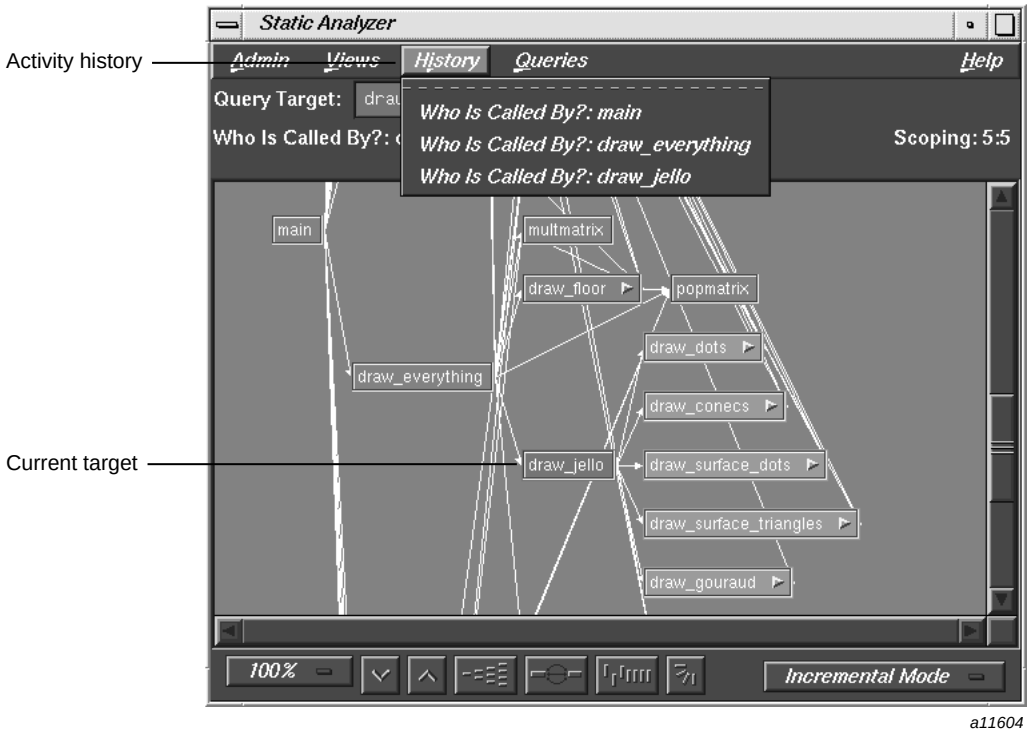

Figure 5-4 Incremental Mode Example

12. Choose **15%** from the **Zoom** menu to set scaling to 15%.

    The call tree reduces in size so that you can see all of the full call tree, although the function names are too small to be readable.

13. Hold down the right mouse button over any node in the tree.

    The corresponding **Node** menu displays, and the name of the function appears at the top of the menu. By using this method, you can see a large part of a tree and orient yourself by displaying the node menus (see Figure 5-5).

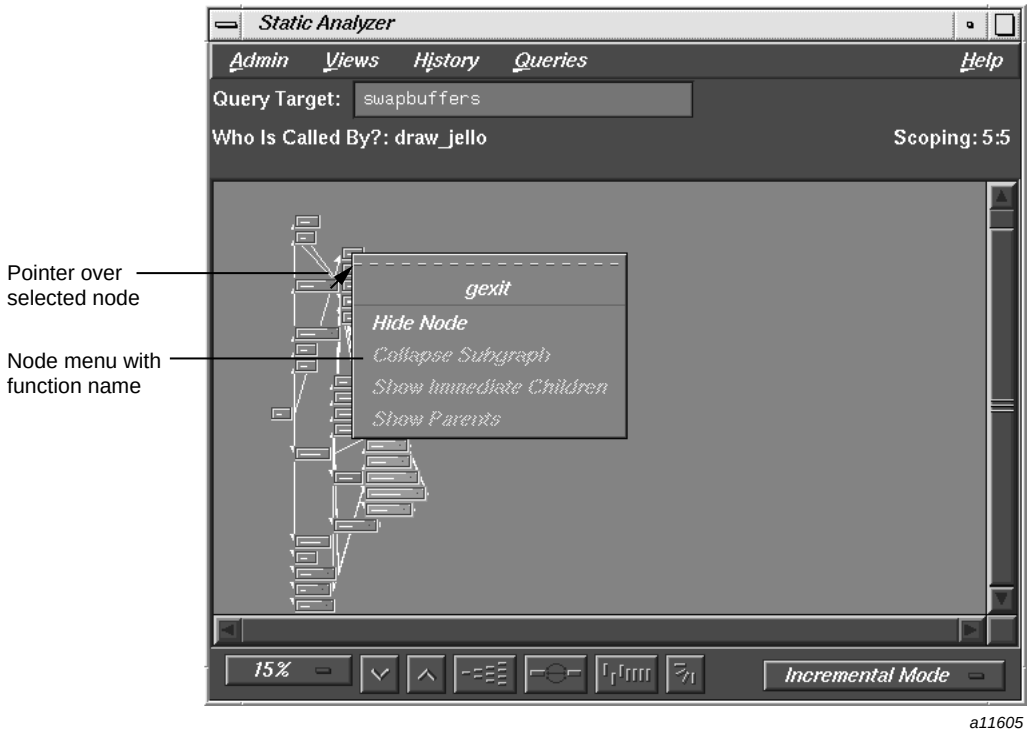

Figure 5-5 Displaying Node Information at Reduced Scale

14. Click a node towards the top of the call tree and choose **100%** from the **Zoom** menu.

    This returns you to viewing at 100% and demonstrates one technique for navigating around a large call tree.

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 is not intended for nonclass elements, and it will not show the results of function, file, and method queries.

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 and 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. A target color indicates target nodes, a results color indicates result nodes, and a nonquery color indicates 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 double-click a node to view source code for a class, but you cannot double-click an arc to see an inheritance.

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 displays only 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.

File Dependency View is particularly useful when you are analyzing Ada source files; it shows you the dependency between packages. If you double-click arcs in this view, you can see from where packages are imported and also definitions of where packages are brought in.

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 tool that works in all of the Static Analyzer's views.

The Results Filter filters the view to show you a subset of all 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 all 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 will see all 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 number is the number of elements returned after filtering and the second is the full number of elements returned by the query. For example, the following sample scoping line 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.

```
Scoping: 78:154
```

Setting Results Filters

To open the **Results Filter** window shown in Figure 5-6, choose **Results Filter** from the **Admin** menu.

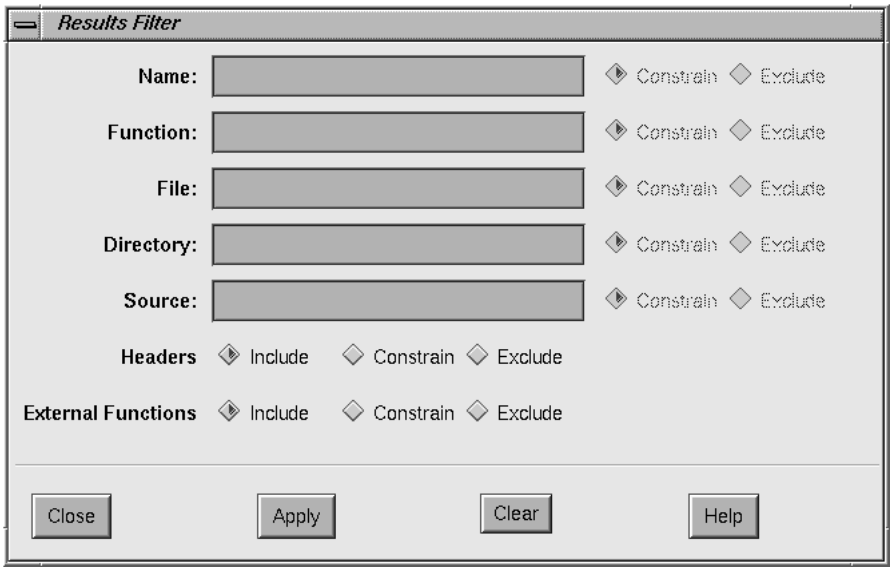

Figure 5-6 The **Results Filter** Window

The Results Filter has seven different scope filters. The first five filters provide fields in which you can enter regular expressions that 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.

Note: Regular expressions accepted by the Results Filter are the same as those supported by the `ed`(1) command. Refer to the `ed`(1) man page for details.

The following filters are available:

- **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.
- **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.
- **Source**: filters by the **Source** field in Text View. This field can exclude or constrain elements according to strings contained in lines of source code.
- **Headers**: filters according to whether elements are contained in a header file.
- **External Functions**: filters according to whether elements are contained in externally defined functions.

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 does not show the **Source** field for each function it displays.

Filtering by Name, Function, File, Directory, and Source

To filter using the first five scope filters, enter a regular expression in the appropriate text area, and then 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 are not displayed in the query results. For example, if you enter `jello.c` in the **File** scope filter and 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 the following 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 has 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 using either of these two filters, 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 the following buttons along 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 click the **Include**, **Exclude**, or **Constrain** button, so you do not usually need to click the **Apply** button.
- **Clear**: clears text from all text fields and returns the bottom two filters to the **Include** setting. 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 see, 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` by entering the following command:

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

2. Enter the following to 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 by entering the following command:

   ```
   % cvstatic &
   ```

4. Use the Fileset Editor to create a fileset for `bounce`. If you need help, refer to "Steps in Static Analysis", page 3

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 `196:196`, which means that all 196 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 `16:196`, which means that you see only 16 elements of the 196 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 5-7, page 67, showing only the methods found in the file `Application.h` that include the string `virtual` in their source code line. The **Scoping** field shows `5:196`.

9. 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 `196:196`.

10. Click the **Close** button to close the **Results Filter** window.

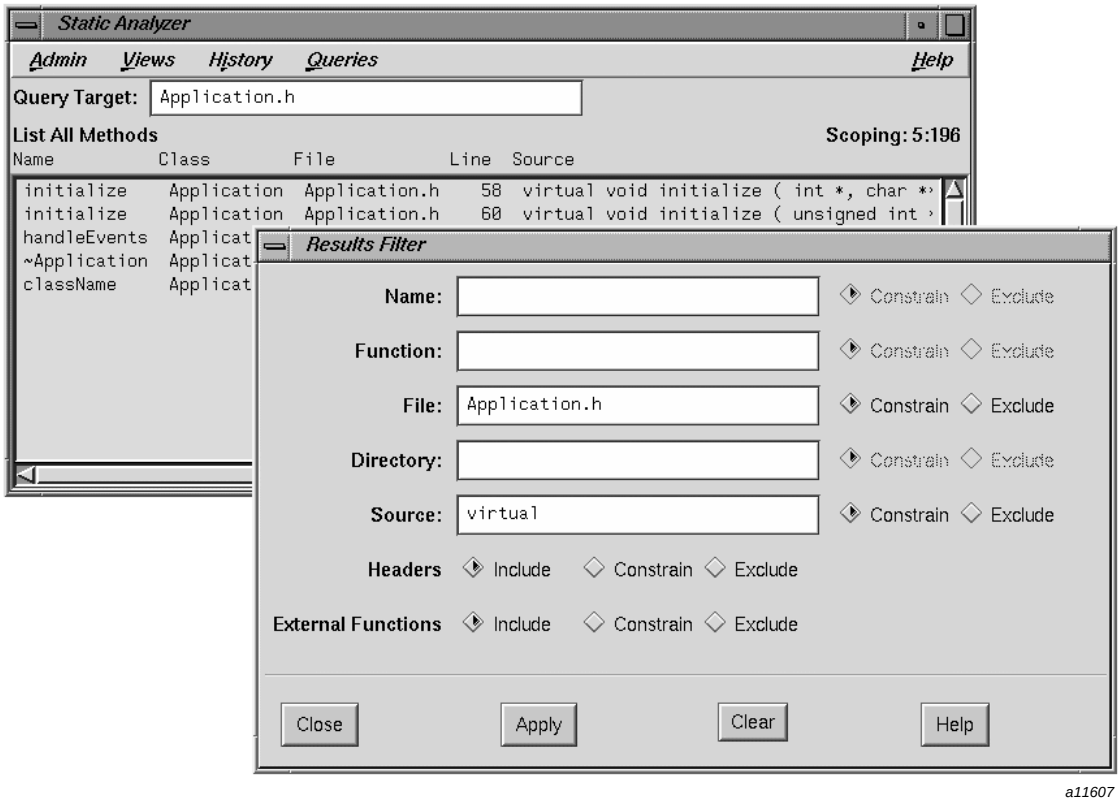

Figure 5-7 The Results Filter Query Results

Working on Large Programming Projects

The Static Analyzer works on uncompilable code, analyzes filesets containing files from completely different programs, and presents query results in a graphic form that is easy to browse. This 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. To be effective, you must narrow your analysis to a meaningful portion of your project, or you may end up with 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 the following topics:

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.

The following lines of code show a shell script that searches through a list of directories for file names with extensions that indicate source code files:

```
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 and assigns a list of directories you want searched. 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 nested 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`. Be sure to put a space between each entry.. 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 do not 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 will not compile without other parts of the system. You can 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. The Results Filter's **Headers** and **External Functions** settings are particularly useful for large programming projects.

If you set **Headers** to **Exclude**, you prevent the Static Analyzer from taking time to display query results that come from header files. And, if you set **External Functions** to **Exclude**, you ensure that the Static Analyzer does not display query results from libraries and other nonfileset files.

For example, consider the function `foo()`, which calls `bar()`, a function in the fileset. It also calls `XtCreateWidget()`, a library function that is not 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()`.

Although the Results Filter does not 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 are using a tree view to display the results of comprehensive queries.

Applying Group Analysis Techniques

Although it is 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 are responsible, sometimes it is necessary to analyze all files in a 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 6-1. This arrangement allows users on personal workstations to query the extensive project database without actually creating the database.

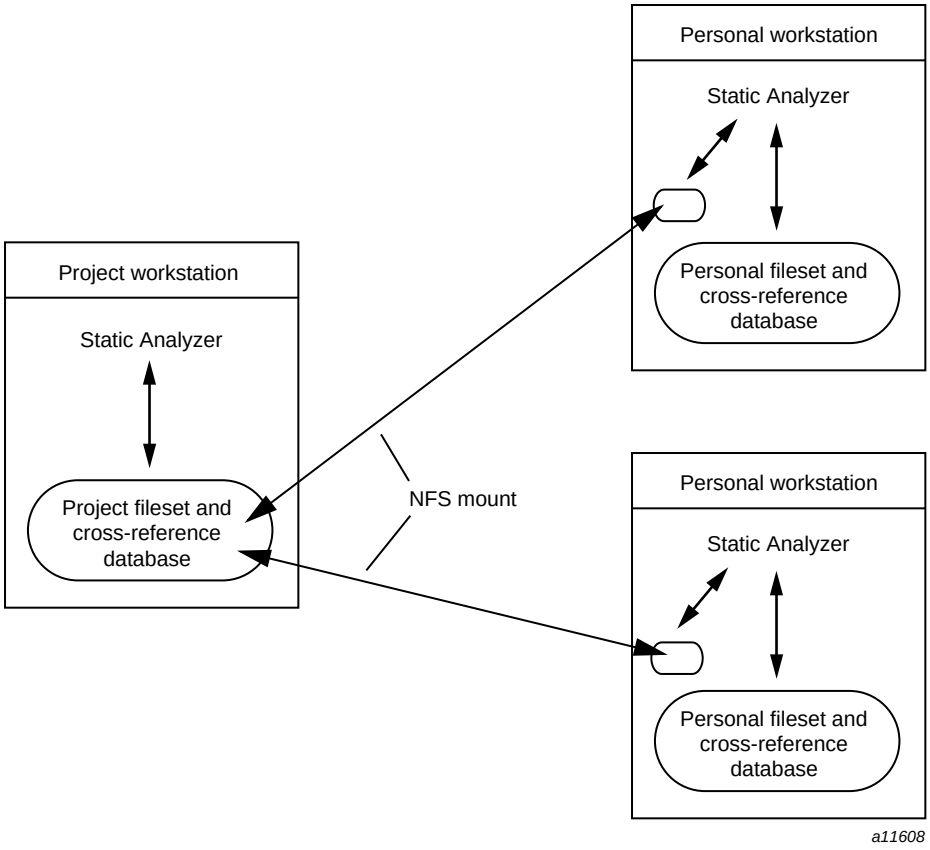

Figure 6-1 A Project Cross-Reference Database

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 update the fileset automatically.

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. This option 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 `dcvstatic.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 the Network File System (NFS). 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 will not try to modify the project database. For example, the following 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 `-readonly` command line option sets the Static Analyzer so that it will not try to rebuild the project database. The `-fileset` command line option 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** will not yield useful results as too much code is displayed at one time. Comprehensive queries like this may also take a good deal of time to complete. It is 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 need to make comprehensive queries on a large database, consider the following viewing tips:

- Use Text View for your queries. Because Text View does not require the Static Analyzer to build a tree containing thousands of elements, it is much faster at displaying the results of a comprehensive query than any of the tree views.

  Although Text View does not show connections between calling and called functions in the query results area, you can easily follow a chain of functions. First, click the function name you want. Then press `Alt-B` to see which functions it calls or press `Alt-C` to see which functions call it.

- Because the tree views show relationships between query elements more clearly than Text View, 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.

  Use the **Query Only** and the **Incremental Mode** viewing options to restrict the number of elements displayed for a query.

  In **Incremental Mode**, you can build a tree from scratch by making very specific queries that identify and follow only the branch of the tree in which you are 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.

Getting Started with the Browser

This chapter is designed to introduce you to the Browser, a facility accessed from the Static Analyzer that shows specific C++ and Ada relationships. This chapter describes what you need to run the Browser, shows you how to start it, and presents a brief overview of its main window and menus. To see examples of using the Browser, see Chapter 8, "Browser Tutorial for C++", page 81, and Chapter 9, "Browser Tutorial for Ada", page 97.

This chapter contains the following sections:

Starting Browser View

After you have created a fileset and built a static analysis database, you are ready to make object-oriented queries using the Browser. To access the Browser, open the **Admin** menu in the Static Analyzer and select **Browser**. The **Browser View** and the **Browsing Choices** windows appear as shown in Figure 7-1.

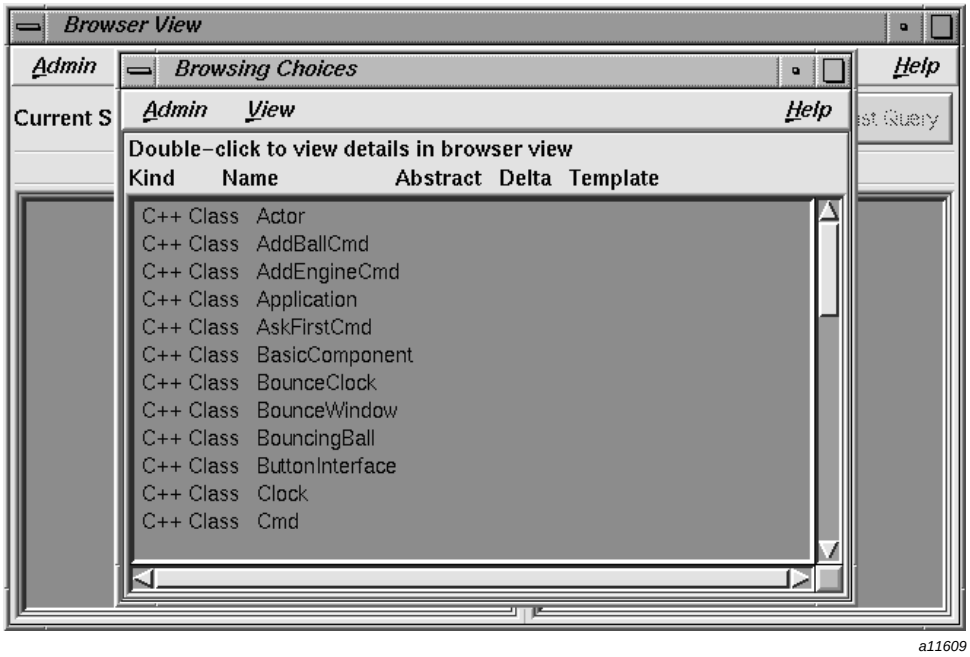

Figure 7-1 Browsing Windows

The **Browsing Choices** window lets you select an item from the fileset to be displayed in the **Browser View** window — either a class if you are using C++; or a package, task, or tagged type, if you are using Ada. The **Browser View** window then displays detailed information on the item.

General Characteristics of the Browser

The **Browser View** window shows you the internal structure and relations of the item you have selected in a textual, outline format. You can also select components of the item and perform queries on them. The results of queries are highlighted in **Browser View** and can also be displayed in the Static Analyzer. **Browser View** can display the contents of C++ and Ada entities. This section describes the features of the Browser common to both languages. For the language-specific characteristics, see Chapter 8, "Browser Tutorial for C++", page 81 (for a C++ sample session), and Chapter 9, "Browser Tutorial for Ada", page 97 (for an Ada sample session).

Browser features that are common to both the C++ and Ada versions are shown in Figure 7-2.

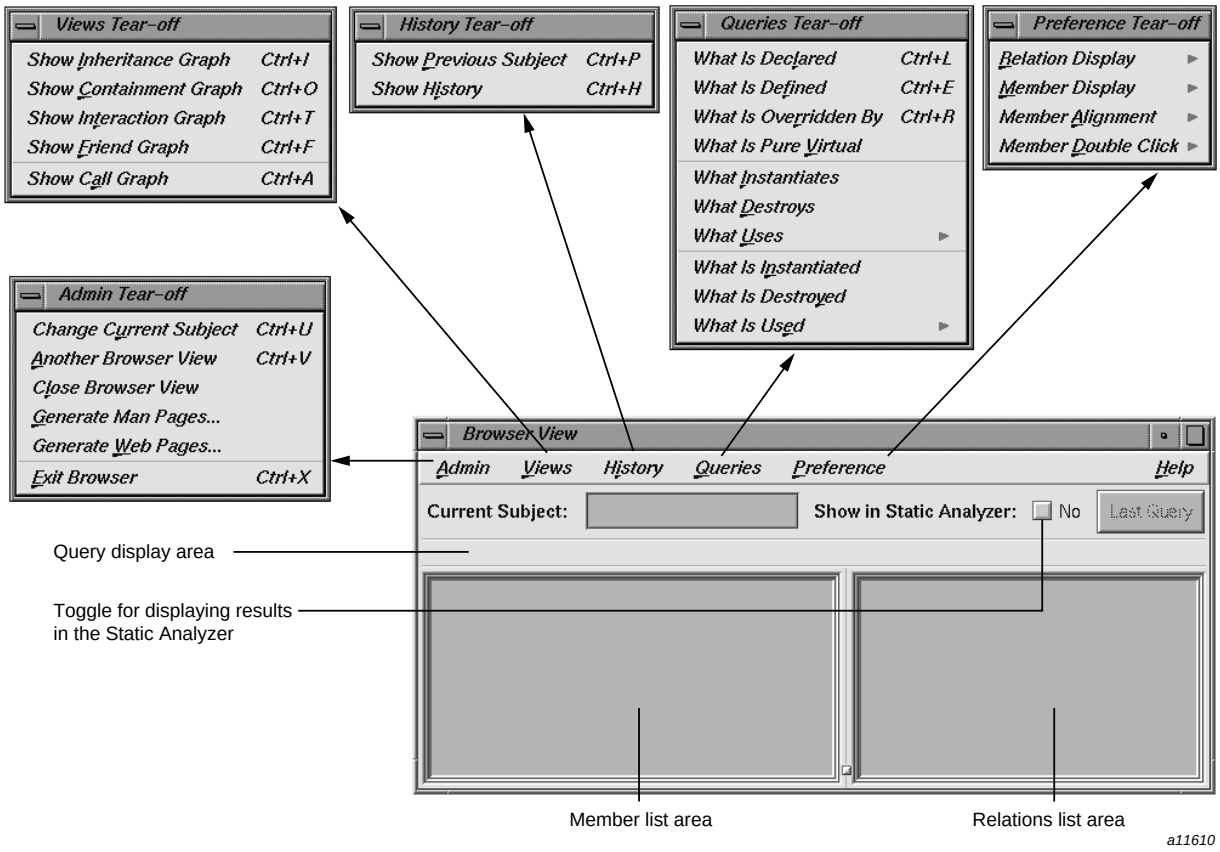

Figure 7-2 Browser View Features

Browser View Outline Lists

The Browser displays its data in outline lists in two side-by-side panes in the **Browser View** window. The lists are in a hierarchical, expandable outline format organized by category. The left pane displays an individual entity and its internals and the right pane displays other items to which that entity is related. When you are looking at C++ code, **Browser View** displays individual classes and their members in the left

pane, and related classes and members in the right pane. The Ada version displays individual packages and their components on the left, and related packages and components on the right.

Outline Icons

An outline icon is a diamond-shaped, concave icon. It appears to the left of component categories in the lists displayed in the Browser. An outline icon is used to expand or collapse a category. The icon contains an arrow pointing downward if the category is expanded (all items displayed) or to the right if the category is collapsed (all items hidden). Clicking the arrow switches back and forth between collapsing and expanding the category. A right-pointing outline icon that appears filled indicates that one or more of the hidden items satisfy the current query. Figure 7-3 illustrates these conditions.

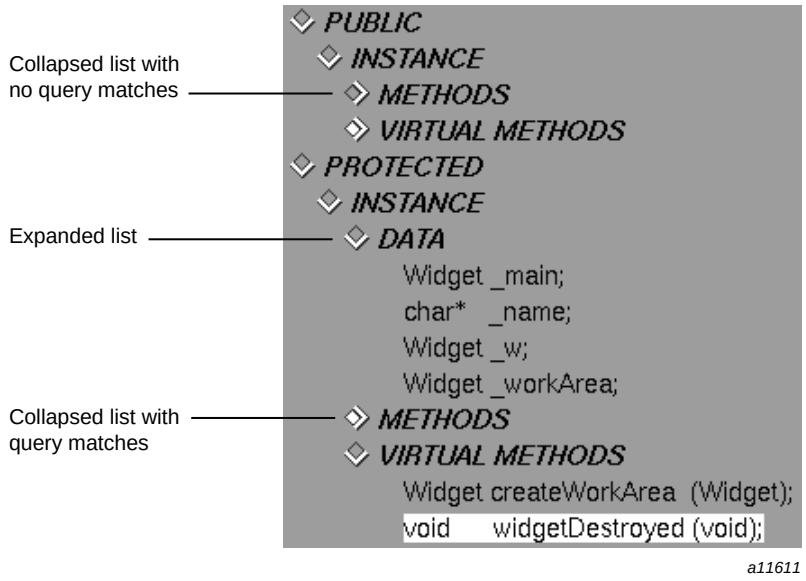

Figure 7-3 Outline Icon Examples

Browser View Menus

The **Browser View** window provides the following menus:

- **Admin** menu—for general housekeeping.
- **Views** menu—for displaying relationships in a graphical format. You can request four variations of class graphs based on these relationships:
  - *Inheritance*, which describes the relationship of parent classes to derived classes (C++) and parent tagged types to derived tagged types (Ada)
  - *Containment*, which describes the relationship of container classes to the classes they contain
  - *Interaction*, which describes the relationship of classes using methods of other classes
  - *Friends*, which describes the relationship of classes declaring other classes as friends

  You can also request a call graph to view the relationships of selected methods or functions.
- **History** menu—for going back to a previous Browser activity.
- **Queries** menu—for performing queries on the current item. (You can also perform queries on a selected element in either pane by holding down the right mouse button. These popup queries menus have different selections depending on the type of element.)
- **Preference** menu—for changing the appearance of the display and the behavior enacted by double-clicking with the mouse.

Other Browser Window Features

The **Current Subject** field displays the name of the item you have selected. Its label indicates the kind of item being displayed. Note that the **Current Subject** field provides a form of file completion; if you enter the partial name of an item and then press the space bar, the name will be completed up to the point that a unique string can be found.

The **Show in Static Analyzer** toggle lets you display the results of any queries in the Static Analyzer window. The Static Analyzer shows more detail, including source information, than the **Browser View** does.

The **Last Query** button lets you display the result of the previous query to the Static Analyzer.

The Browser has annotated scroll bars. This means that when you perform a query, tick marks will appear in the scroll bars (if there are any) to indicate matching elements.

Browser Tutorial for C++

This tutorial demonstrates the main features in the Browser. It outlines common tasks you can perform with the Browser, using sample C++ source code to illustrate the use of each function.

Sample C++ Session

The demonstration directory, `/usr/demos/WorkShop/bounce`, contains the complete source code for the C++ sample application `bounce`. To prepare for the session, you must create the fileset and static analysis database, then launch the browser from the Static Analyzer.

Procedure 8-1 Preparing for the sample session

Prepare for the session by following these steps:

1. Open a shell window and change to the `/usr/demos/WorkShop/bounce` directory.
2. Start the Static Analyzer by entering **`cvstatic &`**

   The **Static Analyzer** window opens.
3. Select **Browser** from the Static Analyzer **Admin** menu.

   This starts the Browser if a parser mode static analysis database has already been built. If none is available, an error message appears, and you must specify a parser mode fileset as shown in Figure 8-1. Then you need to select **Browser** from the **Admin** menu again. This causes a new database to be built using parser mode and takes several minutes to complete.
4. Select **Browser** from the Static Analyzer **Admin** menu.

   This starts the Browser if a parser mode static analysis database has already been built. If none is available, an error message appears, and you must specify a parser mode fileset as shown in Figure 8-1. Then you need to select **Browser** from the **Admin** menu again. This causes a new database to be built using parser mode and takes several minutes to complete.

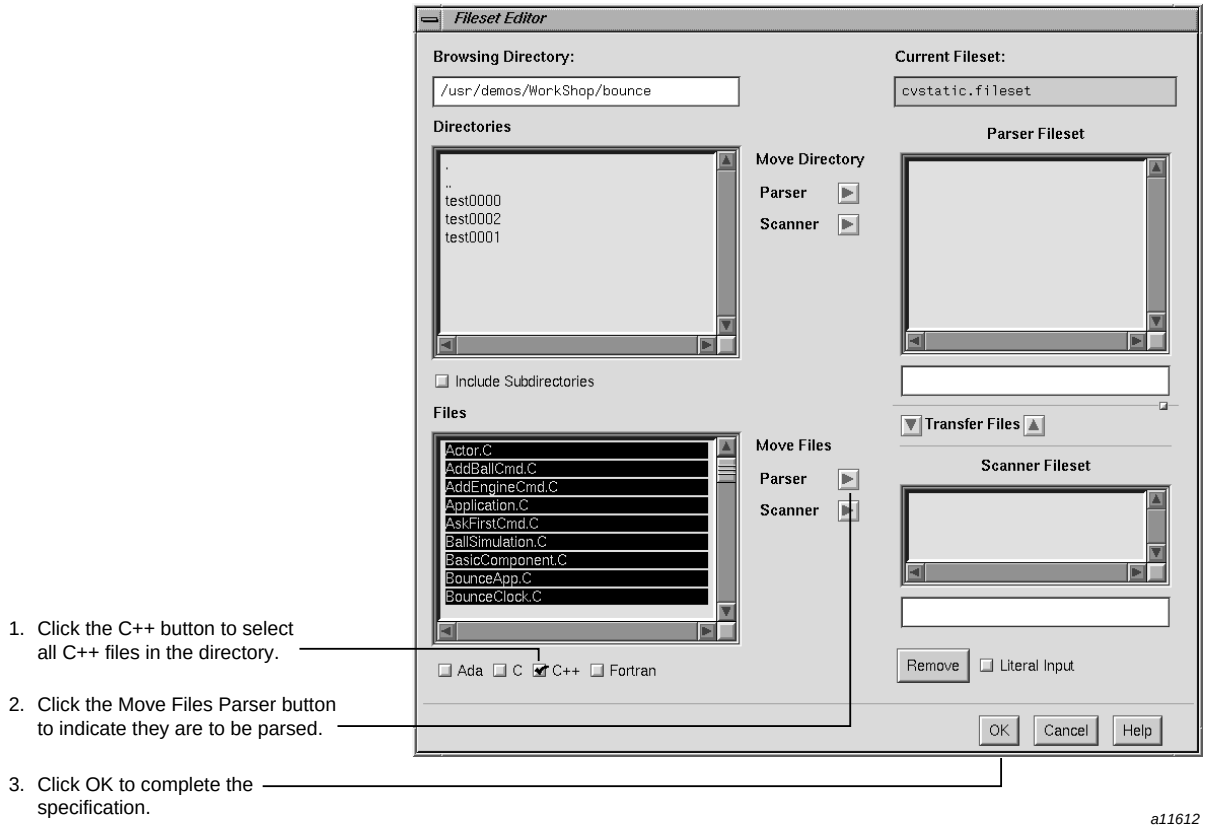

Figure 8-1 Steps in Specifying a Parser Fileset (C++)

The **Browsing Choices** chooser window opens at the same time as the **Browser View** window so that you can select the first class. The **Browsing Choices** chooser window contains the complete list of C++ classes included in the current fileset. Locate the **Actor** class in the chooser window. See Figure 8-2.

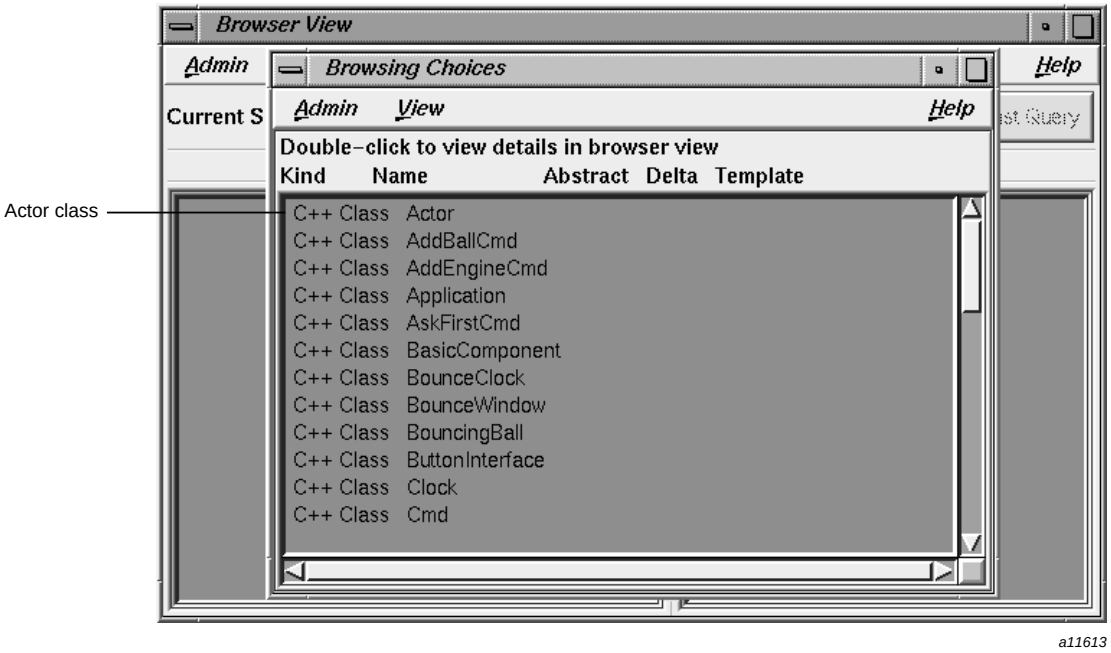

Figure 8-2 Initial Display with Item Selected

Procedure 8-2 Understanding the **Browser View** Window

1. Double-click the `Actor` class in the chooser window. The **Browsing Choices** window closes, and the data for `Actor` now appears in the **Browser View** window. The class name `Actor` is displayed in the **Current Subject** text field. Information about the class appears in the outline list views in the side-by-side panes (see Figure 8-3). `Actor` is now the current subject (class).

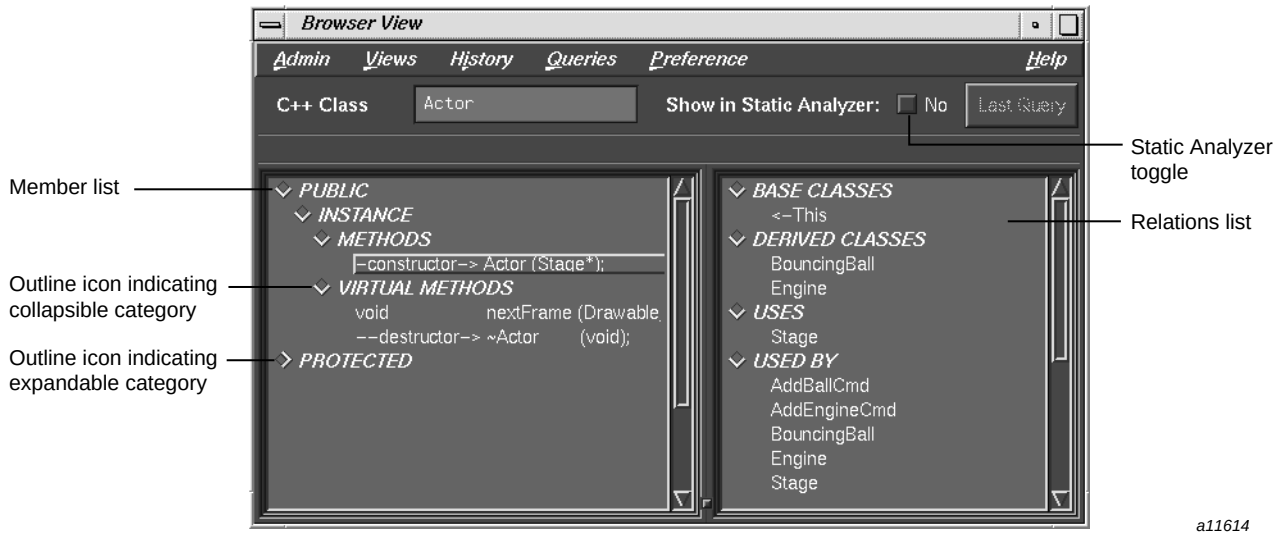

Figure 8-3 Browser View Window with C++ Data

2. Examine the screen contents for the **Browser View** window.

   The member list is on the left. It displays members according to their accessibility: `PUBLIC`, `INSTANCE`, or `PRIVATE`.

   Each kind of member can be `STATIC` or `INSTANCE` (nonstatic). Static objects of a given class contain the same value for a given member. `INSTANCE` members can contain different data values in different instances of that class.

   The member pane displays four kinds of class members: `TYPES`, `DATA`, `METHODS`, and `VIRTUAL METHODS`.

   The relations list, is on the left right side of the **Browser View** window. It displays information on related classes and methods, based on the point of view of the current class: **BASE CLASSES**, **DERIVED CLASSES**, **USES** (classes that the current class uses), **USED BY** (classes that the current class is used by), **FRIENDS**, **FRIEND FUNCTIONS**, and **FRIEND OF** relationships.

   The layout of both list displays are customizable.

Procedure 8-3 Expanding and Collapsing Categories

1. Click the outline icon to the left of the `PROTECTED` category (see Figure 8-3, page 84).

   This displays the elements in the `PROTECTED` category and causes the arrow in the outline icon to point downward.

2. Then click the outline icon to the left of the `PROTECTED` category again. This hides the elements and causes the arrow in the outline icon to point to the right again.

Procedure 8-4 Making Queries

1. Click the **Queries** menu and examine the results.

   Queries search the static analysis database for specific information about classes and their members, including class hierarchy, class and member declarations and definitions, and the interactions among members and classes (for example, which members call which members, where a definition overrides another, where an instance is created or destroyed, and so on).

   The Browser provides two types of menus for making queries:

   - **Queries** menu—accessed from the menu bar, its queries apply to the current class
   - Element-specific popup menus—Accessed by holding down the right mouse button while the pointer is over the selected element you want to query

   The Browser provides answers to queries by highlighting items in the member and related class lists that match the query. Optionally, you can display more detailed query results in the **Static Analyzer** window from which you launched the Browser.

2. Click the **Show in Static Analyzer** toggle button shown in Figure 8-3, page 84.

   This button lets you view the results of queries in the **Static Analyzer** window along with the Browser window. The **Static Analyzer** window has the advantage of showing source lines for your queries

   Select **What Uses** from the **Queries** menu and **To Contain** from its submenu (see Figure 8-4, page 87).

   The **Queries** menu in the menu bar lets you request relationship information for the current class. In addition to highlighting the matching elements in the list, the Browser displays indicator marks in the scroll bar showing the relative locations

of matching elements. Also, the query is identified in the field over the outline list area. If you click on an indicator mark, you will scroll directly to the matching element. Because you turned on the **Show in Static Analyzer** toggle, the results are shown in the Browser as well, including the file, line number, and source line for the classes containing `Actor`. See Figure 8-5.

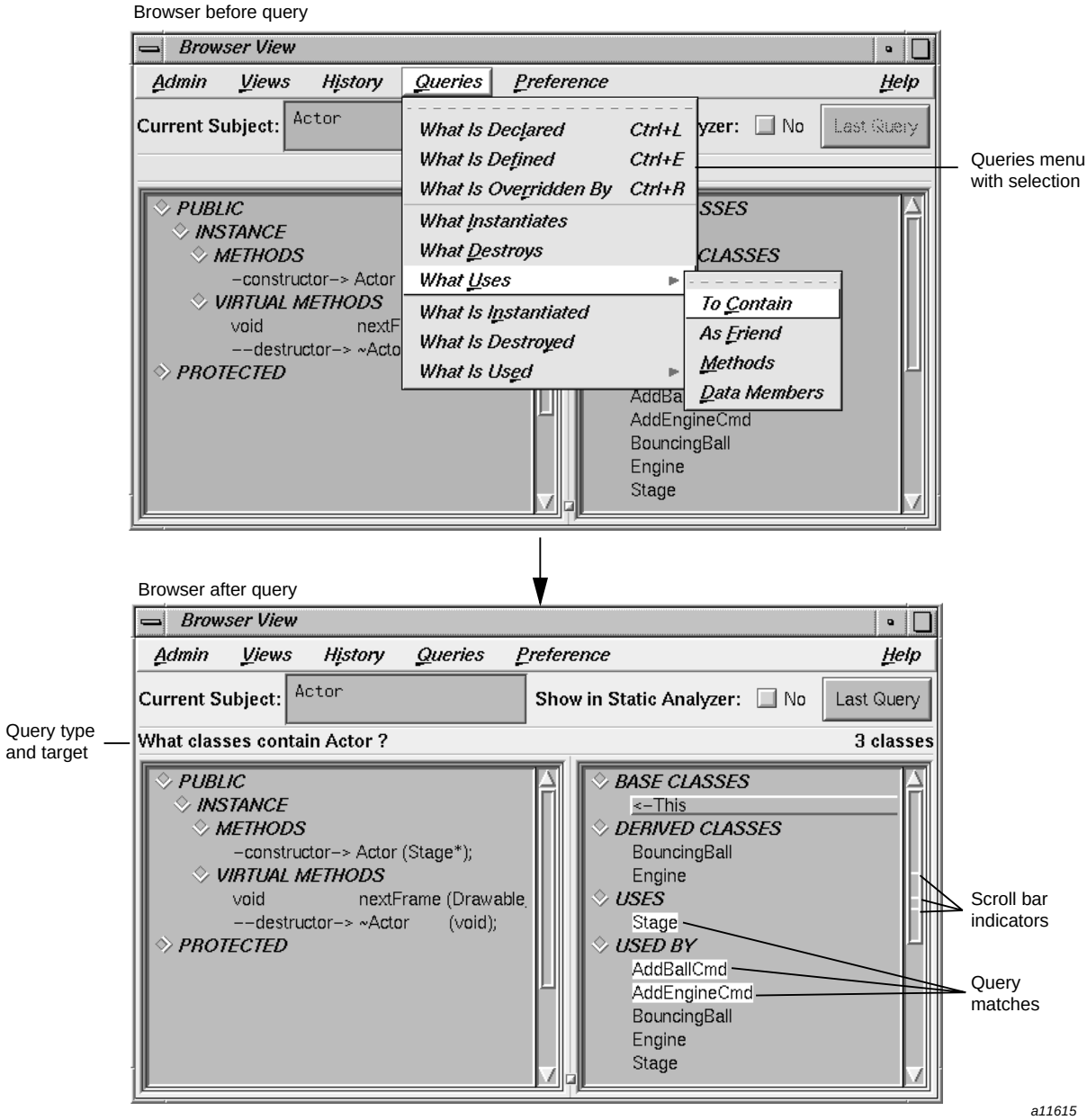

Figure 8-4 Performing a Query on Current Class

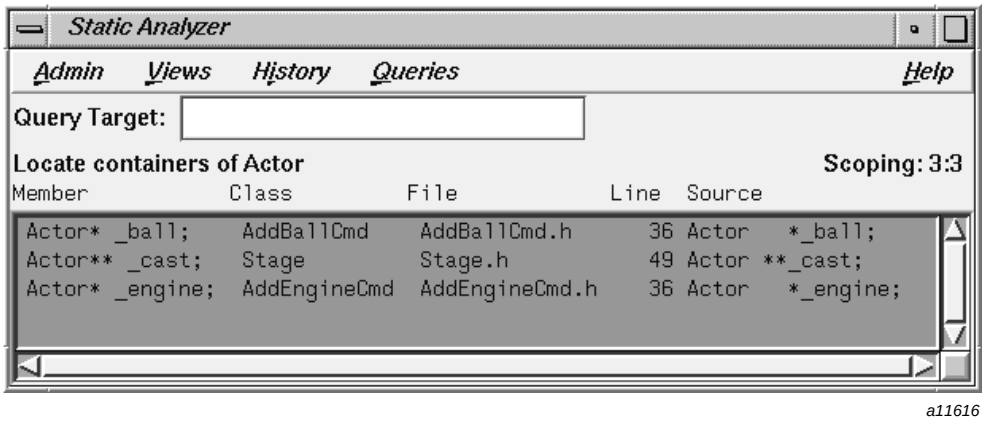

Figure 8-5 Static Analyzer after a Browser Query

3. Select the constructor method in the METHODS category by holding down the right mouse button. Then select **Show Source Where Declared**. See Figure 8-6, page 89.

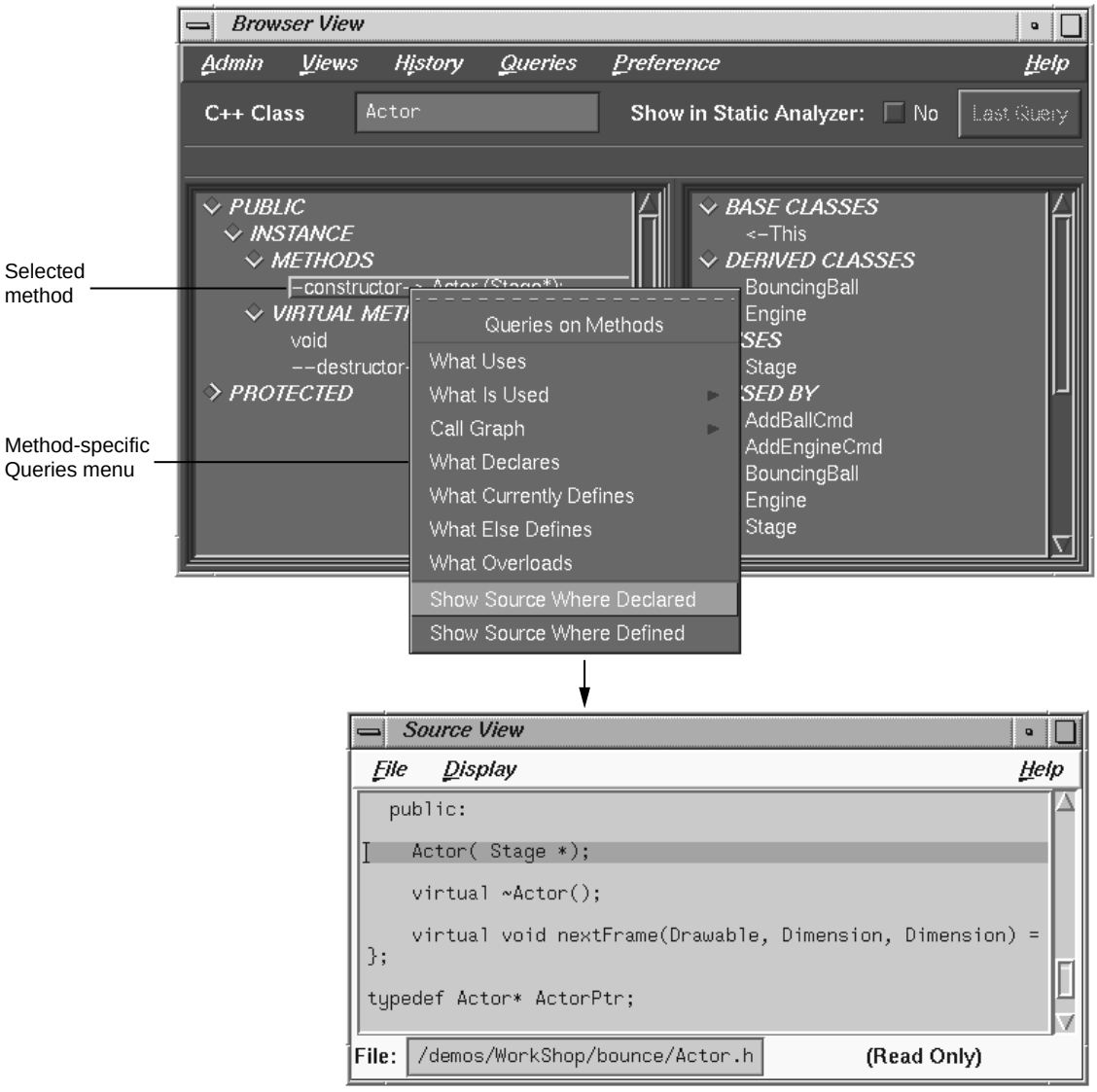

Figure 8-6 Performing a Query on an Element in a List

This displays the **Queries** menu specific to methods. In this case, the query lets us see the source code where it is declared. The **Source View** window now displays with the matching code highlighted.

For practice, try a few random queries.

4. Click the **Last Query** button in the **Browser View** window.

   Clicking this button displays the results of the most recent query in the **Static Analyzer** window from which the Browser was launched.

Procedure 8-5 Using the Browser Graphical Views

1. Look at the graphical views supplied by the Browser and Select **Show Containment Graph** from the **Views** menu in the **Browser View** window.

   The **Graph Views** window is displayed, set to **Containment** as shown in Figure 8-7. You can switch to other relationship modes through the Relation mode menu.

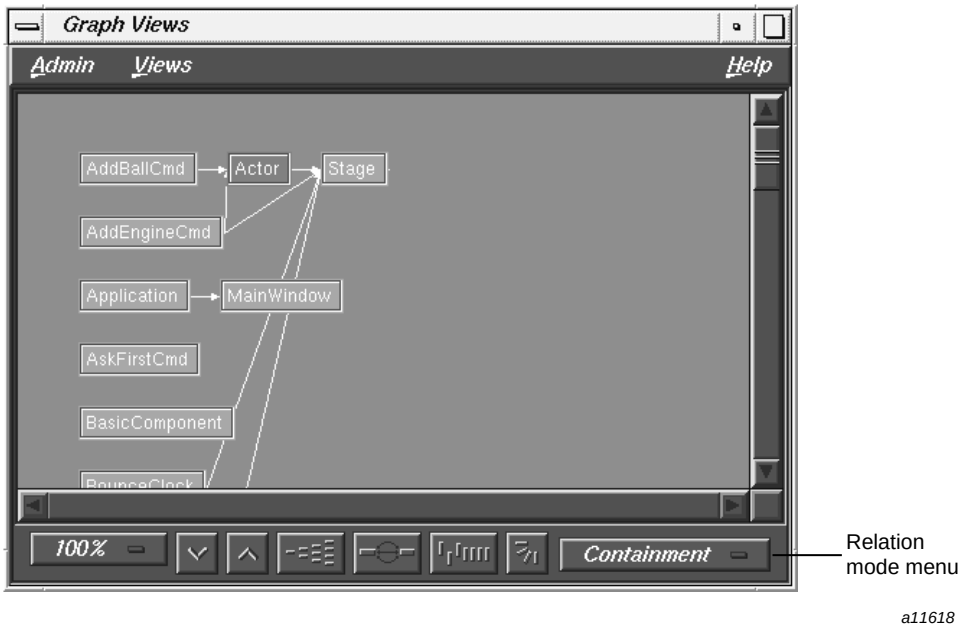

Figure 8-7 Graph Views Window in Containment Mode

2. Pull down the **Views** menu, select **Show Butterfly**, and resize the **Graph Views** Window to be smaller.

   This eliminates extraneous classes from the graph, displaying only those classes that `Actor` contains or is contained by. Now compare the graph with the query results shown in the Browser.

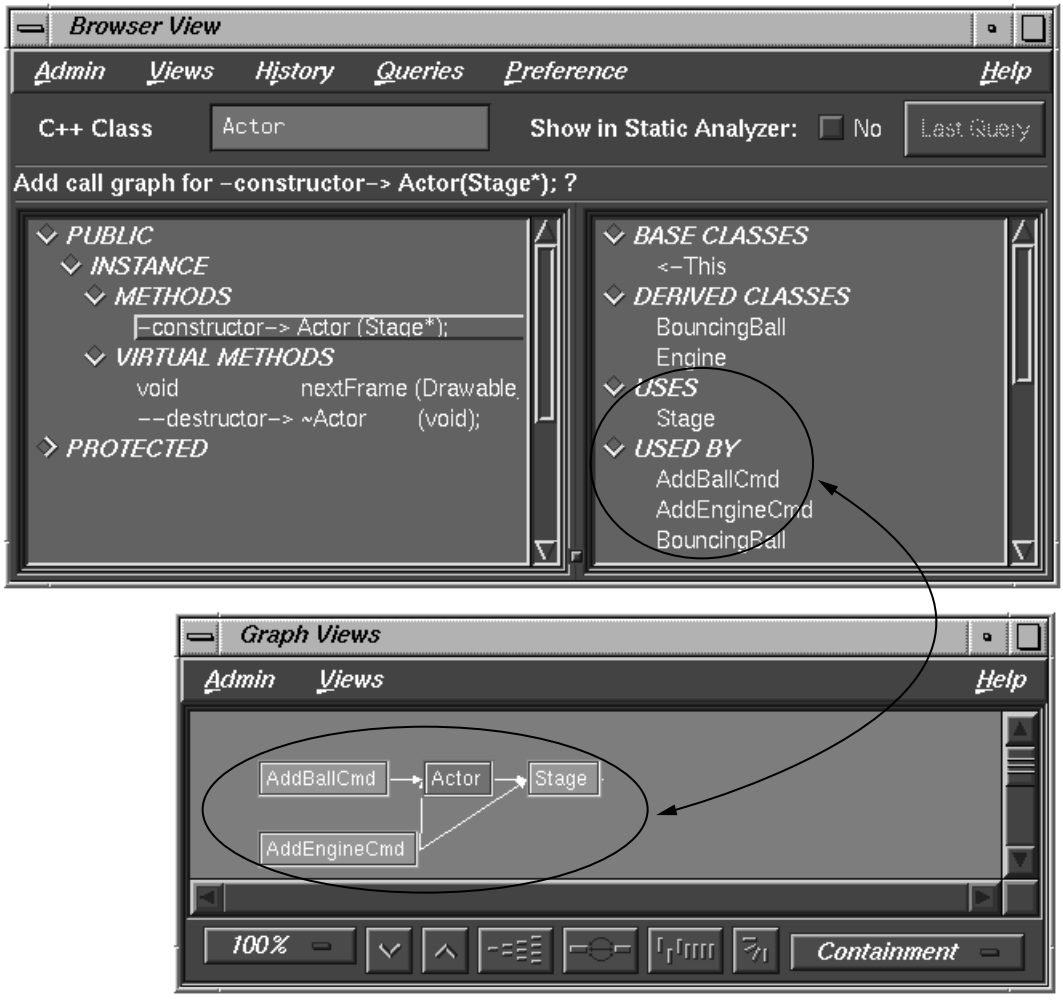

Figure 8-8 Comparison of Data Displayed in a Containment Graph

3. Click on the Relation mode menu in the lower right corner of the **Graph Views** window and select **Inheritance** from the displayed options (see Figure 8-7, page 90).

   This shows the inheritance relationships. In this case, the derived classes `BouncingBall` and `Engine` inherit from `Actor`, as shown in Figure 8-9, page 92.

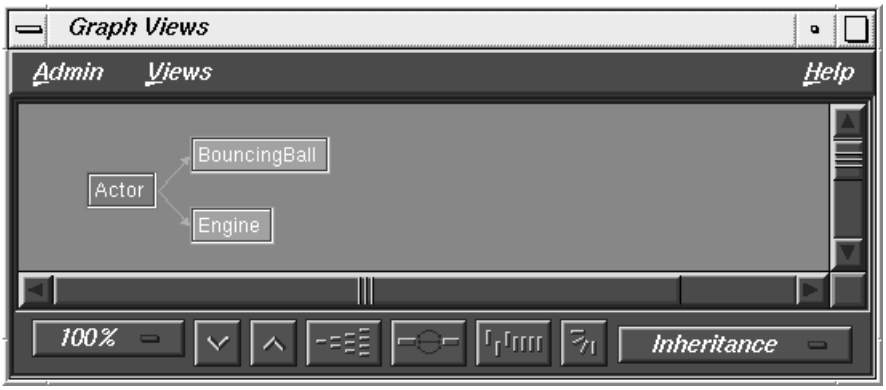

Figure 8-9 Graph Views Window in Inheritance Mode

4. Next, select **Interaction** from the Relation mode menu options.

   This displays the classes that directly interact with `Actor`. Those that use `Actor` appear on the left and those that are used by `Actor` appear on the right. Compare the display results with those from the **Inheritance** display.

Procedure 8-6 Shortcuts for Entering Subjects

1. Go back to the **Browser View** window, clear the **Current Subject** field, and type a question mark (`?`), followed by pressing the `Enter` key.

   This is a shortcut for displaying the **Browsing Choices** window. However, instead of selecting through the **Browsing Choices** window, we are going to demonstrate how name completion works.

2. Type **`Main`** and press the space bar.

   The Browser fills in the rest of the name, `MainWindow` in this example, and its data.

Procedure 8-7 Generating Man Pages

1. The Browser generates man page templates from your classes so that all you have to do is fill in the descriptions and provide comments. To create man pages for classes in the fileset, follow these steps:

2. From the **Browser View Admin** menu, select **Generate Man Pages**.

   The **Man Page Generator** window opens, as shown in Figure 8-10.

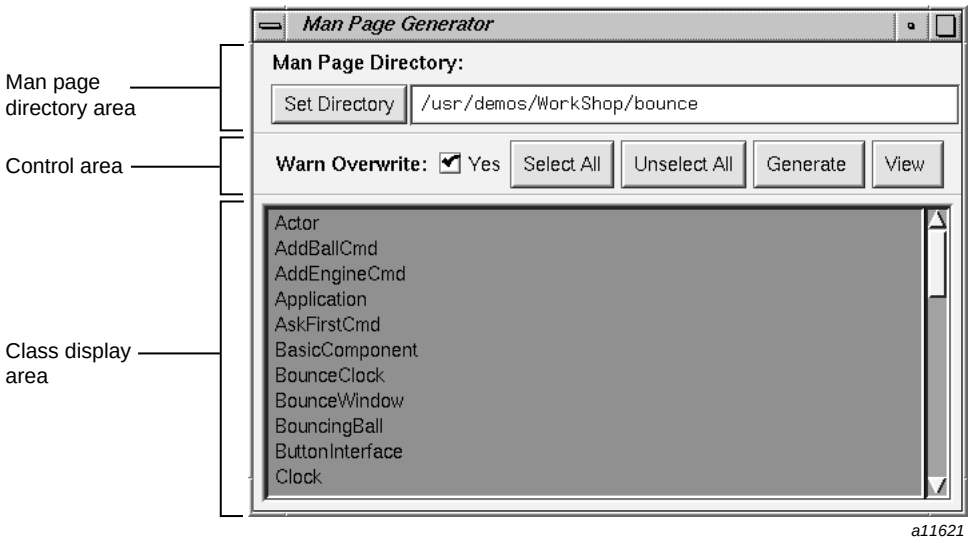

Figure 8-10 Man Page Generator Window

You can specify the target directory in the area at the top of the window, either directly in the **Man Page Directory** field, or by browsing in the dialog box displayed by clicking the **Set Directory** button. The control area lets you receive warnings if a man page already exists, select or unselect all classes, generate new man pages, and display shells showing the new man pages.

Click the **Select All** button in the control area.

This selects all the classes in the class list. If you need only a subset of the list, simply click the desired classes. If you change your mind, you can remove any current selections by clicking the **Unselect All** button.

3. Click **Generate**.

   Wait for a few seconds while your files are generated.

4. Click **View** to view the output files.

   A `winout` window containing the man page text opens, as shown in Figure 8-11. You can edit this file using a text editor, such as `vi`.

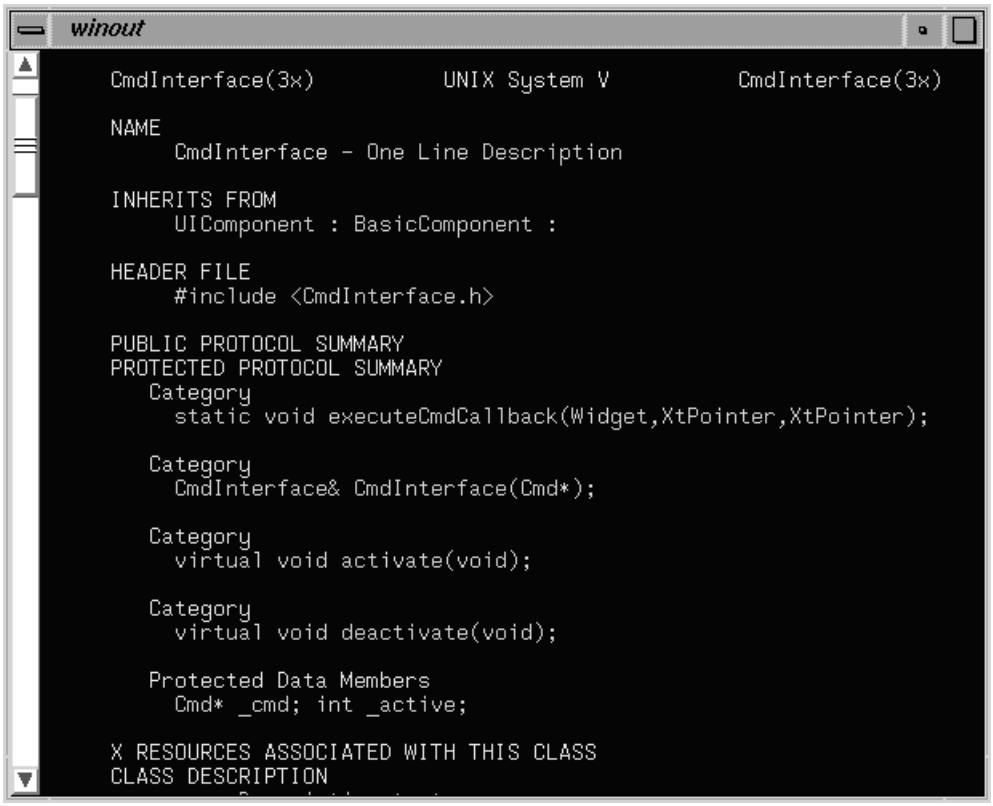

Figure 8-11 Man Page Template

5. Close the `winout` window using the window menu in the upper left corner.

Procedure 8-8 Generating Web Pages

The Browser also lets you generate web pages, that is, documentation in HTML format compatible with World Wide Web readers. To generate a web page, follow these steps:

1. From the **Browser View Admin** menu, select **Generate Web Pages**.

   The **Web Page Generator** window opens, as shown in Figure 8-12.

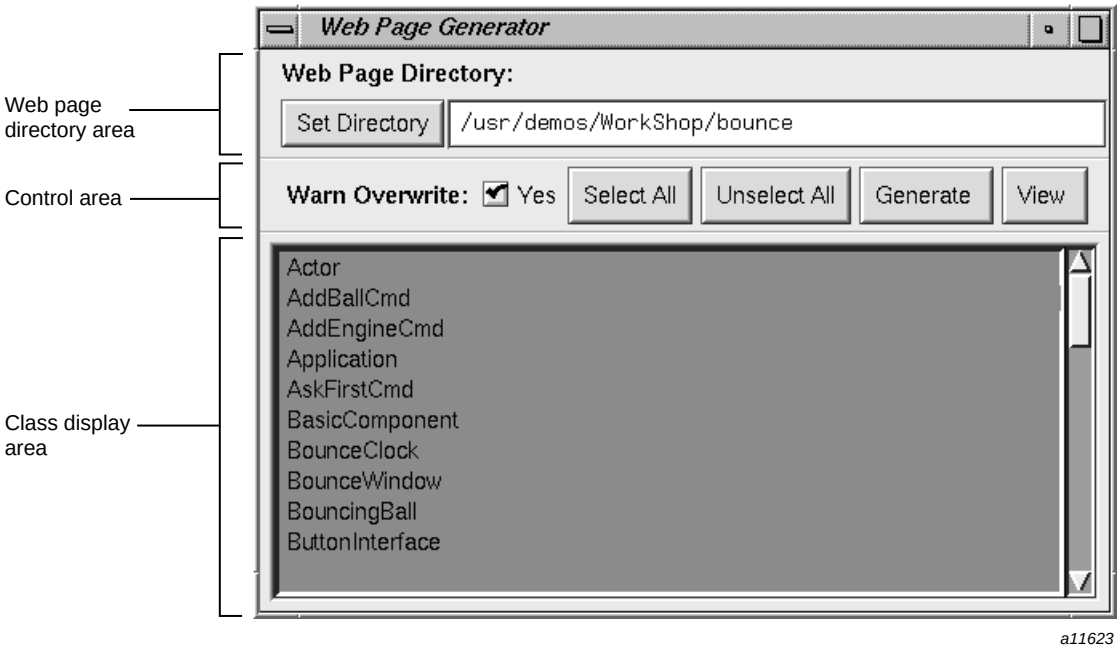

Figure 8-12 Web Page Generator Window

   This window operates in the same manner as the **Man Page Generator** window. You specify the target directory by typing directly in the **Web Page Directory** field or by browsing in the dialog box that comes up when you click the **Set Directory** button. The control area lets you receive warnings if a web page already exists, select or unselect all classes, generate new web pages, and display a shell showing the new web pages.

2. Click the **Select All** button in the control area.

This selects all the classes in the class list. If you need only a subset of the list, simply click the desired classes. If you change your mind, you can remove any current selections by clicking the **Unselect All** button.

3. Click **Generate**.

   Wait for a few seconds while your files are generated.

4. Click **View** to view the output files.

You have reached the end of the C++ tutorial. You can exit both the Static Analyzer and the Browser by pulling down the **Static Analyzer Admin** menu and choosing **Exit**.

Browser Tutorial for Ada

This tutorial demonstrates the main features in the Browser. The session outlines common tasks you can perform with the Browser, using a sample Ada application source to illustrate the use of each function.

Sample Ada Session

The demonstration directory, `/usr/demos/Ada/WorkShop/tagged_example`, contains the complete source code for a simple Ada application called `tagged_example`. To prepare for the session, you first need to create the fileset and static analysis database.

Procedure 9-1 Preparing for the sample session

Prepare for the session by following these steps:

1. Open a shell window, and change to the `/usr/demos/Ada/WorkShop/tagged_example` directory.
2. Start the Static Analyzer by entering **`cvstatic`**.

   The **Static Analyzer** window opens.
3. Pull down the **Admin** menu and select **Edit Fileset**.

   To create a parser mode fileset for this example, follow the instructions shown in Figure 9-1. It takes several minutes to build the database from the fileset.

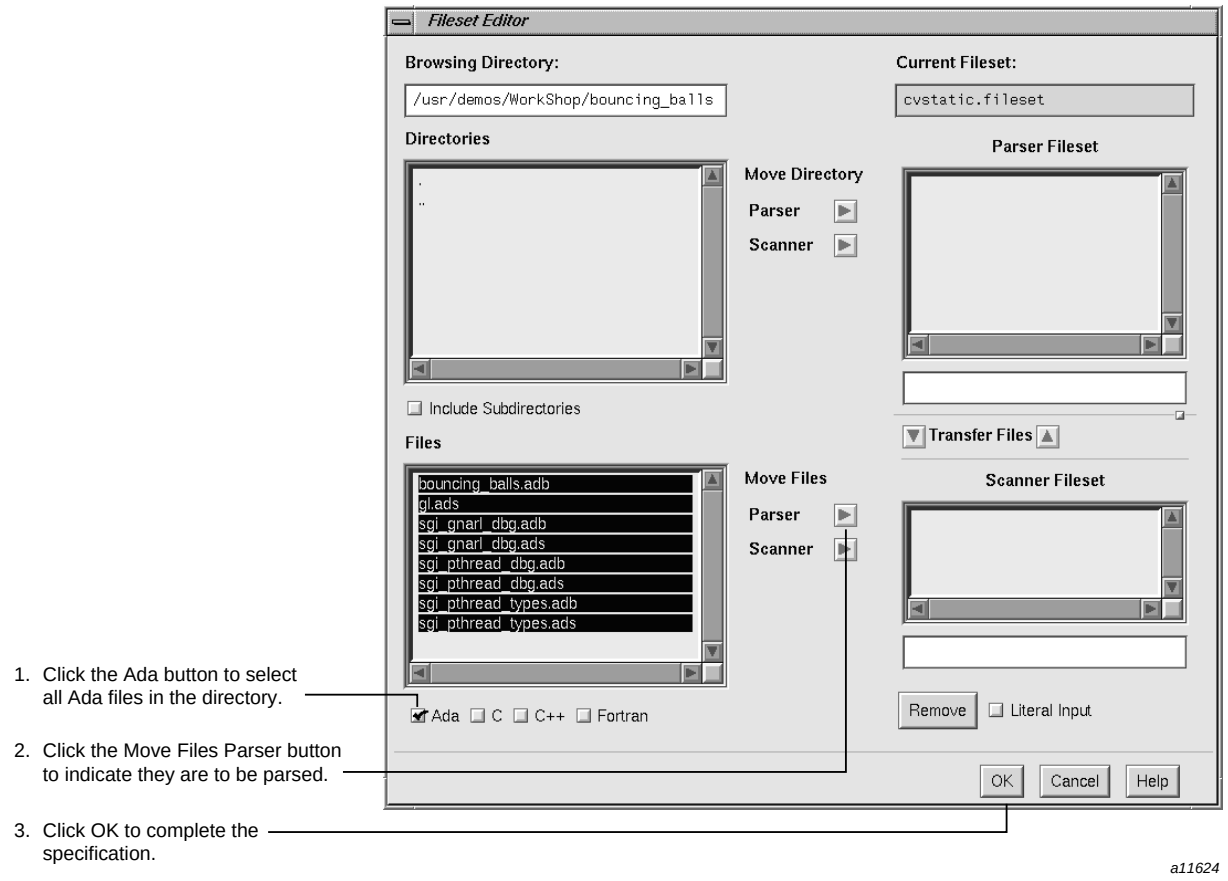

Figure 9-1 Steps in Specifying a Parser Fileset (Ada)

4. When the fileset is built, select **List All Packages** from the **Packages** submenu in the **Queries** menu.

   This displays all the packages in the fileset.

5. Select **List All Tagged Types** from the **Types** submenu in the **Queries** menu.

   This displays all the tagged types in the fileset.

6. Select **List All Files** from the **Files** submenu in the **Queries** menu.

This displays all the source code files in the fileset.

7. Pull down the **Views** menu and select **File Dependency View.**

   The File Dependency View shows you the dependency between packages (packages are defined one to a file). If you double-click arcs in this view, you can see where packages are imported using the `with` clause and also definitions where packages are brought in.

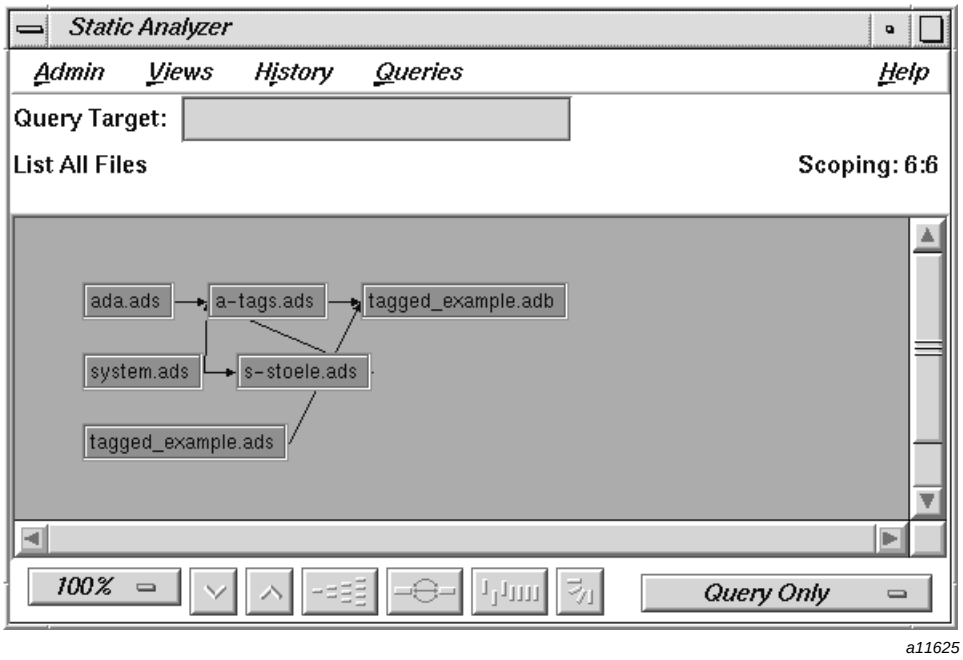

Figure 9-2 File Dependency View Example

Procedure 9-2 Starting the Browser

1. Pull down the Static Analyzer **Admin** menu and select **Browser**.

   This displays the **Browser View** window and the **Browsing Choices** chooser window, which is used to select subjects for browsing. The **Browsing Choices** chooser window contains the complete list of Ada entities (packages, tagged types, and task types) included in the current fileset. See Figure 9-3.

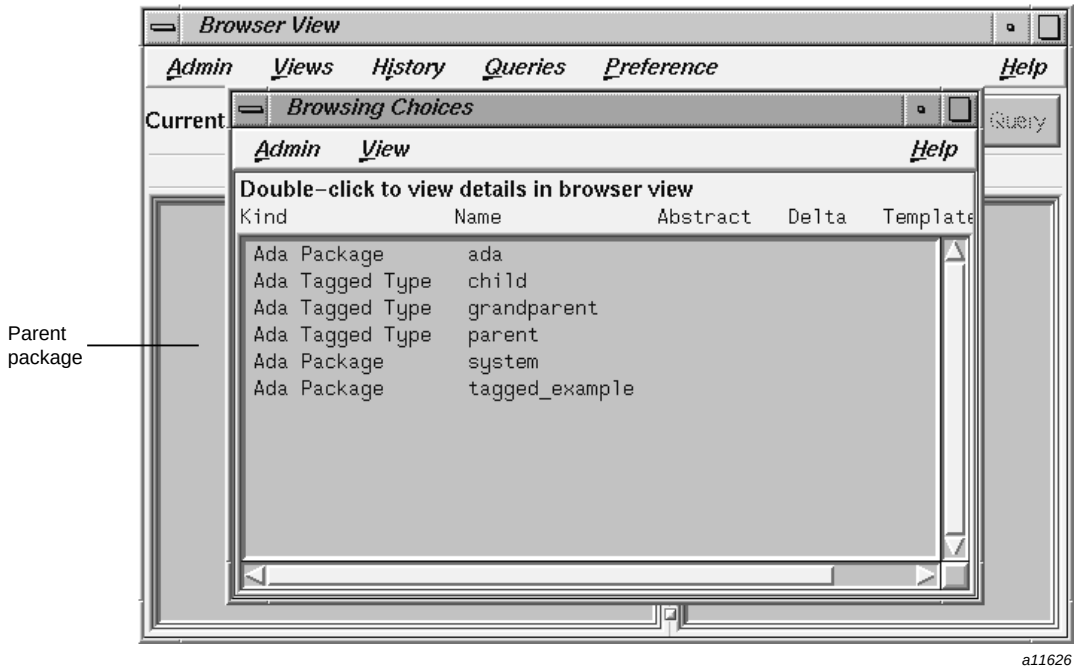

Figure 9-3 Initial Browser Display

Procedure 9-3 Understanding the Browser Window

1. Double-click the parent package in the chooser window.

   The **Browsing Choices** window is lowered, and the data for parent now appears in the **Browser View** window (see Figure 9-4). The subject parent is now displayed in the **Current Subject** text field and is identified as an Ada package. Information about Ada entities appears in the outline list views in the side-by-side panes.

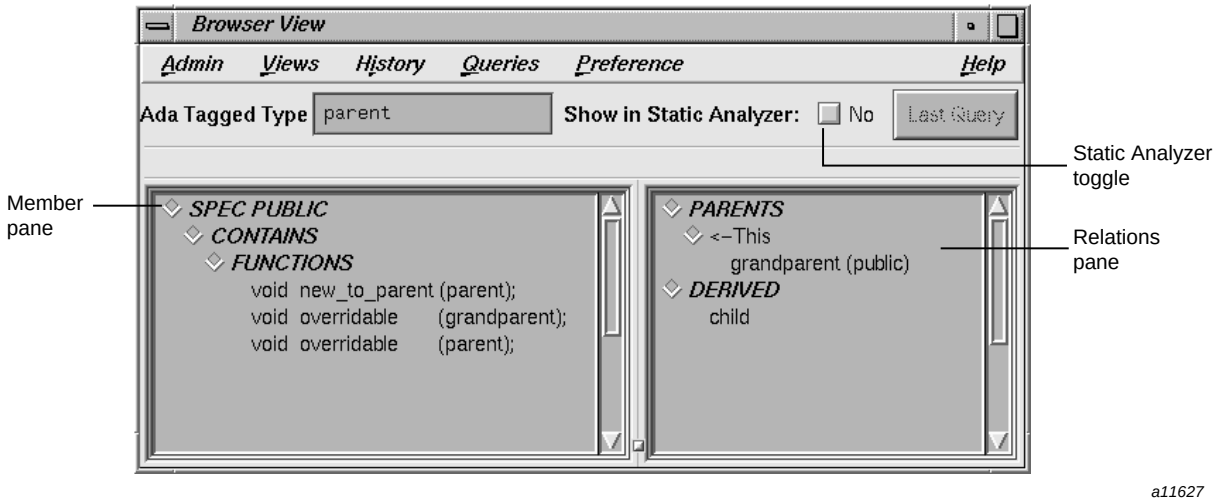

Figure 9-4 Browser View with Ada Data

2. Observe the **Browser View** window results.

   The member pane in **Browser View** is on the left. It displays members according to their accessibility: `SPEC PUBLIC`, `SPEC PRIVATE`, or `BODY`.

   The member pane displays these kinds of Ada members: `DATA`, `TYPE`, `FUNCTIONS`, `ENTRIES`, and `PRIMITIVE OPERATIONS`.

   The relations pane displays information on related Ada entities, based on the point of view of the current subject: `PARENTS` and `DERIVED`.

   You can customize the layout of both list displays.

3. Click the outline diamond icon to the left of the `FUNCTIONS` category (see Figure 9-4, page 101).

   This collapses the category, hiding the items. Outline icons with right-pointing arrows indicate that a category in the list is expandable, that is, that elements in the category are hidden from view.

4. Click the outline icon again to display the items.

Procedure 9-4 Making Queries

1. Click the **Queries** menu.

To learn the details about the structure of your Ada code, you make queries, which are questions about the current subject's members and related entities. Queries are a focused view of a large, complicated structure from the viewpoint of any Ada entity.

Queries search the static analysis database for specific information about subjects and their members. The Browser provides two types of queries menus:

- **Queries** menu — accessed from the menu bar, its queries apply to the current class
- Element-specific popup menus— Accessed by holding down the right mouse button while the pointer is over the selected element you want to query

The Browser answers queries by highlighting items in the member and related class lists that match the query. Optionally, you can display more detailed query results in the **Static Analyzer** window from which you launched the Browser.

2. Select **What Is Declared** from the **Queries** menu.

   The **Queries** menu in the menu bar lets you request relationship information for the current subject. In addition to highlighting the matching elements in the list, the Browser displays indicator marks in the scroll bar showing the relative locations of matching elements. Also, the query is identified in the field over the outline list area. If you click on an indicator mark, you will scroll directly to the matching element. See Figure 9-5.

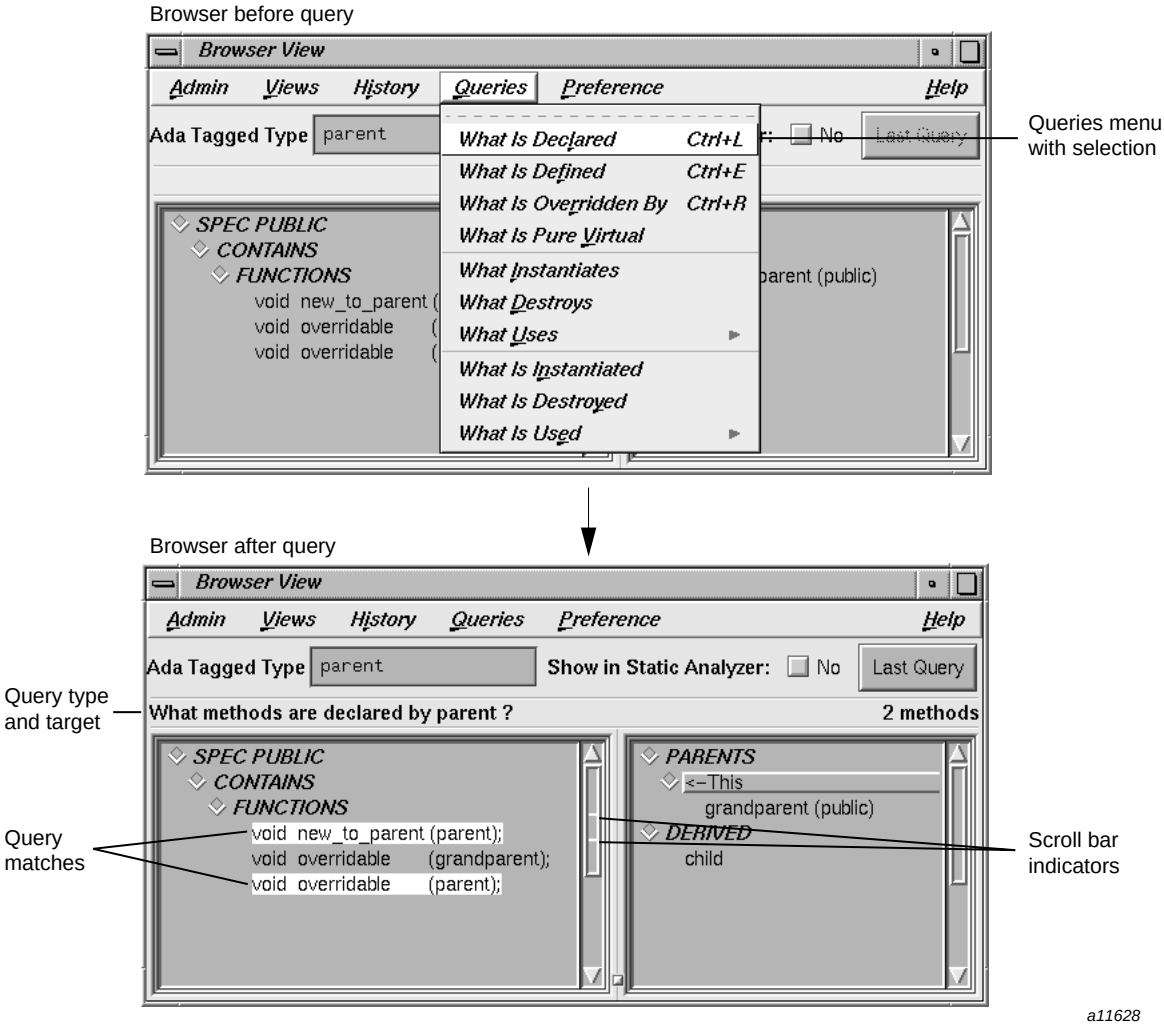

Figure 9-5 Performing a Query on Current Class

Procedure 9-5 Accessing Source Code

1. Select the `New_to_Parent` function, hold down the right mouse button over it, and choose **Show Source Where Declared**.

This displays **Source View** containing the source code where `New_to_Parent` is declared. See Figure 9-6.

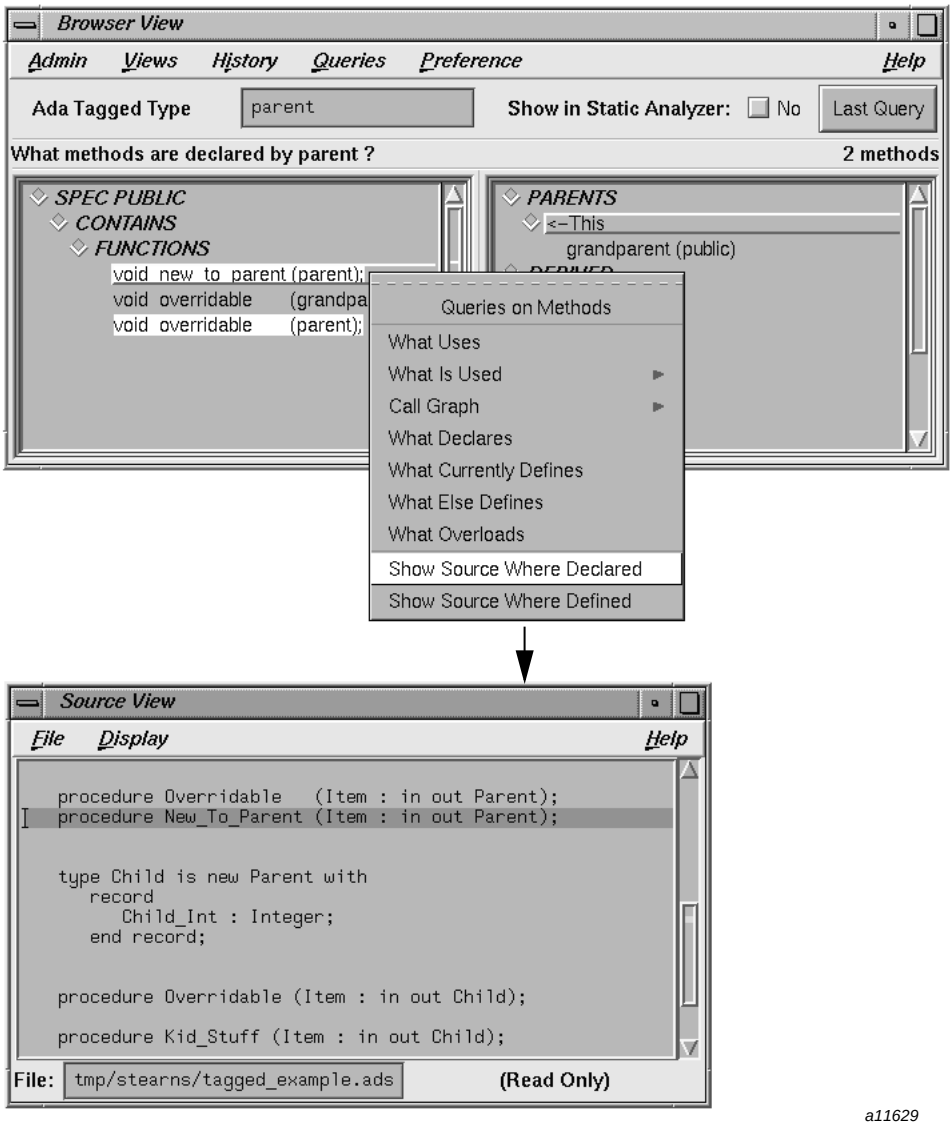

Figure 9-6 Accessing Source Code from the Browser View

Procedure 9-6 Using the Browser Graphical Views

1. Pull down the **Views** menu in the **Browser View** window and select **Show Inheritance Graph**.

   The **Inheritance Graph** window is displayed, as shown in Figure 9-7. You can switch to other relationship modes through the relation mode menu.

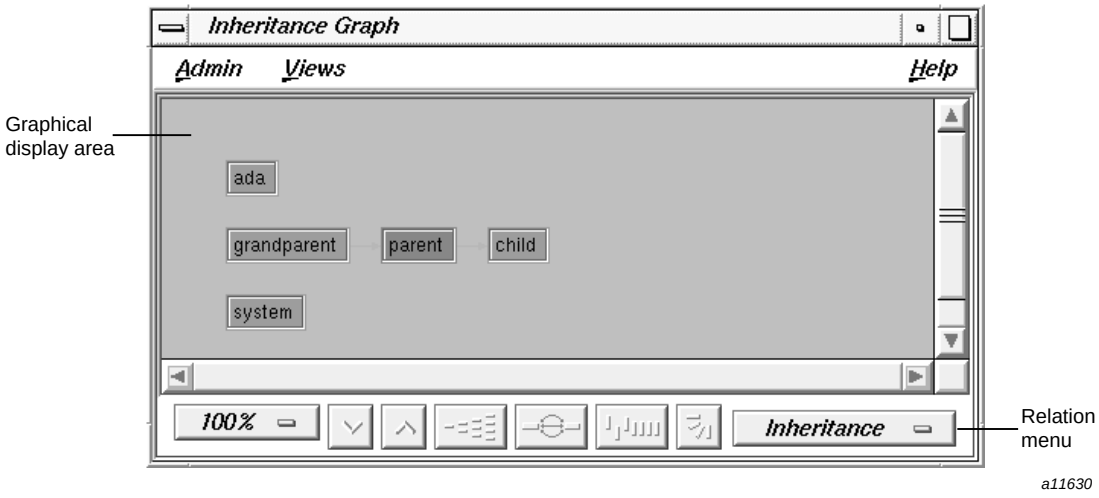

Figure 9-7 Inheritance Graph Example

Procedure 9-7 Shortcuts for Entering Subjects

1. Go back to the **Browser View** window, clear the **Current Subject** field, type a question mark (?), and press `Enter`.

   This is a shortcut for displaying the **Browsing Choices** window. However, instead of selecting the **Browsing Choices** window, follow this tutorial to see how name completion works.

2. Type **`grand`** and press the space bar.

   The Browser fills in the rest of the name (grandparent in this example) and its data.

This is the end of the Ada tutorial. You can exit both the Static Analyzer and the Browser by pulling down the **Static Analyzer Admin** menu and choosing **Exit**.

The Browser Reference

This chapter describes all of the windows and features associated with the Browser.

This chapter contains the following sections:

Browsing Choices Window

The **Browsing Choices** window (see Figure 10-1) lets you select items to be browsed from a list derived from the fileset in the **Browser View** window. Double-clicking an item in the selection list causes the **Browsing Choices** window to be raised (moved to the front) with the chosen item as the current subject for analysis.

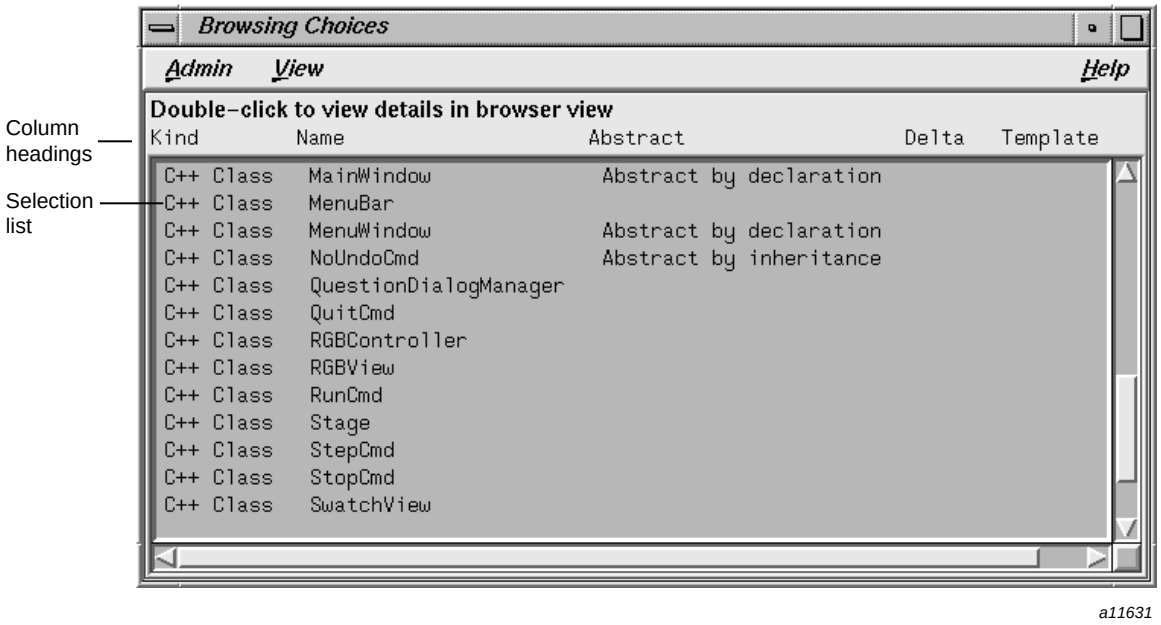

Figure 10-1 Browsing Choices Window

Browsing Choices Window for C++

With C++ code, the **Browsing Choices** window displays one column to indicate the kind of item, a column to identify the item, and three columns indicating properties, as follows:

- **Kind**: classes, template definitions, and template instances
- **Name**: the name of the item
- **Abstract**: abstract property: concrete (blank), abstract by declaration, or abstract by inheritance
- **Delta**: delta property: dynamic, internal dynamic, or non-dynamic (blank)
- **Template**: template property: specific definition, partial instantiation, or normal (blank)

The **Browsing Choices** window provides a facility for sorting items by column. To do this, click in the column you wish to sort on and select **Sort** from the **Admin** menu.

Browsing Choices Window for Ada

If you are using Ada, the **Browsing Choices** window displays packages, tasks, and tagged types in the **Kinds** column. The properties columns are not used in Ada and appear blank. You can sort the items by kind or name by clicking in the appropriate column and selecting **Sort** from the **Admin** menu.

Browser View Window

Browser View is the primary Browser window (see Figure 10-2). It opens when you select **Browser** from the **Admin** menu of the Static Analyzer, but does not display data until you select an item from the list in the **Browsing Choices** window. **Browser View** displays internal and related information for elements in C++ and Ada programs. The information is presented in hierarchical lists shown in outline format.

Browser View lets you perform a variety of static analysis database queries, depending on your current work context. Queries concerning the current subject are accessed from the **Queries** menu in the menu bar. You can also make queries specific to the selected elements in the list area by holding down the right mouse button to display a popup **Queries** menu specific to that type of element. The results of queries are indicated by highlighting matching elements in the **Browser View** window. Matching results are also highlighted in the **Source View** window (if it is displayed) and in the Static Analyzer (if the **Show in Static Analyzer** toggle (see "Show in Static Analyzer Toggle", page 111) is turned on) .

You can also launch graphical views showing hierarchies and call graphs from the **Browser View** window. In addition, you can generate man pages and web pages from **Browser View**.

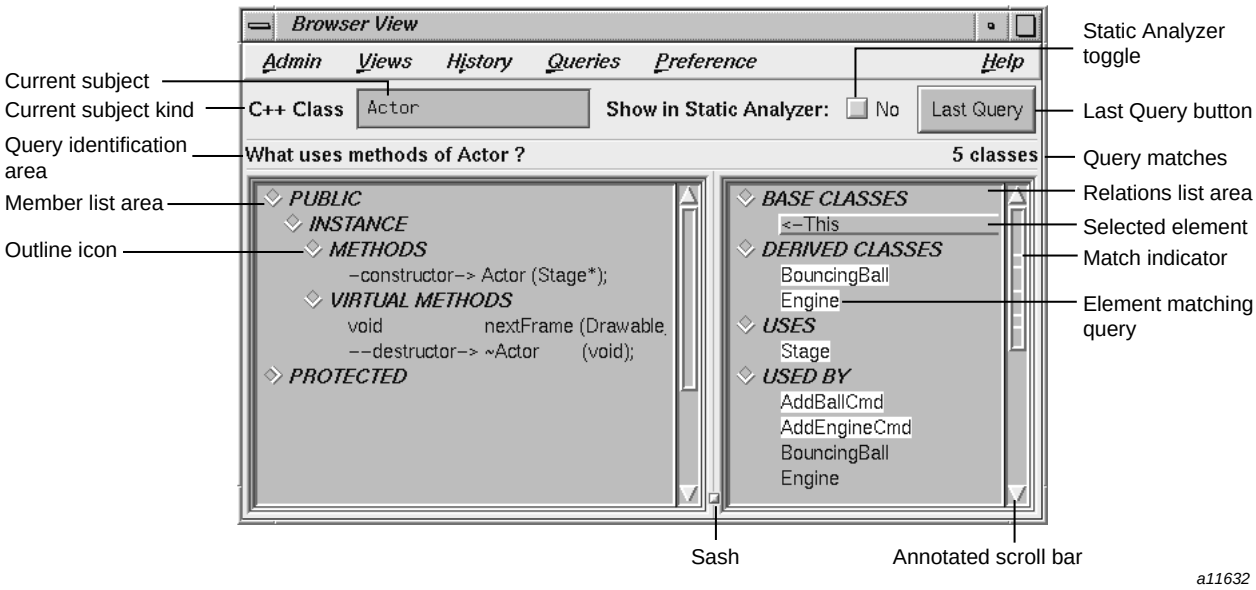

Figure 10-2 Browser View Window Elements

Current Subject Field

The **Current Subject** field indicates the kind and name of the element to be analyzed. It is directly below the menu bar (see Figure 10-2, page 110). The label on this field is initially set to **Current Subject**. To analyze an element, you can type directly into this field (or select from the **Browsing Choices** window). The label changes according to the kind of element you select. You can enter the following kinds of elements:

- C++ class
- C++ template definition
- C++ template instance
- Ada package
- Ada task
- Ada tagged type

If you type a partial string and then press the space bar, the Browser attempts to complete the element name by searching the fileset. A beep indicates that more than one matching name exists. If a match is made, press the `Enter` key to make the change effective.

If you type a question mark (?) into the **Current Subject** field, the **Browsing Choices** window opens. You can select a new item by double-clicking a name in the selection list.

Show in Static Analyzer Toggle

The **Show in Static Analyzer** toggle is directly to the right of the **Current Subject** field (see Figure 10-2, page 110). When the toggle is set (a check mark and the label **Yes** appear), the results of all queries are displayed in the **Static Analyzer** window from which the Browser was launched, including the file, line number, and source line for the matching items. If no results are found and the **Static Analyzer** window is open, the window comes to the front with an error message.

Last Query Button

The **Last Query** button is at the top right of the window, directly beneath the **Help** menu (see Figure 10-2, page 110). Clicking this button displays the results of the most recent query in the **Static Analyzer** window from which the Browser was launched.

Browser View Query Identification Area

The **Browser View** query identification area is directly above the list area (see Figure 10-2, page 110). This area displays the most recent query as a sentence containing both the query question and the name of the object of the query. The number of elements matching the query is displayed at the right end of the line.

Browser View List Areas

The lower two-thirds of the **Browser View** window consists of two lists displayed in side-by-side panes (see Figure 10-2, page 110). The lists contain information about the currently selected subject and are organized by category in an outline format. The lists are:

- member list: a detailed view of the internals of the current subject.

- relation list: items related to the current subject.

You can change the relative widths of the panes that display these lists by moving the sash that separates the panes.

The categories in the lists are different depending on whether you are using C++ or Ada code. Table 10-1, page 112, summarizes the contents of each list by programming language. For more information on the lists, see "C++ Member List", page 113, "C++ Relation List", page 115, "Ada Member List", page 116, and "Ada Relation List", page 118.

Table 10-1 Browser View List Summary

Language	Member List Contents	Relations List Contents
C++	PUBLIC/INSTANCE/PRIVATE INSTANCE/STATIC TYPES/DATA/METHODS/ VIRTUAL METHODS	BASE CLASSES (including the current class)/DERIVED CLASSES/USES/USED BY/FRIEND FUNCTIONS/FRIENDS/FRIEND OF
Ada	SPEC PUBLIC/SPEC PRIVATE/BODY CONTAINS /DATA/TYPE/ FUNCTIONS/ENTRIES/ PRIMITIVE OPERATIONS	PARENTS (including the current subject)/DERIVED

Outline Icons

Each category name appears with an outline icon to its left, that is, a diamond-shaped icon that can be used to collapse (hide) or expand (make visible) the items under that category. Inside the icon there is an arrow that indicates whether the category is in the expanded or collapsed state. If the arrow points downward, the list is in its expanded state, which means all items are displayed. If the icon points to the right, the category is in its collapsed state, which means all items in that category are hidden. Clicking the arrow toggles the state of the category, displaying or hiding the category's contents. Another function of the outline icon is to indicate when a collapsed list contains items matching the current query. This is shown with a filled outline icon. See Figure 10-3.

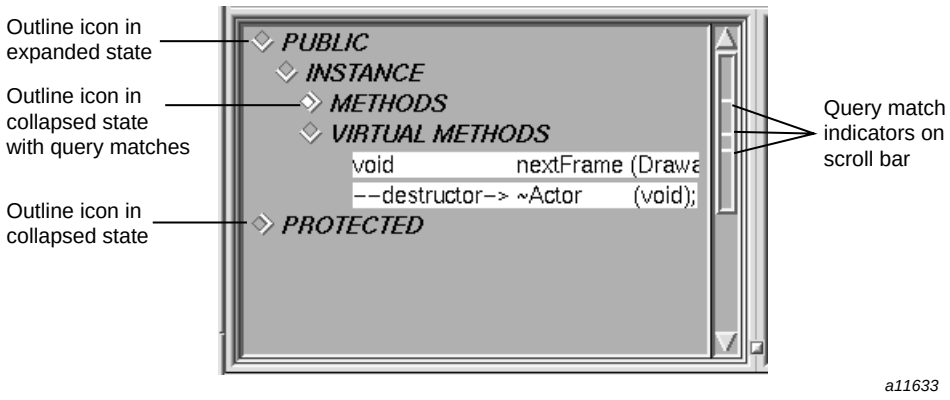

Figure 10-3 Outline List Icons and Indicator Marks

Annotated Scroll Bars and Highlighted Entries

Lists also use annotated scroll bars to locate highlighted list entries. When you make a query on an item in a list, the Browser displays indicator marks in the scroll bars in both panes corresponding to the relative positions of matching items. This informs you about all matches even if they are in collapsed categories or in a portion of the list that is not currently in view. If you click an indicator with the middle mouse button, you scroll directly to the matching item in the list. When the thumb of the scroll bar overlaps a given tick mark, the corresponding entry is visible in the list window. See Figure 10-3, page 113.

C++ Member List

The `Xmember` list displays the types, data members, methods, and virtual methods internal to the current class, template definition, or template instance when you are analyzing C++ code. It labels constructor methods as `-constructor->` and destructors as `-destructor->` .

Display Hierarchy

The members of the current class are sorted recursively into three nested lists according to the access specification (`PUBLIC`, `PROTECTED`, or `PRIVATE`) of each member. Within each of the access categories, the members are sorted by scope into two categories (`INSTANCE` and `STATIC`). Finally, within each category, members are

displayed by member category type in this order: TYPE, DATA, METHODS (member functions), and VIRTUAL METHODS.

Here is a schematic of the outline format for each nested list:

```
Access (PUBLIC, PROTECTED, or PRIVATE)
      Scope (INSTANCE or STATIC)
         TYPES
         DATA
         METHODS
         VIRTUAL METHODS
```

C++ Access Categories

The following access categories are available:

- PUBLIC members: accessible by any method or C-style function
- PROTECTED members: accessible only by methods in derived classes, friend classes, or friend functions
- PRIVATE members: accessible only by methods in the class in which they are defined, friend classes, or friend functions

C++ Scope Categories

The scope categories are as follows:

- STATIC members: all objects of a given class contain the same value for a given member
- INSTANCE (nonstatic) members: members in different instances of that class can contain different data values

C++ Class Member Categories

Class members fall into the following categories:

- TYPES: definitions of data types declared within a class
- DATA: variables that contain state information for a class
- METHODS (or member functions): definitions of how a class interacts with other classes and structures

- VIRTUAL METHODS: methods for an object that ensure that the method invoked is defined by the class from which the object was instantiated, regardless of type casting

The list organization is customizable. For more information, see Appendix A, "Customizing the Browser", page 135.

C++ Relation List

The C++ relations list displays the current class and its related classes in the class list. The categories in the list are:

- BASE CLASSES: ccontains the current class and its ancestors, listed hierarchically
- DERIVED CLASSES: ccontains descendants of the current class, listed hierarchically
- USES: contains classes that the current class uses (that is, instantiates, destroys, interacts with, or contains)
- USED BY: contains classes that the current class is used by
- FRIEND FUNCTIONS: contains global functions declared as friends by the current class
- FRIENDS: contains classes that are declared as friends by the current class.
- FRIEND OF: contains classes that declare the current class as a friend.

Within this list, the current class is displayed as follows:

```
<- This
```

This notation refers to the class in the **Current Class** field.

C++ Relations List Mouse Shortcuts

Double-clicking any displayed class brings up a **Source View** window that highlights the function's definition.

C++ BASE CLASSES Category Hierarchy

The BASE CLASSES category shows the ancestors of the current class, if any. Each indented class is an ancestor of the class listed above it. The BASE CLASSES category

indicates a multiple inheritance relationship by indenting parent classes to the same level. If a given class has ancestors, it is accompanied by an outline icon, which works in a similar manner to the outline icons in the member list. Each ancestor name is followed by its inheritance access type (PUBLIC, PROTECTED, or PRIVATE) listed in parentheses.

This schematic gives an example of a BASE CLASSES category:

```
BASE CLASSES
   <-This
     first_parent_of_This (access type)
                  parent_of_first_parent_class (access type)
     second_parent_of_This (access type)
                  parent_of_second_parent_class (access type)
```

C++ DERIVED CLASSES Category Hierarchy

The DERIVED CLASSEScategory shows the descendants of the current class, if any. Each indented class is a descendant of the class listed above it. If a given class has descendants, it is accompanied by an outline icon, which works in a similar manner to the outline icons in the base classes category and member list.

This schematic gives an example of a possible DERIVED CLASSES category:

```
DERIVED CLASSES
    first_child_of_This
       child_of_first_child_class
    second_child_of_This
       child_of_second_child_class
```

Ada Member List

The Ada version of the **Browser View** member list displays packages, task types, and tagged types as its current subjects. Packages have functions as their internal members. The internal members for task types are entries (under PUBLIC) and functions. Tagged types have primitive operations as their internal members.

Ada Display Hierarchy

The members of the current subject are sorted recursively into three nested lists according to the access specification (SPEC PUBLIC, SPEC PRIVATE, or BODY) of each member. Under each of the access categories lies the INSTANCE subcategory.

Finally, the members are displayed by member category type in this order: TYPES and DATA.

Here is a schematic of the outline format for each nested list:

```
Access (SPEC PUBLIC, SPEC PRIVATE, or BODY)
     Scope (INSTANCE)
        TYPES
        DATA
```

Ada Access Categories

The accessibility categories are different depending on the type of Ada entity.

The following access categories are available for packages:

- SPEC PUBLIC: includes declarations of data, functions, and types made in the public part of the package spec
- SPEC PRIVATE: includes declarations of data, functions, and types made in the private part of the package spec
- BODY: includes declarations and definitions of data, functions, and types made in the implementation of the package. These symbols are usable only within the package body.

The following access categories are available for tagged types:

- SPEC PUBLIC: includes data lists components of the tagged type. Functions list primitive operations of the tagged type.

Note: There is no SPEC PRIVATE or BODY section for a tagged type.

The following access categories are available for task types:

- SPEC PUBLIC: includes functions listed here are entries to the task. Types and data listed here are public (that is, they are usable by a client of the task).
- BODY: includes types, data, and functions used in the implementation of the task. These symbols are usable only within the task body.

Note: There is no SPEC PRIVATE section for tagged types

Ada Type and Data Member Categories

The other categories available are as follows:

- TYPES: definitions of data types declared by a package, task, or tagged type
- DATA: variables that contain state information for a package, task, or tagged type

The list organization is customizable. For more information, see Appendix A, "Customizing the Browser", page 135.

Displaying an Ada Member's Source Code

Double-clicking any member in the member list opens a **Source View** window that contains member code with the declaration highlighted. .

Ada Relation List

The Ada relations list shows parent-derived relationships between tagged types .

Browser View Menu Bar

The following sections describe the menus, found in the **Browser View** window's menu bar (see Figure 10-4).

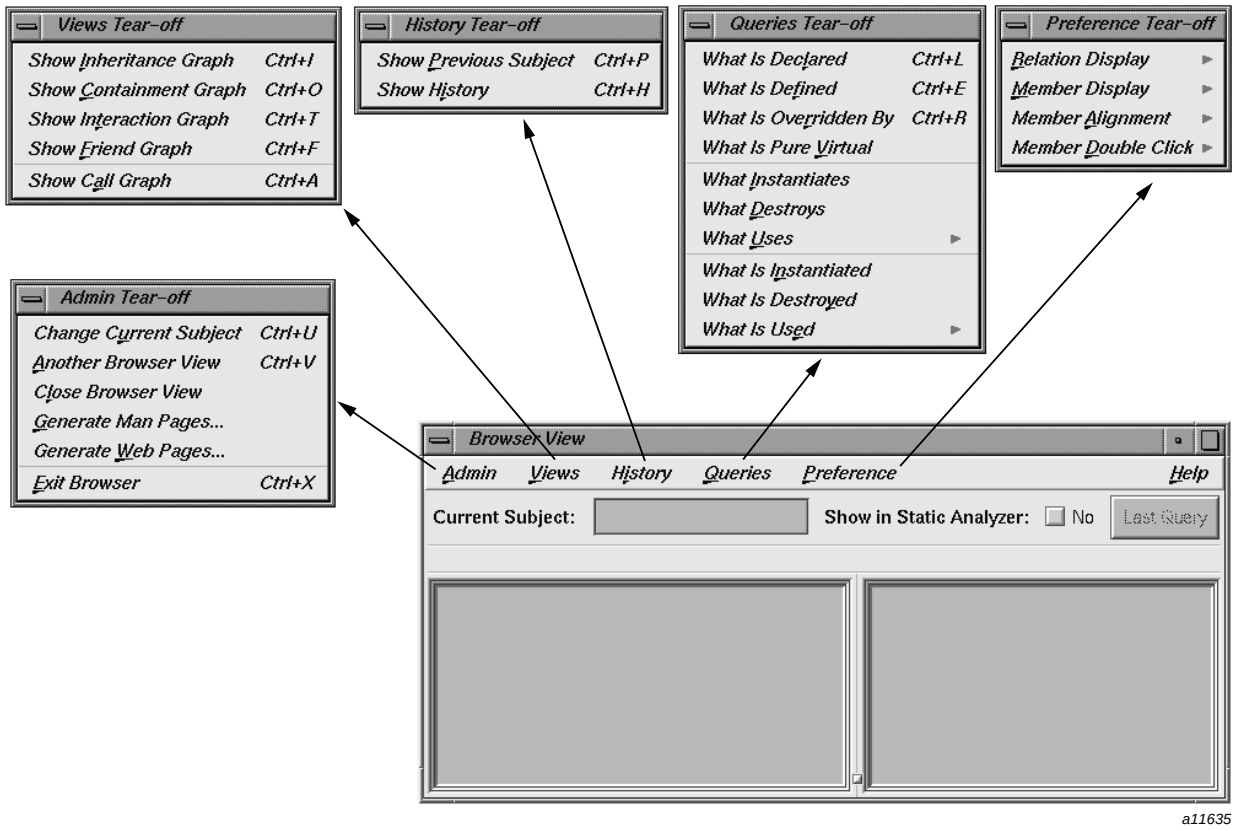

Figure 10-4 Browser View Menu Bar with Menus Displayed

Admin Menu

The **Admin** menu contains the following options for selecting new subjects, manipulating Browser View windows, generating man and web pages, and exiting the Browser View.

- **Change Current Subject**: lets you select a new current subject without manually typing it into the **Current Subject** field. Choosing this option opens the **Browsing Choices** window, which contains a scrolling list of all the classes or packages available from the current fileset. Double-clicking an item selects it for display in the **Browser View** window and closes the **Browsing Choices** window.

- **Another Browser View**: creates an identical copy of the **Browser View** window. All current information displayed within the initial window appears in the copy, but connections to the graphical view windows are not carried over to the new **Browser View** window.
- **Close Browser View**: shuts the **Browser View** window and any associated windows.
- **Generate Man Pages**: opens the **Man Page Generator** window, which lets you create man page templates for classes (C++), packages (Ada), tasks (Ada), and tagged types (Ada).

  Select individual subjects by clicking them. If you want a man page for every subject in the list, click **Select All**. To remove selections you have made, click **Unselect All**. Clicking the **Generate** button creates a man page template for each selected subject. If man pages exist for any selected subjects, the Browser warns you, unless you set the **Warn Overwrite** toggle to `No`.

  Output files go in the directory shown in the **Man Page Directory** field, if it exists. To specify a different output directory, click the **Set Directory** button in the **Man Page Generator** window and enter your choice.

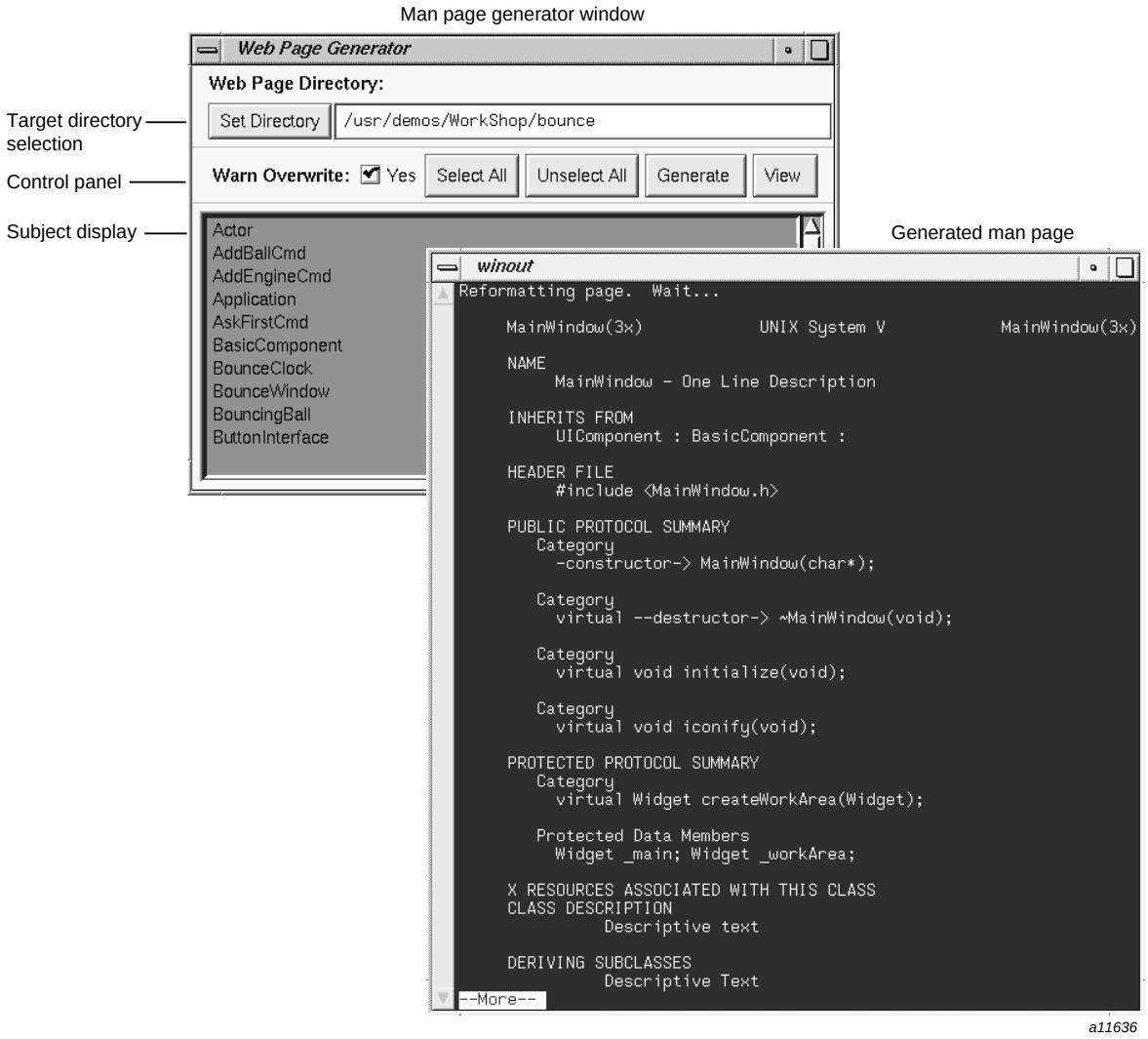

Figure 10-5 Man Page Generator and Typical Man Page Template

- **Generate Web Pages**: opens the **Web Page Generator** window (see Figure 10-6, page 122), which lets you create web page templates for classes (C++), packages (Ada), tasks (Ada), and tagged types (Ada). These templates are in HTML format and can be read by World Wide Web browsers.

Select individual subjects by clicking them. If you want a web page for every subject in the list, click **Select All**. To remove selections you've made, click **Unselect All**. Clicking the **Generate** button creates a web page template for each selected subject. If web pages exist for any selected subjects, the browser warns you, unless you set the **Warn Overwrite** toggle to `NO`.

Output files go in the directory shown in the **Web Page Directory** field, if it exists. To specify a different output directory, click the **Set Directory** button in the **Web Page Generator** window and enter your choice.

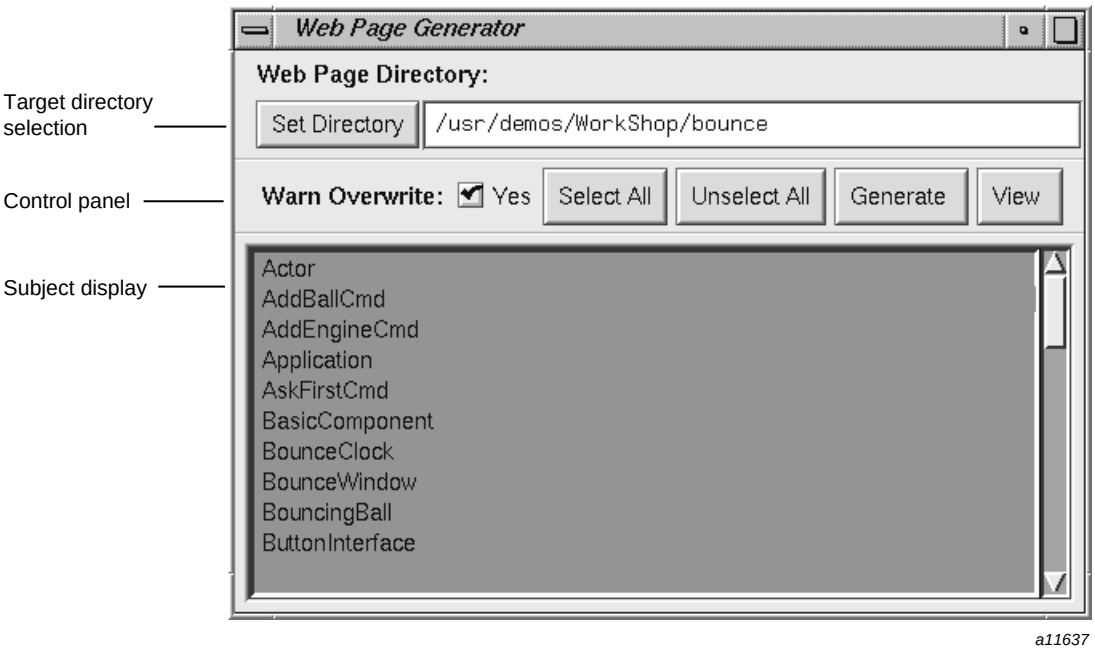

Figure 10-6 Web Page Generator Window

- **Exit Browser**: quits the Browser, closing all windows launched from it (except **Source View**). The **Static Analyzer** window from which the browser was launched is not affected.

Views Menu

The **Views** menu contains options for opening graphical views. Each of the first four selections opens a **Graph Views** window for the current class. The last selection opens a **Call Graph** window. The following selections are available from the **Views** menu:

- **Show Inheritance Graph**: describes the relationship between base classes and derived classes.
- **Show Containment Graph**: describes the relationship of container classes to the classes they use as components.
- **Show Interaction Graph**: describes the relationship of used classes to the classes that are their users.
- **Show Friend Graph**: describes the relationship of classes declaring friends to the classes they declare.
- **Show Call Graph**: opens a **Call Graph** window. To perform operations in it, select a method from the member list display, press the right mouse button to display the **Methods** popup menu, and select **Add**, **Remove**, or **Replace** from the **Call Graph** submenu.

History Menu

The **History** menu contains options that let you quickly select previously chosen subjects for display in the **Browser View** window. If no class was selected previously, a message appears. The following selections are available from the **History** menu:

- **Show Previous Subject**: sets the current subject to the previously displayed class, and the information in the **Browser View** window changes to reflect this.
- **Show History**: opens a **List of Subjects Shown** chooser window for selecting previously viewed subjects. The window presents the previous subjects in reverse chronological order, that is, the most recent subject appears at the bottom of the list.

  To select a subject, click it and press **Apply** or **OK**. Double-clicking a subject has the same effect as selecting OK. It makes the selection and closes the window. The selected class then becomes the current subject in the **Browser View** window.

Queries Menu

The **Queries** menu is accessed from the menu bar and applies to the current subject. The following selections are available from the **Queries** menu:

- **What Is Declared**: displays all methods declared by the current class.
- **What Is Defined**: displays all members defined by the current class.
- **What Is Overridden By**: displays all inherited methods that the current class overrides.
- **What is Pure Virtual**: displays all pure virtual functions in the current subject.
- **What Instantiates**: displays classes that instantiate the current class by invoking its constructors by using its new methods.
- **What Destroys**: displays classes that destroy the current class by invoking its destructors or by using its delete methods.
- **What Uses** submenu: displays classes that use the current class in the following contexts:
  - **To Contain**: displays classes that use the current class as either an embedded or linked component.
  - **As Friend**: displays classes that use the current class as a friend class.
  - **Methods**: displays classes that use the methods defined by the current class.
  - **Data Members**: displays classes that use (by modifying, reading, or taking the address) data members defined by the current class.
- **What Is Instantiated**: displays classes that the current class instantiates by invoking its constructors.
- **What Is Destroyed**: displays classes that the current class destroys by invoking it destructors.
- **What Is Used** submenu : displays those classes used by the current class in the following contexts:
  - **To Contain**: highlights classes that the current class uses as either embedded or linked components.
  - **As Friend**: highlights classes that the current class uses as friend classes.
  - **By Methods**: highlights classes whose methods are used by the current class.
  - **By Data Access**: highlights classes whose data members are assigned, read, or have their address taken by the current class.

Additional queries on subjects, data members, and methods are accessible from the popup menus described in "C++ Member List", page 113 and "C++ Relation List", page 115.

Preference Menu

The **Preference** menu allows you to control how the class information is displayed in the window.

The following selections are available:

- **Relation Display** submenu: allows you to control how the class relations are displayed:
  - **Declaration Order**: displays related classes in the order of their declaration or the detection of their relation.
  - **End To End Sort**: displays a sorted list of related classes.
- **Member Display** submenu : allows you to control how the class members are displayed:
  - **Declaration Order**: displays the members in order of their declaration.
  - **End To End Sort**: performs an end-to-end sort of the member display strings and displays the result.
  - **Name Sort**: performs a sort based on the name of the members and displays the result.
- **Member Alignment** submenu: allows you to control how members line up:
  - **Align Names**: aligns the member names in the display. A radio button indicates if this feature is enabled or disabled.
  - **Align Arglists**: aligns the member function argument lists in the display. A radio button indicates if this feature is enabled or disabled.
- **Member Double Click** submenu: lets you select which related source code is displayed in the **Source View** window when you double-click an item in the member list:
  - **Show Definition**: displays the source code where the item is defined.
  - **Show Declaration**: displays the source code where the item is declared.

– **Show Decl if no Defn**: displays the code where the item is defined; if there is no definition, then the source code containing the declaration is displayed instead.

Browser View Popup Menus

The **Browser View** popup queries menus provide queries for currently selected items in the outline list areas. These menu are accessed by selecting an item and then holding down the right mouse button. Figure 10-7, page 127, shows all of the popup menus available in the **Browser View** window.

This section describes the following menus:

- "Data Members Popup Menu", page 127
- "Methods Popup Menu", page 128
- "Class Popup Menus", page 129

Many of the same queries in the class popup menus appear in more than one menu. To eliminate this redundancy, each query is described once and presented in a single list rather than by menu.

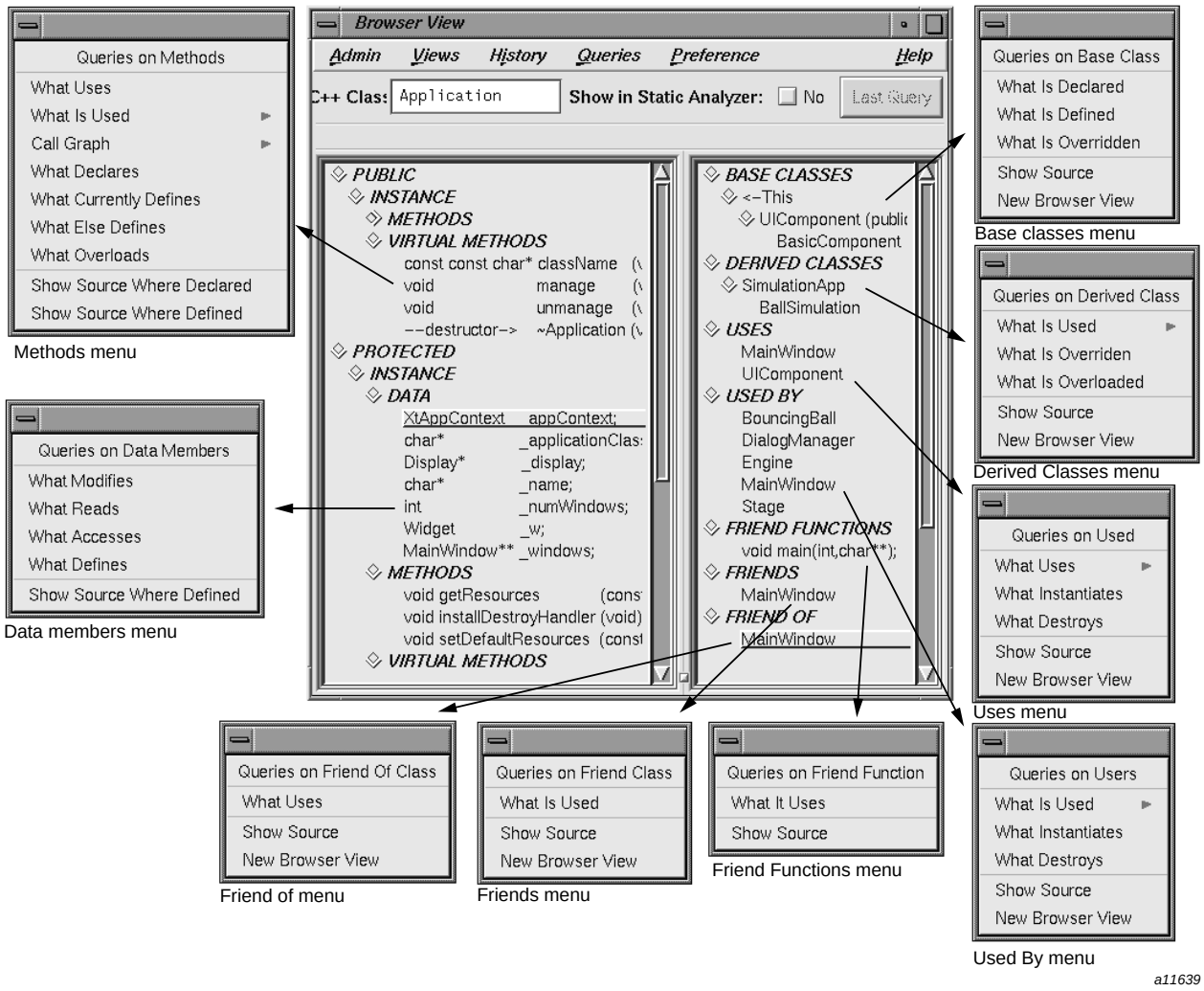

Figure 10-7 Queries Popup Menus in the **Browser View** Window

Data Members Popup Menu

The data members popup menu performs the following queries on data members selected in the member display list:

- **What Modifies**: highlights all methods and classes in which the selected data member is assigned a value.
- **What Reads**: highlights all methods and classes in which the selected data member is read.
- **What Accesses**: highlights all classes where the selected data member is assigned a value, read, or its address is taken.
- **What Defines**: highlights the class that defines the selected data member.
- **Show Source Where Defined**: displays the source code where the data is defined in a **Source View** window.

Methods Popup Menu

The Methods popup menu lets you perform the following queries on methods:

- **What Uses**: highlights all methods and classes that use the currently selected method.
- **What Is Used** submenu: highlights what is used by the currently selected method. Contains the following menu items:
  - **All (method and data access)**: highlights all data members, methods, and classes that the currently selected method uses.
  - **Method Calls**: highlights all methods called by the currently selected method.
  - **Data Access**: highlights all data members that have been assigned, read, or had their address taken by the currently selected method.
  - **Data Modification**: highlights all data members assigned by the currently selected method.
  - **Data Read**: highlights all data members read by the currently selected method.
- **Call Graph** submenu: the **Call Graph** submenu contains the following menu options:
  - **Add**: adds the currently selected method and its calling structure to the **Call Graph** window, if one is open. If not, **Add** opens a **Call Graph** window before adding the method.
  - **Replace**: replaces all methods in the display with the selected method and its calling structure in the **Call Graph** window.

  - **Remove**: removes the currently selected method and its calling structure from the **Call Graph** window.

- **What Declares**: highlights the class that declares the currently selected method.
- **What Currently Defines**: highlights the class that provides the current definition for the method.
- **What Else Defines**: highlights all classes that define the currently selected method.
- **What Overloads**: highlights all methods and classes that overload the currently selected method.

Class Popup Menus

This section describes the popup menus available in the related class list display. (The queries menu that displays when you select **<-This** is not shown here because it is exactly the same as the main **Queries** menu shown in "Queries Menu", page 123.)

Many of the items in the class popup menus are common to more than one menu. To eliminate the redundancy of describing them in each menu, this section presents all the queries in a single list in alphabetical order. The menus they belong to are shown in parentheses. The following selections are available on more than one menu:

- **New Browser View** (all menus except **Friend Functions**): opens a new **Browser View** window displaying the selected class.
- **Show Source** (all menus): opens a **Source View** window on a file containing the declaration of the selected item. The first line of the declaration is highlighted in the source.
- **What Destroys** (**Uses** and **Used By**): highlights all members of the current class that destroy the selected class.
- **What Instantiates** (**Uses** and **Used By**): highlights all members of the current class that instantiate the selected class.
- **What Is Declared** (**Base Classes**): highlights all methods declared by the selected base class.
- **What Is Defined** (**Base Classes**): highlights all members defined by the selected base class.
- **What Is Overloaded** (**Derived Classes**): highlights all members of the current class that are overloaded by the selected class.

- **What Is Overridden** (**Base Classes**): highlights all the methods of the selected base class that are overridden by the current class.
- **What Is Overridden** (**Derived Classes**): highlights all the methods of the current class that are overridden by the selected derived class.
- **What Is Used** (**Friends**): highlights all members of the current class that the selected friend class uses.
- **What Is Used** submenu (**Derived Classes** and **Used By**): contains the following queries:
  - **by Accessing Any Member**: highlights all members of the current class that the selected class uses.
  - **by Calling Methods**: highlights all methods of the current class that the selected class uses.
  - **by Accessing Data Members**: highlights all data members of the current class that the selected class modifies, reads, or takes the address of.
  - **by Modifying Data Members**: highlights all data members of the current class to which the selected class assigns a value.
  - **by Reading Data Members**: highlights all data members of the current class from which the selected class reads a value.
- **What It Uses** (**Friend Functions**): highlights all members of the current class that the selected friend function uses.
- **What Uses** (**Friend of**): highlights all members of the current class that use the friend class.
- **What Uses** submenu (**Uses**): contains the following queries:
  - **by Accessing Any Member**: highlights all members of the current class that use the selected class.
  - **by Calling Methods**: highlights all methods of the current class that use the methods of the selected class.
  - **by Accessing Data**: highlights all data members of the current class that modify, read, or take the address of data members of the selected class.
  - **by Modifying Data**: highlights all data members of the current class that assign a value to data members of the selected class.

– **by Reading Data**: highlights all data members of the current class that read a value from data members of the selected class.

Graph Views Window

The Browser provides a graphical view for showing relationships between classes in the fileset. It depicts classes as nodes and relationships as arcs. The **Graph Views** window shows the following types of class relationships:

- Inheritance
- Containment
- Interaction
- Friends

You can display graphical views by selecting any of the following items from the **Views** menu of the **Browser View** window:

- **Show Inheritance Graph**
- **Show Containment Graph**
- **Show Interaction Graph**
- **Show Friends Graph**

Once the **Graph Views** window is displayed, you can switch to any of the other relationships by using the **Relationship** menu at the bottom right of the **Graph Views** window.

Mouse Manipulations

Double-clicking any subject in the **Graph Views** window causes it to become the new current subject in both the **Browser View** and **Graph Views** windows.

Graph Views Admin Menu

The **Graph Views Admin** selections control which classes included in the current fileset are displayed in the **Graph Views** window. The **Admin** menu has the following selections:

- **Save Graph**: allows you to save the graph to a file. This selection brings up a file selection dialog. When you select a file and click **OK**, you save the graph as a PostScript file with the name specified in the **Selection** field.
- **Close**: closes the **Graph Views** window.

Graph Views Window Views Menu

The **Graph View** menu contains options that allow you to various types of classes.

- **Show All**: displays all classes included in the fileset as nodes, and their relations as arcs, as chosen from the relationship option menu.
- **Show All Related**: displays only those classes included in the chain of relations, which includes the current class.
- **Show Butterfly**: displays only those classes that are the immediate relatives (for example, parents and children for an inheritance relation of the current class).

Call Graph Window

The **Call Graph** window shows all calls made from selected methods in the member list, including calls made from its target methods. You can invoke it by any of the following methods:

- Select **Call Graph** from the **Views** menu in **Browser View**.
- Select a method in the member list, displaying the **Methods** popup menu, and selecting **Call Graph:Add**. This displays the **Call Graph** window the first time and adds new methods to the graph each time you select **Call Graph:Add**.

Figure 10-8 illustrates the second method for displaying **Call Graph**. In this example, the user has selected the initialize method in the **Browser** window and then selected **CallGraph:Add** from the **Methods** popup menu. The initialize method now appears in the **Call Graph** window with the methods that it calls.

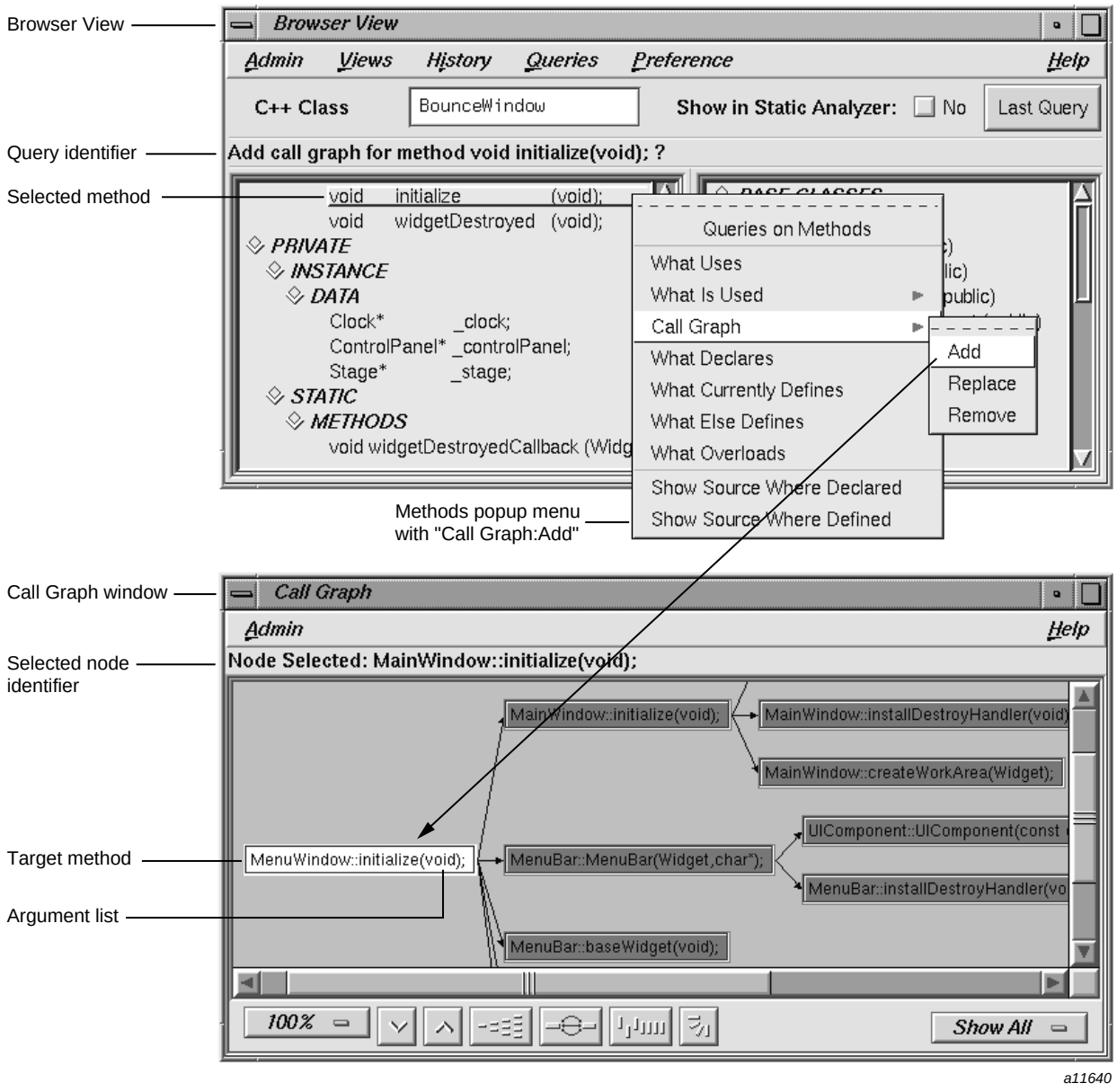

Figure 10-8 Displaying a Selected Method in **Call Graph**

Using the Call Graph Window

You can add, replace, or remove methods in the **Call Graph** window by choosing from the **Call Graph** submenu in the **Methods** popup menu in the **Browser View** member list (see Figure 10-8, page 133), as follows:

- **Add**: adds the currently selected method and its calling structure to the **Call Graph** window, if one is open. If not, **Add** opens a **Call Graph** window and then adds the method.
- **Replace**: replaces all methods in the display with the selected method and its calling structure in the **Call Graph** window.
- **Remove**: removes the currently selected method and its calling structure from the **Call Graph** window.

The action you request is displayed in the message area in **Browser View** window. In the **Call Graph** window, there is also a message area that identifies the method and its arguments.

In the **Call Graph** window, double-clicking any method node opens a **Source View** window that displays the code defining the method. The definition is highlighted in the source.

For information on manipulating graphs, the *ProDev WorkShop: Overview*.

Call Graph Admin Menu

The **Call Graph** window's **Admin** menu contains the following selections:

- **Show Arglist** toggle: lets you display or hide the argument list for each method, as shown in Figure 10-8, page 133.
- **Clear**: removes all methods from the **Call Graph** window.
- **Save Graph**: displays a file selection dialog for saving the graph to a PostScript file.
- **Close**: closes the **Call Graph** window.

Customizing the Browser

The Browser lets you customize your display and the way you work with man pages. These formats are implemented as X application resources that you can redefine in your local `.Xdefaults` file. After editing it, run the following command:

```
xrdb .Xdefaults
```

Then reopen the Static Analyzer.

This appendix covers the following topics:

Customizing the Browser View Lists

The following sections show you how to customize the formats of **Browser View** lists by applying your own keyword headers and rearranging the features of each list.

Member List Resource

The layout of the **Browser View** member list is controlled by the `Cvstatic*memberOrder` resource.

The general format of this resource is as follows:

Level-1-keyword: *HEADING* [*keyword*], *HEADING* [*keyword*], . . ;
Level-2-keyword: *HEADING* [*keyword*], *HEADING* [*keyword*], . . . ;
Level-3-keyword: *HEADING* [*keyword*], *HEADING* [*keyword*], . . . ;

The three keywords are `Protection`, `Scope`, and `Member`. The order in which these keywords are used determines the level of nesting in the outline list used for protection, scope, and member headings, respectively.

Headings may consist of any string you choose to describe the heading category. The headings listed with the level-1 keyword become top-level headings in the outline list, the level-2 headings appear indented under each of the level-1 headings, and the level-3 headings appear indented beneath each of the level-2 headings.

Each heading in a level has an associated keyword that determines the sort of items that appear under the heading. The allowable keywords are as follows for each associated level keyword:

`Protection:`	[`public`], [`protected`], [`private`]
`Scope:`	[`instance`], [`static`]
`Member:`	[`type`], [`data`], [`method`], [`virtualmethod`]

It is also possible to combine the types associated with two or more keywords under one heading by using the construction for any given heading:

HEADING [*keyword1*+*keyword2*+`...`]

You can also control whether a heading is expanded or collapsed when the browser starts up. Placing an asterisk (*) at the end of the heading string causes that heading to be collapsed by default:

HEADING`*` [*keyword*]

The default assignment for the outline resource of the member list can be found in `/usr/lib/X11/app-defaults/Cvstatic`. The contents of the file appear below:

```
Cvstatic*memberOrder:   Protection: PUBLIC [public],
PROTECTED* [protected], PRIVATE* [private]; Scope: INSTANCE
[instance], STATIC [static]; Member: TYPE* [type], DATA
[data], METHODS [method], VIRTUAL_METHODS [virtualmethod];
```

Note: The sample above is a single line.

You can override this definition by placing your own definition in your local `.Xdefaults` file.

Related Class List Resource

The layout of the **Browser View** related class list is controlled by the `Cvstatic*relationOrder` resource.

The construction of this resource is similar to the member list, but simpler:

HEADING [*keyword*], *HEADING* [*keyword*],`...`

The headings and keywords work as described for the member list, but there is no concept of level keywords in the related class list.

The allowable keywords for the related class list are as follows:

`[base], [derived], [uses], [usedby], [friendfunction],[friend], [friendof]`

Note: In the related class list, headings cannot contain multiple keywords, as they can in the member list.

As in the member list, you can control whether a heading in the related class list is expanded or collapsed when the browser starts up. Placing an asterisk (*) at the end of the heading string causes that heading to be collapsed by default:

*HEADING** [*keyword*]

The default assignment for the related class list outline resource can be found in `/usr/lib/X11/app-defaults/Cvstatic`, and is listed below for your convenience:

```
Cvstatic*relationOrder: BASE CLASSES [base], DERIVED CLASSES
[derived], USES [uses], USED BY [usedby], FRIEND FUNCTIONS
[friendfunction], FRIENDS [friend], FRIEND OF [friendof]
```

You can override this definition by placing your own definition in your local `.Xdefaults` file.

Other Browser View List Resources

X Windows System resources, found in `/usr/lib/X11/app-defaults/Cvstatic`, can be modified in your local `.Xdefaults` file. The default values are listed with each resource. You can set any `true` value to `false`.

`Cvstatic*completeClassName: true`

Enables `ClassName` completion. By typing a space in the current class field, you complete a class name from the list of classes in the fileset (if set to `true`, as it is by default).

`Cvstatic*showMessageArea: true`

Enables the message area in the **Browser View** window (if set to `true`, as it is by default).

`Cvstatic*scream: true`

Enables warning beeps when there are 0 results for a query, or when a class name has more than one completion in the current class field (if set to `true`, as it is by default).

`Cvstatic*indentationWidth: 15`

Sets the indentation in the outline lists in pixels. Default setting is `15`.

`Cvstatic*nameAlign: true`

Aligns names of the members under the same parent so that the type declarations and member (variable and function) names form left-justified columns (if set to `true`, as it is by default).

`Cvstatic*arglistAlign: true`

Aligns the argument lists of member functions under the same parent so they form a left-justified column (if set to `true`, as it is by default).

`Cvstatic*sort: true`

Sorts items in the outline lists based on the value of the entire string denoting an item (if set to `true`, as it is by default). For example, given two members, `void f` and `int k`, the Browser lists `int k` before `void f` in the list.

`Cvstatic*nameSort: true`

Sorts items in the outline lists based on the string value of the name of a member (if set to `true`, as it is by default). For example, `void f` would be listed before `int k`).

If you use the last two resources in conjunction, output is sorted first by type and then by name, as shown in .

Table A-1 Sort Resources for Outline Lists

Sort	Name	Sort Effect
`false`	`false`	Members are in declaration order
`false`	`true`	Members are sorted based on the name and not on type or return type.
`true`	`false`	Members are sorted based on the their return type or type. Within the same return type, members appear in declaration order.
`true`	`true`	Members are sorted both on their type or return type and their name. This is the default behavior.

Customizing Man Page Generation

The resources in this section are associated with the **Man Pages for Classes** window, available from the **Browser View Admin** menu item **Generate Man Pages**.

`Cvstatic*manPageDirPath:`*manpage directory path*

The default directory is the current directory (`.`). To place generated man pages in the `windTunnel` directory that you have created use the following command:

```
Cvstatic*manPageDirPath: ./manpage/windTunnel
```

`Cvstatic*manPageSuffix:` `.`*suffix*

The default suffix is 3, which would make the name of a man page:

class_name`.3`

To change the suffix to 4, use the following command:

```
Cvstatic*manPageSuffix: .4
```

`Cvstatic*manPageViewCommand:` *commands*

Clicking the **View** button in the **Man Pages for Classes** window executes the command specified by this resource. The argument given is the set of man pages for the classes that are selected. By

default, **View** displays the most recently generated man page in a read-only window. The default commands are:

```
Cvstatic*manPageViewCommand: winterm -H -c man -d
```

`Cvstatic*manPageCopyRightMessage:` *string*

Lists standard copyright information. You can customize the message.

Index

D

E

F

L

M

S

T

X

Z